DevOps - LLAMA GEMST

tbab, harw, gusm, edtr, mihr

May 2024

1 Tasks

- Miha, Simon: security analysis — open
- Simon add a nice intro page
- Simon: find & install template for latex file — more or less done — nice template, lol
- Simon: look at Mircea's slides and make us a template for graphs — open for later
- finish Docker swarm — SH: fix fluendt
- EDU: Implement Terrafrom
- Gustav: CD -¿ Create bash script to run the stack deploy
- EDU: Proxy after security is done, 21 km run
- Miha and Tomas: type

1.1 DIAGRAMS

- D1: layer distribution for cloud architecture
- D2: General overview including everything
- D2: Detailed overview CI/CD (pre commit: yaml file)
- D3: Detailed: Code architecture
- D4: Detailed: Request diagrams for different stacks:
  - -Grafana/Prometheus Stack: Quantitative
  - -EFK - Kibana/Elastic/Fluented: Qualitative

2 Introduction

The following report described the work done during the Spring 2024 DevOps, Software Evolution and Software Maintenance taught at the IT University of Copenhagen. Our report includes the description of the final state of our system, the process we followed to develop and deploy it, lessons we learned, and reflections on technologies used and high and low level decisions we made throughout the semester.

3 System's Perspective

In this section, we describe the architecture of our ITU-MiniTwit system, the technologies that we used to support it, and we provide a rationale for the technologies we chose. This section also shows how a front-end or API request is processed by our system and what is eventually outputted. Finally, we show the results we obtained from using static analysis tools.

3.1 How a request is processed when it comes from the front-end/simulator

The diagram in figure (note: todo) shows how our system handles a request coming from the front-end or the API. We generally follow a model-view-controller architecture where the main function acts as the entry point of our application. Depending on the request type, the main function then forwards the request to handlers which redirect, handle logic, pass data to HTML templates or pass data to be written/read to/from the database. We chose this architecture as it would allow us to reuse code as much as possible and to seamlessly extend our code base.

For example, since we separated the methods that communicate with the database, we were able to add an abstraction layer without modifying the core logic of the application. Similarly, we were able to migrate our database from SQLite to a managed MySQL database on DigitalOcean very easily since our logic for connecting to the database was contained in one method which was invoked by several other methods when they needed to connect to the database.

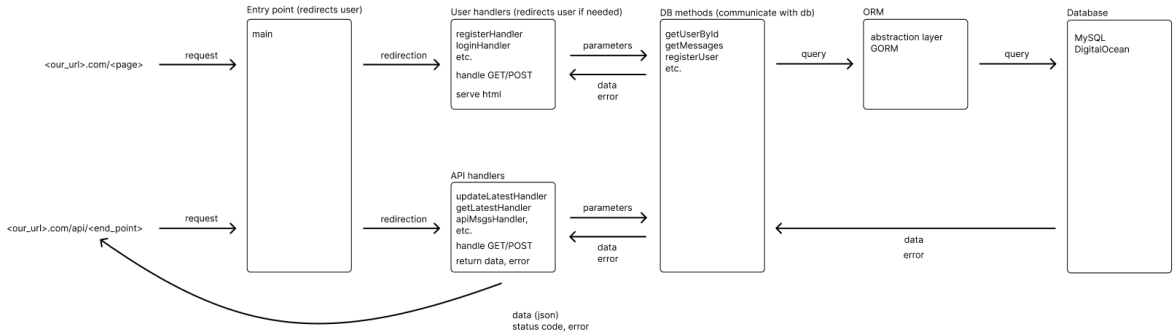

Figure 1: How our system handles a request.

3.2 Architecture graphs of the system

3.3 Technologies used

In this subsection we describe some of the key technologies we used on our system and justify them. The complete list of dependencies can be found in the appendix.

3.3.1 Programming Languages

We chose to refactor our application using GoLang (Go). Go is a high level compiled programming language developed at Google in 2009. We chose to use it to refactor our app due to its simple syntax, automatic garbage collection and very high performance. Because of its easy to approach syntax, GoLang is generally regarded as a simple language to learn compared to others such as C++. For this reason, this made it a clear choice when we all saw the refactor of ITU-MiniTwit as an opportunity for our group members to learn a new language.

The automated tests that we used to monitor development on our app and make sure that no buggy code made it to our deployed application were written in Python. These were provided to us in the course repo and we chose not to refactor them to GoLang as we were able to use them to test our application.

Finally, we used templated HTML files for the front-end of our application.

3.3.2 Containerization and Orchestration Tools

In order to ensure that our system had a high rate of availability (i.e. low down-time), we did horizontal scaling. Even though our cloud provider *Digital Ocean* allows increasing droplet size (vertical scaling), we decided to also scale horizontally and thereby increase availability with *Docker Swarm*. We chose this tool due to three reasons: 1) Ease of setup - since we already had experience with containerization

Docker Swarm was easy to pick up, 2) Load Balancing - *Docker Swarm* has built-in load-balancing thereby increasing performance, and 3) Fault tolerance - by replicating nodes *Docker Swarm* increases fault tolerance and availability.

3.3.3 Database Technologies

We were first provided with a lightweight SQLite database for ITU-MiniTwit. While this worked well for some time, we eventually realized that we needed a more persistent form of storage. We chose to purchase DigitalOcean's managed MySQL database solution. This was an easy choice for us as we were already using DigitalOcean to deploy our app online and using one of their database solutions allowed us to automatically refuse any incoming connection, except if it originated from our online application, which increased the security of our systems greatly. We also chose their MySQL solution as it was very easy to duplicate the schema used in the SQLite database that was provided to us and because we would be storing very structured data, eliminating the need for a document based database.

In addition to this, we implemented a database-abstraction-layer via *GORM*. This was done for several reasons:

1) Since GORM supports multiple databases like SQLite, MySQL and PostgreSQL this added flexibility to our system since we were not tied to one single database for the entire cycle of our system.

2) GORM provides auto-migration, which in short, makes it much easier to import an SQL table and perform operations on it.

3.4 Static Analysis Tool Results

We gave two static analysis tools access to our GitHub repository: SonarCloud and CodeClimate. Both found some minor issues such as code smells, and maintainability issues, but did not raise any major problems such as bad code practices or security vulnerabilities.

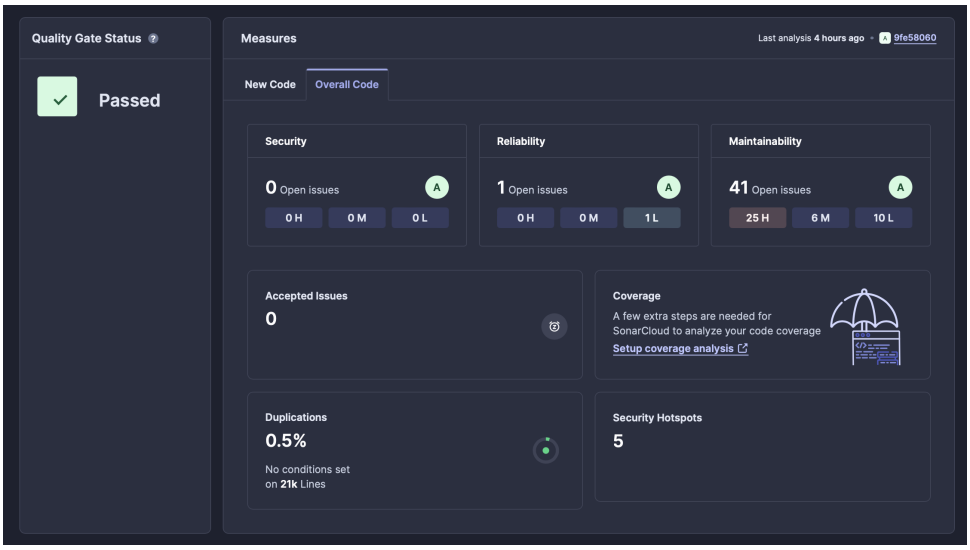

Figure 2: SonarCloud's dashboard after running an analysis on our repository.

Furthermore, installing SonarCloud also gave us access to its bot which reviewed our code quality before we merged features into our main branch.

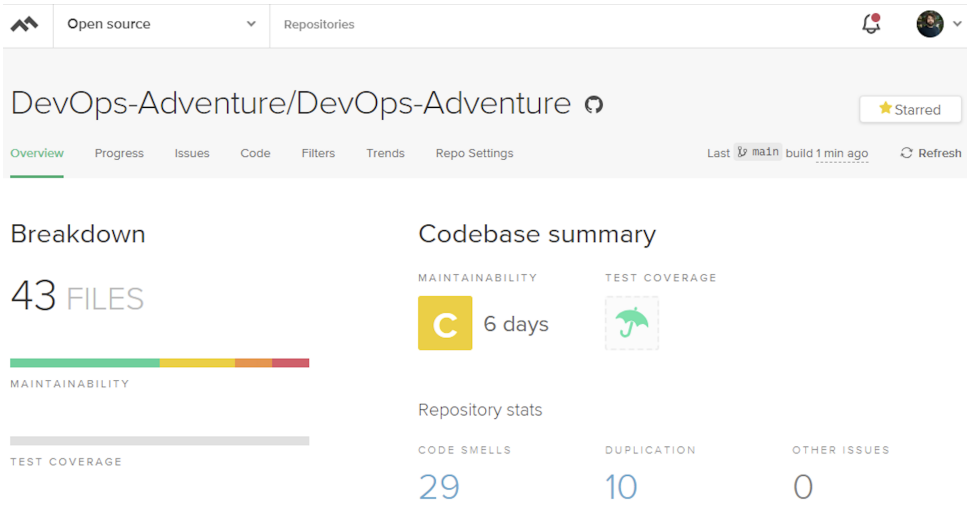

Figure 3: Code Climate's dashboard after running an analysis on our repository.

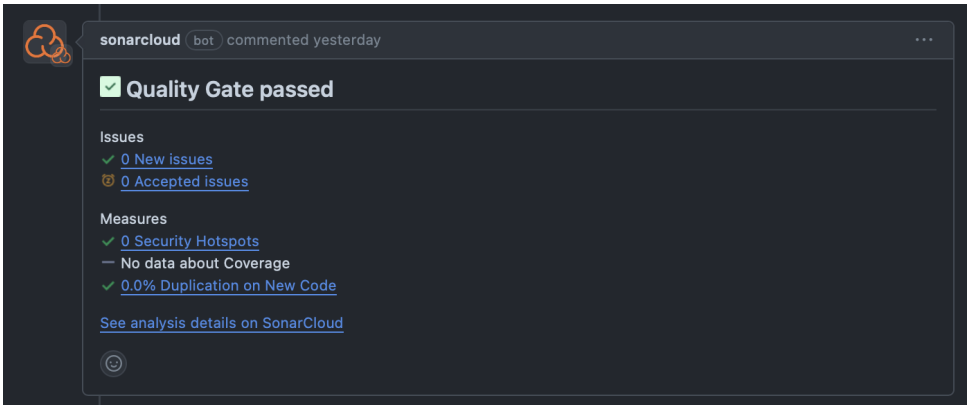

Figure 4: SonarCloud's bot adding comments on our pull requests.

4 Process' Perspective

4.1 Initial Resource Partitioning

Resource setups are made manually, only once - in the beginning.

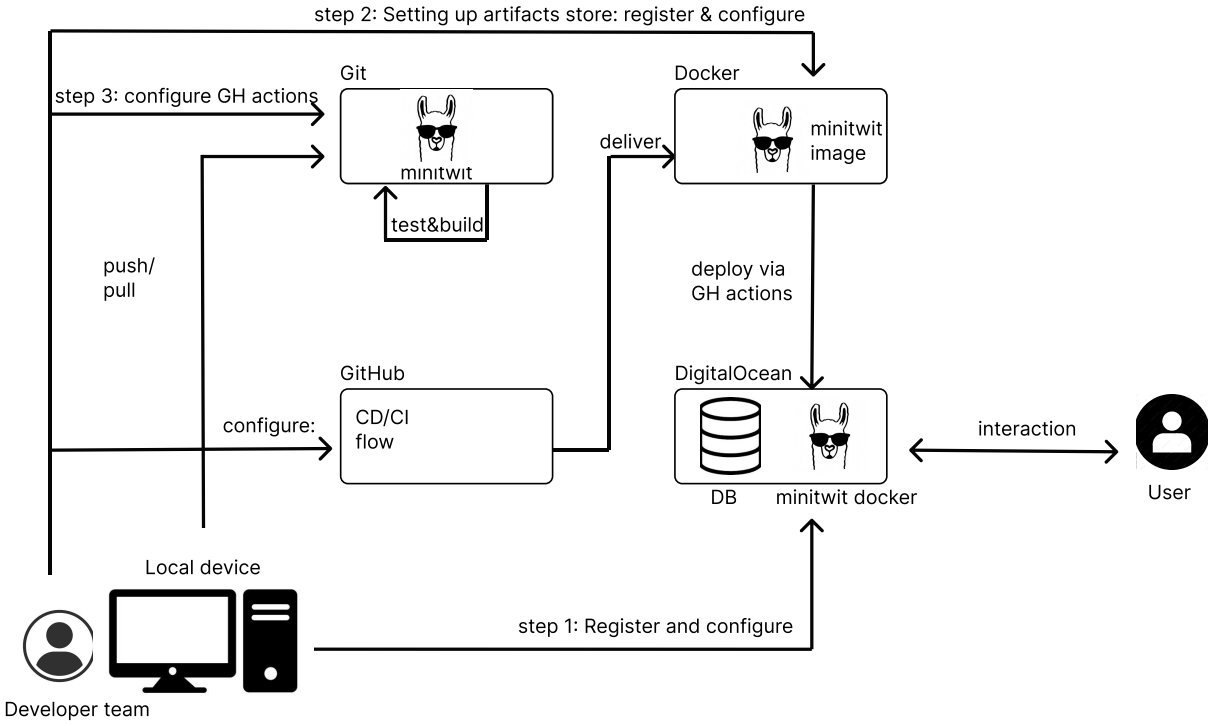

Figure 5: Resources setup.

4.1.1 Step 1: Creation of Virtual Machine and server set up on Digital Ocean

In our vagrant file we start the VM (creating a droplet) (argumentation for size and location), connect to the created droplet (on the server), installing docker container and test docker is working correctly. It also contains a deploy scrips that updates the latest Minitwit docker image from DockerHub and starts the containerized application via docker-compose.

4.1.2 Step 2: Setting up artifacts store

This step includes registration in DockerHub to keep our Minitwit image up to date. We chose Digital Ocean instead of local setup of a private server, to minimise maintenance and risk failure, ease of use and access, and multiple possibilities for integration for our system like docker Swarm and hosting our database on cloud. (why exactly Digital Ocean), student voucher etc

4.1.3 Step 3: Configuring secrets on GitHub repository

We are using GitHub workflow to store our CI/CD jobs, and they are defined via high-level actions in .yaml files. This stage includes generating access token from DockerHub, setting up SSH key from local device and generating access token from could provider which are added to GitHub actions as secrets. The secrets are accessible to the GitHub workflow. The purpose is to provide access to GitHub workflow to the latest Docker mini twit image, and connect to the server to call the deploy script.

4.2 Continuous Integration

We established a continuous integration (CI) pipeline to enable us to often test our code and new features automatically against the tests that were provided to us. We also created this pipeline to shield our main branch, where our ITU-MiniTwit was deployed, from having buggy or malicious code merged into it as any merge into main would automatically be deployed to our live system by our continuous deployment pipeline. Finally, we also setup our CI pipeline to run linters against code being added to the code base and to force a human developer to approve code before having it merged into our main branch and

eventually deployed to our production environment. Figure 6 (note: todo) describes the steps in our CI pipeline and its main stages.

(note: Gustave, if you want to talk about linters, here is a good place, I think :D)

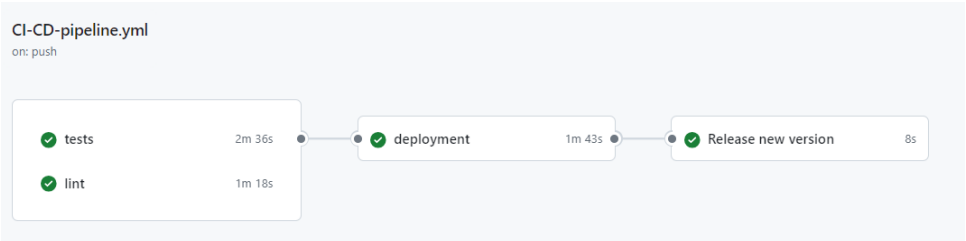

Figure 6: CI-CD stages.

please keep this for discussion before you remove it [mhr], ok sorry :(

The purpose of the CI (Continuous Integration) flow is to integrate changes, create artifacts, and run tests. The CD (Continuous Deployment) flow automates the deployment process from a local device to the server. Both CI and CD flows are defined in a single YAML file and can be triggered manually from GitHub Actions or automatically by committing changes to the main branch.

CI stage 1: Testing [mhr]

Tests

CI acts as a gatekeeper for our code, it includes: Setting up the Go environment and install all GO dependencies and starts the minitwit application. The workflow check that the server is running before it runs the tests. Then is does the same for Python environment, where it install the dependencies for testing. After that checks the ports before running the tests. Runs the following types of tests:

- test itu minitwit ui.py

User interface tests that check whether user exists in the database before registration, then performs the registration and verifies with success message. For this tests purposes it installs Firefox, Selenium and Geckodriver;

- test minitwit sim api.py

These tests check that the functionalities of minitiwt work as expected: user registration, message posting, following/unfollowing, fetching messages and database cleanup; with focus on the latest filed updates

- test refactored minitwit.py

Also standard test to check the application functionalities, focussing on the user registration, login/logout and message post and retriecal, follow/unfollow and correct timeline.

Linters

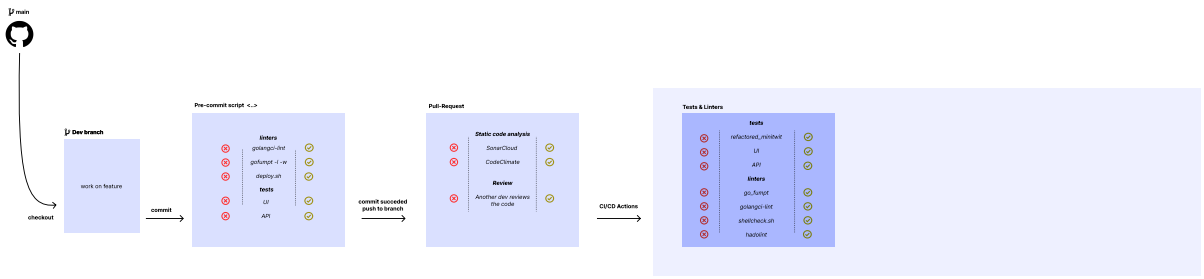

Figure 7: Linters.

CI stage 2: Deployment [mhr] In the deployment stage, after the tests and linters complete, in the

the workflow first fetches the code from the repository, then logs to Docker Hub using the secrets in GH actions. It sets up Docker Buildx and builds and pushes the Docker image of our application to Docker Hub. To optimise the process it is using stored cache. Next, SSH access is configured by setting up the necessary keys. The workflow then deploys the application to the server by copying required files and executing the deployment script via SSH, passing in the database credentials securely.

CI stage 3: Release new version [mhr]
In this final stage, after the deployment is successfully completed, the workflow includes fetching the code from the repository and setting up the Git user identity (needed for tag creation). A new tag is created for each version by incrementing the latest tag number. Finally, a new GitHub release is created based on the new tag (e.g., v35.0). This release generates release notes that include bug fixes and other changes based on the commits since the last release.

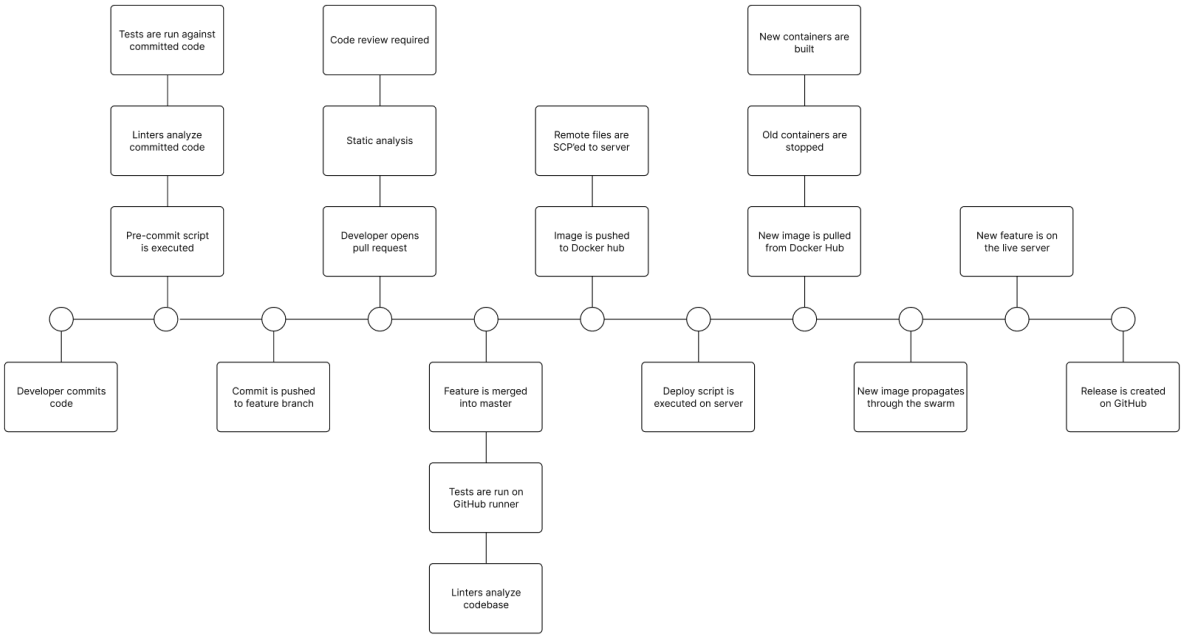

Figure 8: Our CI/CD pipeline.

4.3 Continuous Deployment

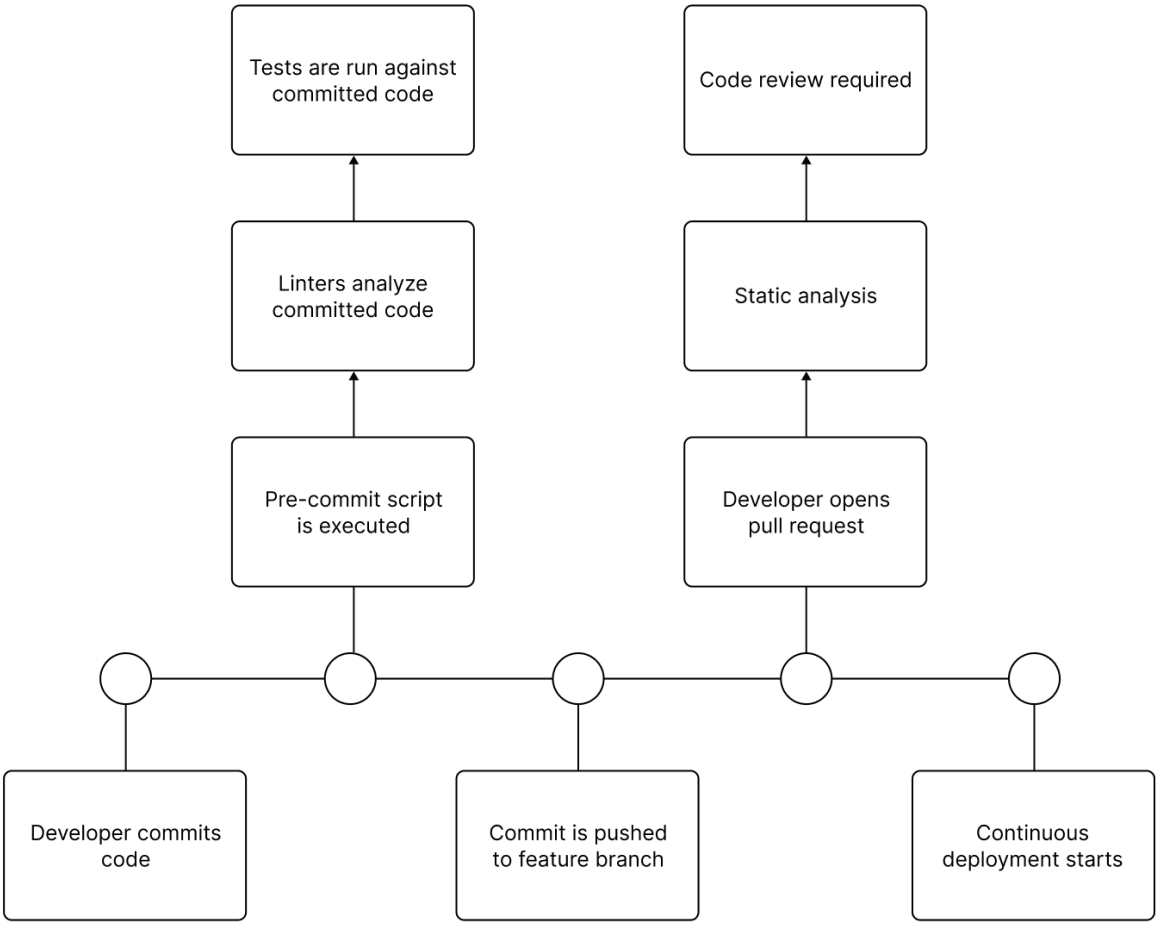

Figure 9: Our continuous integration pipeline.

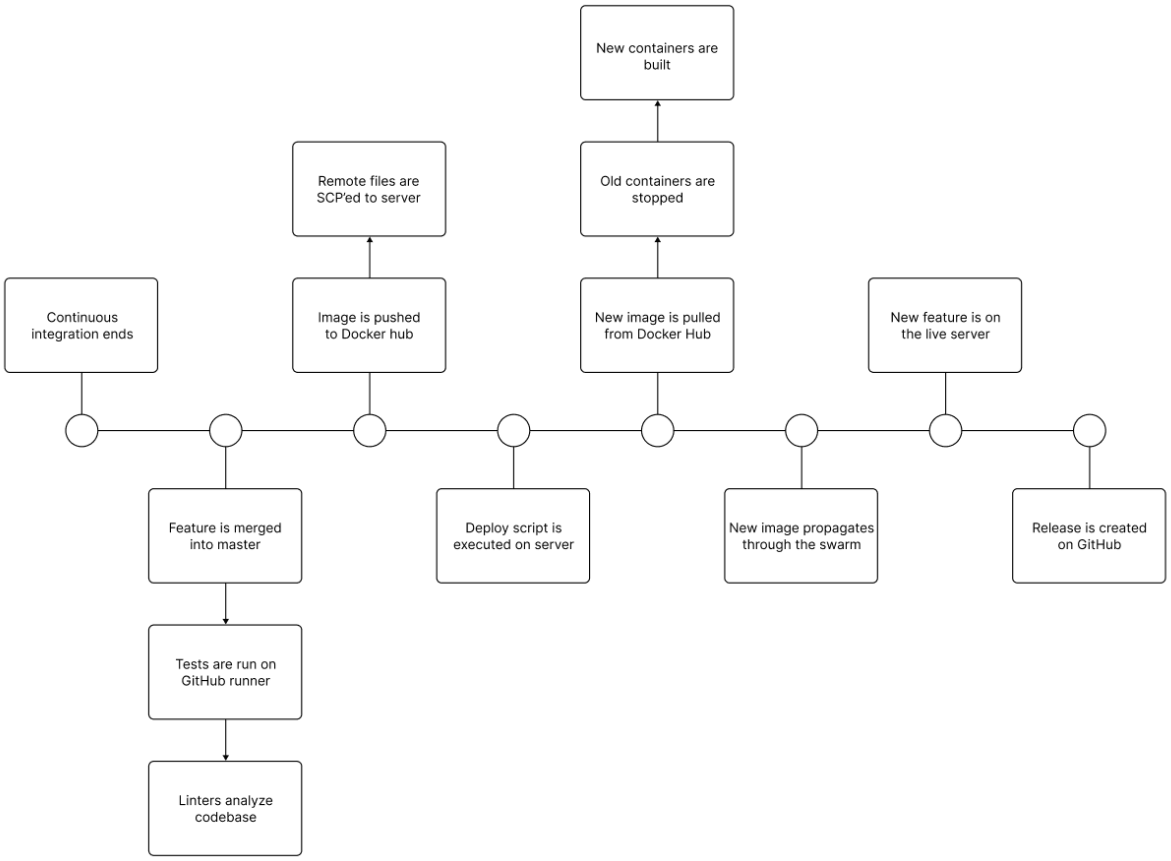

Figure 10: Our continuous deployment pipeline.

We also established a continuous deployment (CD) pipeline to automatically deploy the updated version of our ITU-MiniTwit. Once a new feature was incorporated into the main branch of our repo after the successful merge of a pull request, our CD pipeline would automatically trigger and deploy the new version of our application. Our CD pipeline was executed with a GitHub action and had many responsibilities:

1. Build the new version of ITU-MiniTwit and run tests against it.
2. Build a new Docker image of our app and push it to DockerHub.
3. Upload the deploy script from our repo to the DigitalOcean server
4. Stop current containers running on the server and download the new images from DockerHub.
5. Start the containers with the new images.
6. Release a new version of the ITU-MiniTwit on GitHub.

Once the CD pipeline had terminated, the new feature was deployed to our live system. It is important to note that the containers were brought offline and back online using a rolling update scheme, to ensure that our system would always be available to access. The exact steps of the pipeline can be found in figure 10 (note: todo).

4.4 How we used GIT

Note: Eduardo write this please. Tomas: maybe we don't need a section for it in the end?

4.5 How we monitor our systems and what we monitor/how we use this information to improve our system [mhr]]

Types of monitoring
White box monitoring, where we focus on the processes within our application, providing insights for the internal state and performance of our application
Active monitoring, where we monitor interactions with our application, tracking metrics for active logged users (note: without tracking their location)
Use of monitoring

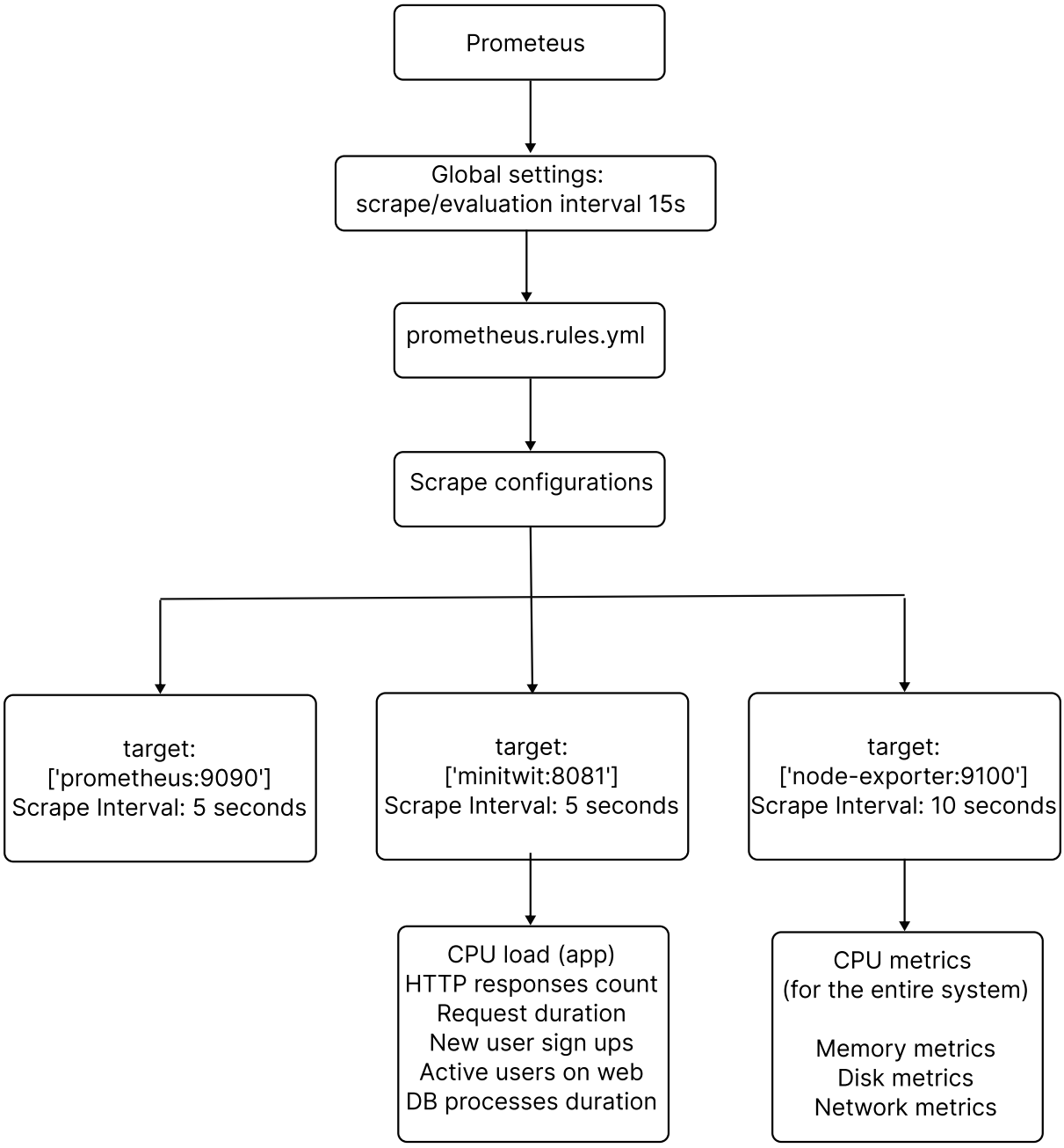

Figure 11: Prometheus configuration.

On fig. 8 is shown the hierarchy of the orders that Prometheus is executing, how and from where it collects information. We have chosen to monitor metrics on two levels: application and system level.
1/CPU load on application level:
We monitor the CPU load of our application. We discovered a correlation between CPU spikes and application crashes. This was due to differing load permissions between Docker (6GB) and the server

size (4GB). The dashboard confirmed our hypothesis, and once we increased the RAM in our droplet, the application started working fine.

2/Linux server

Monitoring CPU, Memory, and Network metrics at the system level helps us determine whether issues originate locally within the application or at the server level. To be informed about overall health of the hosting environment is of a crucial importance as it could affect our application performance.

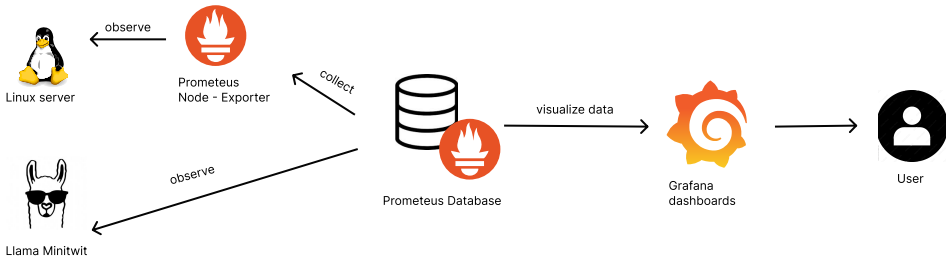

Figure 12: Visualization flow.

Figure 9 shows that after scraped data is stored in Prometheus server it reaches the final user via Grafana visualization dashboards. We chose Grafana and Prometheus stack because of numerous advantages: FOSS helps quick and low cost start and flexible scale up, without hidden costs. Tt is widely used as industry-standard way of monitoring, has good integration with cloud-origin environments and variety of plugins, provides detailed visualizations and it is very user-friendly.

4.6 Logging and all that witchcraft crap. What we log and why we used those technologies, how we make sure that we're not logging too little but also not logging too much SIMON

4.7 Security Assessment

4.7.1 Risk Identification

The following assets have been identified and considered in the context of the security assessment:

1. **User Data**: User data, includes any information directly connected to users, such as emails, passwords, and blog posts.
2. **Production VMs**: Servers used to run the MiniTwit application.
3. **Logging Data**: The system generates logs, especially qualitative logs that could expose behavior patterns and personal information.
4. **Source code**: The source code of the MiniTwit application.
5. **Private Keys and Access Credentials**: Private keys and credentials used to access various parts of the system.

These five assets of our system have been identified and could be at risk. Based on these, eight risk scenarios have been constructed. For each scenario, a short description is given, the impact is described and they have been categorized in terms of three characteristics of security as defined in the CIA triad: confidentiality, integrity, and availability.

4.7.2 Risk Scenarios

Risk Scenario 1: *Data Breach via SQL Injection*
Description: An attacker exploits a SQL injection vulnerability in the MiniTwit application to gain unauthorized access to the database.database.
Impact: Exposure of sensitive user data (emails, passwords, blog posts).
Category: Confidentiality, Integrity

Risk Scenario 2: *Compromised Production VM*
Description: The production VM is compromised due to weak SSH credentials, allowing an attacker to gain control over the server.
Impact: Unauthorized access to application and

data.
Category: Integrity, Confidentiality, Availability

Risk Scenario 3: *Distributed Denial of Service (DDoS) Attack*
Description: The MiniTwit application is targeted by a DDoS attack, overwhelming the servers and causing service disruption.
Impact: Unavailability of the MiniTwit application.
Category: Availability

Risk Scenario 4: *Theft of Private Keys and Credentials*
Description: An attacker gains access to private keys and access credentials stored in a misconfigured server.
Impact: Unauthorized access to encrypted data and critical systems. Potential for man in middle attacks.
Category: Integrity, Confidentiality, Availability

Risk Scenario 5: *Source Code Leak via Public Repository*
Description: The source code of MiniTwit is accidentally made public on a version control platform like GitHub.
Impact: Impact: Exposure of application vulnerabilities and sensitive information (e.g., hardcoded credentials)
Category: Integrity, Confidentiality

Risk Scenario 6: *Third-Party Software Breach*
Description: A third-party software component used in the MiniTwit application is compromised, leading to security vulnerabilities or data breaches.
Impact: Software vulnerabilities and data leaks
Category: Integrity

Risk Scenario 7: *Exposure of Sensitive Information in Logs*
Description: Qualitative logs containing sensitive user information and operational details are exposed due to misconfigured logging settings.
Impact: Leakage of user behavior patterns eventually personal information.
Category: Confidentiality

Risk Scenario 8: *Brut Force Attack*
Description: An attacker that uses trial and error to gain access to passwords, login credentials, and encryption keys.
Impact: Unauthorized access to user accounts, data leaks, data manipulation.
Category: Confidentiality

4.7.3 Risk Analysis and Mitigation

Following we have determined the severity of each threat and placed them in a risk matrix to prioritize appropriate mitigation strategies. The severity categories are defined as follows:

	Insignificant	**Marginal**	**Critical**
Likely			
Possible		4	1, 6
Unlikely	8	5, 7	2, 3

Furthermore, the outcomes are discussed and potential solutions are presented.

	Low risk
	Medium risk
	High risk

4.8 Strategy for scaling and upgrading our system (talk about moving to a cloud db, and buying more droplets when moving to swarm

4.9 Our use of AI assistants [mhr]

Chat GPT can be an invaluable tool for various purposes:
- It's helpful when we needed suggestions for different ways to solve a problem, especially when uploading error messages.
- To provide simple examples that clearly demonstrate the concept that we were trying to implement.

- Can show different language implementation of how a particular function can be implemented in different programming languages.

and not very useful with:
- It is not particularly useful for direct translations of a function from one programming language to another.
- it cannot provide direct (correct) answers, especially for complex or highly specific questions.

Considerations for refactoring strategy:

5 Lessons Learned Perspective

In this section, we describe some of the technical and managerial difficulties we ran into during the semester, the lessons we learned from them and reflect on how we these difficulties could have been avoided or how we could have recovered more gracefully.

5.1 Technical Issues

- Database cleared for some reason every now and then?
- Swarm setup issues

5.1.1 Resource Starvation

During the 4th session of the course, we were tasked with deploying our ITU-MiniTwit system to the cloud so that the simulator could eventually interact with it through our API. We partitioned a droplet on DigitalOcean and deployed our app through a Docker container. We had a few additional services running on other containers but on the same droplet. Once the simulator started, our Docker container would seemingly randomly be shut down every few hours or every few days. After a few weeks of debugging, we eventually realized that there was a direct correlation between our container being shut off and when there was a spike in traffic on our ITU-MiniTwit. We quickly realized that the sum of the resources that we had allocated to each container running on the virtual machine installed on our droplet was higher than the resources we had purchased from Digital Ocean. More specifically, our containers were allowed to consume up to 6GB of RAM, but our droplet only provided us with 4. Thus, when the traffic spiked, the container consumed more RAM than it physically had access to and the OS would kill it.

While it's easy for us to look back and recognize that this was a simple oversight, we had seen many tutorials that recommended capping the resources each Docker container could consume and had simply glanced over the information.

5.1.2 Slow ITU-MiniTwit

After the simulator had been running on our database for some time, we noticed a significant increase in the dataset size, which resulted in longer query times. To address this issue, we implemented indexes directly in the database using SQL queries. To determine which indexes to create, we focused on the processes that were performing the slowest: accessing the public timeline, retrieving messages, and managing followers. For these, we implemented B-tree indices and we managed to optimize the performance of our database queries and ensuring efficient data retrieval.
For user table (unique):

```
CREATE INDEX idx_username ON user(username(255));
CREATE INDEX idx_email ON user(email(255));
```

For message table (non-unique):

```
CREATE INDEX idx_author_id ON message(author_id(255));
CREATE INDEX idx_pub_date ON message(pub_date(255));
```

For follower table:

```
CREATE INDEX idx_who_id ON follower(who_id(255));
CREATE INDEX idx_whom_id ON follower(whom_id(255));
```

5.2 What tools we used during the semester, how we divided work, how we shared knowledge?

5.2.1 Good Git practices (moved) [mhr]

We have established a routine where we worked on a new branch every week and tested our code locally before we push it to main. In order to be able to merge we need to have at least one reviewer to approve the code. Apart of this, also tests run on the code before it is pushed which significantly reduces the risk.

5.2.2 The three DevOps ways and what we used? [mhr]

The Principles of Flow

- Make Work Visible
- Visibility: In our group we apply the requirement of visibility in our work by defining and sharing our tasks in a common task board in Notion, bellow (by 03-03-2024). This helps us to keep track of the work that needs to be done and be informed what is each team member occupied with.

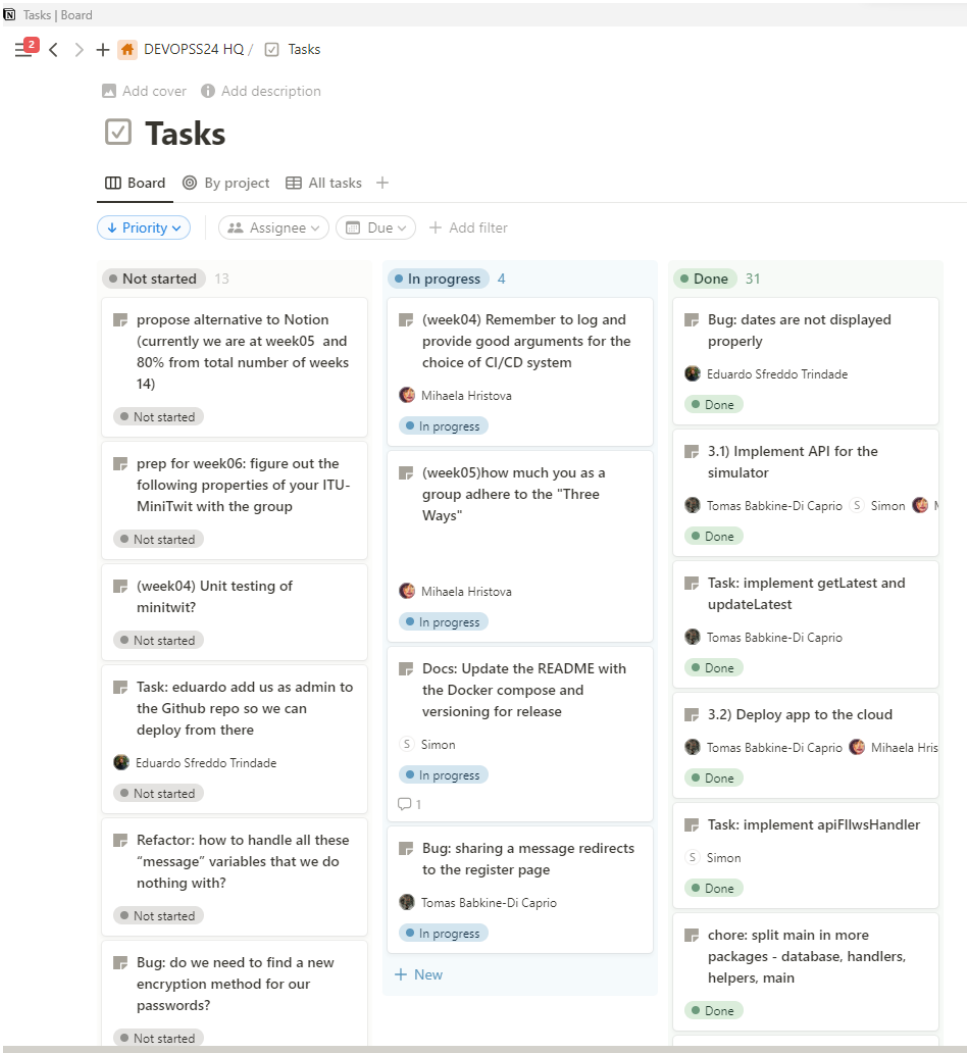

Figure 13: Notion dashboard where we keep track of our tasks.

- Limit Work in Progress (to be implemented)

  We have set a limit of the tasks that can be in each column in the Notion board) This means that we can only add a task when one is being finished. In case we are waiting on somebody's task to

be completed and we are free to start new task we will rather see what is causing the delay for our team member to finish his task and help him instead.

here we could also implement a rule that new week cannot be started without completion of the task for the previous week OR having max three tasks that can be "carried" over.

- Reduce batch sizes:

  in our case where we develop itu-minitwit application we could look at the different features for example post a message as a single unit operation that is being refactored, tested and deployed with the next *push* to main branch. The idea is instead of trying to implement all features at once (post messages, follow user, register etc) we develop and test one at a time. This shortens the lead time significantly and most importantly help us to detect and act on bugs much faster.

- Reduce the number of hand-offs

  Since we are a small organization that consist of 5 members we do not have a risk of wasting time on this principle. However we have implemented CI/CD flow that makes deployment to the cloud automated just with push to main branch. In out deployment process we use configured .yaml file "continuous deployment" that creates VM in Linus server, sets up a docker and tests out application in it (to be implemented), created a docker image of our application that sends to the server provider.

- Continually Identify and Evaluate Constraints

  improve work capacity by following the steps environment creation → code deployment → test setup and run → overly tight architecture

  We have detached the dependency from local device by working with VM and Docker where we make sure that all needed requirements to run our application are present. Our deployment process is automated, that is consisted of automated testing of our application on Docker (to be implemented, Postman testing to be mentioned here perhaps?)

  We have designed our app architecture with loose couples to achieve independency and safety when implementing changes.

- Eliminate Hardships and Waste in the Value Stream

  /using any material or resource beyond customer requirement is a waste/. Categories of waste:

  -partially done work: all team members are responsible for completion of the task assigned. Even they happen to need help in completion, the initially assigned person has to keep track of the status of the task.

  -extra processes: we follow the mandatory assignments that are release each week without trying to develop unrequested features or any other way of deviating of project course work

  -extra features: look above

  -task switching: we try to keep one person for a task, in case the task is more difficult we might assign more than one person. We do not assign a person to multiple task in case he is unable to complete at least one of them.

  -waiting: in case we need to wait for a member to complete his task we always offer help and accelerate the process.

  -motion: we meet regularly in person for our lectures and we make sure to have at least one meeting in person per week. When we are not co-located we are distributing our tasks accordingly. Person that is not present gets rather independent task and vise versa.

  -defects: we make sure to be completely informed for a task before we take over.

  -nonstandard or manual work: we make sure that nobody is using non-rebuilding servers, test environments and configuration. All our dependencies and operations are aimed to be automatic, self-services and available on demand.

  -heroics: unfortunately we cannot ensure that all of our work is happening smoothly and often we depend on each other. This is a common scenario when we have to implement operation or a feature that we are all unexperienced in and we might need another member to give us a green light - for example we have enabled revision/review of the code in our gitHub that requires at

least one more member to accept review of the code before we are able to merge. We work on a separate branch each week that is being merged before submission. In case of debug on the main (for example when implementing CI/CD frow) that required multiple PR and therefore someone else on a stand by to accept the review request (in the middle of the night).

6 References

1. Official website of Go https://go.dev/ -for installation and tutorials

2. LinkedIn learning “Learning Go” by David Gassner https://www.linkedin.com/learning/learning-go-8399317/explore-go-s-variable-types?u=55937129

3.reference for indexing the database https://github.com/dbeaver/dbeaver/wiki/Creating-Indexes