MSc_group_c

ITU MiniTwit Report Skeleton

May 2024

1 Introduction

2 System perspective

3 Process Perspective

3.1 CI/CD (GitHub Actions)

GitHub Actions was chosen based on its simplicity, familiarity, and free pricing [@githubactions_vs_jenkins], [@20_cicd_comparison]. A motivating factor, was the suite of services supported natively in Github, of these a few were utilized:

- GitHub Action Secrets & Variables for storing ssh-keys, passwords, etc.
- GitHub Tags, Releases & Artifacts Storage for artifact versioning of the GoLang application.
- GitHub Applications for code quality evaluations with CodeClimate, SonarQubeCloud, and qtlysh.
- GitHub Projects, Tasks & Backlog for managing task formulation and distribution.

3.1.1 CI/CD Pipelines

A total of **7** pipelines are established, these are:

Table 1: List of GitHub Actions workflows employed.

File	Purpose	Invoked on
`continous-development.yml`	Primary CI/CD flow against PROD	Pushing `main`
`codeql.yml`	Analyzes go source code using `CodeQL`	Push & PRs to `main`.
`generate-report.yml`	Generates `report.pdf` from files in `/report/*`	Push to `/report/*`

File	Purpose	Invoked on
`linter-workflow.yml`	Runs golangci-lint on go source code.	Push `main` or any PR
`pull-request-tests.yml`	Runs python tests.	All PRs
`test-deployment.yml`	Secondary CI/CD flow against TEST.	Tag `test-env*`
`sonarcube_analysis.yml`	Analyses go source code using SonarCloud.	PRs to `main`

3.1.2 CI/CD Specific Technologies

- The `golangci-lint` linter is implemented in `linter-workflow.yml` (see tasks #119 and #129)
- The `pandoc` library is used to generate laTeX reports from markdown in `generate_report.yml`
- The `CodeQL` code analysis engine is used in `codeql.yml` to check for security vulnerabilities.
- Original `pytest` files are used in `continous-development.yml`–now functioning as a `Test` stage (see `minitwit_tests.py` and `sim_api_test.py`).

3.1.3 Choice of CI/CD

- Since GitHub was chosen, GitLab CI/CD and BitBucket Pipelines were discarded, as they are specific to alternative git repository management sites.
- Commercial automation tools such as Azure DevOps and TeamCity were discarded due to the pricing and limitations of their free plans.

As such, the choice was between GitHub's native GitHub Actions or a CI/CD system agnostic to repository management sites.

It was decided that time-to-production, in the case of establishing working CI/CD pipelines, was the biggest priority. As an alternative, the self-hosted automation system Jenkins was considered, but the perceived learning curve along with the self-hosted infrastructure setup [@20_cicd_comparison] dissuaded it as the choice of CI/CD system.

Table 2: Comparison between CI/CD systems.

CI/CD Tool / Platform	**GitHub Actions**	**Jenkins**	**Azure DevOps**	**TeamCity (JetBrains)**
Ease-of-use	Simple [@githubactions_vs_jenkins]	Medium [@githubactions_vs_jenkins]	*Undetermined*	*Undetermined*

CI/CD Tool / Platform	GitHub Actions	Jenkins	Azure DevOps	TeamCity (JetBrains)
Version Control	Native GitHub Integration [@20_cicd_comparison]	Agnostic [@20_cicd_comparison]	Agnostic [@20_cicd_comparison]	Agnostic [@20_cicd_comparison]
Hosting	Primarily cloud-based [@20_cicd_comparison]	Self-hosted [@20_cicd_comparison]	Cloud-based [@20_cicd_comparison]	Cloud-based or self-hosted [@20_cicd_comparison]
Pricing Model	Free for public repositories, tiered for private [@20_cicd_comparison]	Open-source (MIT License), only cost is for hosting [@20_cicd_comparison]	Commercial with a limited free tier [@20_cicd_comparison]	Commercial [@20_cicd_comparison]

3.2 Monitoring

- We use Prometheus as an Echo middleware, with additional custom made metrics to scrape our application every 5 seconds.
  - Custom metrics:
    - User follower (gauge)
    - User followees (gauge)
    - VM CPU usage (gauge)
    - Messages posted (by time) (counter)
    - Messages posted (by user) (gauge)
    - Mesages flagged (by user) (gauge)
    - New user (counter)
    - Total users (gauge)
  - Prometheus was chosen on the background of:
    - Demonstrated in Class
    - Easy integration with golang/echo via. middleware
    - Wide spread usage and easy to integrate with e.g. Grafana
    - Free to use
- Grafana
  - As of writing this, the dashboards does not work due to swarm scaling. All pictures are from the day of the simulator stopping.
  - Users:
    - Admin user with password shared with the group.
    - Helge and Mircea specific login as described on Teams.
  - Was chosen on the background of:
    - Demonstrated in Class
    - Rich Visualization
    - Free to use

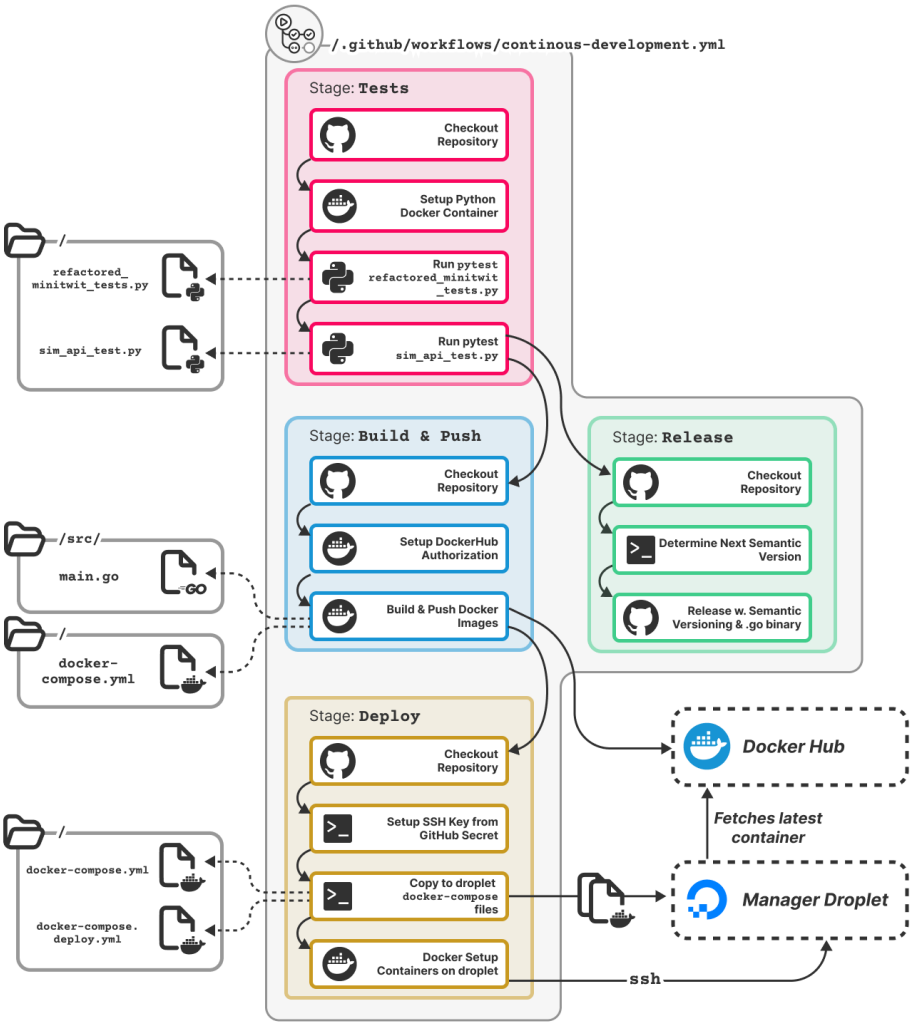


Figure 1: Informal visualization of `continous-development.yml` (primary pipline), with stages `Tests`, `Build & Push`, `Release`, and `Deploy`

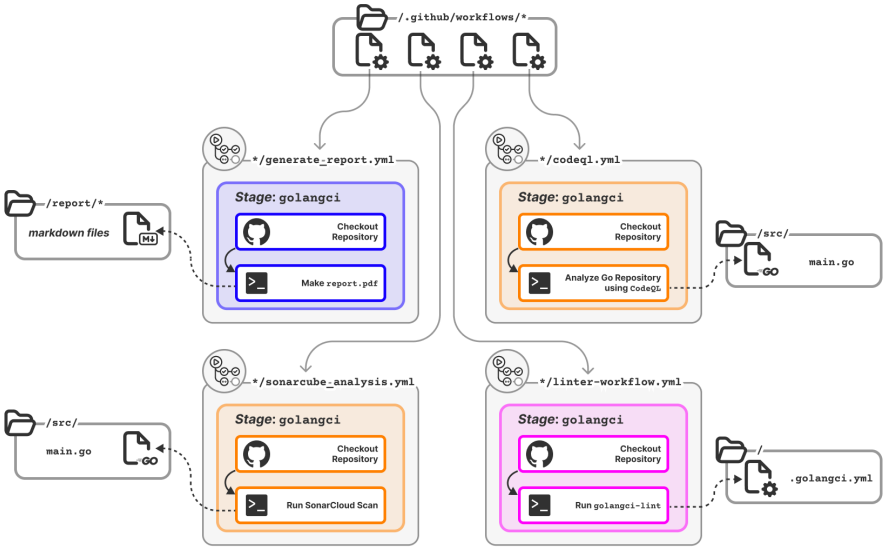


Figure 2: Informal visualization of other pipelines

3.2.1 Grafana Dashboards

Whitebox Request and response monitoring dashboard:

Timeframe: last 30 minutes:

Timeframe: Last 2 days:

Whitebox User action dashboards monitoring:

Timeframe: Last 7 days:

Whitebox Virtual memory dashboard monitoring:

Timframe: last 5 minutes:

3.2.2 Black box monitoring

Black box user side error monitoring was given by the Helge and Mircea in form of the Status and Simulator API errors graf. We were encouraged to just use this as our client side error monitoring.

3.2.3 DigitalOcean monitoring

DigitalOcean provides some monitoring capabilities (Bandwidth, CPU usage, and Disk I/O). This did help to identify an attack. More on that [Insert refrence here]

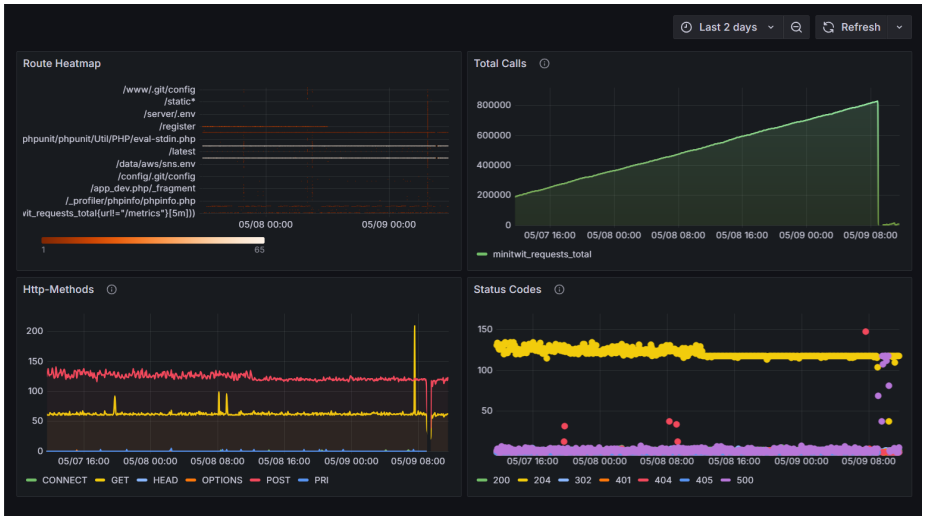


Figure 3: Request and response dashboard last 30 minutes

Figure 4: Request and response dashboardLast 2 days

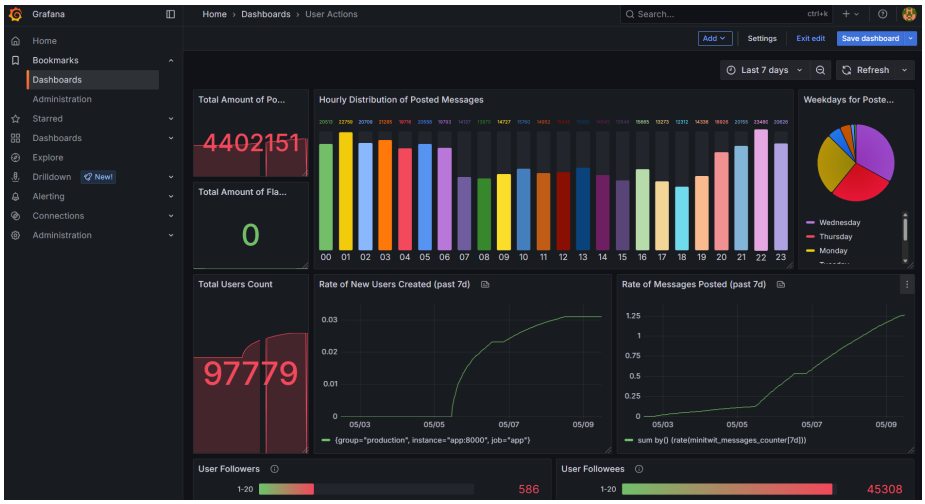


Figure 5: User action dashboards Last 7 days

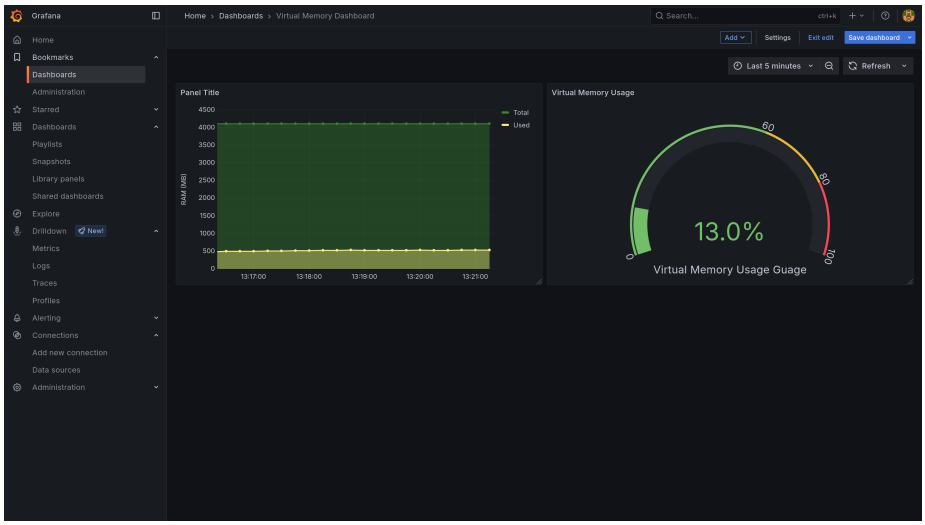
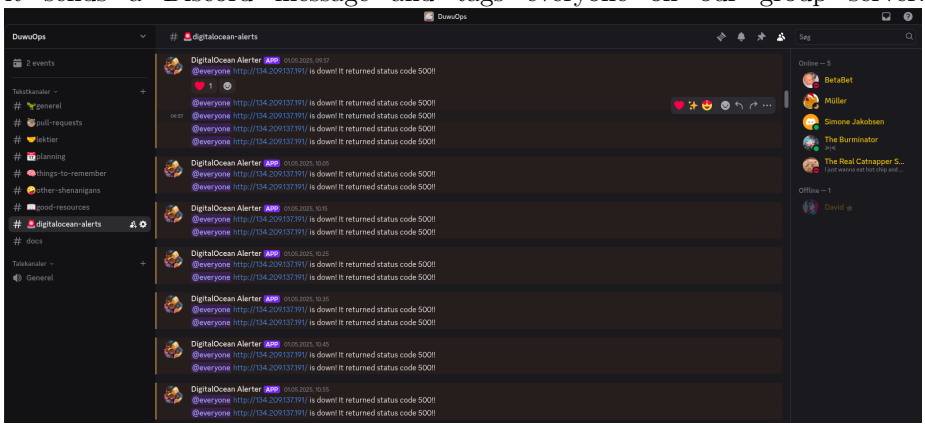


Figure 6: Virtual Memory dashbord Last 5 minutes

3.2.4 Alert System

An alert system was set up via a Discord bot that on the server via a cronjob that checks every 5 miniutes. If the application is not up it sends a Discord message and tags everyone on our group server.



3.3 Logging

- The ELK method was implemented but ultimatly scraped in favor of using loki/alloy that intergrate with Grafana which gather our logging and monitoring the same place.
- Practical Principles:
  - TODO: A process should not worry about storage
  - TODO: A process should log only what is necessary
  - TODO: Logging should be done at the proper level: Mention emoji use
  - Logs should be centralised: All logs can be found via Grafana->Drilldown->Logs

3.4 Strategy for scaling and upgrades

- We used docker swarm with docker stack so that we can leverage the docker compose setup that was already made. Some changes were made to accommodate the swarm set up. These are: a network overlay, setting how many replicas per service, where nesecary setting where the service should be placed, and update confirguration.
- The update config for the minitwit application is set so that it updates one at a time. This is set as we only have two instances of minitwit and if an update fails we don't want more than one instance to be down. On failure we do a rollback.
- We do rolling updates as this is nativly supported on docker swarm.

3.5 AI use

Throughout the development process, all team members leveraged artificial intelligence tools to varying degrees and for diverse applications. The primary AI systems employed included ChatGPT, Claude, DeepSeek, and GitHub Copilot. Team members provided contextual information regarding code issues or implementation challenges, utilizing AI-generated responses as foundational guidance for problem-solving methodologies rather than direct solution implementation. This methodology facilitated the identification of potential problem domains and remediation strategies while preserving critical assessment of AI-derived recommendations. In accordance with transparency requirements, AI tools have been formally acknowledged as co-authors in relevant version control commits where their contributions influenced the development process. (This paragraf was written using AI lol)