DevOps, Software Evolution and Software Maintenance

Alexander Palsson[1] Emil Krogsgaard Ravn[2] Gianmarco Murru[3]
Henrik Bellsund[4] Tor Arnth Petersen[5]

IT University of Copenhagen

{[1]apal, [2]erav, [3]gimu, [4]hbel, [5]toap}@itu.dk

June 21, 2022

Contents

1 System's Perspective

1.1 Design of MiniTwit-Go

We use GitHub Actions to manage our CI/CD setup. When a developer makes a change to the code and makes a pull request, a new GitHub Action is invoked. To host our virtual machines and store data we used Digital Ocean as our Cloud Provider. An Docker image defines the containerization of the MiniTwit application, the API, logging and monitoring and database, marked as the cluster in Figure 1.

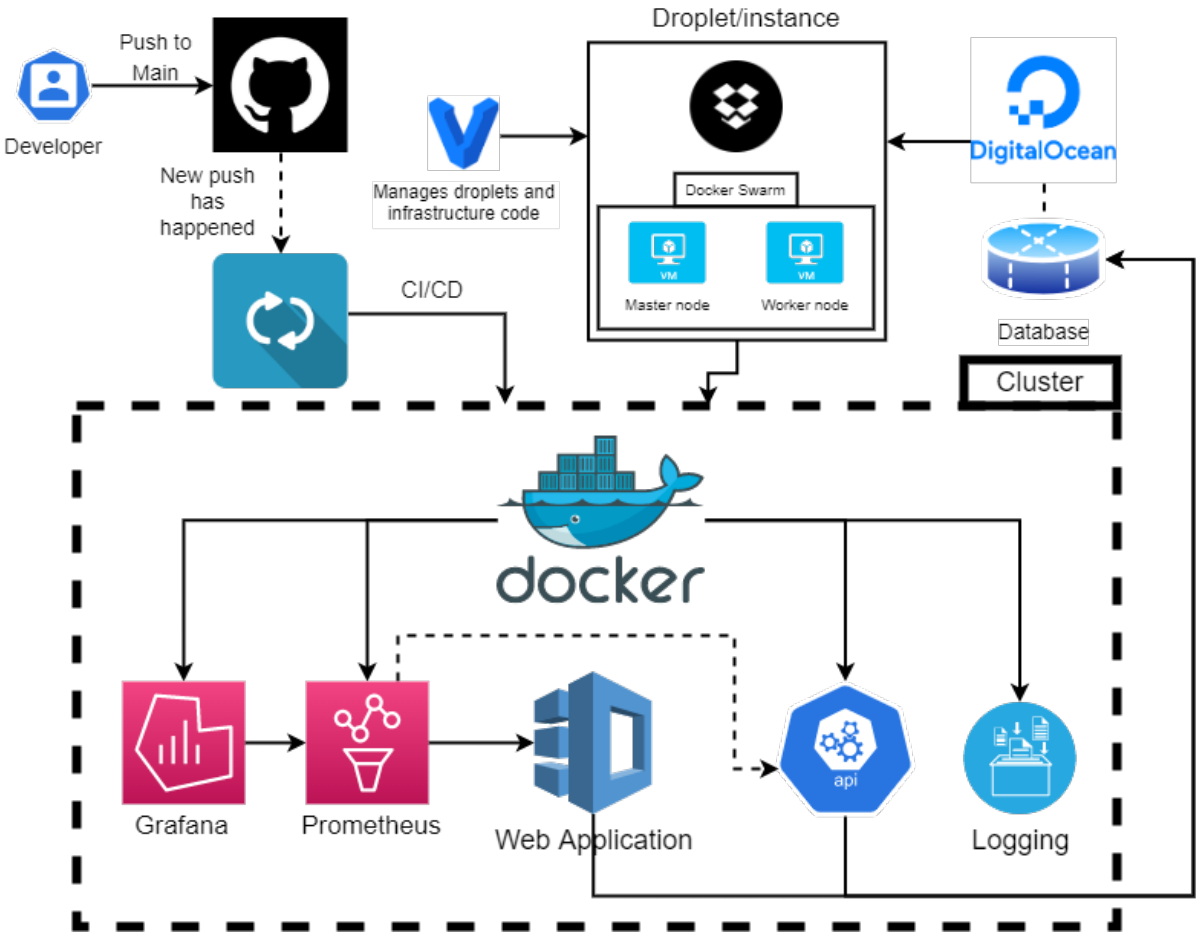

Figure 1: Complete design of the MiniTwit-Go system

1.2 Architecture of MiniTwit-Go

Figure 2 shows a top-level diagram of the overall packages in the system. Furthermore, the dependencies between the packages are shown in the diagram. The list below describes and motivates the purpose of each package in the package diagram:

- **Models**: The model is the most dependable package in the system since it is the interface that defines the data to be displayed in the user interface. The data defined here are *structs* that in Go are typed collections of fields, which is useful for grouping data together to form records such as LoginForm for forms and Message for users.
- **Web**: The web solely depends on Models and its responsibility is to visualize data defined in Models. Web defines a stylesheet (CSS file) as well as the login, register and timeline pages.
- **Controller**: The controller depends on both Models and Database as its responsibility is to determine which Web view to display in response to any incoming action, including when the system loads.

- **API**: The API solely depends on Models since its responsibility is to use and call the application defined data in Models, and does not depend on any other package.
- **Database**: The database depends on the controller and is responsible for long term storage.

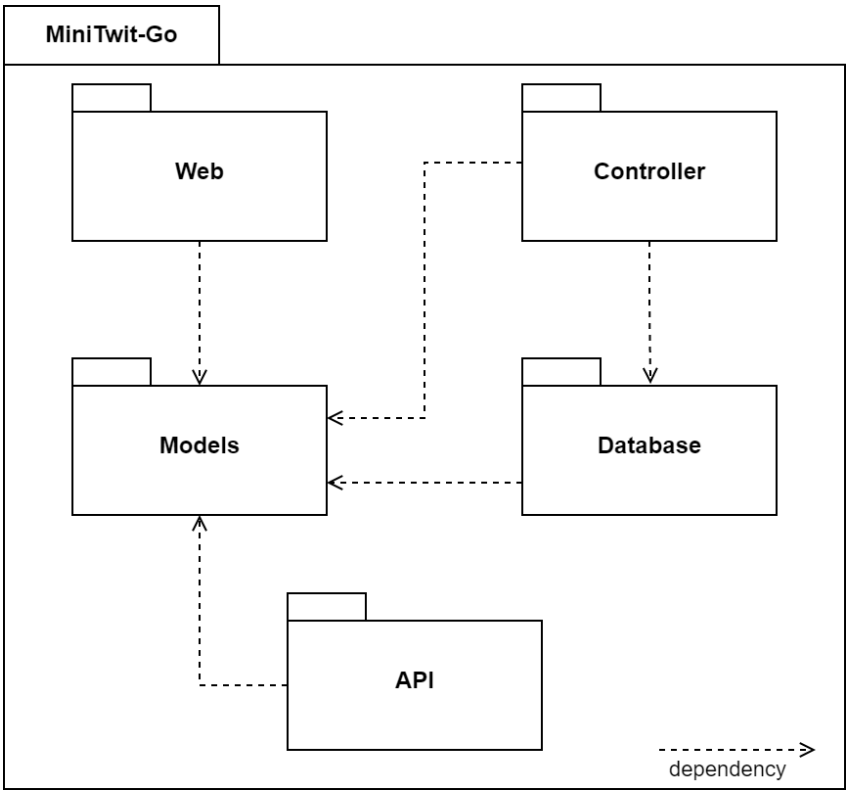

Figure 2: Package diagram of the MiniTwit-Go System

1.3 Dependencies of our system

Below is a list and a brief description of all the technologies and tools we applied and are dependent on:

- **Go** [6]: We chose Google's relatively new programming language Golang since it is good for developing web applications, easy to maintain, and faster than many other programming languages. It is often compared to C in terms of speed and efficiency. Furthermore, Go does not require an interpreter, freeing up power and boosting performance. Lastly, there was a wish of working with Golang as it is a new programming language that no one within the group has worked with before [14].
- **Gin Framework** [4]: We chose Gin because it is a high-performance Golang framework for creating web applications and contains a lot of useful modules. Furthermore, it reduces boilerplate code by providing commonly used functionalities, such as routing, middleware support, rendering, etc. Thus, it makes it easier to build web applications.
- **DigitalOcean** [2]: We chose DigitalOcean because it is coupled with the GitHub education pack. DigitalOcean provides free credits, which is advantageous for us as students. Furthermore, it is a simple cloud service provider that is suitable for small applications and projects and simple to setup. The documentation for DigitalOcean is straightforward and comprehensible to get started quickly.
- **Vagrant** [9]: Vagrant was chosen since combining DigitalOcean with Vagrant provides a clean way to boot up virtual machines to host our web application. In short, Vagrant can be seen as a scripting engine for VirtualBox. Another point is that it can be source controlled with ease since everything is defined in a single text file. You can of course version control a snapshot of the virtual machine, but that will take up a lot more space than just a Vagrantfile.

- **GitHub Actions** [5]: We first went with Travis CI but changed to GitHub Actions, simply because Travis was going to start charging for its usage.
- **GORM (ORM library)** [11]: This library was chosen because its the main ORM (Object-Relational Mapping) library for Golang. Moreover, it allowed us to automatically migrate schemas which would automatically create tables based on our model. Finally, GORM is developer-friendly and well documented, which makes it easy to work with.
- **Docker Swarm** [10]: A separate tool within Docker that manages and orchestrates "swarms" of Docker containers. It is useful for providing scaling and load balancing strategies.
- **ELK stack (Kibana, filebeat, elasticsearch)** [3]: A stack of data analytic tools for collecting, processing, and visualizing system log output.
- **Prometheus and Grafana**: Prometheus [15] is a monitoring system and alerting toolkit used for monitoring our virtual machines. Grafana [13] is an analytic and interactive visualization web application that provides queries, clear presentation through charts and graphs, and customizable dashboards. The output of Prometheus can be used as input to Grafana to produce analytic data.
- **Bugsnag** [16]: An error-monitoring tool used to identify, prioritize and replicate bugs found in our system. Helps us as developers to easier spot the effects of our code and address issues before they escalate.
- **DuckDNS** [1]: A dynamic DNS service that points a DNS to an IP of our choice. It eliminates the need to remember a specific IP address.

1.4 Important interactions of subsystems

An example where we have an interaction between some of the subsystems in MiniTwit-Go is when the user registers through the frontend. The logic is sent to the *registration.go* module in the Controller package, which uses the model from the Models package and updates the information in the database. This way we can separate our logic into different functionalities and divide them in a separate subsystem. This can be seen in Figure 3.

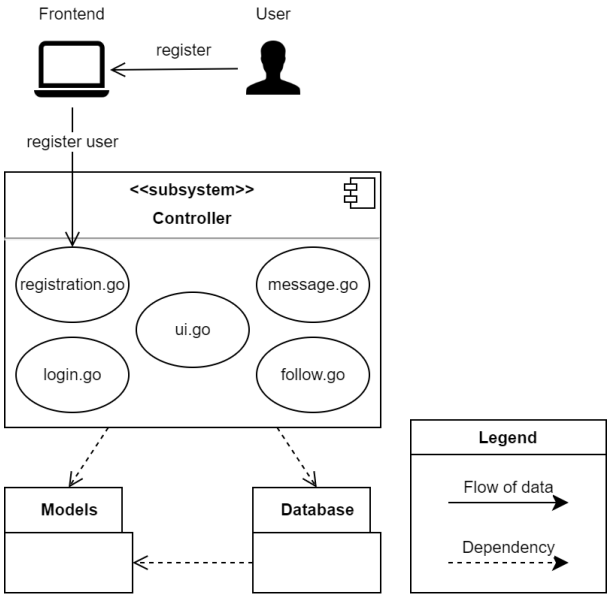

Figure 3: User registration shown with interaction of subsystem

1.5 Current state of Minitwit-Go

The system will be assessed with the 3 focus points: **operability**, **simplicity**, **evolveability** from Kleppmann' definition of software maintainability [12]. The state of the program will be described from

a developer perspective, which is the focus of Kleppmann's maintainability operationalization of the term.

- **Operability** - *Making Life Easy for Operations*:
  - **Health monitoring**: we use Prometheus for storing times series data about the health of our system and Grafana for visualising them
  - **Cause of issues**: Bugsnag helps us detect code bugs and their origin and automatically creates GitHub Issues. Whereas the Grafana gives an overview of the performance e.g., CPU usage, memory usage etc. Finally Elasticsearch stores logs from our service and through Kibana it is possible to visualize them and filter and research them.
  - **Automation and integration of tools**: We use GitHub Action to ensure automatic Continuous Integration (with several static analysis tool) and Continuous Deployment. Every time a pull request is merged into the *main* branch, the new version of the system is deployed into production and a new release is published on GitHub.
  - **Single point of failure**: We have a single point of failure on the manager node of Docker Swarm. If the Docker service in the manager virtual machine stops working that would make the entire infrastructure stop working. To ensure High Availability and avoid Single-point of failure we should have multiple manager nodes, which we have avoided because of the cost.
  - **Documentation**: The entire system is well-documented everything from deployment to configuration of the individual components.
  - **Self-healing/roll-back mechanism**: We don't have any automatic roll-back mechanism, however, since any image is tagged with the GitHub workflow build number, it is easy to roll back to the previous working version. Docker Swarm ensures self-healing techniques such as restarting a container in case of failure.
- **Simplicity** - *Managing Complexity*
  - **Tight coupling vs loose coupling**: All of our components are built independently, even though they might rely on others to work (e.g. API and Web-App rely on Database, Kibana rely on Elasticsearch etc.)
- **Evolvability** - *Making Change Easy*
  - How easy is the system to change: as most of the components are loosely coupled in the architecture it is relatively easy to substitute the individual components.

Quality assessment
From Arachni we did a quality assessment of our API[1]. Apart from the penetration part, which we will touch upon later, more *interesting* was the HTTP not-expected responses e.g., sometimes we get a 200-OK when we should not. These issues explain some of the issues we had with the API simulation test during the course. In short, we still have some issues with our code, which we at this point have not managed/prioritized to fix.

1.6 License

To scan our project for licenses of our dependencies we used Lichen[2]. The dependencies of our project use the licenses: MPL-2.0, Apache-2.0, BSD-3-Clause and BSD-2-Clause. All these aforementioned licenses are compatible with the MIT-license of our project, which means we can bundle all the licenses together under the MIT license.

[1] An interactive dashboard can be found here: `https://www.arachni-scanner.com/reports/report.html/#!/summary/charts`

[2] `https://github.com/uw-labs/lichen`

2 Process' Perspective

2.1 Communication between developers

The communication between team members was equally conveyed through both physical and remote working sessions. We worked and communicated to a large extent on the days when the course lectures were given. Furthermore, the use of Microsoft Teams allowed us to work and communicate remotely whenever it was needed. Each week the team were updated on tasks that were not finished from previous weeks and planned prioritization of said tasks based on the state of the system. If anyone had a question or a concern, a team member could ask in the Teams chat and resolve the problem rather quickly.

2.2 Team organization

Everyone in the team had the same responsibilities and roles in the team. We focused on including every group member in the development process, such that everyone was up to date. We would provide and go through an agenda for the day we were working, where the agendas were based on the state of the project and the latest lecture.

Furthermore, the team was composed of people with varying experiences with DevOps ranging from zero to professional experience prior to starting the course. This meant that we had to get everyone on the same level during some sessions together to prevent some group members from falling behind on the course material.

Finally, in relation to getting everyone in touch with most parts of the code and project, we did not enforce any strict rules about whom works on what, and we endorsed everyone to work on things that they would like to, such as a specific technology or language (Go, Docker, Vagrant etc.) to be able to take ownership of the application.

2.3 Stages and tools in CI/CD chain

We initially wanted to use Travis CI for our CI/CD chain. Unfortunately, Travis CI was out of the scope economically since they changed their policy recently related to their pricing model. Therefore, we resorted to GitHub Actions instead. We used GitHub Actions to automate our integration and release pipeline. We have three GitHub action workflows:

- *Continuous Integration*: The integration workflow is running some static code-checks[3] and tries building the Golang. application.
- *Continuous Deployment*: The deployment workflow will build Docker images for both the API and the application and push them to Docker Hub. After it will login into the machine, which is being hosted on DigitalOcean, and deploy the system.
- *GitHub release*: Fetches the latest version of the application from an API endpoint and makes a new GitHub release automatically.

A visual representation can be seen in Figure 6 found in appendix A.

2.4 Repository organization

At the top level of our repository, we have our configuration files (vagrantfile, Dockerfile, docker-compose, iac.sh, etc.), and other documentation files such as licenses, service level agreements, `README.md` and a bash script (wait_for_release.sh) that is used in the CD to wait for the new release version is deployed before starting the "GitHub release" GitHub Action workflow. The script iac.sh is used to create the infrastructure on DigitalOcean from scratch, it runs vagrant and then it connects to the machines to set up the Docker Swarm Cluster. To separate functionality we have organized folders in a way to isolate features: `src`, `monitoring`, `api` and `.github/workflows` are the most critical packages that are involved in our system.

- The `src` folder contains most of our codebase, containing structure and models for the database, test files, controllers, and static front-end layout files.

[3]See: `https://github.com/ITU-DevOps-N/go-minitwit/blob/develop/.github/workflows/continuous-integration.yml` for a list of services used

- The rest of the codebase is stored in `api`, where we keep our API for interacting with the user simulator.
- In `monitoring` we have our relevant monitoring files (Prometheus/Grafana).
- In `.github/workflows` we have the configuration files for Github Actions.

2.5 Applied Branching Strategy

Branching model

- `main`: Entirely stable code should reside here, possibly only code that has been or will be released. Code in this branch will go through the CI/CD pipeline.
- `develop`: A parallel branch that is worked from or used to test stability — it is not necessarily always stable, but whenever it gets to a stable state, it can be merged into main. Used to pull in topic branches, i.e. hotfixes, features etc. Tested on develop and merged into main.
- `feature/`: A short-lived branch that you create and use for a single particular feature or related work.

Whenever a group member wants to add code to the repository, the first step they do is to create a new branch, `feature/<feature_name>`, where `<feature_name>` is to be replaced with a descriptive name. Once the feature is ready, the developer will create a pull request from the `feature/<feature_name>` branch to the `develop` branch. All the main functionalities of the system should work in the `feature/<feature_name>` branch before approving the pull request. At least one reviewer will be required to approve the pull request. Once the pull request has been merged, the `feature/<feature_name>` branch should be deleted locally and remotely. Once a stable release is made, the `develop` branch will be merged into the `main` branch. At least two reviewers will be required to approve the pull request. The process is illustrated in the figure below.

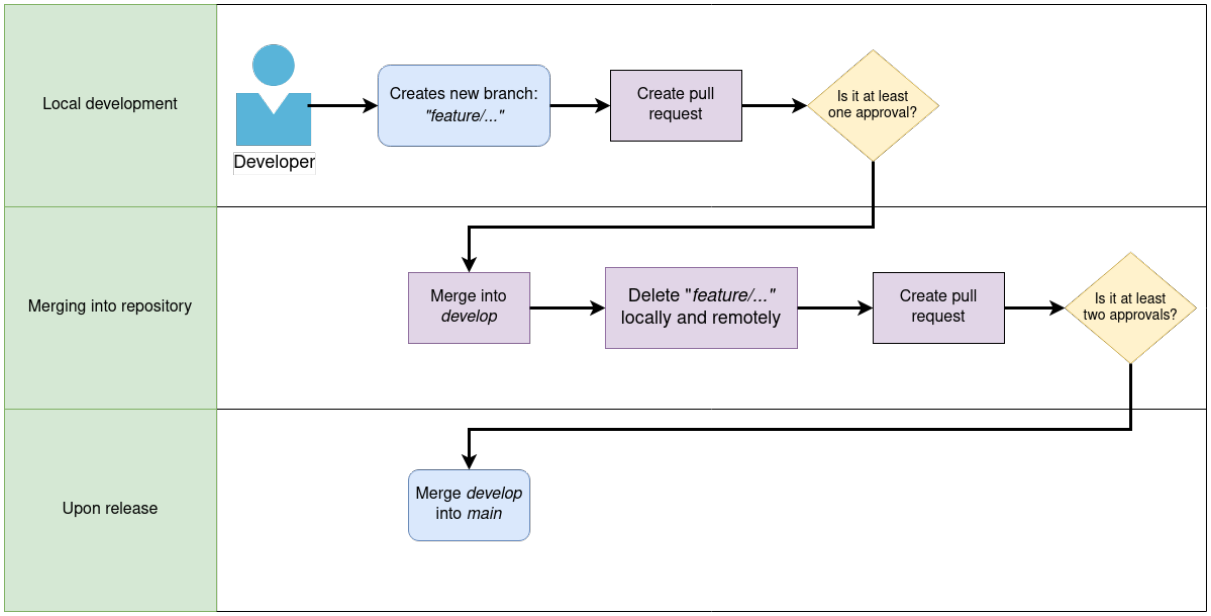

Figure 4: Branching strategy

2.6 Applied Development Process and Tools Supporting It

We used Github issues to declare new tasks that needed our attention. Whenever a group member encountered a problem that needed to be addressed, they would create a new issue describing what needs to be done. In addition to Github issues, we also configured Bugsnag to automatically create GitHub issues. If something was critical and needed to be done urgently, we made use of the ”urgent

message" function in Microsoft Teams, which notifies all members of a chat in time intervals until it is toggled off.

2.7 How do you monitor your systems and what precisely do you monitor?

For monitoring our system we are using Prometheus, an open-source monitoring system offering dimensional data models, flexible query language and a modern alerting approach. It collects and stores its metrics (numeric measurements) as time-series data, e.g. the metrics information gets stored along with a timestamp indicating at which time it was recorded, additionally with key-value pairs called labels.

Prometheus is configured in `Prometheus.yml`

We choose to monitor the following metrics of our system:

- **CPU Load Percentage** - provides information about the processor (CPU) load, in percentage, of both the API and Web systems.
- **Gin Total Requests** - shows the number of total requests on web-app and API
- **Prometheus Process Memory** - give information about the amount of memory the Prometheus process is using from the kernel
- **Go Memory Stats** - displays how much memory the go process is using at a certain time.

Then we use Grafana and its dashboard to analyze/visualize data queries from Prometheus, to give a nice monitoring overview of the system. See Grafana Dashboard at appendix C.

2.8 What do you log in your systems and how do you aggregate logs?

For logging, we use the *ELK stack* to store, search, ingest and visualize data collected from our system, in addition to *filebeat* to aggregate logs. Within our `filebeat.yml` we specify that we take input from all containers in our docker setup. The output filters the data to the index which contains logs in form of response codes from our *ITUMiniTwit* application, including messages that contain the keywords "`ERR`", "`WARN`" and "`GIN`". The configuration for our logging output can be seen in the figure below.

```
35  #---------------------------- Elasticsearch output ------------------------------
36
37  output.elasticsearch:
38    hosts: ["elasticsearch:9200"]
39    indices:
40      - index: "filebeat-minitwit-%{[agent.version]}-%{+yyyy.MM.dd}"
41        when.or:
42          - contains:
43              stream: "stderr"
44          - contains:
45              message : "ERR"
46          - contains:
47              message : "WARN"
48          - contains:
49              message : "GIN"
50  logging.json: true
51  logging.metrics.enabled: false
```

Figure 5: Snippet from `filebeat.yml`

Finally, we use Kibana to visualize the filtered data to dashboards, where the data can be analyzed into for example graphs, charts and heatmaps.

2.9 Brief results of the security assessment

We used Arachni to do a penetration test of our API, which showed that we have some issues with cross-site-forgery, unencrypted password forms, common directories, password fields with auto-complete,

and missing X-frame Options header[4]. However, these issues were either reported once or twice during a 24 hours penetration testing and the other group who were penetrating testing our API were not able to find any vulnerabilities. [5]

2.10 Applied strategy for scaling and load balancing

Scaling: we can summarize the scaling strategies into Vertical and Horizontal Scaling.

- **Vertical scaling** - we resized the virtual machines into DigitalOcean, increasing the CPU and RAM of these latter. This was the solution that was chosen instead of increasing the number of machines because of cost convenience.
- **Horizontal scaling** - Docker Swarm cluster makes it possible to scale our containers horizontally. This can be achieved either through manual maintenance or editing the docker-compose file.

Both of them were not automated and therefore human interaction was required in order to perform scaling of any kind. If we observed that one or more of the virtual machines that were hosted on DigitalOcean had too much load on for example CPU or memory usage, we would perform vertical scaling to the existing machines by upgrading their specs or horizontally by increasing the number of containers.

Load Balancing: currently there are three virtual machines hosted on DigitalOcean and they are all part of the Docker Swarm cluster, one manager and two worker nodes. All the containers are spread among these machines. Our main endpoint is exposed by DuckDNS and it points to the manager node's IP. Every time a service is called through this endpoint, it is resolved by Docker Swarm, no matter in which node the service's container is located.

3 Lessons Learned Perspective

3.1 Evolution and Refactoring

The main issues during the development phase and in general the evolution of the course were:

- Pace of the course
- Learning Golang
- Revising architecture often
- Backtracking errors.

The first issue was the **pace of the course** and the number of technologies which we had to add to our tech stack was challenging and work heavy. This limited our ability to keep up with the course material in the beginning since we simultaneously had to **learn a new language** and web framework.

During the implementation of new components and **revisiting the architecture**, we spent a remarkable amount of time designing our system to make sure it was built the right way. With the load generated by the Simulator API, **backtracking errors** and performing bug fixing was a long and continuous work. It was often necessary to investigate metrics, logs and Bugsnag reports to identify the issues.

Initially, we only released new application versions on Docker Hub with each new image in our continuous deployment pipeline. However, the image did not capture the entire state of the repository for each release, only the source files for building and running the application itself. We fixed this late in the course by implementing a new GitHub Action to automatically create a GitHub release whenever we pushed to main. This release automatically made a changelog of all the commits that went into main since the last release, and the release version number was also updated automatically. In hindsight, it should have been there from the beginning to comply more with the DevOps way of thinking with more frequent releases.

[4]Some of these issues were fixed
[5]See full Risk identification and analysis here [8] and pen-test report here [7]

3.2 Operation

We had a difficult time implementing both monitoring and logging, which resulted in a hard time handling issues with our application. Also, documentation was often made after implementation and sometimes not together with pushing into development/main. Ideally, we should have been doing implementation and documentation parallel and enforcing documentation before pushing into development or at least production.

Another issue was related to the developer experience. We wanted to have a philosophy of 'keeping the machine clean', meaning that we did not want everyone to download specific dependencies that the system needed, such as the Go version and DBMS version. We wanted a seamless setup of the environment where everyone was using the same versions with Docker. In Go, you can test your code modifications without compiling them into an executable. However, the way our architecture was setup meant that we had to compile the application for each change. For development purposes, this is very inconvenient, since compiling takes a lot of time for even a simple change.

3.3 Maintenance

We were very static in our approach to working with maintenance as we mostly only worked once or twice a week on developing/maintaining our system. Ideally, we should have been more flexible in our way of working with a larger focus on maintenance, hence, working less but more days. This was also reflected in the many errors generated by the simulator, which during a week would aggregate to a lot of issues not handled. Our attempt to accommodate for this was to meet during weekends to catch up, however, this approach was still reactive and not proactive.

3.4 Reflection and description of our DevOps style

For this project the application itself was rather secondary as setting up the whole system infrastructure was the focus. This means we actually never got to the point where we were adding that much new functionality or visualization to the application, thus hard to comply with the DevOps way of working. We also focused on making development as easy as possible since being efficient in implementing new features was a major key to the success of this project.

4 References

[1] *Duck DNS*. https://www.duckdns.org/.

[2] *DigitalOcean*, 2022. https://www.digitalocean.com/.

[3] Elasticsearch B.V. *What is the ELK Stack?* https://www.elastic.co/what-is/elk-stack.

[4] GitHub. *Gin Web Framework)*, 2022. https://github.com/gin-gonic/gin#gin-web-framework [Accessed: 25/05/2022].

[5] GitHub. *Automate your workflow from idea to production*, 2022. https://github.com/features/actions.

[6] Google. *Go*. https://go.dev/.

[7] Group-N. *Pen-test*, 2022. https://www.arachni-scanner.com/reports/report.html/#!/summary/charts [Accessed: 30/05/2022].

[8] Group-N. *Risk Identification and analysis*, 2022. https://github.com/ITU-DevOps-N/go-minitwit/wiki/Security [Accessed: 25/05/2022].

[9] HashiCorp. *Vagrant*. https://www.vagrantup.com/.

[10] Docker Inc. *Swarm mode overview*. https://docs.docker.com/engine/swarm/.

[11] Jinzhu. *The fantastic ORM library for Golang*, 2022. https://gorm.io/.

[12] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[13] Grafana Labs. *Grafana*, 2022. https://grafana.com/.

[14] Victor Osadchiy. *Why use Go)*, 2022. https://yalantis.com/blog/why-use-go/ [Accessed: 22/05/2022].

[15] The Linux Foundation Prometheus. *From metrics to insight*, 2022. https://prometheus.io/.

[16] SmartBear Software. *Balance agility with stability*, 2022. https://www.bugsnag.com/.

A CI/CD Chain

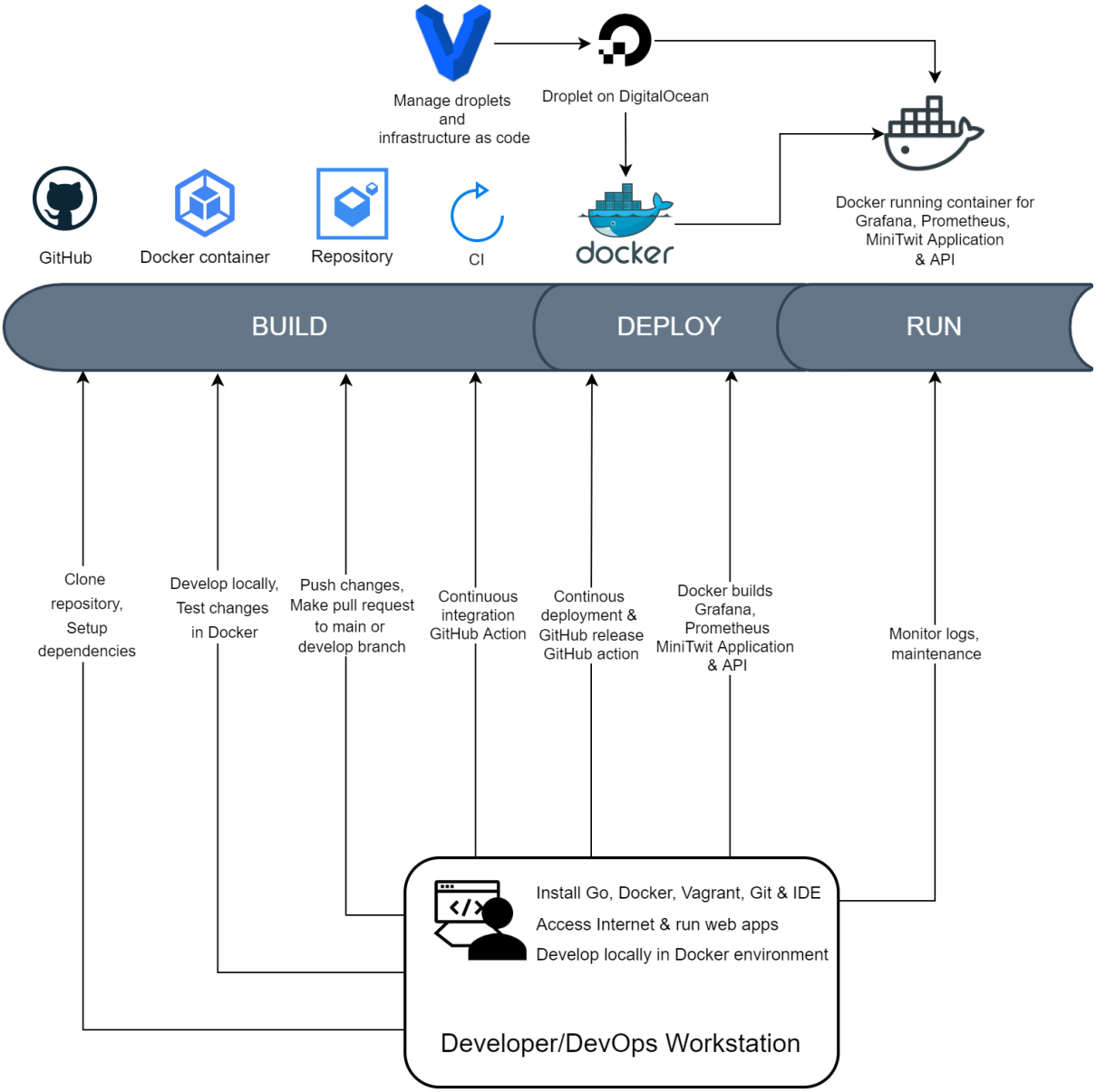

Figure 6: CI/CD Chain

B Risk analysis matrix

Risk scenario	Likelihood	Impact	Risk	Todo
1a	M	L	L	-
1b	L	H	M	-
2a	H	M	H	Encrypt communications to ensure adversary cannot read information sent. Authenticate packets to prevent spoofed packets. Implement EDR (Endpoint Detection and Response software)
3a	H	H	H	Sanitize SQL inputs
4a	L/M	M	M	Investigate logs to find design flaw
5a	L	H	M	Secure credentials/secrets for gaining access to logging services better
5b	L	H	L	Review awareness and security best practices between developers/staff
6a	L	M	M	-
6b	L	M	L	Investigate how to harden vulnerable system component (if not possible - investigate replacements)
7a	L	L	L	Salt + hashed passwords in database
7b	L	H	L	Review awareness and security best practices between developers/staff
8a	M	M	M	Investigate our libraries and dependencies, ensuring they are only consuming trusted repositories
9a	M	H	M/H	Ensure log data is encoded correctly to prevent injections or attacks on the logging or monitoring systems.
9b	L	H	M	Review our choice of logging systems
10a	M	M	M	Sanitize and validate all client-supplied input data.
				Restrict connectivity to internal ports.

Figure 7: Risk analysis matrix

C Grafana monitoring

Figure 8: Risk analysis matrix