Test1 - Group Q

DevOps, Software Evolution and Software Maintenance

KSDSESM1KU

IT-University of Copenhagen

May 23, 2024

Carl Bruun	carbr@itu.dk
Christian Stender	ches@itu.dk
Nadja Brix Koch	nako@itu.dk
Theis Helth Stensgaard	thhs@itu.dk

Contents

1 Introduction **3**

2 System's Perspective **3**

2.1 Subsystem interactions . . . 4

2.2 Current state . . . 5

3 Process' Perspective **5**

3.1 CI/CD Pipeline . . . 6

3.2 System Monitoring . . . 6

3.3 Log Aggregation Techniques . . . 8

3.4 Security Assessment . . . 8

3.5 Scaling and Upgrade Strategy . . . 9

3.6 Use of AI tools . . . 10

4 Lessons Learned Perspective **10**

4.1 Challenges with Evolution and Refactoring . . . 10

4.2 Challenges with Operations . . . 11

4.3 Challenges with Maintenance . . . 11

4.4 Other Reflections on DevOps Practices . . . 11

5 Appendix **13**

A Choice of programming language . . . 13

B List of Technologies and Tools . . . 14

C Choice of CI tool . . . 15

D Static code analysis . . . 16

1 Introduction

During this semester, we have refactored and maintained the ITU-Minitwit system. As visualised in the informal context diagram in figure 1, we have used a plethora of tools and practices to do this. This report highlights our implementation, DevOps process and the challenges we have faced throughout the project.

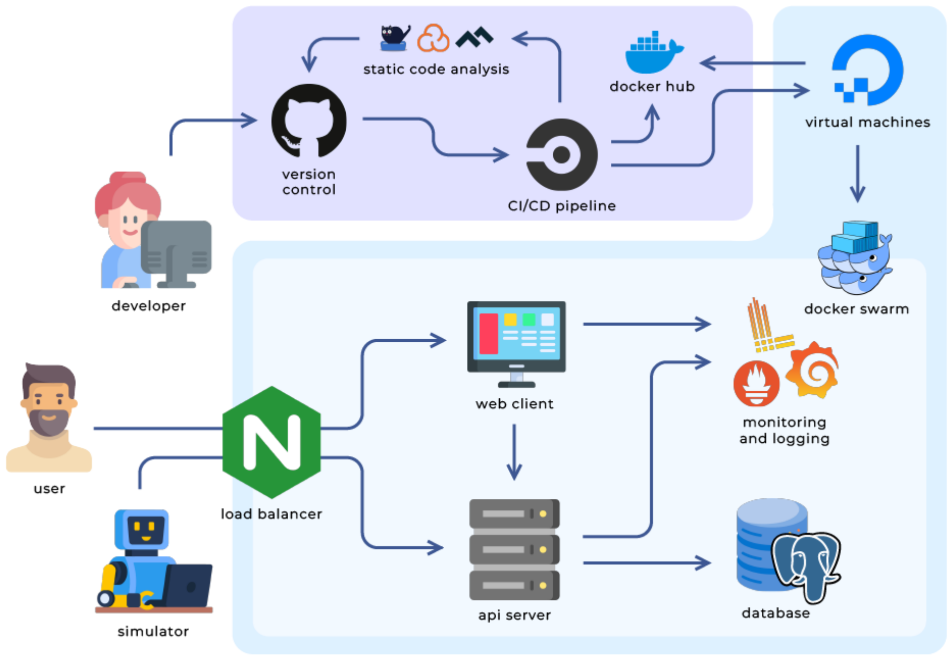


Figure 1: Informal context diagram.

2 System's Perspective

Our ITU-MiniTwit system consists of a web service and an API service which act in a client-server architecture deployed in a Docker Swarm on virtual machines on Digital Ocean. The whole system is load balanced using Nginx and data is stored in a PostgreSQL database that is deployed on Digital Ocean. The UML deployment diagram in figure 2 shows how the different components in our system are deployed.

The web client and API server are both written in Golang, using Go/Gorilla for routing and session management. Initially, we considered four different languages: Python, Crystal, Ruby and Golang. Both Crystal and Golang are statically typed and are highly performant. However, after trying out Crystal, we decided on using Golang due to its comprehensive documentation and working out of the box on Windows, Linux and MacOS. A detailed feature mapping of the system and a comparison of programming languages can be found in appendix A.

For an overview and concise description of the technologies and tools utilised throughout our project work, see appendix B. Each technology will also be introduced throughout the report.

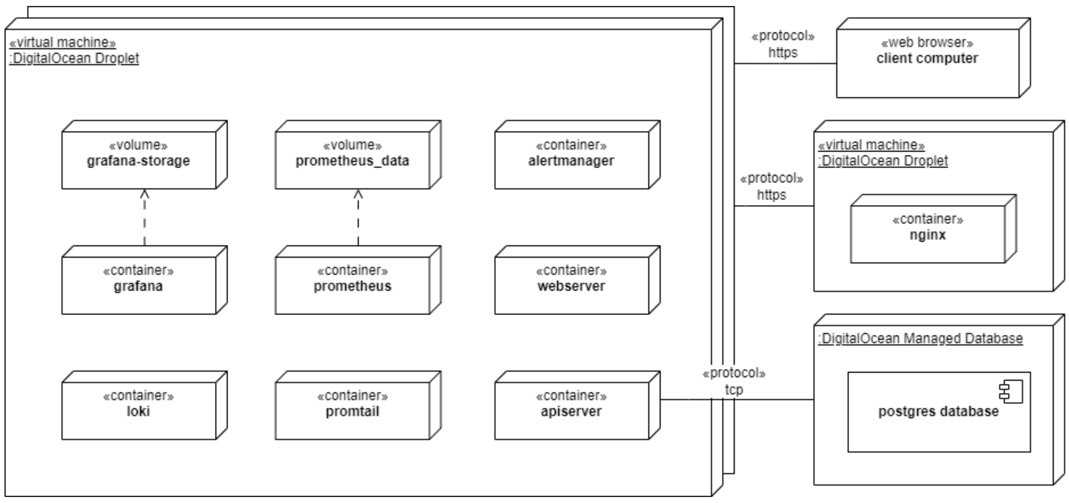


Figure 2: UML Deployment Diagram

2.1 Subsystem interactions

The interactions that a user accessing our web client and the interactions the simulator undergo are similar. Figure 3 is a sequence diagram depicting the order of operations when a user visits the web client and hits the public timeline – from Nginx to the database and back again. Similarly, figure 4 is a sequence diagram depicting the order of operations when the simulator sends a request to post a message for a given user.

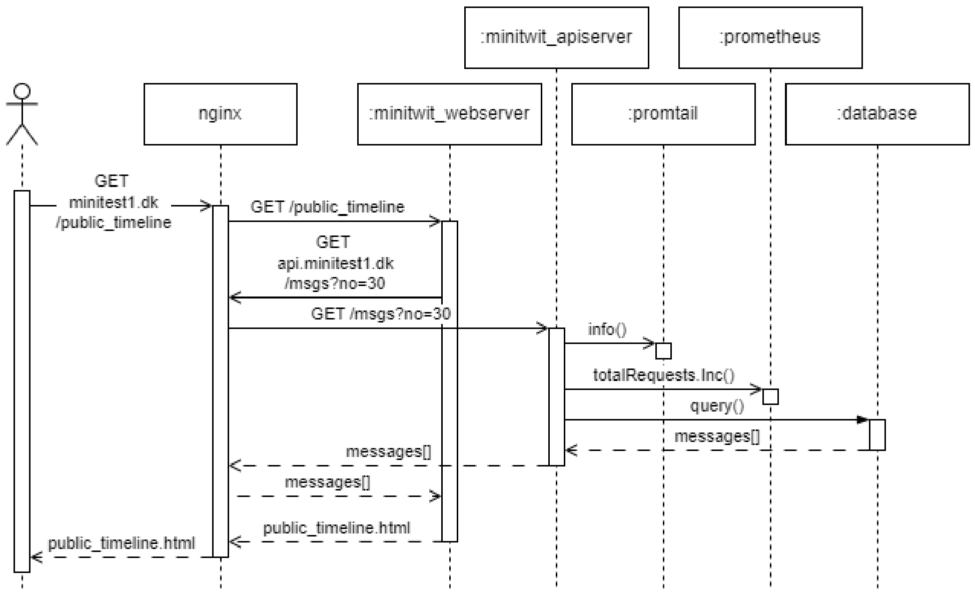


Figure 3: Sequence diagram of the user interacting with the system.

One thing to note is that we have not been stress testing the user perspective in this course as all requests have come from the simulator. This means that we do not have data to reflect on the performance of our nginx setup which results in a lot of back and forth messaging as seen in the first diagram.

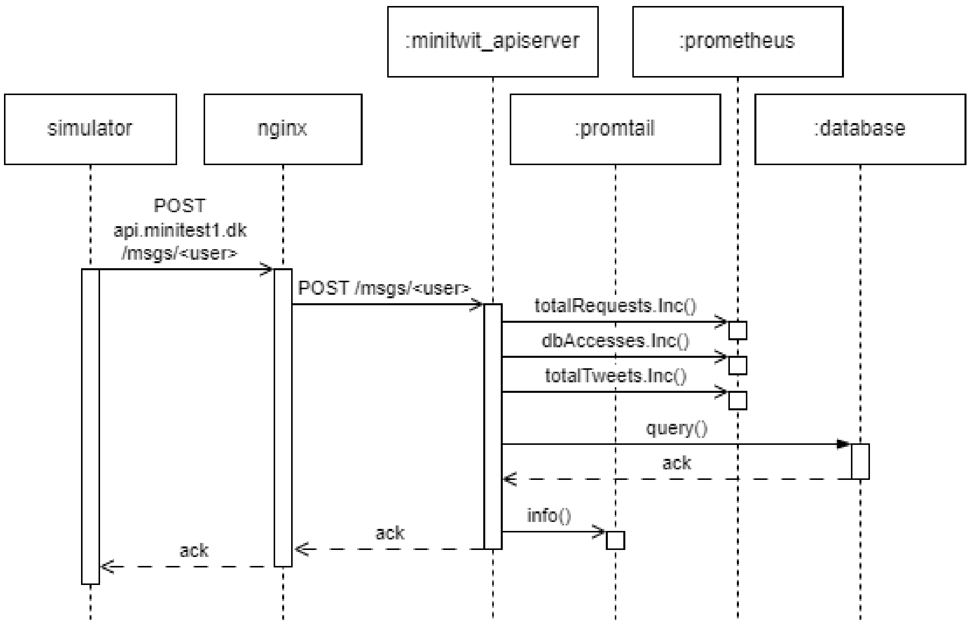


Figure 4: Sequence diagram of the simulator interacting with the system.

2.2 Current state

At the time of writing, our ITU-MiniTwit system is being evaluated using the static code analysis tools, Code Climate and SonarCloud (see appendix D). In summary, both tools reveal that while the system has a good level of maintainability and security, the following areas require improvement:

- **Code Smells**: Identified in Code Climate, these primarily consist of functions with an excessive length and functions with multiple return statements.
- **Duplications**: Identified in SonarCloud, we exceed the suggested amount of duplicated code. This is primarily duplicated code blocks between our API server and the web client, related to imported library code.
- **Test Coverage**: Implementing and configuring comprehensive test coverage to accurately reflect this metric in both Code Climate and SonarCloud.
- **Maintainability issues**: The specific issues noted in SonarCloud should be resolved according to their estimated effort to maintain and ensure a low technical debt ratio.

Addressing these areas will improve the current state of our code base and ensure long-term maintainability. During the project we have made changes based on the feedback from these tools. An example of this is the structure of the code of our web client and API server. Each component is only a single Golang package, initially written as one long file. Code Climate assessed this to be a threat to maintainability, which motivated us to refactor the code into multiple files based on their responsibilities.

3 Process' Perspective

This section provides an overview of our process, specifically detailing CI/CD chains, system monitoring, logging, security assessments, scaling and availability strategies, as well as the use of AI-assistants.

3.1 CI/CD Pipeline

Our pipeline for CI/CD is depicted in figure 5 and is managed on CircleCI.

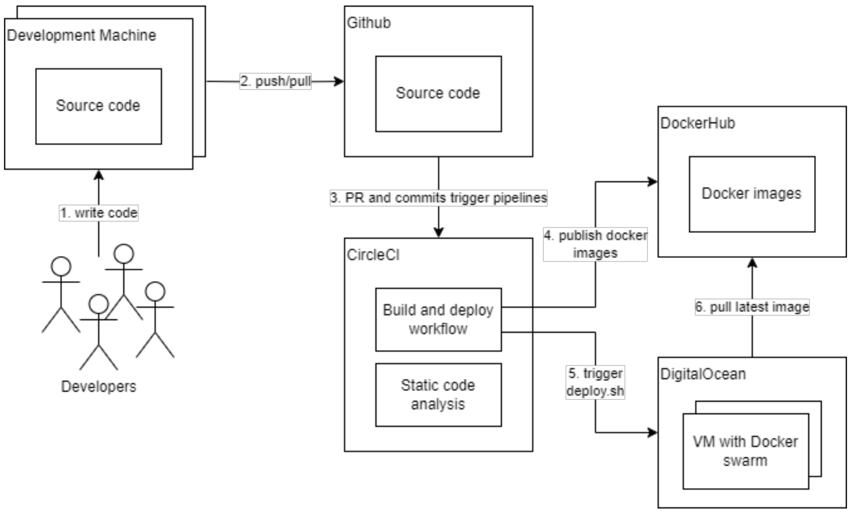
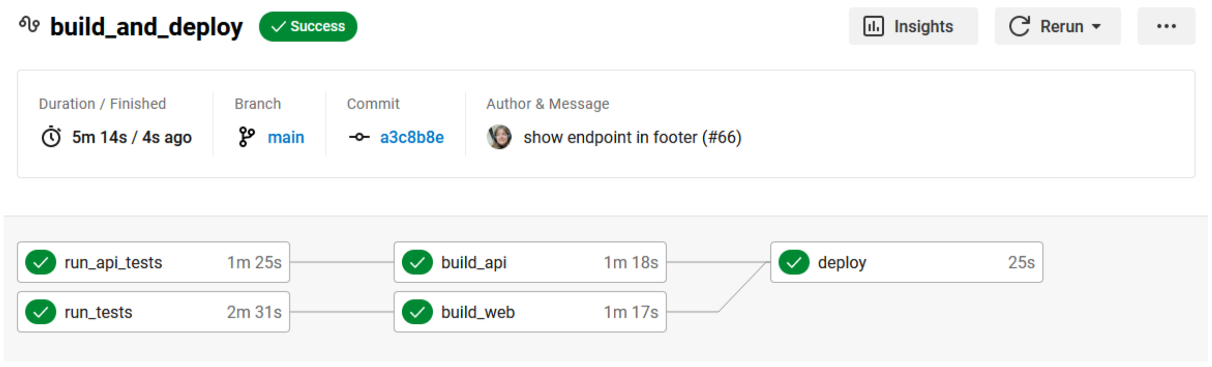


Figure 5: Visualisation of the CI/CD pipeline.

The source code on Github is linted using Hadolint and Shellcheck on each commit, and is analysed by the static code analysis tools mentioned in section 2.2 when a pull request is opened. When a pull request is merged into main, our `build_and_deploy` workflow is triggered, which can be seen in figure 6. Releases have been done manually on Github once every Thursday evening.



Duration / Finished	Branch	Commit	Author & Message
5m 14s / 4s ago	main	a3c8b8e	show endpoint in footer (#66)

Figure 6: The `build_and_deploy` workflow on CircleCI.

The decision to manage our CI/CD pipeline on CircleCI is based on the tool comparisons in appendix C. Github Actions would also have been a good choice for this, but as there were no obvious downsides to either, we went for the technology that we had not worked with before to gain experience with it.

3.2 System Monitoring

In managing our system's health, we rely on two key tools: Prometheus and Grafana. We instrument the system with the Prometheus *client_golang* library, from which we create the metrics that we monitor, register each metric using the default register of Prometheus, finally creating HTTP endpoints exposing the metrics for Prometheus to scrape [1]. Our approach is mainly whitebox monitoring in application-based metrics tracking the internal parts of the system. However, we do use a few blackbox monitoring elements by tracking specific endpoint responses.

We use Grafana as a visualisation tool that uses our Prometheus server as the data source to provide real-time insights of our measures.

As from the Monitoring Maturity Model by James Turnbull [4], our monitoring strategy is mainly composed of reactive elements such as tracking specific error codes and status of specific user requests to respond promptly to issues that arise.

Key Metrics Monitored

We have defined monitoring metrics for both the API and the web client. However, given the API's consistent workload from the simulation, this has been our primary focus. We have defined a total of 14 metrics for the API:

- **http_requests_total**
- **database_accesses_total**
- **errors_total**
- **get_user_id_requests**
- **get_user_id_requests_failed**
- **follow_requests**
- **follow_requests_failed**
- **unfollow_requests**
- **unfollow_requests_failed**
- **post_message_requests**
- **post_message_requests_failed**
- **not_found**
- **bad_request**
- **internal_server_error**

These monitoring metrics are used to create the Grafana dashboard in Figure 7.

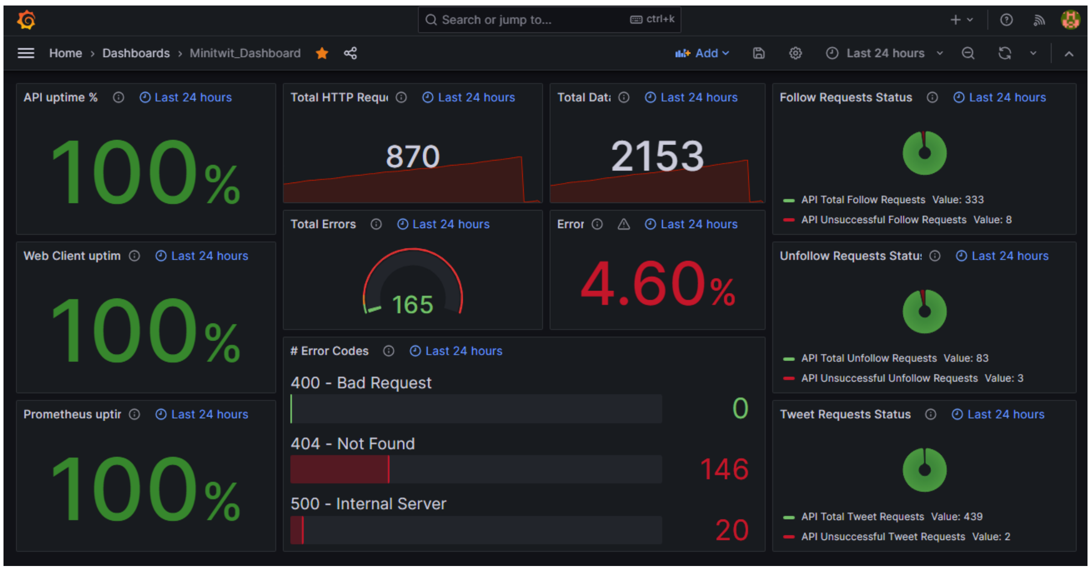


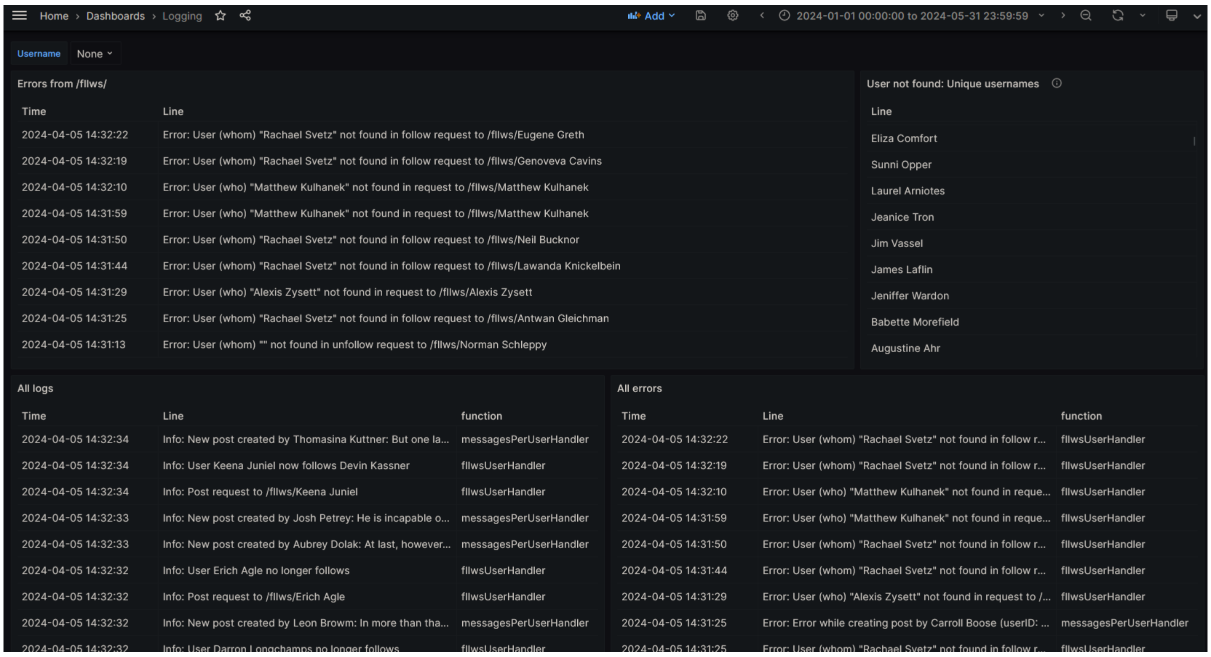
Figure 7: Minitwit_Dashboard – API Monitoring.

The dashboard includes (1) widgets for showing the uptime percentage of different parts of our system,

(2) several widgets for showing the total amount of errors and error rate and (3) the distribution of successes and failures on specific user requests.

3.3 Log Aggregation Techniques

To gain insight into how our system was being used, we have implemented a logging stack using Promtail, Loki and Grafana. The decision to not go for the otherwise popular ELK stack, is partly based on the fact that we could not get it to work, and partly based on the fact that we already used Grafana as a data visualisation tool for our monitoring setup.


Home › Dashboards › Logging
Add
2024-01-01 00:00:00 to 2024-05-31 23:59:59
Username None

Errors from /fllws/

Time	Line
2024-04-05 14:32:22	Error: User (whom) "Rachael Svetz" not found in follow request to /fllws/Eugene Greth
2024-04-05 14:32:19	Error: User (whom) "Rachael Svetz" not found in follow request to /fllws/Genoveva Cavins
2024-04-05 14:32:10	Error: User (who) "Matthew Kulhanek" not found in request to /fllws/Matthew Kulhanek
2024-04-05 14:31:59	Error: User (who) "Matthew Kulhanek" not found in request to /fllws/Matthew Kulhanek
2024-04-05 14:31:50	Error: User (whom) "Rachael Svetz" not found in follow request to /fllws/Neil Bucknor
2024-04-05 14:31:44	Error: User (whom) "Rachael Svetz" not found in follow request to /fllws/Lawanda Knickelbein
2024-04-05 14:31:29	Error: User (who) "Alexis Zysett" not found in request to /fllws/Alexis Zysett
2024-04-05 14:31:25	Error: User (whom) "Rachael Svetz" not found in follow request to /fllws/Antwan Gleichman
2024-04-05 14:31:13	Error: User (whom) "" not found in unfollow request to /fllws/Norman Schleppy

User not found: Unique usernames

Line
Eliza Comfort
Sunni Opper
Laurel Arniotes
Jeanice Tron
Jim Vassel
James Laflin
Jeniffer Wardon
Babette Morefield
Augustine Ahr

All logs

Time	Line	function
2024-04-05 14:32:34	Info: New post created by Thomasina Kuttner: But one la...	messagesPerUserHandler
2024-04-05 14:32:34	Info: User Keena Juniel now follows Devin Kassner	fllwsUserHandler
2024-04-05 14:32:34	Info: Post request to /fllws/Keena Juniel	fllwsUserHandler
2024-04-05 14:32:33	Info: New post created by Josh Petrey: He is incapable o...	messagesPerUserHandler
2024-04-05 14:32:33	Info: New post created by Aubrey Dolak: At last, however...	messagesPerUserHandler
2024-04-05 14:32:32	Info: User Erich Agle no longer follows	fllwsUserHandler
2024-04-05 14:32:32	Info: Post request to /fllws/Erich Agle	fllwsUserHandler
2024-04-05 14:32:32	Info: New post created by Leon Browm: In more than tha...	messagesPerUserHandler
2024-04-05 14:32:32	Info: User Darron Longchamps no longer follows	fllwsUserHandler

All errors

Time	Line	function
2024-04-05 14:32:22	Error: User (whom) "Rachael Svetz" not found in follow r...	fllwsUserHandler
2024-04-05 14:32:19	Error: User (whom) "Rachael Svetz" not found in follow r...	fllwsUserHandler
2024-04-05 14:32:10	Error: User (who) "Matthew Kulhanek" not found in reque...	fllwsUserHandler
2024-04-05 14:31:59	Error: User (who) "Matthew Kulhanek" not found in reque...	fllwsUserHandler
2024-04-05 14:31:50	Error: User (whom) "Rachael Svetz" not found in follow r...	fllwsUserHandler
2024-04-05 14:31:44	Error: User (whom) "Rachael Svetz" not found in follow r...	fllwsUserHandler
2024-04-05 14:31:29	Error: User (who) "Alexis Zysett" not found in request to /...	fllwsUserHandler
2024-04-05 14:31:25	Error: Error while creating post by Carroll Boose (userID: ...	messagesPerUserHandler
2024-04-05 14:31:25	Error: User (whom) "Rachael Svetz" not found in follow r...	fllwsUserHandler

Figure 8: Grafana dashboard of logs.

Our strategy for logging was to use the *information* level to log the beginning of every function call and when alterations to the database were successfully made, and to use the *error* level to log when errors occured.

As we had a high amount of ”User not found” errors on our `/fllws` endpoint, we created a widget on our dashboard that showed all unique usernames that were not found in the database. This enabled us to check whether these users actually did not exist. The setup of our dashboard can be seen in figure 8.

3.4 Security Assessment

In our efforts to ensure the security of the ITU-MiniTwit system, we have utilised Snyk, Metasploit and SonarCloud, however, these tools have not identified any immediate vulnerabilities. To further assess the security state of our system we have identified potential threats and performed a risk analysis.

Risk Identification

Our primary asset is the ITU-MiniTwit web application, which includes web client, API server, and a database. One of the most common vulnerabilities for web applications involves injecting malicious

SQL statements. Other threats consists of cross-site scripting (XSS) or misconfiguration, e.g. incorrectly configured servers, databases or applications. From this we can construct several risk scenarios:

- **Scenario 1**: An attacker performs SQL injection on the web application to download sensitive user data.
- **Scenario 2**: An attacker exploits a cross-site scripting vulnerability to steal session cookies and impersonate users.
- **Scenario 3**: An attacker exploits a misconfiguration of the web client allowing unauthorised access to system resources.
- **Scenario 4**: A developer with legitimate access exposes sensitive data or sabotages the system, either intentionally or accidentally.

Risk Analysis

To analyze the identified risks we constructed the risk matrix in table 1.

Risk Scenario	Likelihood	Impact	Priority
SQL Injection	Low	High	Medium
XSS	Low	Medium	Low
Misconfiguration	Medium	High	High
Developer error	Low	High	Medium

Table 1: Risk matrix based on identified risks.

To mitigate the risk scenarios the following strategies are employed:

- **Static code analysis**: The likelihood of our risk scenarios is covered in part by SonarCloud that has security rules setup, which analyses the source code and allows detecting SQL Injection and XSS.
- **Secrets**: To minimize the risk of exposing sensitive data, we use secrets in Docker swarm, GitHub and CircleCI.
- **Automation**: Automatic configuration e.g. using tools like Vagrant ensures we avoid misconfigurations.

Security Hardening

For further security hardening we employ the HTTPS protocol. Our HTTPS certificate was obtained through CertBot. However, due to difficulties with the standard HTTP validation method, especially concerning load balancers as described in [3] we opted for an alternative approach. We used the acme-dns-certbot tool [2], which interfaces CertBot with a third-party DNS server for validation instead of HTTP. This method is especially beneficial for issuing certificates for individual servers behind a load balancer, where traditional HTTP certificate validation would require setting validation files on each server.

3.5 Scaling and Upgrade Strategy

Our monitoring dashboard detected several API crashes on our Digital Ocean node due to 100% CPU usage. To improve service reliability and eliminate the single point of failure, we implemented scaling and replication using Docker Swarm.

Docker Swarm

We chose Docker Swarm as our container orchestration platform because it is included by default with Docker and is simpler to use compared to alternatives like Kubernetes and OpenShift. Docker Swarm allowed us to set up a cluster consisting of manager and worker nodes, effectively creating a distributed system. In this setup, worker nodes run our stateful services, API and web client, independently, while the manager node handles service scheduling across the nodes using the routing mesh, Docker Swarm's built-in load balancing mechanism, and maintains the swarm state.

We first deployed the Docker Swarm as a separate cluster beside our single-node production server in Digital Ocean to ensure a transition that would not disrupt our live server. For each service, including our API, we added an additional replica in order to distribute the workload across multiple containers.

However, the inherent load balancer of Docker Swarm, the routing mesh, is rather low-level, and from-the-shelf it only redirects traffic to replicas when a node is "unhealthy" or dead. Therefore, we opted for a dedicated load balancer to complement the routing mesh by employing a more sophisticated traffic distribution. Simply redirecting traffic when a node is crashed that initiated the whole focus on scaling and replication.

Nginx

We use Nginx as both reverse proxy and load balancer. This involved adding a new virtual machine in our Digital Ocean project outside the Docker Swarm to run an instance of Nginx, which we configured to handle a more sophisticated traffic distribution between the replicated services.

To configure Nginx, we separated the configuration files into two parts, one for the API and one for the web client. The configuration file for the API includes an upstream block that lists the IP addresses of the replicas in the swarm cluster along the API ports, allowing Nginx to direct traffic to the specified instances. Additionally we added server blocks for HTTP and HTTPS traffic, effectively redirecting all HTTP traffic to HTTPS. Similarly, the configuration file for the web client follows the same pattern but includes the IP addresses and service ports of the web client in the upstream block instead.

3.6 Use of AI tools

To automate the production of easy or recurring code such as a new Prometheus monitoring metric, generating skeleton code, code completion and helping us gaining a high-level understanding of the ITU-MiniTwit system, GitHub Copilot was used by multiple developers. However, it was quite bad at guessing the intended implementation logic of new features, and therefore served no purpose in the development in these cases.

4 Lessons Learned Perspective

4.1 Challenges with Evolution and Refactoring

We experienced that when adding new technologies to our system, other parts of our system stopped working. When parts of our system failed, we tried getting them to work again by finding guides online that was specifically targeted at making the newly added technology and the existing 'broken' technology work together. This has taught us the importance of modular solutions that are open for extension and are easily refactored. One of the agile principles state that "Working software is the primary measure of progress“, and in line with this, we have truly experienced in this project how broken software halts progress.

4.2 Challenges with Operations

A challenge with our CI/CD management tool was the inability to run our automated tests until a pull request was merged to our main branch.

Running tests locally was a lengthy process as it required several docker containers being built and started. As a result, we sometimes approved pull requests with failing code. This resulted in situations such as commit `7661b99`[1] and `8071acb`[2], requiring a new branch and pull request to salvage the main branch. To solve this, we considered setting up Github Actions with the same tests as in our CircleCI workflow, but decided it was redundant. However, we have learned that if running tests locally is too difficult, it will not be done, highlighting the importance of automated tests.
Moreover, we found that pushing failing code to main is not as dangerous as it sounds, highlighting the importance of a fault tolerant CI/CD pipeline. Additionally, having the code on Github decoupled from the code on our VMs makes it safe to push code to our main branch. Ultimately this has allowed us to continuously deliver and deploy, which again is in line with the agile principles[3].

4.3 Challenges with Maintenance

Early in the project, monitoring was limited to the official status-page[4], where it was easy to see if *something* was wrong, but not what it was. This is where logging becomes a powerful tool. Our logging stack consisting of Promtail, Loki and Grafana allowed us to see that there were a lot of usernames that were not able to be found. However, when we started to work with Docker Swarm, our stack failed, and again it left us completely blind to the cause of the errors we experienced. In the end, it meant that one of our Github issues[5], which was identified early in our process, was never solved and closed.

4.4 Other Reflections on DevOps Practices

We created a DevOps anti-pattern where all changes to the virtual machine were made manually via SSH, without version control and traceability when making configuration changes later in the project. This led to an erosion of our automated cloud provisioning, deviating more and more from our initial setup, making it increasingly difficult and time-consuming to maintain. Initially we had used Vagrant to provision our single-node cloud infrastructure, but did not maintain it throughout the project. Ultimately, this led to more time spent on fixing human errors and configuring new virtual machines. Additionally, if the infrastructure were to crash, it would take days to reproduce the current state of the service.

Besides including further virtual machine configuration to our automated deployment, it would have been beneficial to include a security aspect to our pipeline, namely snyk and wmap to provide reactive identification of vulnerabilities.

[1] `https://github.com/TheisHS/test1-itu-minitwit/commit/7661b997afbc0de3742e7df066b37245ef7f18bd`
[2] `https://github.com/TheisHS/test1-itu-minitwit/commit/8071acb95fa1010bb102b45b32634bb6b6790e70`
[3] ”Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale“.
[4] `http://206.81.24.116/status.html`
[5] `https://github.com/TheisHS/test1-itu-minitwit/issues/41`

References

[1] The Prometheus Authors. *Prometheus Go Client Library*. Accessed: 2024-05-19. 2024. URL: https://github.com/prometheus/client_golang.

[2] Joohoi. *acme-dns-certbot-joohoi*. Accessed: 2024-05-19. 2019. URL: https://github.com/joohoi/acme-dns-certbot-joohoi.

[3] Jamie Scaife and Kathryn Hancox. *How To Acquire a Let's Encrypt Certificate Using DNS Validation with acme-dns-certbot on Ubuntu 18.04*. Accessed: 2024-05-19. 2020. URL: https://www.digitalocean.com/community/tutorials/how-to-acquire-a-let-s-encrypt-certificate-using-dns-validation-with-acme-dns-certbot-on-ubuntu-18-04.

[4] James Turnbull. *A Monitoring Maturity Model*. 2022. URL: https://www.kartar.net/posts/a-monitoring-maturity-model.markdown/ (visited on 05/18/2024).

5 Appendix

A Choice of programming language

Feature Category	**Description**
Database Operations	Connect to the database
	Create the database tables
	Query the database
	Get the user id
	Format a timestamp for display
	Get a gravatar image for an email
Request Handling	Before a request, it connects to the DB and gets the user, and after a request it closes the connection to the DB.
UI	A page showing the public timeline, including all public messages.
	A page showing the timeline for the logged-in user. If no user is logged in, redirect to the public timeline. The private timeline includes all public messages, and the messages from the people the logged-in user follows.
	A page for user profiles. If the user is followed, display a different message (”You follow this user”/”You don’t follow ...”)
Routes	A route to follow other users
	A route to unfollow other users
	A route for adding messages
	A route to register
	A route to login
	A route to logout

Table 2: Feature mapping of ITU Minitwit

	Python3 Flask	Crystal Kemal	Ruby Sinatra	Golang
Our experience	Moderate experience	No experience	No experience	Moderate experience
Types	Dynamically typed	Statically typed	Dynamically typed	Statically typed
Performance	Moderate performance	High performance	Moderate performance	High performance
SQLite support	yes	yes	yes	yes
Middleware support*	yes	yes	yes	yes
Release date	2010	2016	2007	2011
Deployment	Deployed using virtual environments to manage dependencies.	Compiled into a single binary executable with all its dependencies.	Requires the presence of the Ruby runtime environment and dependencies.	Can be deployed as single binary executables.
Documentation	-	Insufficient	-	Comprehensive
Operation System	Windows and Unix based	Unix based, but Windows is ”preview”**	Windows and Unix based	Windows and Unix based

Table 3: Programming language strengths

* Middleware in web APIs is used as a design pattern to intercept and manipulate HTTP requests. `https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-middleware`

** `https://github.com/crystal-lang/crystal/issues/5430`

B List of Technologies and Tools

- **Go**: A statically typed, compiled programming often used for backend development.
- **Go/Gorilla**: A toolkit for Golang, that provides packages for building web applications. Used for routing and session management.
- **SQLite**: A lightweight SQL database engine that is ideal for embedded database applications. Used in initial setup and for testing.
- **Docker**: A platform for shipping and running applications, ensuring consistency across different environments. Containerization is used deliberately throughout our system and in delivery.
- **Docker Swarm**: A container orchestration tool, included in Docker by default, to manage a cluster of Docker nodes as a single virtual system.
- **DigitalOcean**: A cloud infrastructure provider offering scalable compute and storage solutions. Used for deploying and managing the application and database.
- **Prometheus**: An open-source monitoring toolkit used for monitoring metrics in our cloud environments.
- **Grafana**: An open-source analytics and monitoring platform that integrates various data sources to visualize our data and metrics.
- **PostgreSQL**: A open-source object-relational database system, known for extensibility and speed. Used after the transition from SQLite managed by DigitalOcean.
- **Promtail**: An agent that ships log file to Loki and part of the Grafana logging stack. Used for gathering logs.
- **Loki**: A log aggregation system designed to be scalable. Used with Grafana for log querying and visualization.
- **Nginx**: A high-performance web server and reverse proxy server. Used for load balancing.
- **CertBot**: A open-source tool for automatically using Let's Encrypt certificates to enable HTTPS on web servers.
- **CircleCI**: A continuous integration and delivery platform that automates the build, test and deployment processes of our project.
- **CodeClimate**: A platform that provides code review, offering insights into code quality, maintainability and test coverage, helping us ensure high standards and to improve code health.
- **Github**: Version Control Management
- **Hadolint**: A lint-tool that analyzes Dockerfiles to find common issues.
- **Shellcheck**: A shell script analysis tool that identifies syntax errors and common issues.
- **Sonarcloud**: A cloud-based code quality and security service that analyzes code to detect bugs and vulnerabilities.
- **Vagrant**: A tool for building and managing virtualized development environments. Used initially for VM instantiating.

C Choice of CI tool

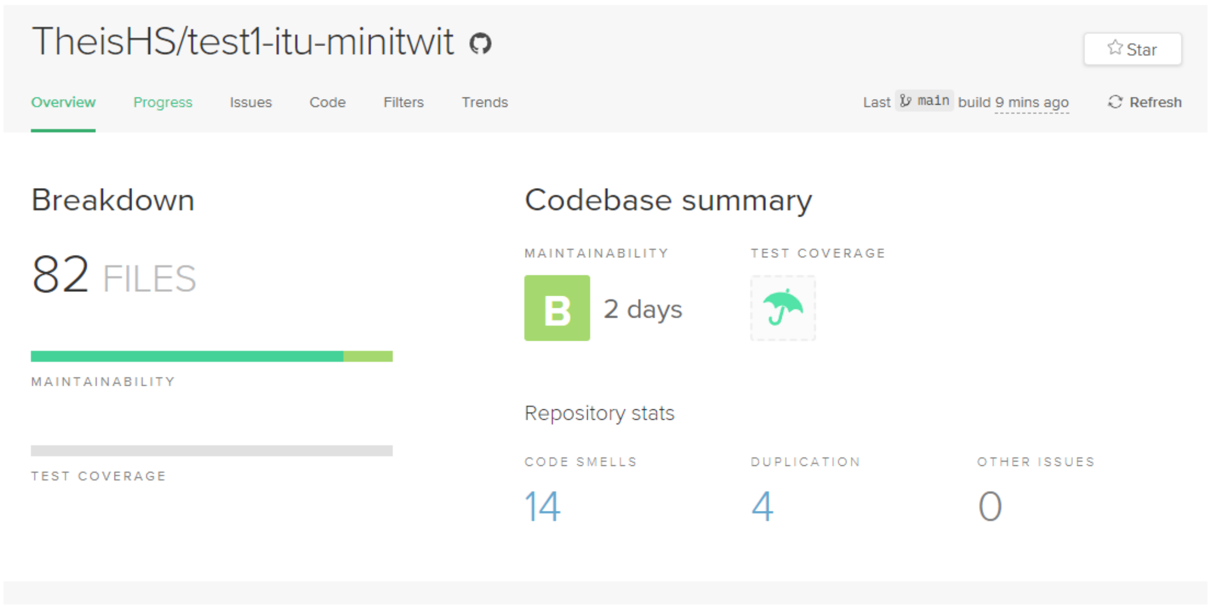
As our code is stored on Github, we've eliminated Gitlab CI as as an option. Travis CI is only free for a single month (for students), and we've also eliminated this.

Github Actions	CircleCI	Our considerations
Is free ... "Cheapest for people with public repositories"	3000 minutes for free pr month	We have a public repo.
Runs full pipeline automatically	Can be paused and wait for human interaction	We don't have a usecase for needing human intervention before deploying if the code passes all the tests we stup, and the CircleCI feature (even though nice) is not needed.
More than CI/CD - can also automate manual tasks like generating changelogs or versioning releases	Only CI/CD, but specialised in this.	We only need CI/CD for now.
Slower than CircleCI	Faster than Github Actions	Do we need speed?
Only Windows, MacOS and Linux	Every operating system	We only need Linux
Configuration can be split in mulitple files	Single file configuration	Cleaner setup with GHA?
Docker support is still a bit buggy on GHA, and works only with Linux.	CircleCI has perfected its Docker support over the years to make it (almost) the de-facto environment for running builds.	We will use Docker, but still only Linux.
More granular control by exposing all commands. Complexity increase.	Less complex, has built in commands for often-used services. Less control.	We don't know what we need yet - so maybe more control is nice, but it being easy is also nice.

Table 4: CI tool strengths.

D Static code analysis

Here we provide a snapshot of our static code analysis tools from May 19, 2024.


TheisHS/test1-itu-minitwit

Star

Overview Progress Issues Code Filters Trends

Last main build 9 mins ago Refresh

Breakdown

82 FILES

MAINTAINABILITY

TEST COVERAGE

Codebase summary

MAINTAINABILITY B 2 days

TEST COVERAGE

Repository stats

CODE SMELLS	DUPLICATION	OTHER ISSUES
14	4	0

Figure 9: Code Climate summary 19/05/2024.

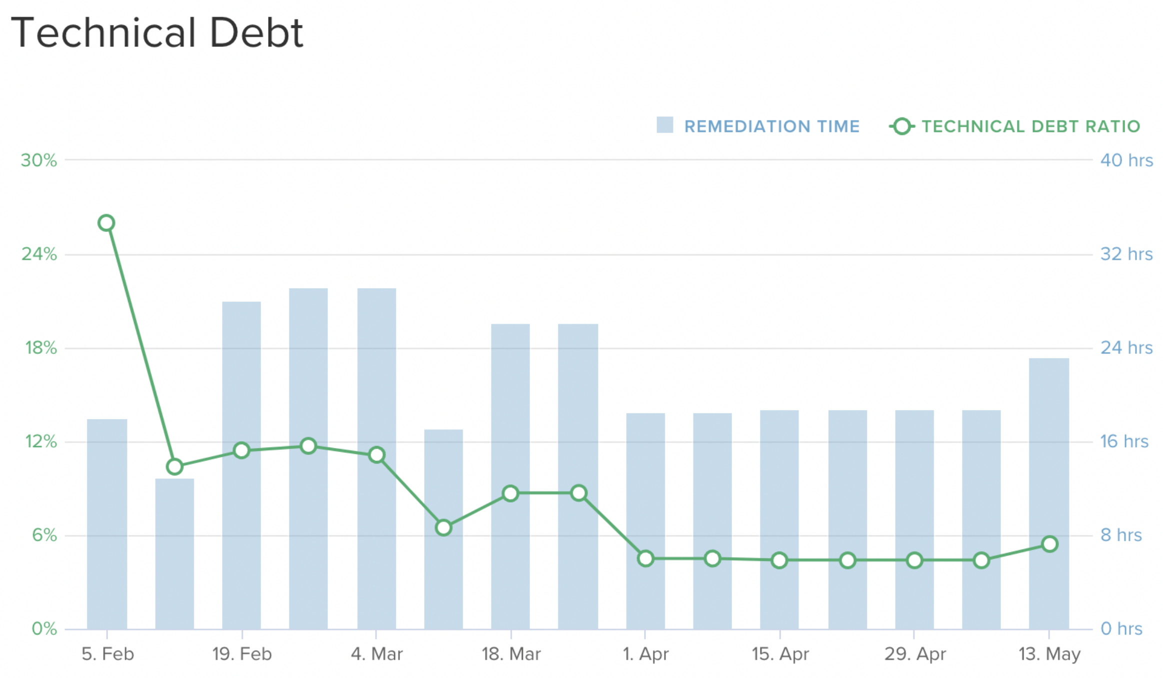
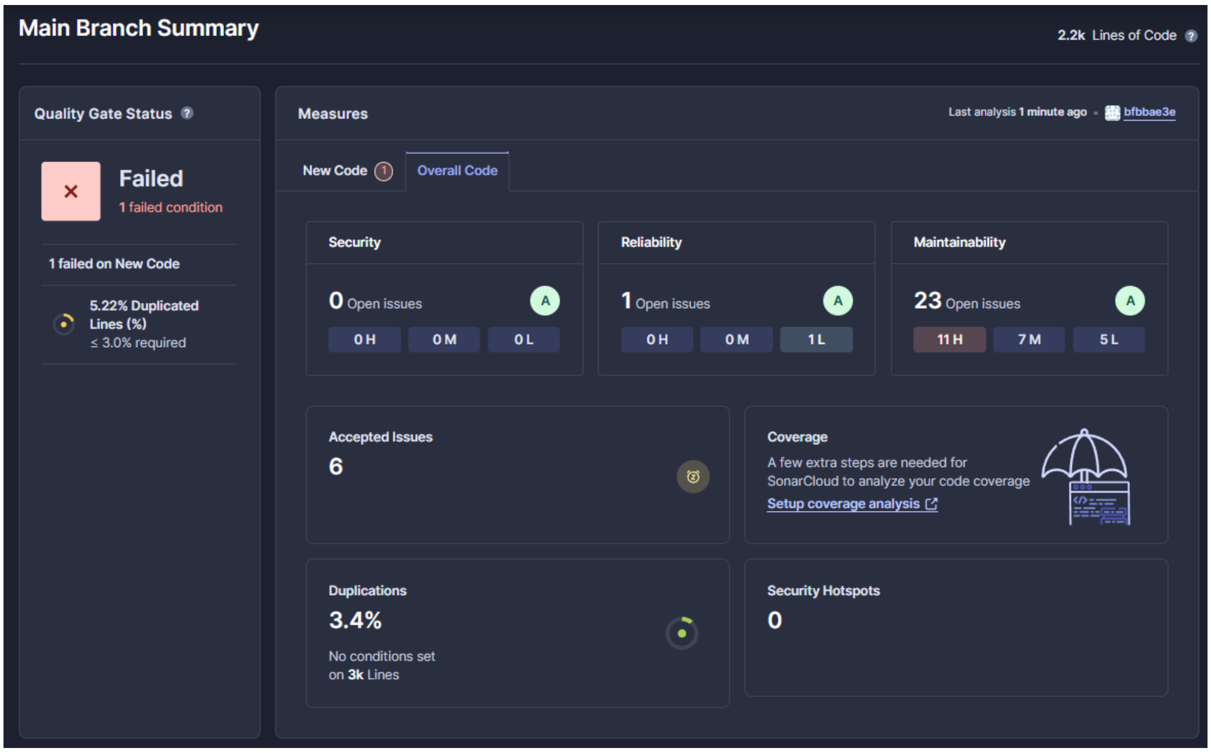


Figure 10: Code Climate Technical Debt Trends 19/05/2024.


Main Branch Summary

2.2k Lines of Code

Quality Gate Status

Failed
1 failed condition

1 failed on New Code

5.22% Duplicated Lines (%)
≤ 3.0% required

Measures

Last analysis 1 minute ago · bfbbae3e

New Code | Overall Code

Security
0 Open issues A
0 H | 0 M | 0 L

Reliability
1 Open issues A
0 H | 0 M | 1 L

Maintainability
23 Open issues A
11 H | 7 M | 5 L

Accepted Issues
6

Coverage
A few extra steps are needed for SonarCloud to analyze your code coverage
Setup coverage analysis

Duplications
3.4%
No conditions set on 3k Lines

Security Hotspots
0

Figure 11: SonarCloud summary 19/05/2024.