DevOps

Spring 2022

Group L final report

Lars Djursner Rasmussen	ladr@itu.dk
Alexander Majgaard Wermuth	alwe@itu.dk
Jacob Mølby	jacmo@itu.dk
Tobias Vestergaard Svendsen	tovs@itu.dk
Christian Gerdes	cger@itu.dk
Aske Wachs	askw@itu.dk

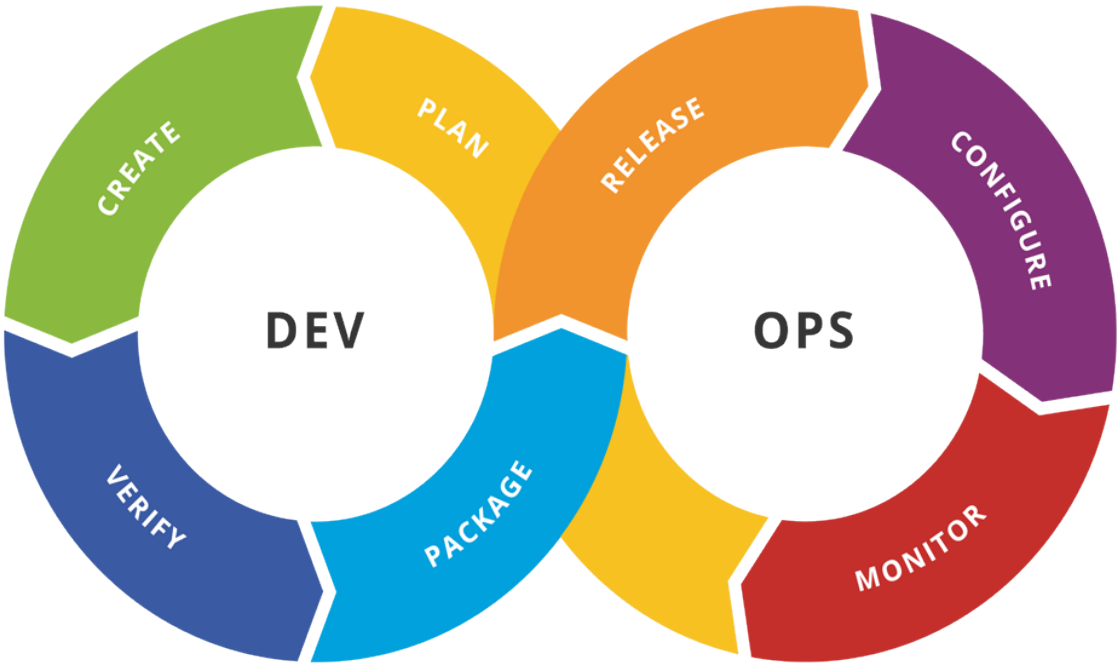


Figure 1: Source: `infolytx.com`

Software development

IT-University of Copenhagen

Denmark

June 1, 2022

Contents

1 Introduction

This report details our approach to designing, developing, and maintaining the ITU-minitwit system while learning DevOps practices in the course *DevOps, Software Evolution, and Software Maintenance*.

The report's content is divided into three sections. The first section, *System's Perspective*, describes the system's design and architecture, as well as its dependencies and licenses. The second section, *Process' Perspective*, revolves around developer interactions, project organization, CI/CD, and security, among other relevant topics. The third section, *Lessons Learned Perspective*, discusses the experiences and knowledge gained throughout the project.

Project repository: https://github.com/aske-w/itu-minitwit

2 System's Perspective

2.1 Design and Architecture of your ITU-MiniTwit systems

In this section we will show the design and architecture of the systems we implemented. To do this we use the 3+1 model by Christensen et al. [3].

2.1.1 Module viewpoint

The module viewpoint is visualized with a package diagram. This can be seen in figure 2.

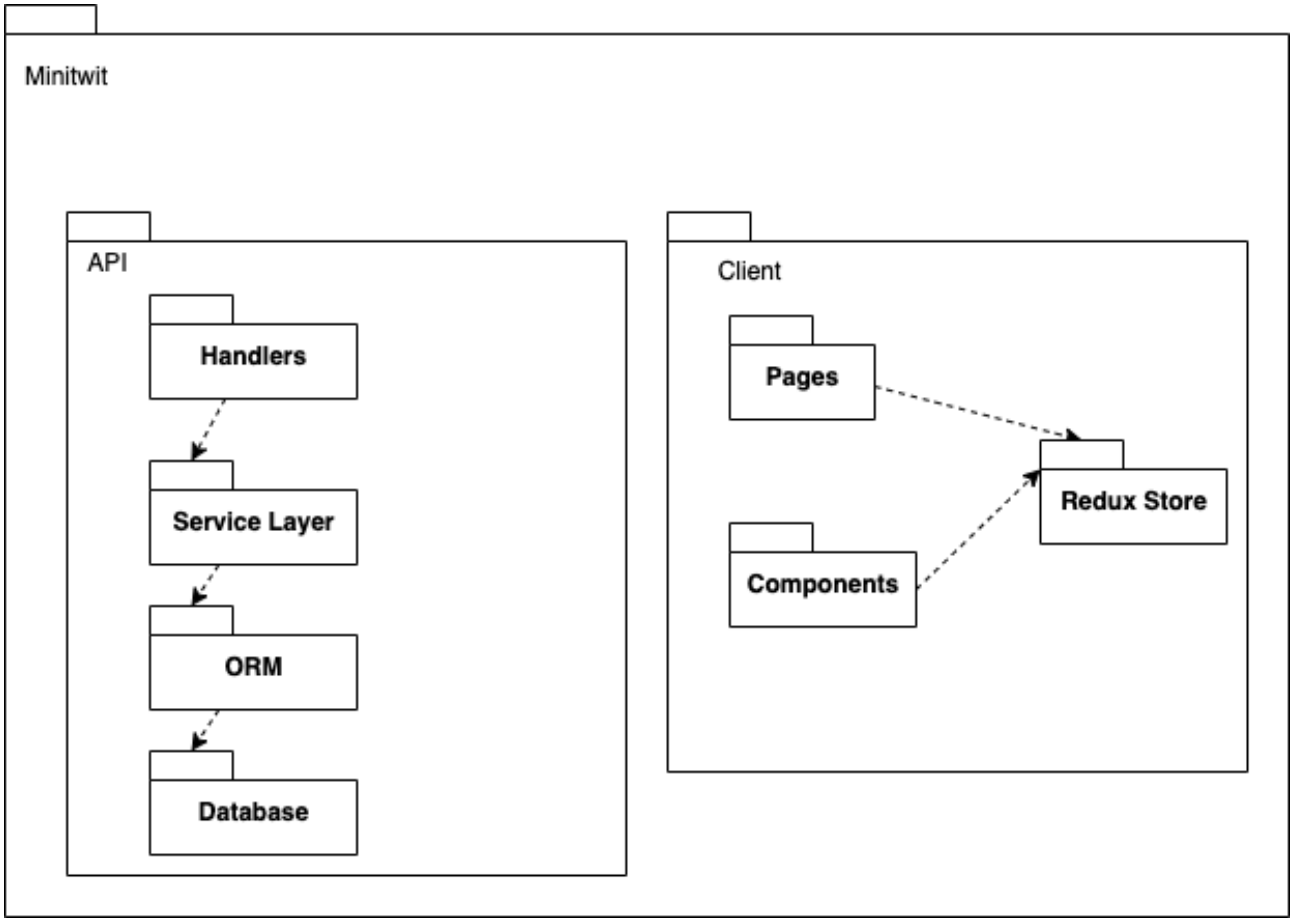


Figure 2: Shows the different packages for the API (server) and the Client (web application).

2.1.2 Component & Connector viewpoint

Figure 3 shows the components of our system and how they can be accessed with HTTP from the browser.

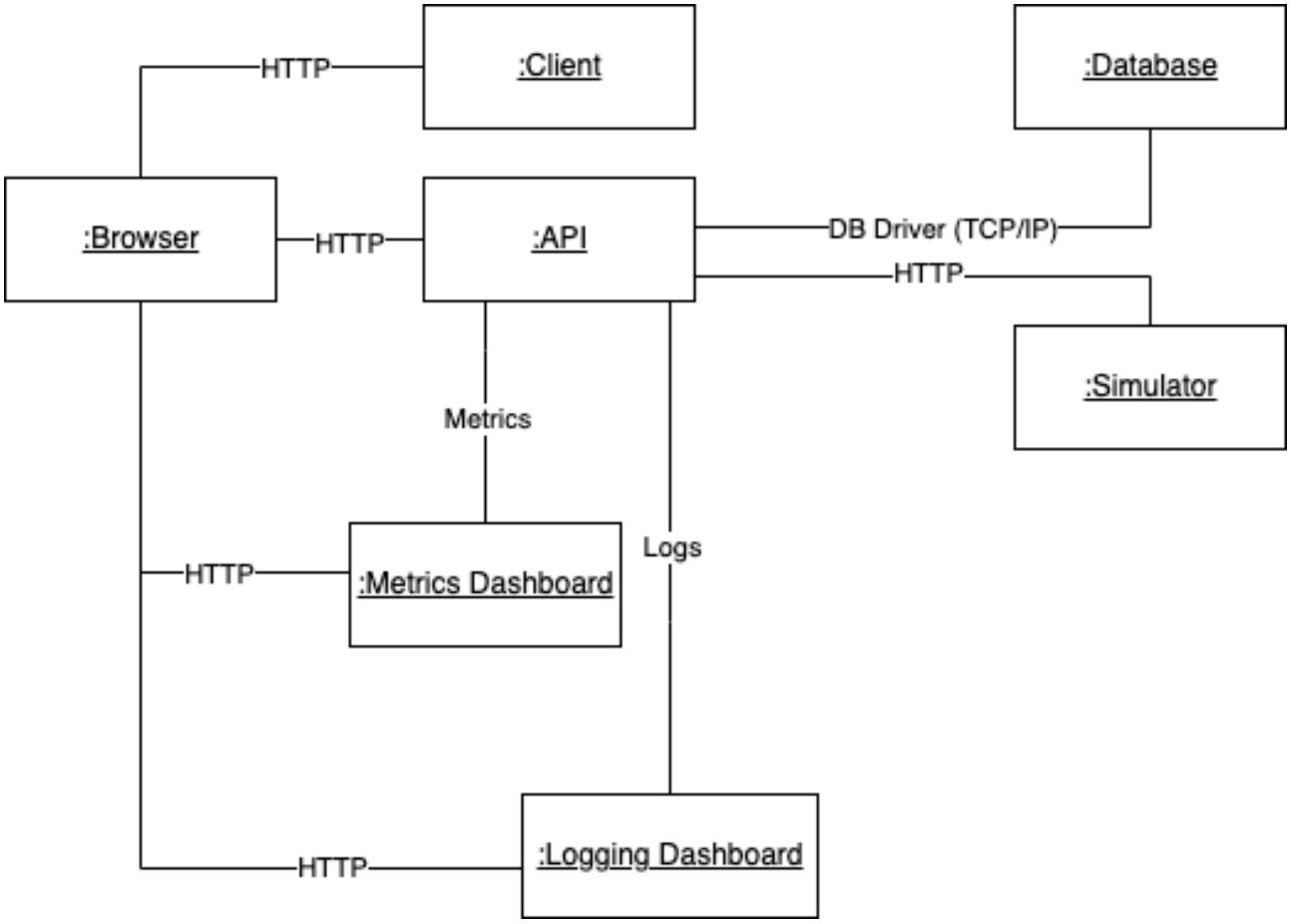


Figure 3: Diagram over our components and how they connect with each other.

2.1.3 Allocation viewpoint

The allocation viewpoint concerns itself with the deployment of the systems. This can be seen in figure 4.

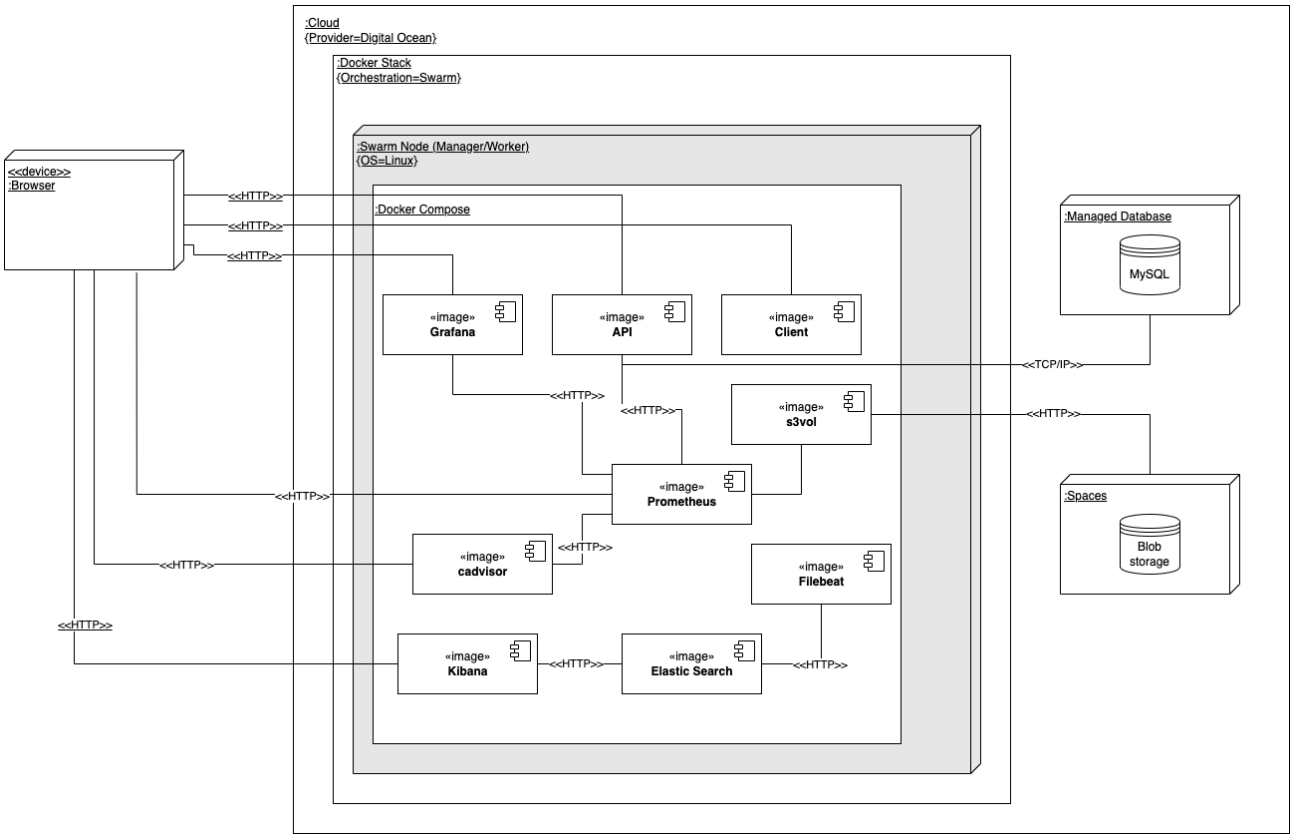


Figure 4: Deployment diagram over the different items in our system.

2.2 Dependencies

We chose to replace the original Python implementation, with an implementation in Golang for the server application. Additionally, we split the application into server and client, and chose to implement the client in React. The dependencies associated with the client/server approach is listed below, alongside the dependencies associated with logging & monitoring, infrastructure, and the development process.

2.2.1 Backend application dependencies

- **Iris**: Go web framework.
- **GORM**: Object-relational mapper for Go, with MySQL and Prometheus drivers.

2.2.2 Frontend application dependencies

- **React**: JavaScript library for building user interfaces.
- **React-router-dom**: Client based routing.
- **Redux**: State management.
- **Axios**: HTTP client.
- **TailwindCSS**: CSS framework.

2.2.3 Logging & Monitoring dependencies

- **cAdvisor**: Container metrics.
- **Prometheus**: Collecting monitoring data.
- **Elastic Search**: Storing and search logs.
- **Filebeat**: Export log collections to Elastic Search.
- **Kibana**: Visualization of logs in Elastic Search.
- **Grafana**: Dashboard to visualize Prometheus metrics.

2.2.4 Infrastructure dependencies

- **DigitalOcean Droplet**: Virtual machine instances.
- **DigitalOcean Managed Databases**: MySQL database cluster with horizontal and vertical scaling of read-only nodes and vertical scaling for single write node (master).
- **Docker Hub**: For storing and retrieving docker images.

2.2.5 Development dependencies

- **Git**: Distributed version control system
- **Github Actions**: Automatic actions for testing, analysis, deployment and releases.

2.3 Important interactions of subsystems

To show the interactions of the subsystems, we used the action of a client posting a message as an example. This shows how the different subsystems of Client, API, Logging, Database and Metrics interact in a typical scenario, which is visualized in a sequence diagram in figure 5.

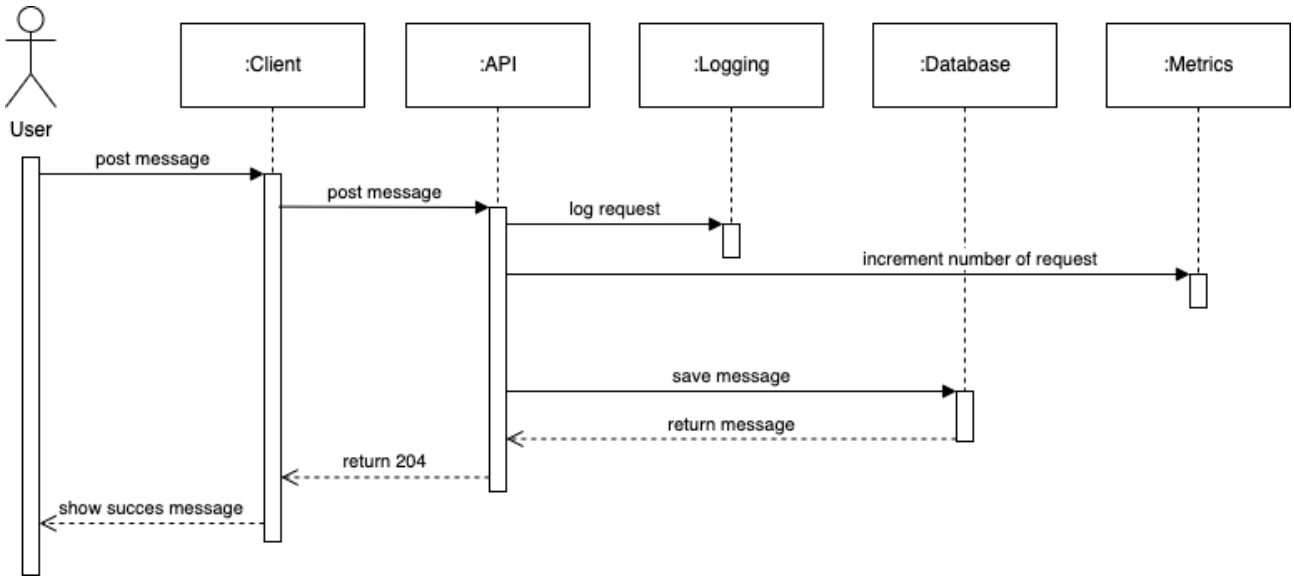


Figure 5: Sequence diagram over the action of *posting a message*.

2.4 Static analysis and quality assessment

To ensure a certain level of quality within the repository, various tools for static analysis have been used. Every time a pull request is made to the *development* branch, a GitHub Action is run. This action use the following tools *gofmt*, *gosec*, *vet* and *lichen*. *Gofmt* formats the code programmatically to ensure consistency. Vet checks the code for errors that the compiler might not find. *Gosec* scans the binary for potential security vulnerabilities. Without the tools above running from the command-line, we also integrated with third-party tools *SonarCloud*, *Code Climate* and *Better Code Hub*. The tools mostly show no problems related to the Go application. However, certain warnings are caused by the simulator and the original Python implementation, which are both included in the repository. The following are some of the areas that can be improved. Badges for overview can be seen in appendix 15.

- Explicit error handling everywhere.
- Reduce cognitive complexity and length of some functions.

Our API has throughout the project been among the fastest, as seen in the figure 6.

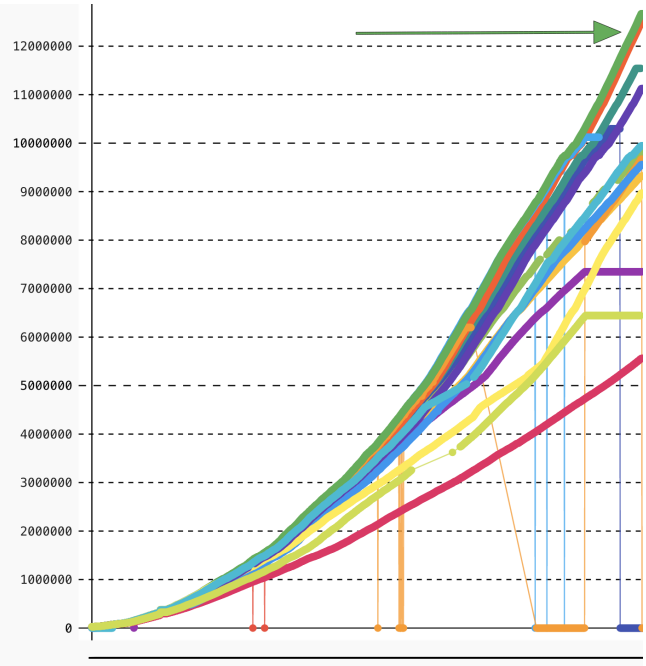


Figure 6: Latest number returned by the API. The green line, which the arrow points to, is ours.

A few times, we had issues with our database giving the error *failed to read auto-increment value from storage engine*. This resulted in errors being thrown from our application as seen in figure 7. We fixed the error and since then, no errors have been returned from the API.

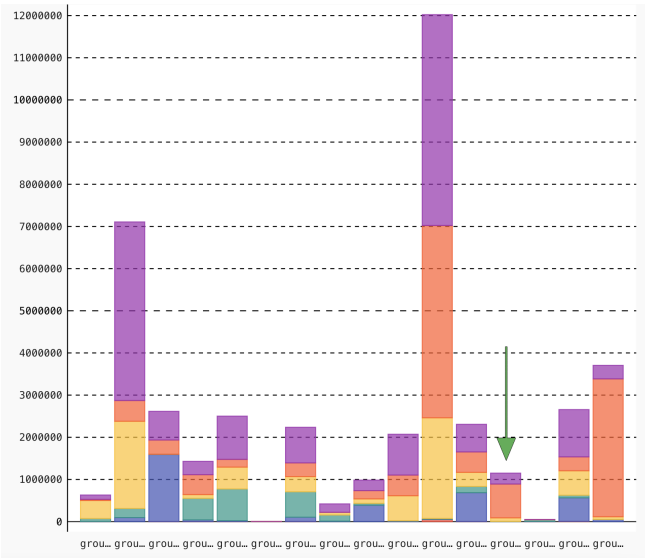


Figure 7: Errors returned by the API. The green arrow shows ours.

2.5 License

We chose the MIT license, which is a permissive license that is commonly used with open-source software. We use a Go package called *Lichen* to check if we are compliant with the licenses of our dependencies. This has been implemented as a Github Action CI/CD check. The MIT license is compliant with all of our dependencies, which can be seen in appendix B.

3 Process' perspective

3.1 Developer interactions and teamwork

Throughout the project, we have held meetings alongside development in person every Tuesday, following the weekly lectures in the course. In between these meetings, we have resorted to using Facebook Messenger and Discord for asynchronous communication. Usually, the group has been divided into sub-groups of developers, dependent on the complexity of the work.

3.2 CI/CD

We use GitHub Actions as our continuous integration and continuous delivery platform. We have three separate workflows/pipelines configured, that run independently of each other. Pull requests to our development branch will trigger the test workflow. This workflow will run all tests against the development branch. We also run a static analysis workflow concurrently with the test workflow. Static analysis includes multiple steps. Initially, the code is formatted and then vetted for suspicious elements. Afterwards, a security scanner will check the entire code base for security flaws. Finally, we scan all of our dependencies' licenses to ensure we are using these dependencies under their license agreements. If any steps fail, the entire workflow will render itself a failure. We will only be able to merge the pull request into the development branch if both workflows succeed.

When pull requests have been merged into the development branch, we can deploy the changes by creating another pull request from the development branch to the main branch. This will trigger the deployment

workflow. The deployment workflow includes a few steps:

1. Checkout the code
2. Authenticate with Docker Hub
3. Build and push a new tagged docker image for the server to Docker Hub
4. Build and push a new tagged docker image for the client to Docker Hub
5. Copy ./swarm/docker-compose.yml, ./swarm/filebeat.yml, ./swarm/grafana/ and ./swarm/prometheus/ to the server using SSH
6. SSH into worker server and ensure correct permissions
7. SSH into server and pull the newly pushed images from Docker Hub and deploy the swarm
8. Tag the commit and create a GitHub release with a change log

3.3 Organization of repository and branching

We use a mono-repository structure to store all of our code and assets. The client and server code is split into separate folders within the same repository. The repository also contains configuration folders for Filebeat, Grafana and Prometheus.

3.3.1 Branching Strategy

We have chosen to keep our branching strategy simple with the Git-flow branching strategy combined with a locked main branch. This means that nobody can push directly to it. Every change to the main branch must be made through pull requests from the development branch. All pull requests to the main branch are required to be reviewed by at least one collaborator before it can be merged. It must also pass all automated actions.

The development branch should always be either ahead of main or up-to-date with main. When developing new features, collaborators branch out from development and later create a pull request to development. Basically, we have the following branches: main, development, hotfix/*, and feature/*.

The *main* branch is used as production, and a *development* branch is used for development. A new feature will be worked on in a separate *feature* branch that is branched out from *development*. When a feature is complete, the associated branch is merged into *development*. The *main* branch will pull from development to release stable changes to *production*. We used the naming convention of prepending *feature/** and *hotfix/** to distinguish between feature and hotfix branches.

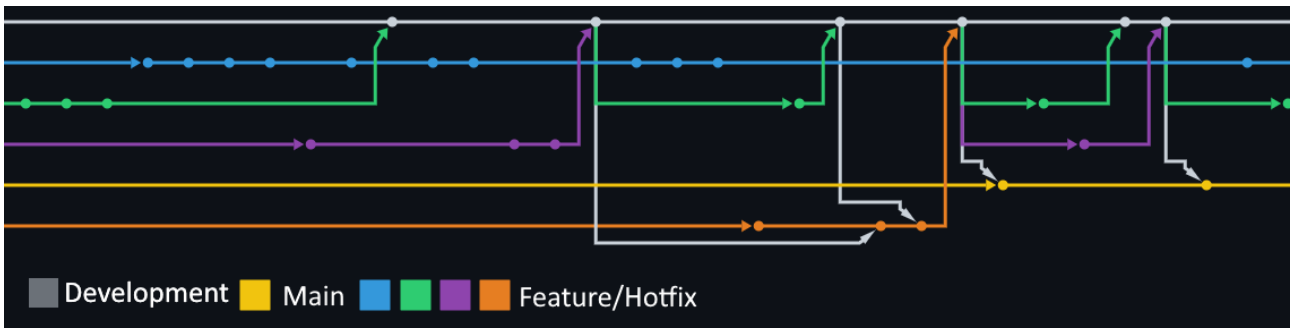


Figure 8: Visualized network graph of branches from Github. The graph shows the *main* branch (yellow), *development* (grey) and various *feature* and *hotfix* branches (blue, green, purple, orange). The graph shows how features and hotfixes are merged into development before being merged into production. The graph also shows that some features or hotfixes have been merged with development to minimize merge conflicts.

3.4 Development process

We have used the Projects feature, essentially a Kanban board, on GitHub to handle the organization of the work that is to be done 9. The reasoning behind this, was to centralize our workflow as much as possible. Whenever a task had been defined during a meeting, we would add the task and assign the person(s) responsible for it.

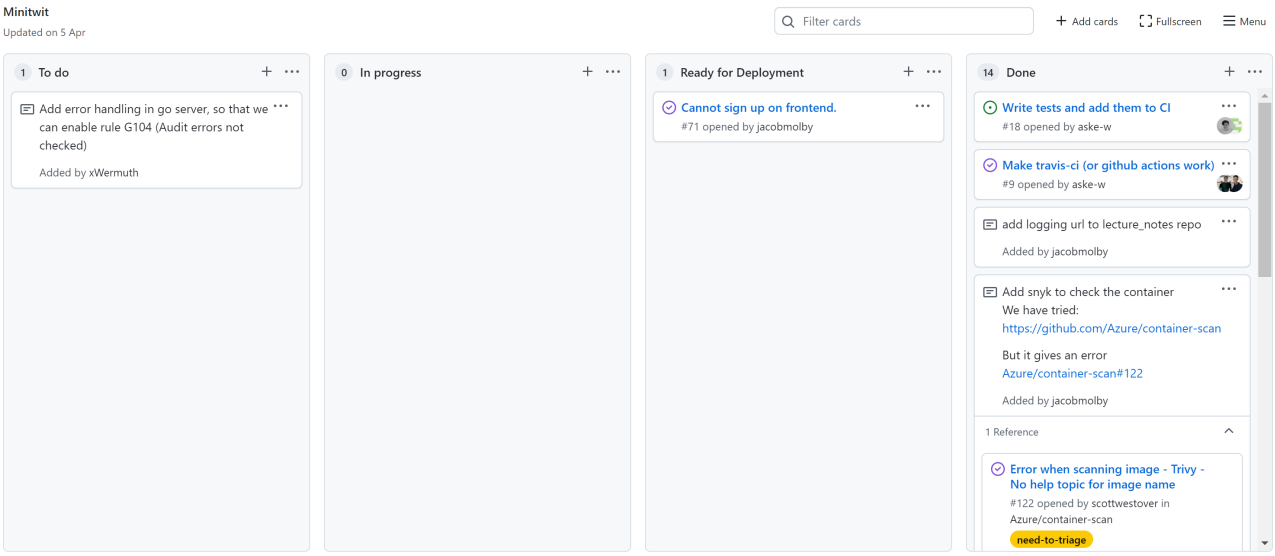


Figure 9: GitHub Projects for organization of tasks.

3.5 Monitoring and logging

3.5.1 Logging

For logging, we use the EFK-stack that consists of the following three elements Elastic, Filebeat and Kibana. The EFK-stack is a way of aggregating large volumes of logs to quickly sort through and analyze them [7]. This is especially helpful if a significant part of the infrastructure is hosted in the cloud.

Filebeat collects, transforms, and ships logs to the ElasticSearch back end [6]. Furthermore, Filebeat is used to configure an index for ElasticSearch. In our case, we created an index on the Docker container running our server to filter by its logs and not get overwhelmed by the other containers' logs. In Kibana the logs are visualized, where we can filter by multiple fields such as timestamp, log type (error, warning, etc.), and many others [8].

Figure 10 shows our logging infrastructure.

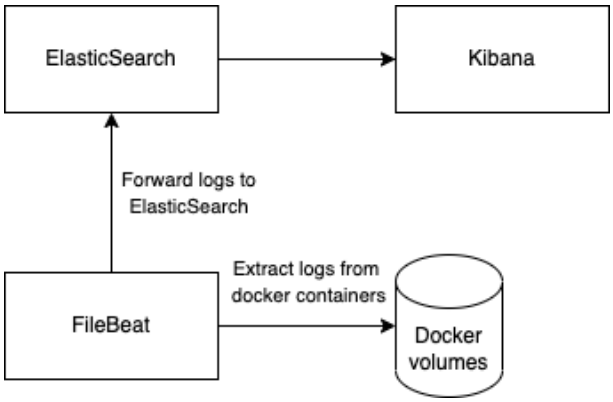


Figure 10: Simple overview of our ELK-stack

If we wanted to have our logs more structured, we might have considered including Logstash in our stack. That would have made the logs easier to analyze. Logstash like Filebeat can also send data to ElastichSearch, but it also acts as an aggregator, where it pulls data from various sources and transform the data into meaningful logs before pushing it to ElasticSearch [2]. However, Logstash processing has the disadvantage of being slow and since our infrastructure is small, enhancing the logs is not needed.

3.5.2 Monitoring

Grafana and Prometheus are used for monitoring our application. Prometheus is used to store the collected metrics, and Grafana is used as the interface for visualizing the data. We are monitoring metrics within four different categories: Web Server, Container Statistics, Database and Business Intelligence. The metrics monitored are listed below. Screenshots of the dashboard can be seen in appendix A.2.

Web Server:

- Simulator endpoints response time
- Regular endpoints response time
- Number of accesses to endpoints

Container Statistics:

- CPU usage
- Memory usage

Database:

- Uptime
- Active connections
- Idle connections
- Number of threads running
- Number of slow queries

Business Intelligence:

- Average followers per user
- Total number of users

3.6 Security

In this section we will describe the results of our security assessment.

Risk Identification

We identified the following assets that could be vulnerable.

- API/server (Go application)
- Webclient (React.js application, served with NGINX)
- Graphana
- Prometheus
- cAdvisor
- Elastic Search
- Kibana
- Filebeat
- MySQL database

For these assets certain threat sources were also identified:

1. SQL injection on web client.
2. XSS on web client.
3. Malicious actor accessing secrets.
4. DDoS on VPS.
5. Hacker guessing the passwords of users.

Based on the threat sources different risk scenarios where devised.

1. Attacker performs SQL injection to download or destroy data from the database.
2. Attacker inputs JavaScript in an input field and accesses data of another user.
3. Attacker is able to socially engineer a group member to expose a secret.
4. Attacker uses DDoS to crash or halt our server or database.
5. Since we have no requirements for passwords, it is possible for the users to create single letter or number passwords. This would make it very easy for the attacker to exploit using a dictionary or brute force attack.

Risk Analysis

Based on the identification of risks above, we conducted a risk analysis. For this purpose, a risk matrix has been constructed as seen in table 1. The matrix consists of likelihoods on the x-axis and impacts on the y-axis. Each of the scenarios from above has been inserted based on their likelihood and impact.

Impacts \ Likelihoods	Rare	Unlikely	Possible	Likely	Certain
Catastrophic		**1**			
Critical	**3**		**2**		**5**
Marginal			**4**		
Negligible					
Insignificant					

Table 1: Risk matrix for the scenarios. The numbers indicate a scenario from above.

To mitigate the risk the following is planned to safeguard against possible attacks.

- **Scenario 1:** Fix the injections and restore backups.
- **Scenario 2:** Issue an apology to the user and fix the injections.
- **Scenario 3:** Give the exposed group member a security course and change all secrets.
- **Scenario 4:** Restart the server. Setup DDoS protection like a firewall or CloudFlare.
- **Scenario 5:** Reset the users' password. To mitigate future attacks, we should implement minimum requirements for the passwords on user creation.

We also did a penetration test. With tools available in the Kali Linux distribution we tried a variety of automated penetration tools like `wmap`, `nmap` and `metasploit`. None of them were successful in having an impact on our system.

3.7 Scaling and load balancing strategy

During the course we have applied different strategies for scaling our Minitwit application. We were able to use the smallest droplet available on DigitalOcean until the implementation of logging with the EFK-stack. We had to scale vertically since we needed more RAM.

After the lecture on *deployment strategies, scalability and load balancing* we implemented the ability to scale horizontally for high availability for the Minitwit application. We discovered Docker Stack which allows you to deploy a compose file into a swarm. We used Docker Swarm as an orchestrator to distribute our services between two nodes. Then the manager node schedules the services as replica tasks on nodes in the swarm [1].

We have not defined any services to be replicated for load balancing. Nevertheless, some services are replicated on all nodes out of necessity, such as *cAdvisor*, which collects container data for containers running on a node. *Filebeat* is also replicated on all nodes to pull logs from the containers on each node.

Docker Swarm's routing mesh obfuscates container placement from the point of view of requests and automatically routes requests to nodes with the requested containers. If we had automatically scaled the services, the routing mesh would automatically load balance requests between nodes and containers.

4 Lessons Learned Perspective

4.1 Testing & Splitting Web Client and API

We chose to split the Web Client from the API to conduct tests on the API. Before this, our tests revolved around asserting the text embedded in the HTML, which the API would serve. This also allowed us to change the authentication to JSON Web Tokens, which simplified testing compared to the previous authentication, that used cookies stored in sessions (see pull request #89).

4.2 ElasticSearch

In our current setup, we have two servers with 4GB of memory each. Ideally, we would have at least three servers running to increase resilience. However, the ElasticSearch image in Docker uses at least 2GB of memory and our free credits in DigitalOcean can only afford two servers with 4GB of memory.

If we wanted to have three servers running, we could have used a managed service like Datadog for log management. Especially, since our infrastructure is small, it could be unnecessary with the ELK-stack, since we do not use many of its features.

4.3 GitGuardian

GitGuardian was a helpful tool to alert if secrets were committed by accident. The alerts were sometimes false positives, however, it did successfully detect the inclusion of secrets in the commit d7e4d9b.

GitGuardian sent an email to the repository owner since the repository was not set up as a team, with a notification of a possible secret exposure. The email can be seen in appendix 11 The alert helped us minimize the time in which the secrets were exposed. Otherwise, we might never have noticed the exposure.

To mitigate malicious use, the secrets were changed as soon as we noticed the alert. Further, we added the directory in which the files with the secrets were in to our `.gitignore`, as they were not necessary to commit anyway.

4.4 Persistence of Prometheus Data

It did not occur to us before the end of April that saving the monitoring data from Prometheus was a good idea, especially to show this data at the exam. Furthermore, in a real-life production setting, you might want to have historical monitoring data, to some extent, to accompany your logging data.

To persist this data was a relatively simple exercise. We created a volume to map the directory in which Prometheus saves its data inside its container to a directory on the host machine. Then, we used a container called `s3vol` to store the directory on the host machine in a blob storage at DigitalOcean. This was done by creating a volume mapping the host directory to the directory in s3vol which it sends to the blob storage. The persistence of Prometheus data can be seen in commit 11844f8.

4.5 Terraform

At the beginning of the project, Vagrant was used to provision the server. However, we could not provision Docker Swarm with Vagrant, as some unsolvable errors kept occurring. Therefore, we switched to provisioning

via bash scripts. However, bash scripts are imperative and can be difficult to understand. Therefore, late in the project, we changed to use Terraform, which is written in a declarative language. It also provides features such as first-party integrations with cloud providers, drift detection and validation of the infrastructure. This can be seen in commit 3b0684a.

4.6 Logging

In the current state of the application, logs are stored on the servers. However, if the servers restart the old logs will be overwritten, resulting in a loss of critical information. In addition, having the logs in one place presents a security issue. If an attacker gains access, the person can delete the logs and cover up his tracks. With externalized logs, it might help warn an attack is occurring and gives a better chance of preventing it. Therefore, it would preferable to send the logs to a blob storage like `s3vol` within a given interval.
Furthermore, to make sure additional critical logs are not deleted, we should also configure the default log rotation of ElasticSearch, where it creates a new log file each day and persists it for a week before deleting it [4].

4.7 DevOps Reflections

This project has differentiated itself from other code projects, because of the focus on infrastructure and automation. Previous projects have mostly revolved around the development of new features. By automating the infrastructure and implementing continuous integration and continuous deployment, we were able to reduce the amount of time spent on integration and deployment. This allowed us to work more efficiently by deploying smaller increments more frequently, resulting in less merge conflict.

The DevOps Handbook characterize DevOps by three principles; flow, feedback and continual learning and experimentation [5, Ch. 1-4].

Our workflow followed all the characteristics of the flow principle. We could improve on making work more visible. We have a GitHub projects Kanban board, but it was not used much, since we immediately started doing the weekly assignments. It was more used for tracking long running tasks and bugs. We made small batches of work, which we deployed continuously. We had minimal handoffs since only our team worked on the Minitwit application. We reduced many constraints through automation of code deployment and testing. Using Docker allowed us to replicate the production environment locally, but could have been better since we had to use separate docker-compose files.

For the feedback characteristic, we have not adhered to swarming to solve problems and build new knowledge. If one person struggled they have been able to get help from others, but we have not had a problem where all group members sat around one computer to solve it. To see problems as they occur, we have automated build and testing through GitHub Actions. Changes would automatically be rejected if these failed. We have used Grafana to monitor the state of our application. What we looked for, was if containers were online or using excessive amounts of resources, and how fast our simulator endpoints responded. We have always been the ones to make decisions and review others work, and therefore we have pushed quality close to the source, i.e. the evolution and maintenance of the Minitwit application. We have optimized our work for downstream work centers, i.e. ourselves, through a well-considered software architecture and contributing helpful information to our README.

Our workflow does not exhibit the trait of continual learning and experimentation to the same extent as the other two characteristics of DevOps. We have not institutionalized the improvement of daily work by time boxing error fixes. Rather, we have fixed problems as they have occurred and written helpful information in the README. We also lack in terms of resilience patterns. We rely a lot on GitHub, but we are aware of the consequences of this.

In terms of a learning culture, we have respected that some things have taken longer than others. Many of the concepts during the course have been new to us. Therefore, we have spend time understanding these concepts and adapting them for the Minitwit application. Even though it has been a source of frustration, we have all done tasks which turned out to be more difficult than anticipated.

5 Conclusion

Working on this project has emphasized the benefits of infrastructure and automation within software development. Automation saves time and resources and establishes structure, which is important when multiple developers are working on a project simultaneously. The structure also contributes to the stability of the system whenever new features or changes are deployed. The use of continuous integration and continuous deployments encourages faster and more frequent delivery, which helps ensure that written code actually gets deployed to production. Altogether, the practices of DevOps streamlines the development process while ensuring high quality software.

References

[1] visited 24-May-2022. May 2022. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#services-tasks-and-containers.

[2] Daniel Berman. *Filebeat vs. Logstash: The Evolution of a Log Shipper*. visited 24-May-2022. Feb. 2020. URL: https://logz.io/blog/filebeat-vs-logstash/.

[3] H Christensen, Aino Corry, and K Hansen. "An approach to software architecture description using UML". In: *Technical Report* (2004), pp. 271–350.

[4] *Configure Log Rotation Policies for Monitoring Services*. visited 24-May-2022. URL: https://docs.datafabric.hpe.com/62/AdministratorGuide/LogCollection_Rotation.html#:~:text=Elasticsearch%5C%20Log%5C%20Rotation%5C%20Policy,%5C%2Fetc%5C%2Felasticsearch%5C%2Flogging..

[5] Gene Kim et al. *The DevOPS handbook*. Portland, OR: IT Revolution Press, Dec. 2016.

[6] *Lightweight shipper for logs*. visited 10-May-2022. URL: https://www.elastic.co/beats/filebeat.

[7] *The ELK stack*. visited 24-May-2022. URL: https://aws.amazon.com/opensearch-service/the-elk-stack/#:~:text=The%5C%20ELK%5C%20stack%5C%20is%5C%20an,Elasticsearch%5C%2C%5C%20Logstash%5C%2C%5C%20and%5C%20Kibana..

[8] *Your window into the Elastic Stack*. visited 24-May-2022. URL: https://www.elastic.co/kibana/.

Appendices

A Figures

A.1 GitGuardian notification

Figure 11: Email from GitGuardian notifying of secrets accidentally being commited

A.2 Grafana monitoring dashboard screenshots

For some monitoring measures the dashboard does not aggregate data correctly from different servers. Therefore, it may seem like there are 3 nodes instead of 2, and some data is duplicated.

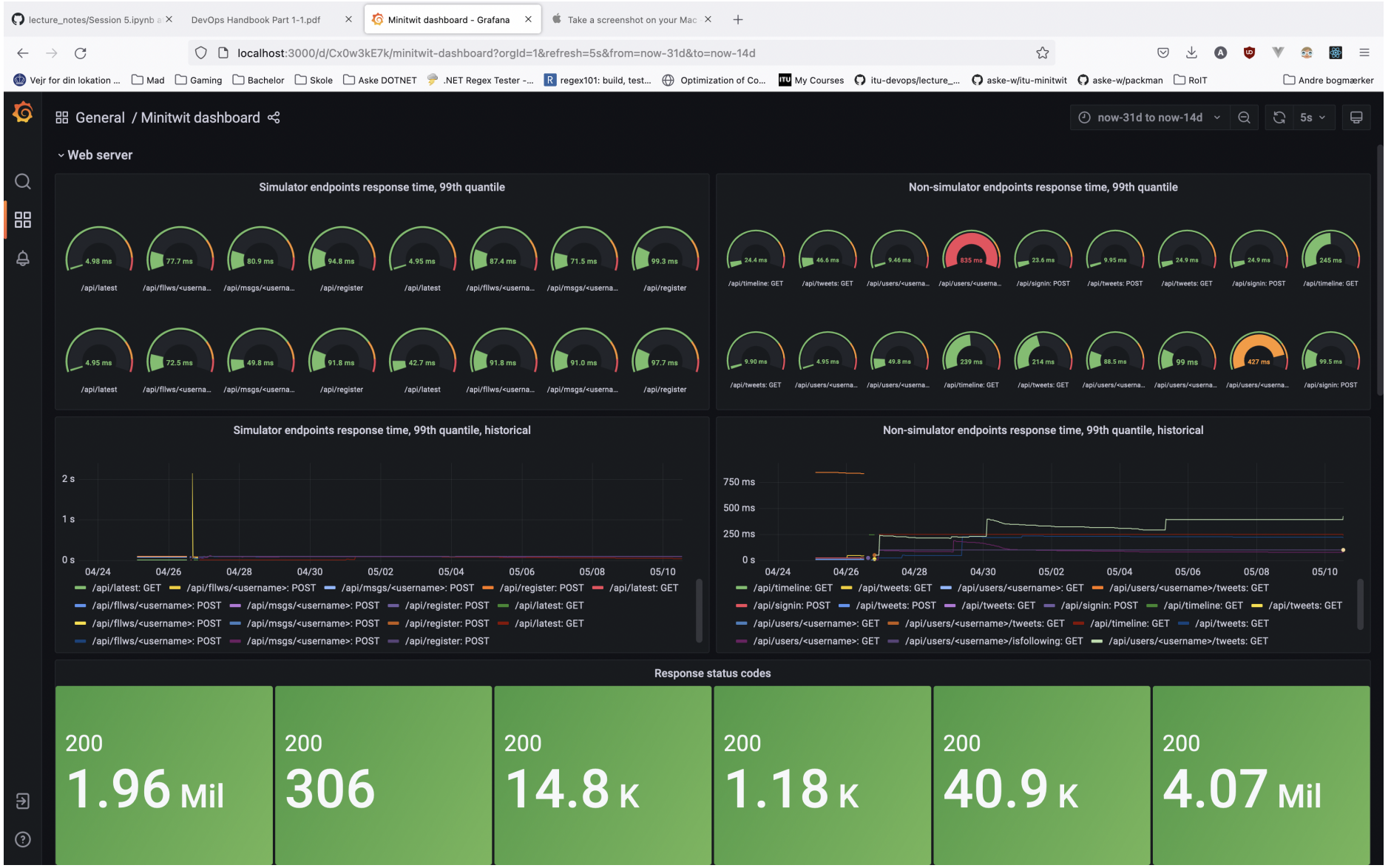


Figure 12: Screenshot of the Grafana dashboard showing endpoints response time and status codes

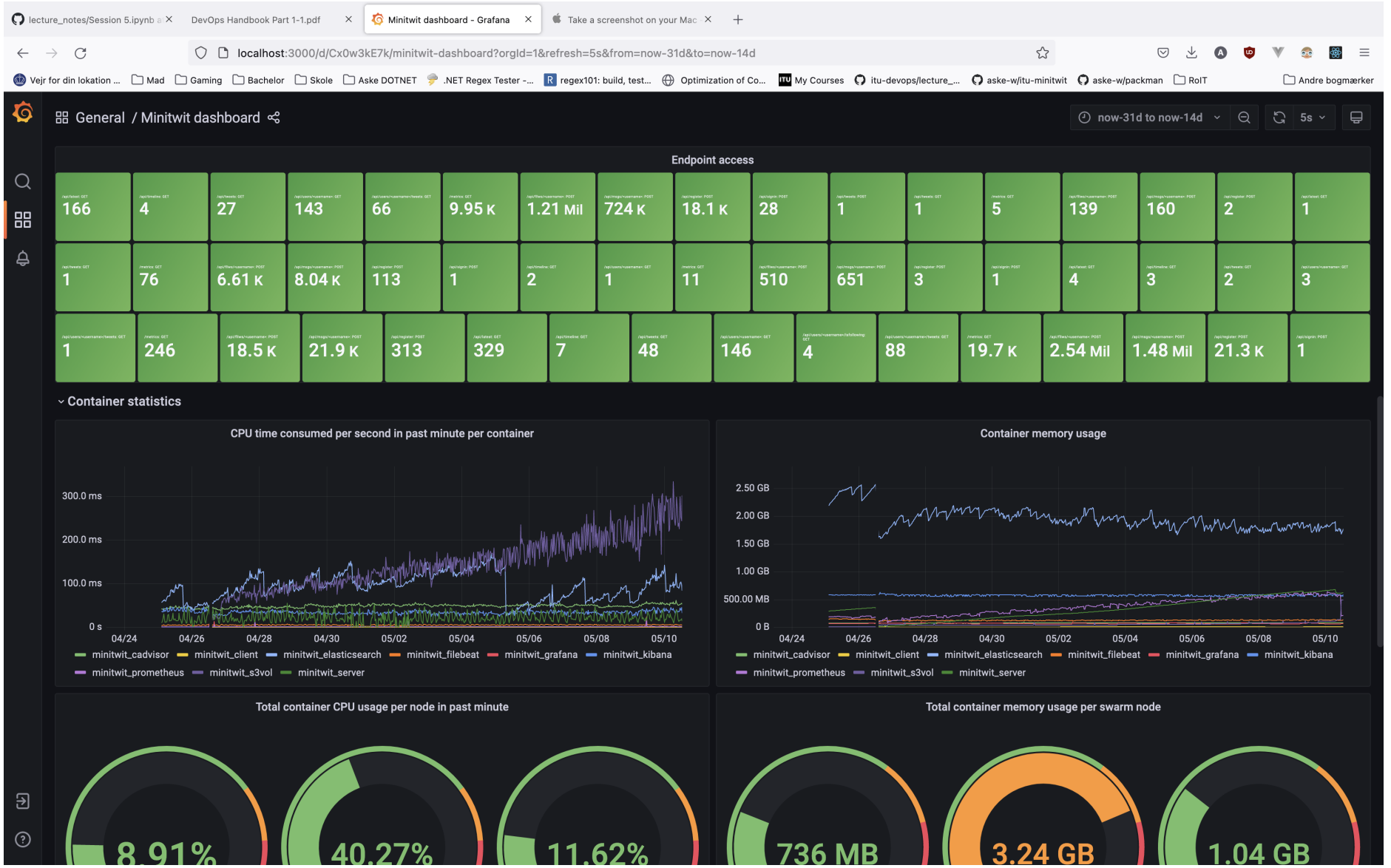


Figure 13: Screenshot of the Grafana dashboard showing the access count for each endpoint and CPU and memory statistics for containers

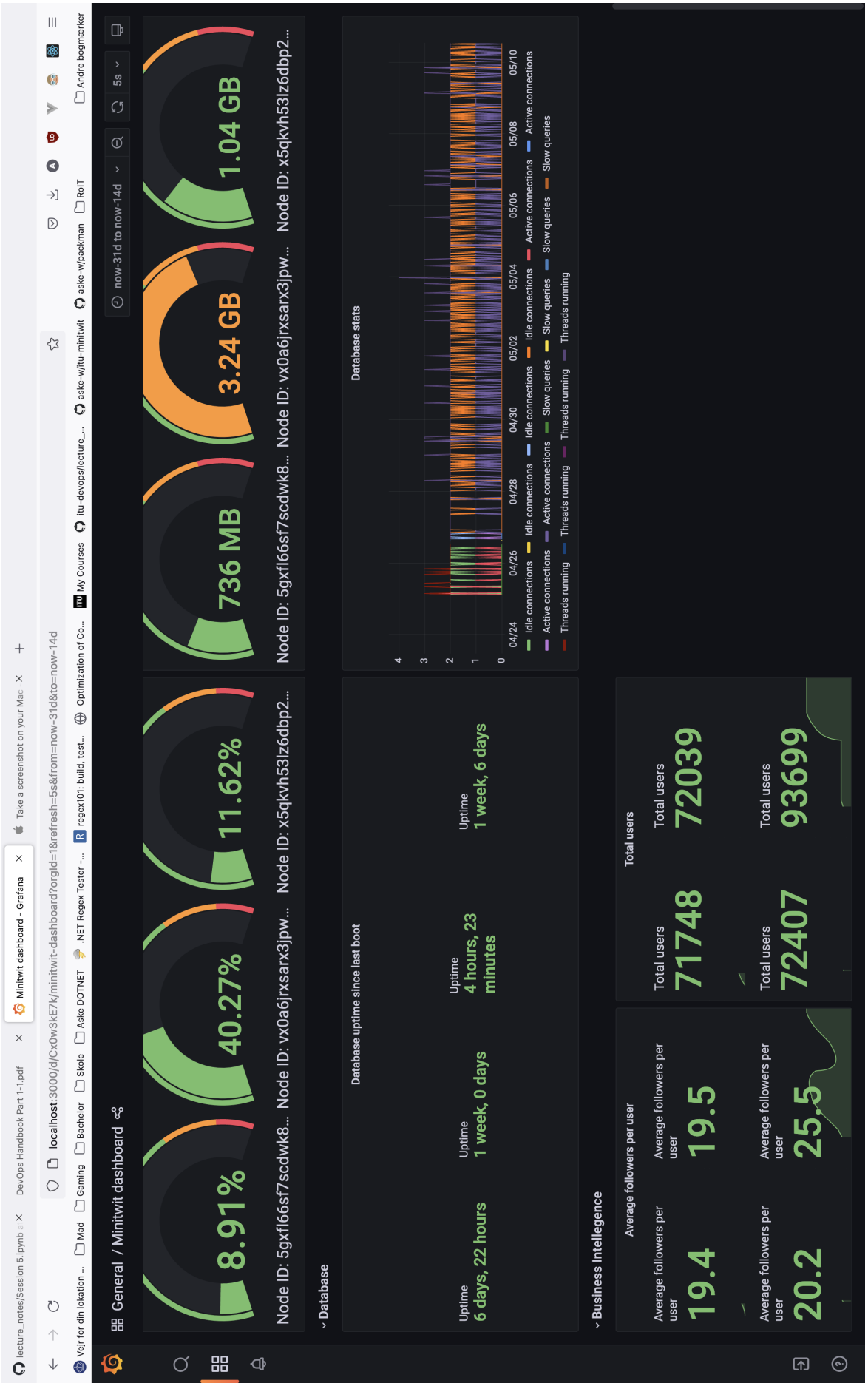
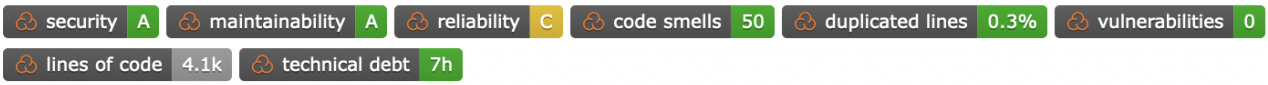


Figure 14: Screenshot of the Grafana dashboard showing the current memory and CPU usage for containers, database statistics and business intelligence

Sonarqube



Code Climate

Better Code Hub

Figure 15: Static analysis overview

B Lichen

```
jacobmolby@Moelbys-MacBook ~/P/i/server (development) [1]> lichen --config ../lichen.yml app

github.com/BurntSushi/toml@v1.0.0: MIT (allowed)
github.com/CloudyKit/fastprinter@v0.0.0-20200109182630-33d98a066a53: MIT (allowed)
github.com/CloudyKit/jet/v6@v6.1.0: Apache-2.0 (allowed)
github.com/Shopify/goreferrer@v0.0.0-20210630161223-536fa16abd6f: MIT (allowed)
github.com/andybalholm/brotli@v1.0.4: MIT (allowed)
github.com/aymerick/douceur@v0.2.0: MIT (allowed)
github.com/aymerick/raymond@v2.0.3-0.20180322193309-b565731e1464+incompatible: MIT (allowed)
github.com/beorn7/perks@v1.0.1: MIT (allowed)
github.com/blang/semver/v4@v4.0.0: MIT (allowed)
github.com/cespare/xxhash/v2@v2.1.2: MIT (allowed)
github.com/eknkc/amber@v0.0.0-20171010120322-cdade1c07385: MIT (allowed)
github.com/fatih/structs@v1.1.0: MIT (allowed)
github.com/flosch/pongo2/v4@v4.0.2: MIT (allowed)
github.com/go-sql-driver/mysql@v1.6.0: MPL-2.0 (allowed)
github.com/gobwas/httphead@v0.1.0: MIT (allowed)
github.com/gobwas/pool@v0.2.1: MIT (allowed)
github.com/gobwas/ws@v1.1.0: MIT (allowed)
github.com/goccy/go-json@v0.9.4: MIT (allowed)
github.com/golang/protobuf@v1.5.2: BSD-3-Clause (allowed)
github.com/golang/snappy@v0.0.4: BSD-3-Clause (allowed)
github.com/google/uuid@v1.3.0: BSD-3-Clause (allowed)
github.com/gorilla/css@v1.0.0: BSD-3-Clause (allowed)
github.com/gorilla/websocket@v1.5.0: BSD-2-Clause (allowed)
github.com/iris-contrib/go.uuid@v2.0.0+incompatible: MIT (allowed)
github.com/iris-contrib/jade@v1.1.4: BSD-3-Clause (allowed)
github.com/iris-contrib/schema@v0.0.6:  (allowed)
github.com/jinzhu/inflection@v1.0.0: MIT (allowed)
github.com/jinzhu/now@v1.1.4: MIT (allowed)
github.com/joho/godotenv@v1.4.0: MIT (allowed)
github.com/josharian/intern@v1.0.0: MIT (allowed)
github.com/json-iterator/go@v1.1.12: MIT (allowed)
github.com/kataras/blocks@v0.0.5: MIT (allowed)
github.com/kataras/golog@v0.1.7: BSD-3-Clause (allowed)
github.com/kataras/iris/v12@v12.2.0-alpha9: BSD-3-Clause (allowed)
github.com/kataras/jwt@v0.1.2: MIT (allowed)
github.com/kataras/neffos@v0.0.19: MIT (allowed)
github.com/kataras/pio@v0.0.10: BSD-3-Clause (allowed)
github.com/kataras/sitemap@v0.0.5: MIT (allowed)
github.com/kataras/tunnel@v0.0.3: MIT (allowed)
github.com/klauspost/compress@v1.14.4: MIT, BSD-3-Clause, Apache-2.0 (allowed)
github.com/mailru/easyjson@v0.7.7: MIT (allowed)
github.com/mattn/go-sqlite3@v1.14.9: MIT (allowed)
github.com/matttproud/golang_protobuf_extensions@v1.0.1: Apache-2.0 (allowed)
github.com/mediocregopher/radix/v3@v3.8.0: MIT (allowed)
github.com/microcosm-cc/bluemonday@v1.0.18: BSD-3-Clause (allowed)
github.com/modern-go/concurrent@v0.0.0-20180306012644-bacd9c7ef1dd: Apache-2.0 (allowed)
github.com/modern-go/reflect2@v1.0.2: Apache-2.0 (allowed)
github.com/nats-io/nats.go@v1.13.1-0.20220121202836-972a071d373d: Apache-2.0 (allowed)
github.com/nats-io/nkeys@v0.3.0: Apache-2.0 (allowed)
github.com/nats-io/nuid@v1.0.1: Apache-2.0 (allowed)
github.com/prometheus/client_golang@v1.12.1: Apache-2.0 (allowed)
github.com/prometheus/client_model@v0.2.0: Apache-2.0 (allowed)
github.com/prometheus/common@v0.32.1: Apache-2.0 (allowed)
github.com/prometheus/procfs@v0.7.3: Apache-2.0 (allowed)
github.com/russross/blackfriday/v2@v2.1.0: BSD-2-Clause (allowed)
github.com/schollz/closestmatch@v2.1.0+incompatible: MIT (allowed)
github.com/tdewolff/minify/v2@v2.10.0: MIT (allowed)
github.com/tdewolff/parse/v2@v2.5.27: MIT (allowed)
github.com/valyala/bytebufferpool@v1.0.0: MIT (allowed)
github.com/vmihailenco/msgpack/v5@v5.3.5: BSD-2-Clause (allowed)
github.com/vmihailenco/tagparser/v2@v2.0.0: BSD-2-Clause (allowed)
github.com/yosssi/ace@v0.0.5: MIT (allowed)
golang.org/x/crypto@v0.0.0-20220214200702-86341886e292: BSD-3-Clause (allowed)
golang.org/x/net@v0.0.0-20220225172249-27dd8689420f: BSD-3-Clause (allowed)
golang.org/x/sys@v0.0.0-20220310020820-b874c991c1a5: BSD-3-Clause (allowed)
golang.org/x/text@v0.3.7: BSD-3-Clause (allowed)
golang.org/x/time@v0.0.0-20220224211638-0e9765cccd65: BSD-3-Clause (allowed)
golang.org/x/xerrors@v0.0.0-20200804184101-5ec99f83aff1: BSD-3-Clause (allowed)
google.golang.org/protobuf@v1.27.1: BSD-3-Clause (allowed)
gopkg.in/ini.v1@v1.66.4: Apache-2.0 (allowed)
gopkg.in/yaml.v3@v3.0.0-20210107192922-496545a6307b: MIT, Apache-2.0 (allowed)
gorm.io/driver/mysql@v1.3.2: MIT (allowed)
gorm.io/driver/sqlite@v1.3.1: MIT (allowed)
gorm.io/gorm@v1.23.1: MIT (allowed)
gorm.io/plugin/prometheus@v0.0.0-20220223061010-d8bdd50fdfc7: MIT (allowed)
```

Figure 16: Output of Lichen