DevOps, Software Evolution and Software Maintenance

COURSE CODE: BSDSESM1KU

BACHELOR IN SOFTWARE DEVELOPMENT

DevOps: ITU-MiniTwit

GROUP R — RHODODEVDRON

IT UNIVERSITY OF COPENHAGEN

Name	Email
Adrian Valdemar Borup	adbo@itu.dk
Albert Rise Nielsen	albn@itu.dk
Joachim Alexander Borup	aljb@itu.dk
Thomas Wolgast Rørbech	thwr@itu.dk

May 31, 2022

Table of Contents

1 System Perspective

1.1 System architecture

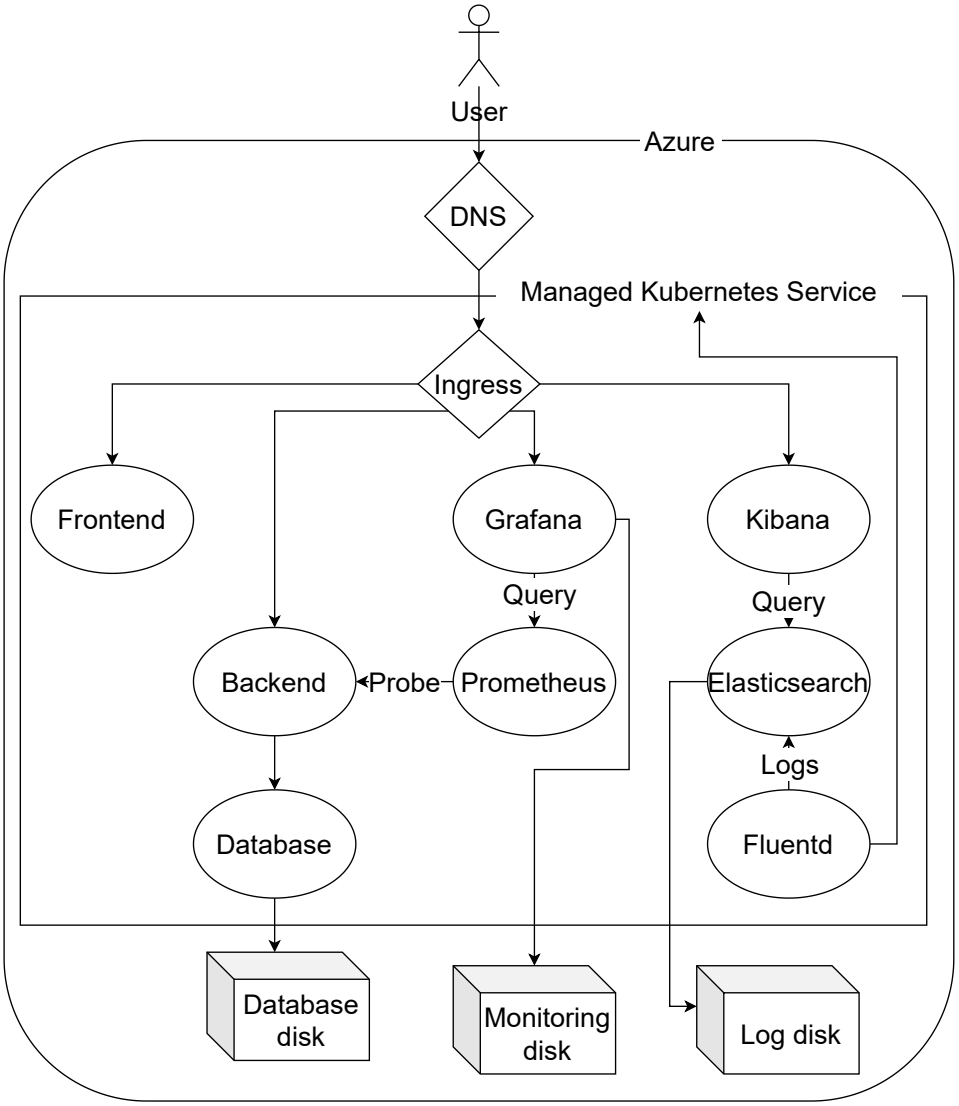


Figure 1: *System architecture and it's components*

We've built our system twice, according to 2 specifications. The first was a, mostly, one to one rewrite of the original project into Go [16]. While the second, and current iteration, was according to the API specification [28]. This iteration consists of 4 subsystems and supporting infrastructure. The subsystems we'll go through below are: backend, frontend, monitoring and logging. The overview and interaction between these systems are modelled in figure 1.

1.1.1 Backend

The core system of the project is the backend. It is responsible for the actual computations and controlling of the database. It's built with Go [16], runs in a Docker container and uses

Microsoft SQL server [40] as the database. We deep dive into this component in the "System Design" section - section 1.2.

1.1.2 Frontend

As the backend changed from server rendered website to a REST API, we decided to move the frontend into it's own subsystem. The frontend is built with Vue.js [9] and is written in TypeScript [42]. It makes use of the public API that the backend exposes, and therefore provides a more user friendly access layer to our service. The subsystem is built in small components that can be reused when expanding the project. We had a focus on the visual elements being mostly identical to the original frontend, to ensure user familiarity.

1.1.3 Monitoring

To keep track of the health of our system, its performance, usage and other metrics, we use Prometheus [47] to collect metrics and Grafana [17] to visualize said metrics. Our metrics include: CPU usage, memory usage, number of requests per endpoint, average request latency and amount of registered users. We explain how we use these metrics in section 2.5.

1.1.4 Logging

For more detailed diagnostics we have a logging subsystem. It's built with Fluentd [10], Elasticsearch [7], and Kibana [8]. It works like this:

1. A pod in the cluster is logging to stdout
2. The stdout is sent to log files by Kubernetes
3. Fluentd reads the logs from the Kubernetes host and parses them according to the configured log format
4. Fluentd tags them and sends them to Elasticsearch
5. Elasticsearch indexes them, such that they are easily searchable, and stores them
6. Kibana provides a web interface to search, read and display the logs in a structured way

Having this logging system allows us to diagnose what happened when something goes wrong, as we can easily find and read the info, warnings and errors that our system outputs.

We explain how we use these logs in section 2.6.

1.1.5 Kubernetes and Azure

To deploy this system we use Kubernetes [33] deployed on Azure [38] through AKS [39].

Using Kubernetes makes us host agnostic. A few times during the project we've had to move our host, which we've done without issue due to the emphasis on containerization and host abstraction that Kubernetes has. Kubernetes also allows us to easily scale our system, as we can easily deploy new nodes and load balance our services.

We've used Azure as our deployment target, as it was reliable, familiar and cheap. AKS is Azure's managed Kubernetes service.

1.1.6 Supporting infrastructure

To make this infrastructure function we have some supporting infrastructure. Such as our domain registrant where we bought our domain - one.com [44]. Our domain administrator that controls our domain and handles nameserver changes - DK Hostmaster [4]. And our Azure hosted DNS zone [3] that handles dns records. This informs systems to route traffic to the correct infrastructure.

1.2 System design

Our main system design goals were to make the system extensible and maintainable. That is, adding new features and replacing current implementations should be easy to do, and you should be able to trust that the system works once the change is made.

1.2.1 Bridge and repository design patterns

We've tried to achieve this by using the repository and bridge design patterns when possible, particularly to communicate with our database. The repository pattern is used to collect all our database queries in one place, and the bridge pattern is used to abstract away the implementation. Rather than talking directly to the database, our controllers communicate via an interface that defines which operations can be made. For an example of this, see the `IUserRepository` interface [23].

Because we depend on the abstraction rather than the implementation, we can easily replace the database with a NoSQL database, create a wrapper that does extra operations on top, or similar—and the code depending on the interface won't have to change.

1.2.2 Object-relational mapping

In the same spirit as above, we've used GORM [29] for object-relational mapping. With this ORM, the database provider is abstracted away, making it possible to replace the database provider without changing the ORM code.

1.2.3 Separate frontend

We moved the frontend code to a separate project [20]. This way, the Go server has only a single responsibility. This also serves as a way of using the bridge pattern to add another layer of abstraction, because the frontend is now only dependent on the API and not the Go implementation. This would also allow for different frontends to be implemented, like you obtain with MVC.

1.2.4 Program modules

In order to improve the single-responsibility aspect of our code, we've used modules to separate code into units of single responsibility as shown in Table 1. Figure 2 shows how these packages are used by each other.

Package	**Responsibility**
`controllers`	Managing the API routes and implementing the controllers
`database`	Communicate with the database
`internal`	Logging output destination and format — and error handling
`models`	Model definitions
`monitoring`	Managing Prometheus and monitoring
`password`	Password hashing and validation
`test/controllers`	Integration and unit tests for all controllers
`main`	Starting the application

Table 1: *The packages in our Go server and their responsibilities.*

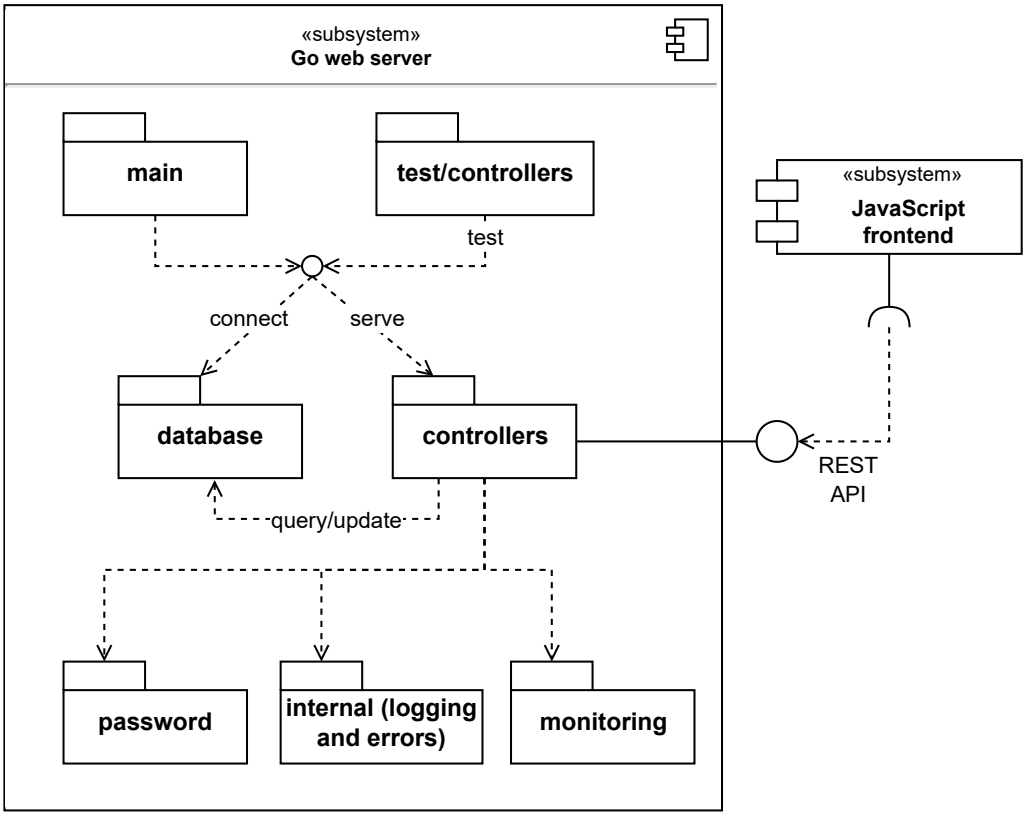


Figure 2: *A UML diagram showing how our packages and self-coded subsystems are related.*

1.3 Software license compatibility

To check if the software licenses of the libraries we've used are compatible with the MIT license in our repository, we have used lichen [53] to automatically check the licenses of all used libraries. This process is documented in Appendix A. The results showed that all the libraries were compatible, except one because the tool couldn't find its license. However, checking its repository manually revealed a compatible MIT license.

1.4 System dependencies

Table 2 shows most of the major tools, libraries, and other technologies our systems depend on.

Category	Tools and technologies	Description
Version control	Git [13], GitHub [14]	Source code accessible for all developers, tracking issues, creating notes and tasks.
Pipeline CI/CD	CircleCi [1]	Automate task such as; testing, building, deploying, releasing, generating changelog.
Containerization	Docker [5], DockerHub [6]	Ensure same development environment and deployment to DockerHub.
Infrastructure	Terraform [27]	Infrastructure as Code, Make it easier to swap host, automate tasks for releases.
Service provider	Azure [38], own server [32]	Hosting our app.
Service management	Kubernetes [33]	Orchestrate containers, scaling up / down, load balancing.
Database	SQL Server [40], SQLite [51]	To store user data.
Logging	Elasticsearch [7], Fluentd [10], Kibana [8]	To gather logs into a centralized log and sort, filter, search the centralized log.
Monitoring	Grafana [17], Prometheus [47]	Gather and organize metrics from our system, based on those create alerts and a visual representation.
Languages	Golang [16], TypeScript [42]	Backend and frontend.
Frameworks	GORM [29], testify [52], gingonic [12], VueJS [9]	ORM, Testing, HTTP Framework, Frontend SPA.
Code analysis	SonarCloud [50], Better Code Hub [49], lint [15] cyclomatic [11]	Increase code quality and motivate clean code.
Security	Snyk [48], OWASP Zap [46], Metasploit [37] WMAP [43]	Notify about Security issues and updates to dependencies and scan for vulnerabilities.
Documentation	LaTeX [36], Markdown [30]	Report, notes, changelog, instructions.

Table 2: *The tools and technologies that our system depends on.*

1.5 Current state of the systems

The system architecture and system design have all been implemented as described, tests have been written, and integration and delivery is automated—see more in subsection 2.4. Using static code analysis tools, we have measured the quality of our code to be good (see Appendix C).

2 Process Perspective

2.1 Team interaction

The team consists of 4 developers, who meet once or twice a week to work on the project. At each meeting, the developers discuss the tasks at hand, create issues, divide the work, and then spends the rest of the time working on their tasks. There is no hierarchy - only 4 like-minded developers who makes decisions in unison.

New tasks are created as issues on GitHub, and then assigned to the developer who will be working on it. Maintaining a set of issues on GitHub lets us keep an overview of the planned tasks, and allows for any of the developers to easily add a new task at any time.

2.2 Organisation of repositories

The group uses GitHub to manage repositories. On GitHub, we have a dedicated organization that owns all of the project repositories. This way, we can keep related repositories together and reuse permission settings on GitHub. Currently, we have the following repositories:

1. `Devops-2022-Group-R/itu-minitwit`, which is the main project repository. Here, we have the backend web server which handles all business logic.
2. `Devops-2022-Group-R/itu-minitwit-frontend`, containing our frontend web-app.
3. `Devops-2022-Group-R/flag-tool`, which is a rewrite of the original flag tool given with the project template. The tool has been rewritten and moved to a separate repository.
4. `Devops-2022-Group-R/bump-tool`, which is a small tool to help with finding the next version bump based on a pull request's tag (major/minor/patch) and the project's current version.

The general philosophy has been to separate parts that can exist alone with a single responsibility. This way, all issues, pull requests, and releases are related to the topic of the repository. In a monolithic repository, you have to put in more effort to specify which part of the repository is relevant for a PR or issue. This also allows for easier and more simple CI/CD pipelines, and it becomes more obvious what a release changes. One example of this separation is the frontend web application, which is completely separated from the backend, with Kubernetes specs and CI/CD pipelines in its own repository.

However, we have not been as good at following this philosophy as we would like to. For example, our monitoring is a separate entity from our web server, but all the monitoring configuration is stored in the `itu-minitwit` repository [19] [24]. Instead, we should have moved this to a separate repository because it does not have anything to do with how the web server operates.

The same goes for the LaTeX files that this report consists of. We would have liked to have a separate repository for this in order to keep Git history clean and CI/CD more separate, but in this case it is a project requirement to have it in the main repository.

2.3 Git branching strategy

We have applied trunk-based development, meaning we branch out from the main branch, have a branch that lives shortly until the changes are implemented, and then merge it directly into the main branch. This keeps unrelated changes separate and makes PRs easier to review and merge. It's worth noting that we do not utilise release branches from trunk-based development, because we release continuously for each merge into the main branch.

2.4 CI/CD chains

We use CircleCI as a continous integration and continous delivery platform, and have configured a workflow [22] with jobs to build, test, and deploy our service, generate a changelog, as well as building our LaTeX project. At the end of this section, there is a diagram (Figure 3) showing our full CI/CD flow.

2.4.1 Building, testing, and static analysis

This part of the workflow quite simply installs the dependencies, and then builds and tests the project. Additionally it runs the following three static code analysis jobs:

- A Go linters runner: `golangci-lint`, which runs dozens of linters in parallel.
- A cyclomatic complexity calculator: `gocyclo`, to ensure that the code has low complexity. Otherwise, the job will exit with code 1 until we've refactored the functions in question.
- A developer security platform, `Snyk`, to find and fix security vulnerabilities in code, dependencies, and containers.

Besides this, we have set up SonarCloud and Better Code Hub, which provide us with more general static code analysis. For the overall results of these two and some examples of warnings, see Appendix C.

2.4.2 Deploying our service

CircleCI will also:

- Publish a GitHub release when merging to the master branch

- Push our changes to the four images in Docker Hub (the server, frontend, Prometheus, and Grafana)
- Deploy by performing a rolling-update to the aforementioned services in Kubernetes
- Apply infrastructure changes via Terraform

2.4.3 Generating a changelog, and building the LaTeX project

We use CircleCI Github Changelog Generator [2] to generate a changelog based on our tags, issues, labels and pull requests on GitHub.

In order to include a build of the LaTeX project in the CircleCI workflow, we've written a script to let CircleCI run a custom Docker image, build a PDF, and then commit it to the branch.

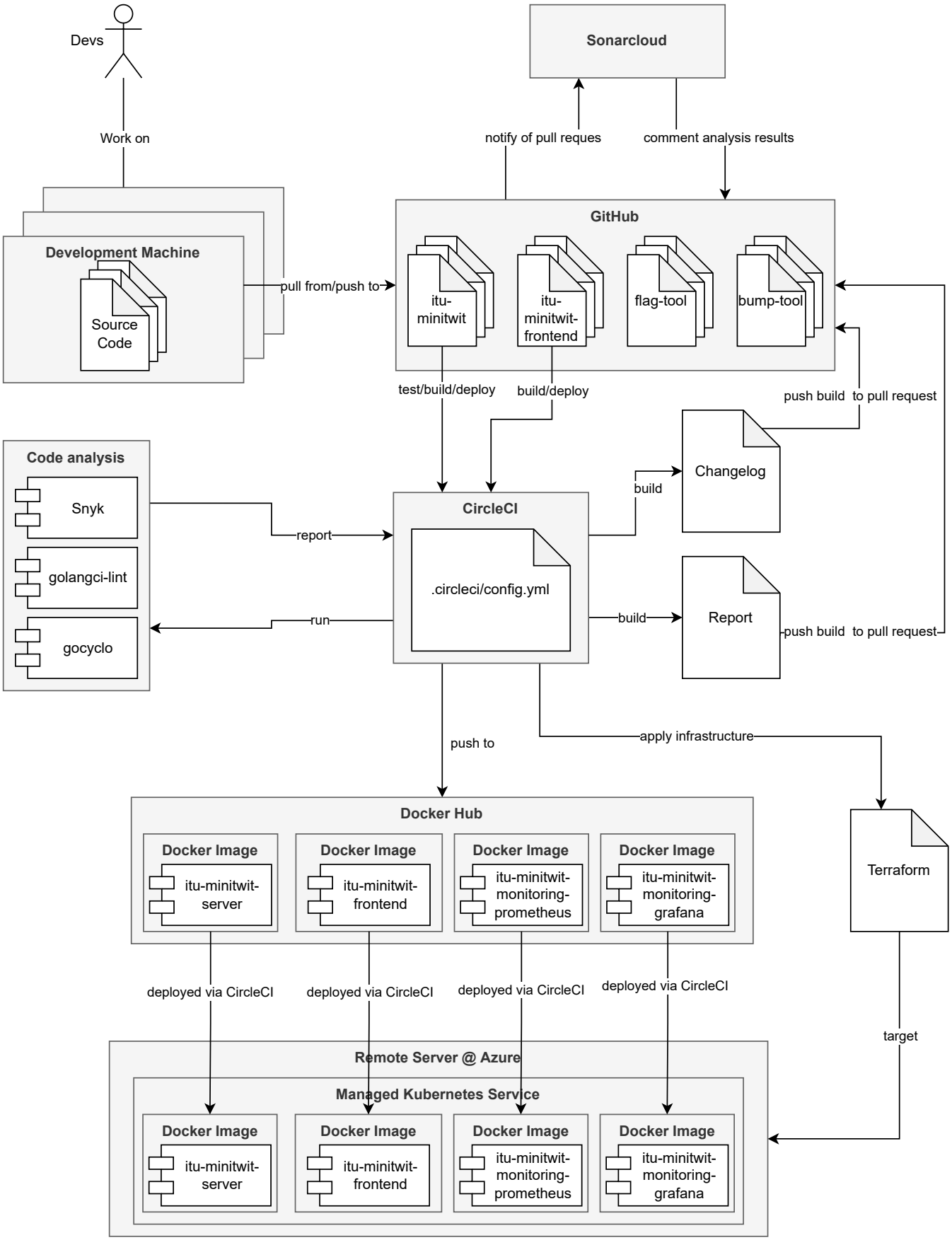


Figure 3: *A diagram showing our CI/CD flow.*

2.5 Monitoring

We chose to monitor the number of requests and request latency for each endpoint on our server, and we monitored the CPU and memory usage of it. We chose these because they give a quick overview of how the system is performing and because they allow us to diagnose issues faster, by locating when something happened, at which endpoint and if it had an effect on system resources.

As an example, we used the request latency to add SQL indexes to our database, as we had some slow endpoints. For a different example where we had an issue with our system, which was reflected in the monitoring, see Appendix B.

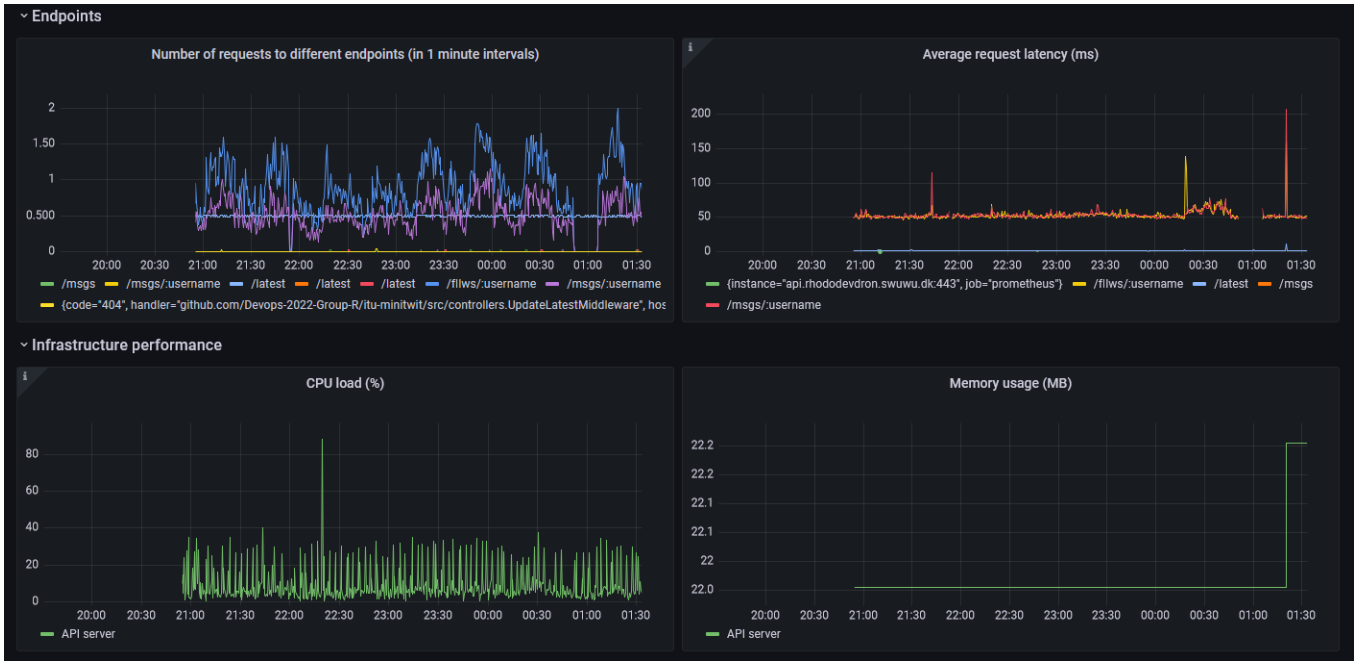


Figure 4: *A sample of our Grafana dashboard.*

2.6 Logging

We log using JSON, which allows us to select specific fields to display in Kibana, making the logs easy to read, filter, and sort by. Figure 5 shows how we display logs in an easy-to-read format.

Time	kubernetes.container_name	method	route	status	duration
Apr 5, 2022 @ 20:33:17.000	itu-minitwit-backend	POST	/fllws/Joan Exler?latest=2498757	204	53.309503ms
Apr 5, 2022 @ 20:33:16.000	itu-minitwit-backend	POST	/fllws/Steffanie Kohut?latest=2498754	204	69.819242ms
Apr 5, 2022 @ 20:33:16.000	itu-minitwit-backend	POST	/fllws/Jessie Mootispaw?latest=2498755	204	52.715469ms
Apr 5, 2022 @ 20:33:16.000	itu-minitwit-backend	GET	/latest	200	1.210268ms
Apr 5, 2022 @ 20:33:16.000	itu-minitwit-backend	POST	/fllws/Takako Mix?latest=2498756	204	69.464013ms
Apr 5, 2022 @ 20:33:15.000	itu-minitwit-backend	POST	/fllws/Daphine Tarnowski?latest=2498751	204	58.167685ms
Apr 5, 2022 @ 20:33:15.000	itu-minitwit-backend	POST	/fllws/Pearl Mccaskell?latest=2498752	204	47.46478ms
Apr 5, 2022 @ 20:33:15.000	itu-minitwit-backend	POST	/fllws/Evelyn Lohmeier?latest=2498753	204	49.529896ms
Apr 5, 2022 @ 20:33:14.000	itu-minitwit-backend	POST	/msgs/Major Zieglen?latest=2498750	204	79.164869ms
Apr 5, 2022 @ 20:33:14.000	itu-minitwit-backend	GET	/latest	200	1.191267ms

Figure 5: *Example of how we view logs in Kibana.*

It's not shown on the figure, but we also have a column for error messages, which we've experienced makes it easy to diagnose problems in the system. To diagnose performance issues, the "duration" column has also been helpful. By the end of the project, the mentioned log output has been enough to diagnose bugs and get an overview of what's happening in the system.

2.7 Security assessment

We intitially identified our assets by looking at which parts of our program might be vulnerable and of value to an intruder, we used the OWASP top 10 web application security risk [45] to identify sources of threats and from these we constructed risk scenarios to get a initial overview of the risks.

After having identified the risks, we used this information during Risk analysis to create a risk matrix and discuss possible solutions for the different threats.

Some of the threats like SQL Injection and vulnerable or outdated dependencies were resolved by using an ORM [29] as middleware for the database to clean user input and static analysis tools integrated with our pipepline like Snyk [48] checking for vulnerable dependencies.

Risk Severity / Risk probability	Catastrophic	Critical	Marginal	Negligible	Insignificant
Certain					
Likely		DDOS			
Possible					
Unlikely			Access to Monitoring		
Rare	SQL Injection Access Azure crendentials	Access to CircleCi	Security flaw of vulnerable component		

Figure 6: *Risk matrix*

After risk analysis we used two vulnerability scanners OWASP ZAP [**owasp-zap**] and Metasploit [37] WMAP [43] to test our system for vulnerabilities targeting our root endpoint

[21]. We tried to add an extra middleware beside our CORS middleware to handle vulnerabilities flagged by the scanners, but we did not want to invest a lot of time and resources into it as we investigated the potential risks and realised they were just warnings and did not apply in our case. We dealt with one of our risk scenarios regarding monitoring, by ensuring changes to the dashboard were only accessible through code.

Read more in our security session notes [25].

2.8 Strategy for scaling and load balancing

We have deployed all of our components with Kubernetes deployments [34], and configured them to run behind Kubernetes services [35], this ensures component level load balancing. Using deployments also allows us to manually scale the number of instances of each component. While deployments do have the ability to autoscale, after some configuration, we have decided against it due to cost issues and because our monitoring indicated that the system was not getting enough load to justify scaling.

Deployments also support zero downtime updates. This is achieved by changing settings on the deployment, such as the image to be used, then Kubernetes creates a new pod, and then kills an old pod when the new is ready, repeating the process until all pods have the new configuration.

3 Lessons Learned Perspective

3.1 Evolution and refactoring

3.1.1 Initial refactoring

At the start of the project, we hosted a large collaborative session, using Live Share [41] to let everyone see the same context of the workspace of one developer, allowing us to co-edit and co-debug the same code. In retrospect, perhaps it would have been better to commit a basic boilerplate app, and then have each developer slowly commit more changes.

Having attempted a 1:1-translation, we ended up with one large file. This is where we started separating responsibilites, moving them to smaller files, as well as moving the front-end to a separate repo.

3.2 Operation

3.2.1 Uptime

We aimed for 100% uptime, as is the spirit of DevOps. Unfortunately, we did not quite live up to this DevOps spirit during the Easter break, where the service was down for several days,

because we ran out of student credits on Azure. Most of us were busy, so we knowingly let the problem remain until the week after, which is not the DevOps mindset.

3.2.2 Service-level agreement

We have provided a service-level agreement [26] for our API, in order to establish an agreement between us and the customer, clearly defining what we can provide to the customer, and which rights they have.

3.2.3 Backwards compatibility in migrating URLs

When we moved to Kubernetes, we found out that we couldn't test Ingress on the `swuwu` domain, so we moved the name servers. However, in order to keep the simulator working, we would need to submit a pull request to the lecture_notes repo, and wait for someome to merge it. This takes time, requires coordination between us and the repo owners, and so on. Therefore, to instead ensure backwards compatibility, we've made sure that every link that we've used at some point still works. For instance, both of these domains point to the same place:

- `https://rhododevdron.dk/`
- `https://rhododevdron.swuwu.dk/`

3.3 Maintenance

3.3.1 Replacing database management systems

We've switched from SQLite to PostgreSQL, and then later to SQL Server. This could be done almost seamlessly due to the abstraction of our repository controllers, as described in section 1.2. In particular, to see how little change was required to switch to PostgreSQL, see pull request #34, particularly the changes to `main.go`, and how it interacts with `database/database.go` in commit b483145 [18].

3.3.2 Switching hosts with Kubernetes

At some point, we had issues with subscriptions on Azure due to missing credits, so we temporarily stopped hosting on Azure, and instead switched to a local computer. Thanks to Kubernetes, this process of switching hosts was very easy.

3.3.3 Infrastructure as Code

Using Terraform as an IaC software tool has made it simple to deal with student credits running out, as well as switching subscriptions on Azure. We can simply just rebuild all the infrastructure on a new subscription, with a few commands.

3.4 Reflection

We believe we have achieved a high flow rate in our project, by fully automating the process from code to server to "customer". This has allowed us to incrementally change our system, and get quick feedback from the customers and monitoring systems. Our automation systems also allows for easy rollback of any changes that may have caused issues.

This way of deploying often instead of large deployments is very different from our usual projects. As they would usually be building the project and then releasing the entire thing at once. Instead we here start with a project that works and incrementally update, fix, and improve from there, while having it deployed and receiving requests. It's an interesting and much more reassuring way of working, as it allows us to instantly determine whether or not our changes had a positive, or negative, effect in a production environment. This, in turn provides us with a constant feedback loop that we can use to improve the system even more. This way of working is worth considering in later projects, though it does require some way of creating system load.

This course emphasises risk taking and continual learning, by having us learn and use a new technology, mostly, every week. But also encourages trying something outside of the preset parameters. We've taken this to heart and tried a host of different technologies until we found the one that fit our needs. First we used Vagrant with DigitalOcean, then Terraform with Azure WebApps, and finally Kubernetes on AKS. In databases we've tried different technologies as well, first using an SQLite database, moving to Postgres, then to Azure SQL database and lastly Microsoft SQL server in Kubernetes. This, while risky, has enabled us to learn new technologies and improve the deployment and hosting process, which provides value both to us as developers, and to our customers who get a faster service and more updates. We've enjoyed working this way and will try to do so in later projects. It seems, though, like a work method that mostly makes sense in projects where the system is already in a production ready state and running. Then you can start experimenting with technologies and new methods. The biggest takeaway here is that moving to a new technology or method is risky, and time consuming, but can provide a lot of value to customers and developers alike.

4 References

[1] CircleCI. *CircleCI*. Accessed: May 25 2022. URL: https://circleci.com/.

[2] Comino, Murillo. *CircleCI Github Changelog Generator*. Accessed: May 18 2022. Oct. 2020. URL: https://github.com/onimur/circleci-github-changelog-generator.

[3] Contributors. *DNS Zones and Records overview - Azure DNS | Microsoft Docs*. Accessed: May 24 2022. Mar. 2022. URL: https://docs.microsoft.com/en-us/azure/dns/dns-zones-records.

[4] DK Hostmaster. *DK Hostmaster*. Accessed: May 24 2022. URL: https://www.dk-hostmaster.dk/da.

[5] Docker. *Docker*. Accessed: May 25 2022. URL: https://www.docker.com/.

[6] Docker. *Docker Hub*. Accessed: May 25 2022. URL: https://hub.docker.com/.

[7] Elasticsearch B.V. *Elasticsearch*. Accessed: May 24 2022. URL: https://www.elastic.co/.

[8] Elasticsearch B.V. *Kibana*. Accessed: May 24 2022. URL: https://www.elastic.co/kibana.

[9] Evan You and Contributors. *Vue.js*. Accessed: May 24 2022. URL: https://vuejs.org/.

[10] Fluentd Project. *Fluentd*. Accessed: May 24 2022. URL: https://fluentd.org/.

[11] fzipp. *Gocyclo*. Accessed: May 25 2022. URL: https://github.com/fzipp/gocyclo.

[12] Gin-Gonic. *Gin*. Accessed: May 25 2022. URL: https://github.com/gin-gonic/gin.

[13] Git. *Git*. Accessed: May 25 2022. URL: https://git-scm.com/.

[14] GitHub. *GitHub*. Accessed: May 25 2022. URL: https://github.com/.

[15] Golangci. *Golangci-Lint*. Accessed: May 25 2022. URL: https://github.com/golangci/golangci-lint.

[16] Google. *Go*. Accessed: Febuary 8 2022. URL: https://golang.org/.

[17] Grafana Labs. *Grafana*. Accessed: May 24 2022. URL: https://grafana.com/.

[18] Group R. *Commit b483145 of pull request #34 - Created database layer*. Accessed: May 25 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit/pull/34/commits/b4831458ae940e9b4c42dd929fe9be73143580d4.

[19] Group R. *Configuration files for ITU-MiniTwit monitoring*. Accessed: May 17 2022. Apr. 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit/tree/master/monitoring.

[20] Group R. *ITU MiniTwit frontend repository*. Accessed: May 24 2022. Mar. 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit-frontend.

[21] Group R. *ITU-MiniTwit - A microblogging platform for ITU students*. 2022. URL: https://rhododevdron.dk/.

[22] Group R. *ITU-MiniTwit CircleCI workflow*. Accessed: May 17 2022. Apr. 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit/blob/master/.circleci/config.yml.

[23] Group R. *IUserRepository interface*. Accessed: May 24 2022. Mar. 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit/blob/master/src/database/user-repository.go#L37-L48.

[24] Group R. *Kubernetes specs for ITU-MiniTwit monitoring*. Accessed: May 17 2022. Apr. 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit/tree/master/.infrastructure/kubernetes/monitoring.

[25] Group R. *Security notes for ITU-MiniTwit*. Accessed: May 18 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit/blob/2e19621aed54a6b9c0185ba7053dfb63a6e5948a/notes/session09_Security.md.

[26] Group R. *Service-level agreement for Group R's implementation of ITU-MiniTwit*. Accessed: May 25 2022. URL: https://github.com/Devops-2022-Group-R/itu-minitwit/blob/master/SLA.md.

[27] HashiCorp. *Terraform*. Accessed: May 25 2022. URL: https://www.terraform.io/.

[28] Helge Pfeiffer. *minitwit_sim_api.py*. Accessed: Febuary 15 2022. Feb. 2022. URL: https://github.com/itu-devops/lecture_notes/blob/master/sessions/session_03/API%20Spec/minitwit_sim_api.py.

[29] Jinzhu. *GORM - Golang ORM*. Accessed: May 18 2022. 2022. URL: https://gorm.io/index.html.

[30] John Gruber And Aaron Swartz. *Markdown*. Accessed: May 25 2022. URL: https://en.wikipedia.org/wiki/Markdown.

[31] Gene Kim. *The DevOPS Handbook - How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. It Revolution Press, 2016. ISBN: 9781942788003.

[32] Kubernetes. *Kubeadm*. Accessed: May 25 2022. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/.

[33] Kubernetes Authors. *Kubernetes*. Accessed: May 24 2022. URL: https://kubernetes.io/.

[34] Kubernetes Website Contributors. *Deployments | Kubernetes*. Accessed: May 18 2022. Feb. 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[35] Kubernetes Website Contributors. *Service | Kubernetes*. Accessed: May 18 2022. May 2022. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[36] Leslie Lamport - LaTeX. *LaTeX*. Accessed: May 25 2022. URL: https://www.latex-project.org/.

[37] Metasploit. *Metasploit - The Penetration Testing Framework*. Accessed: May 18 2022. 2022. URL: https://www.metasploit.com/.

[38] Microsoft. *Azure*. Accessed: May 24 2022. URL: https://azure.microsoft.com/en-us/.

[39] Microsoft. *Azure Kubernetes Service*. Accessed: May 24 2022. URL: https://azure.microsoft.com/en-us/services/kubernetes-service/.

[40] Microsoft. *Microsoft SQL Server*. Accessed: May 24 2022. URL: https://www.microsoft.com/en-us/sql-server/sql-server-2019.

[41] Microsoft. *Visual Studio Live Share*. Accessed: May 24 2022. URL: https://visualstudio.microsoft.com/services/live-share/.

[42] Microsoft and Contributors. *TypeScript*. Accessed: May 24 2022. URL: https://www.typescriptlang.org/.

[43] OffSec. *wmap web scanner*. Accessed: May 18 2022. 2022. URL: https://www.offensive-security.com/metasploit-unleashed/wmap-web-scanner/.

[44] one.com. *one.com*. Accessed: May 24 2022. URL: https://one.com/.

[45] OWASP. *OWASP Top 10 2021 - Web Application Security Risks*. Accessed: May 18 2022. 2022. URL: https://owasp.org/www-project-top-ten/.

[46] OWASP. *OWASP ZAP - Web Application Security Scanner*. Accessed: May 18 2022. 2022. URL: https://zaproxy.org/.

[47] Prometheus Authors. *Prometheus*. Accessed: May 24 2022. URL: https://prometheus.io/.

[48] Snyk. *Snyk - Security Scanning Tool*. Accessed: May 18 2022. 2022. URL: https://snyk.io/.

[49] Software Improvement Group. *Better Code Hub*. Accessed: May 25 2022. URL: https://bettercodehub.com/.

[50] SonarCloud. *SonarCloud*. Accessed: May 25 2022. URL: https://sonarcloud.io/.

[51] SQLite. *SQLite*. Accessed: May 25 2022. URL: https://www.sqlite.org/.

[52] Stretchr, Inc. *Testify*. Accessed: May 25 2022. URL: https://github.com/stretchr/testify.

[53] UW Labs. *lichen*. Accessed: March 20 2022. URL: https://github.com/uw-labs/lichen.

Appendices

Table of Contents

A Checking software license compatibility

To run the license scan tool, we used:

```
# Install
go install github.com/uw-labs/lichen@latest
# Run in root of Devops-2022-Group-R/itu-minitwit/
go build -o my-binary ./src
lichen my-binary
```

This reveals that the only disallowed library we use is `Azure/azure-sdk-for-go/sdk/internal` because it's missing a license. However, if you check the repository manually, you can find a MIT license in the root.

The output of the command is:

```
github.com/Azure/azure-sdk-for-go/sdk/azcore@v0.19.0: MIT (allowed)
github.com/Azure/azure-sdk-for-go/sdk/azidentity@v0.11.0: MIT (allowed)
github.com/Azure/azure-sdk-for-go/sdk/internal@v0.7.0:  (not allowed - unresolvable license)
github.com/beorn7/perks@v1.0.1: MIT (allowed)
github.com/cespare/xxhash/v2@v2.1.2: MIT (allowed)
github.com/denisenkom/go-mssqldb@v0.12.0: BSD-3-Clause (allowed)
github.com/gin-contrib/sse@v0.1.0: MIT (allowed)
github.com/gin-gonic/gin@v1.7.7: MIT (allowed)
github.com/go-playground/locales@v0.14.0: MIT (allowed)
github.com/go-playground/universal-translator@v0.18.0: MIT (allowed)
github.com/go-playground/validator/v10@v10.10.0: MIT (allowed)
github.com/golang-sql/civil@v0.0.0-20190719163853-cb61b32ac6fe: Apache-2.0 (allowed)
github.com/golang-sql/sqlexp@v0.0.0-20170517235910-f1bb20e5a188: BSD-3-Clause (allowed)
github.com/golang/protobuf@v1.5.2: BSD-3-Clause (allowed)
github.com/jinzhu/inflection@v1.0.0: MIT (allowed)
github.com/jinzhu/now@v1.1.4: MIT (allowed)
github.com/leodido/go-urn@v1.2.1: MIT (allowed)
github.com/mattn/go-isatty@v0.0.14: MIT (allowed)
github.com/mattn/go-sqlite3@v1.14.11: MIT (allowed)
github.com/matttproud/golang_protobuf_extensions@v1.0.1: Apache-2.0 (allowed)
github.com/pkg/browser@v0.0.0-20180916011732-0a3d74bf9ce4: BSD-2-Clause (allowed)
github.com/prometheus/client_golang@v1.12.1: Apache-2.0 (allowed)
github.com/prometheus/client_model@v0.2.0: Apache-2.0 (allowed)
github.com/prometheus/common@v0.32.1: Apache-2.0 (allowed)
github.com/prometheus/procfs@v0.7.3: Apache-2.0 (allowed)
github.com/shirou/gopsutil@v3.21.11+incompatible: BSD-3-Clause (allowed)
github.com/sirupsen/logrus@v1.6.0: MIT (allowed)
github.com/tklauser/go-sysconf@v0.3.10: BSD-3-Clause (allowed)
github.com/tklauser/numcpus@v0.4.0: Apache-2.0 (allowed)
github.com/ugorji/go/codec@v1.2.6: MIT (allowed)
github.com/zsais/go-gin-prometheus@v0.1.0: MIT (allowed)
```

```
golang.org/x/crypto@v0.0.0-20220210151621-f4118a5b28e2: BSD-3-Clause (allowed)
golang.org/x/net@v0.0.0-20211112202133-69e39bad7dc2: BSD-3-Clause (allowed)
golang.org/x/sys@v0.0.0-20220209214540-3681064d5158: BSD-3-Clause (allowed)
golang.org/x/text@v0.3.7: BSD-3-Clause (allowed)
google.golang.org/protobuf@v1.27.1: BSD-3-Clause (allowed)
gopkg.in/yaml.v2@v2.4.0: Apache-2.0, MIT (allowed)
gorm.io/driver/sqlite@v1.2.6: MIT (allowed)
gorm.io/driver/sqlserver@v1.3.1: MIT (allowed)
gorm.io/gorm@v1.23.1: MIT (allowed)
2022/03/20 14:23:40 1 error occurred:
      * github.com/Azure/azure-sdk-for-go/sdk/internal@v0.7.0: not allowed - unresolvable
        license
```

B Monitoring example: High CPU usage

Figure 8 and Figure 9 show the status of our system in Grafana and Azure when we tried to implement logging. The initial implementation used up high amounts of our CPU resources, which is reflected in the response times and CPU recording metrics.

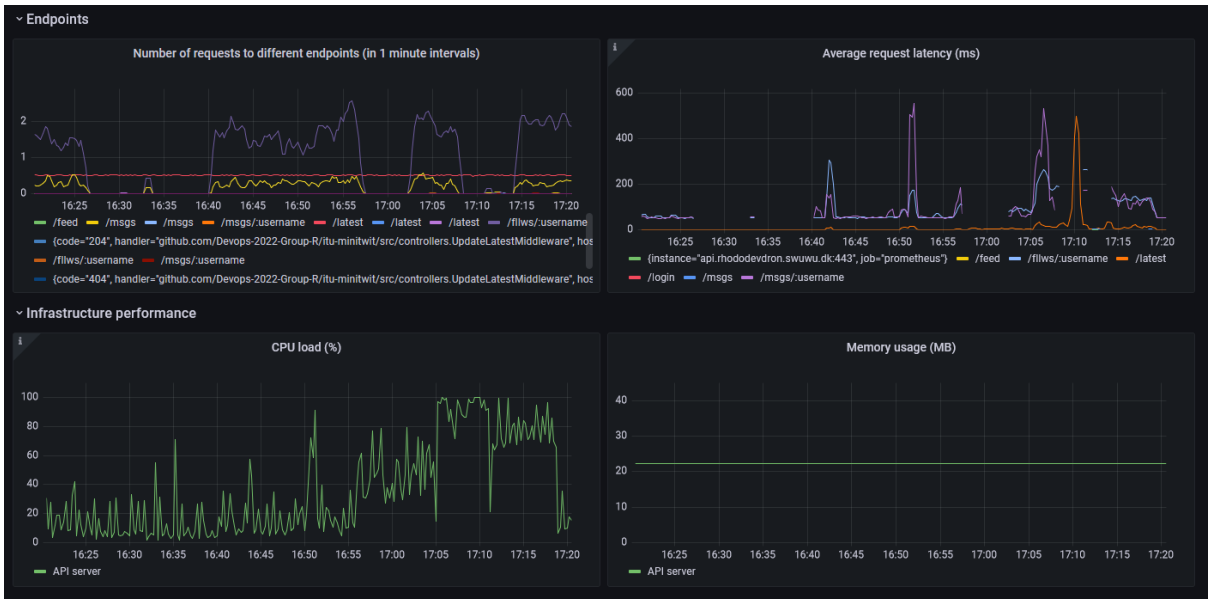


Figure 8: *A snapshot of our Grafana monitoring dashboard.*

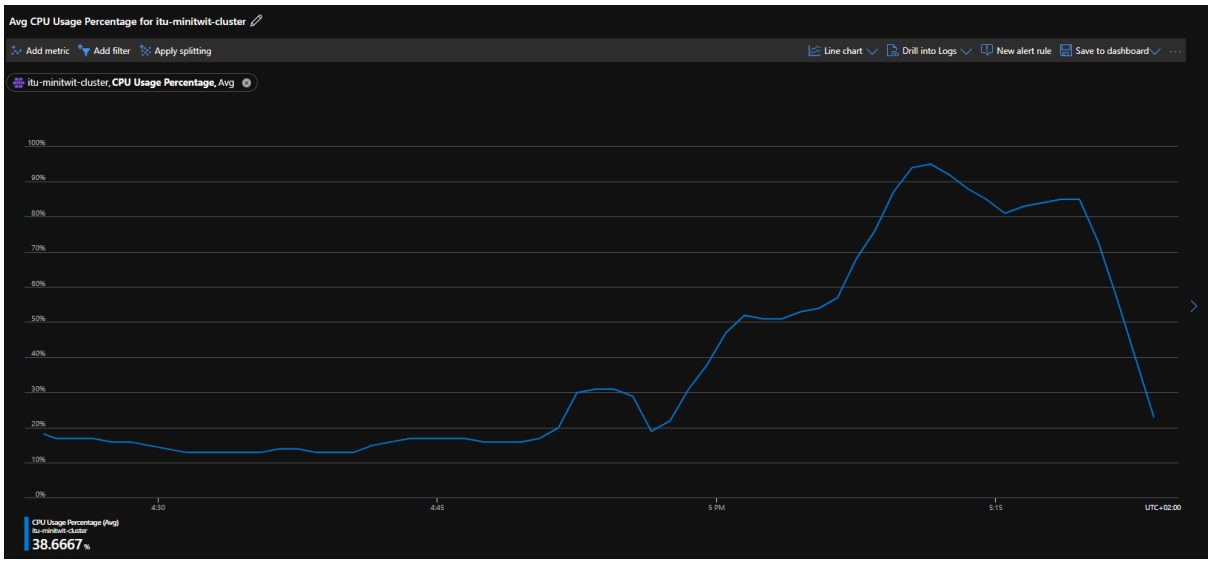
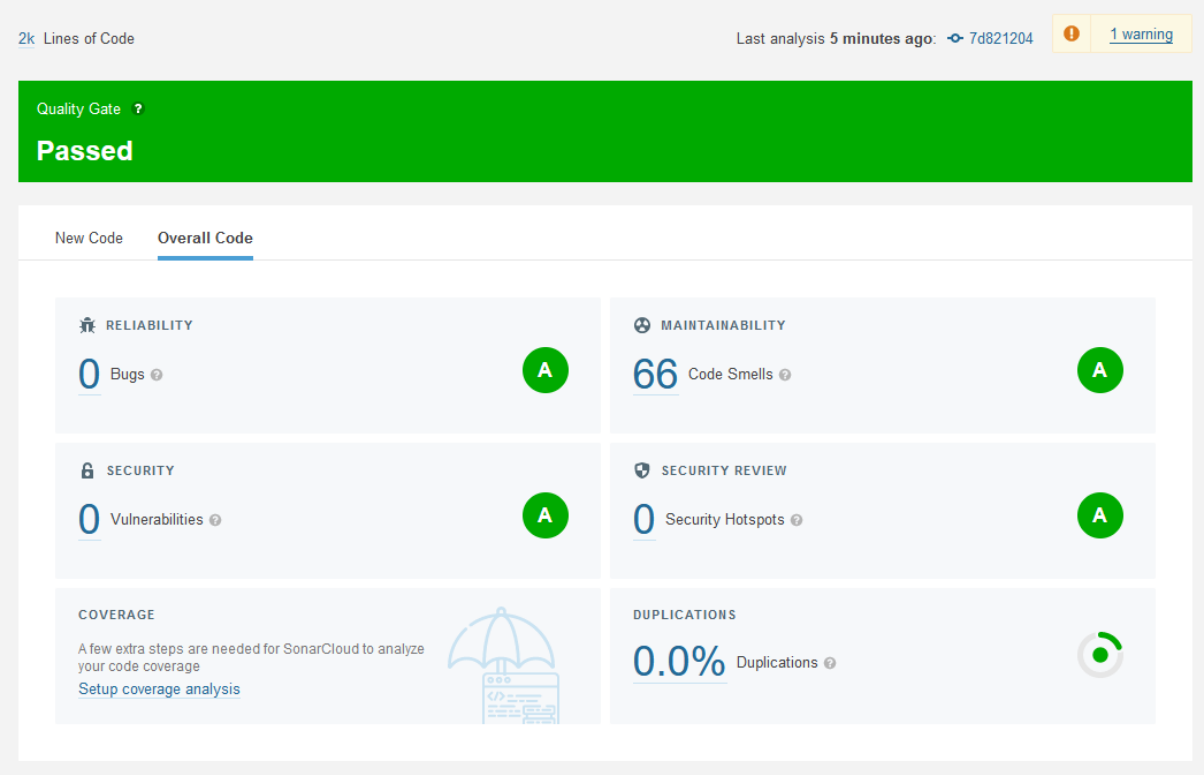


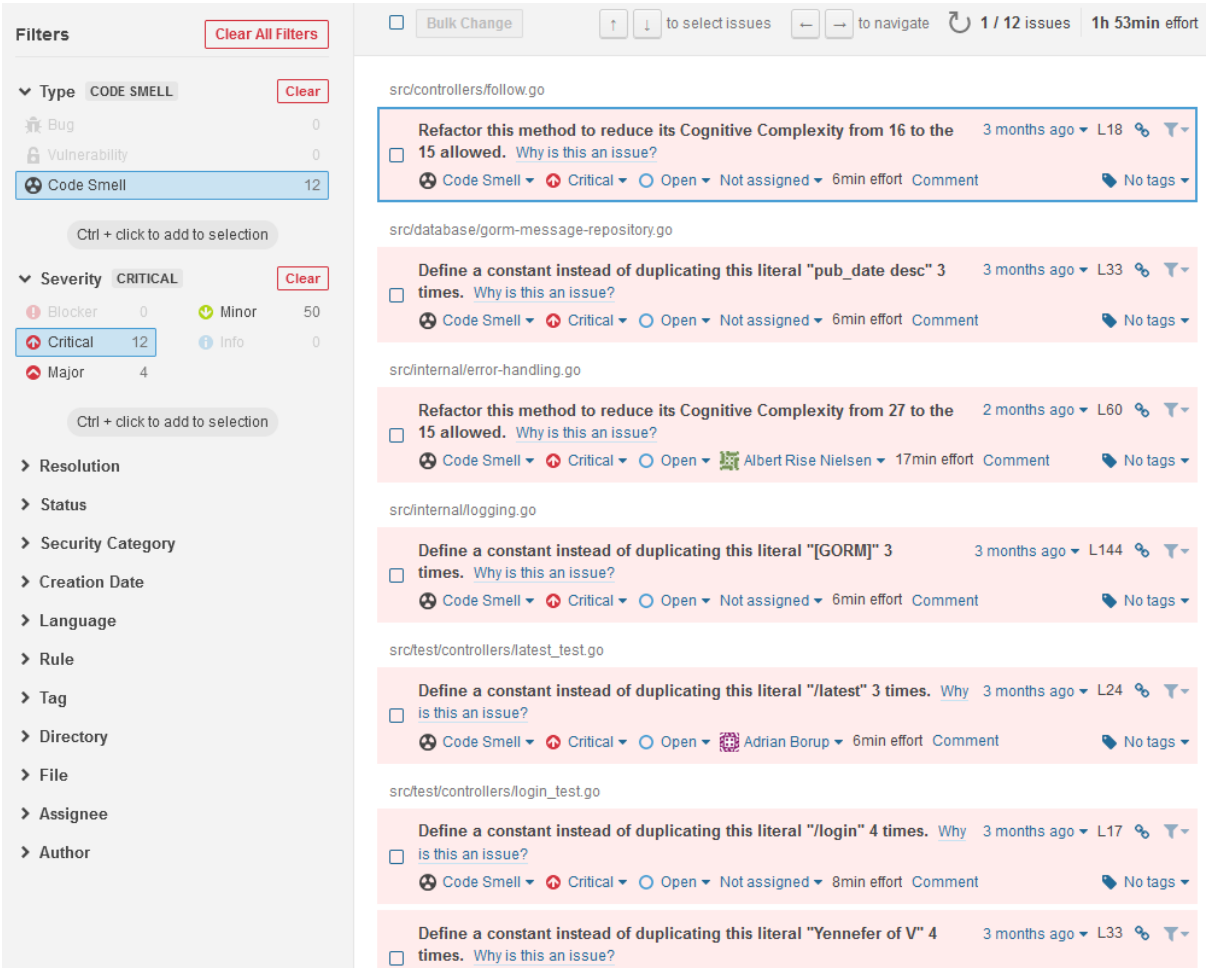
Figure 9: *A graph from Azure's monitoring showing the CPU usage of our cluster.*

C Results of static code analysis with SonarCloud and Better Code Hub

C.1 SonarCloud results



C.2 SonarCloud “critical” code smell examples



C.3 Better Code Hub results