DevOps, Software Evolution and Software Maintenance

Alekxander Baxwill (abax@itu.dk)

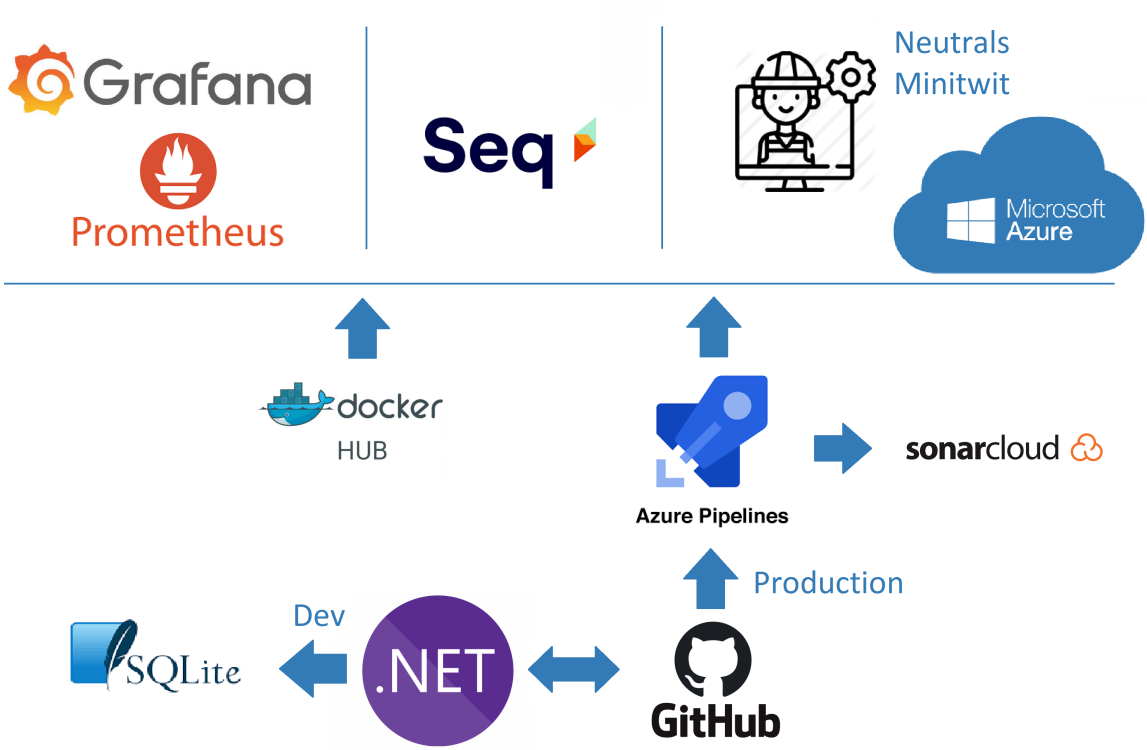
Jacob Sjöblom (jsjo@itu.dk)

Frederik Peter Volkers (frvo@itu.dk)

Albert Bethlowsky Rovsing (arov@itu.dk)

Rasmus Andreas de Neergaard (rade@itu.dk)

May 18, 2021



1 System's Perspective

1.1 Design and Architecture of the system

As a basis for this project, we as a team decided on rebuilding the solution in the C language. As a alternative Java was also discussed, since there was previous experience with it in the group. We decided on C over Java to give our self an opportunity to explore a new language, while still having some experience to fall back on due to the similarities between the two. In addition C is widely used in the industry, and gaining experience with .NET could be good addition to our future job aspects.

For the cloud provider, Digital Ocean was initially considered but due to the synergy between Microsoft Azure and other Microsoft Services (Azure DevOps, Azure SQL, Monitoring etc.) in addition to some team members having experience with Azure. Microsoft Azure also provides students with free credits.

To give an overview of the architecture and design of the MiniTwit application, the 3+1 approach is used. The architecture of our MiniTwit implementation is based on a ASP-NET MVC which is based of the Model-View-Controller pattern.

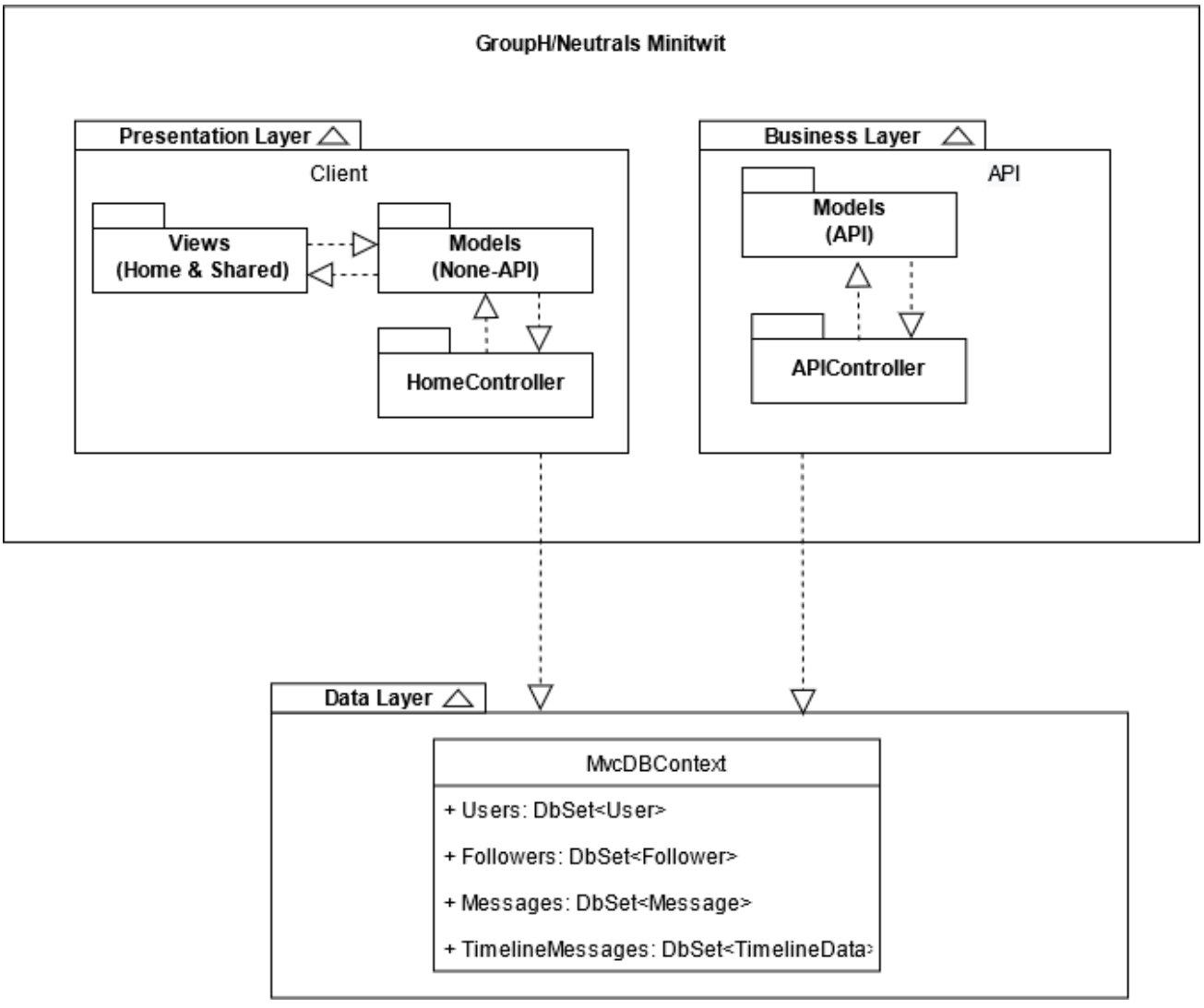


Figure 1: Module View from 3+1

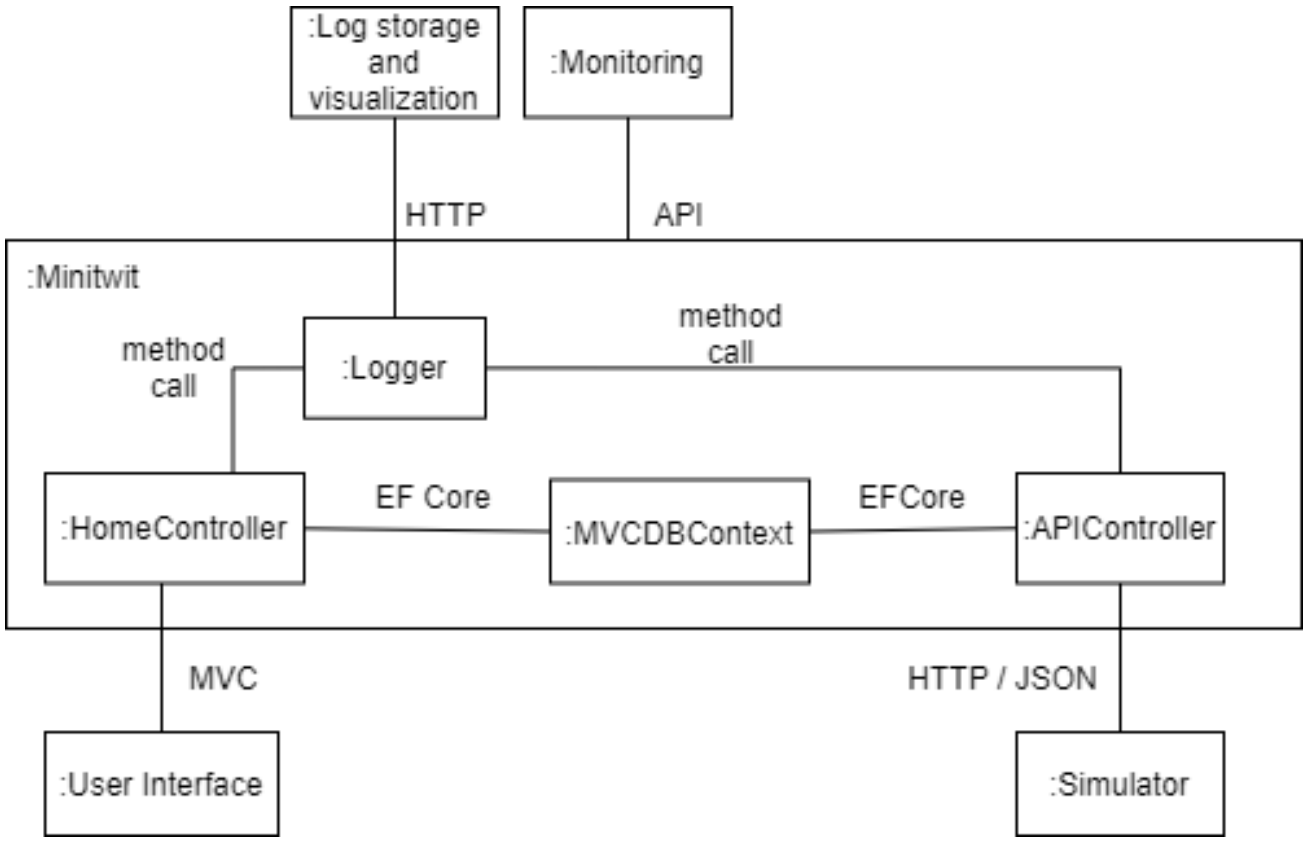


Figure 2: Connector Component View

1.2 Dependencies

1.2.1 Software dependencies

The dependencies of the system is derived via GitHub from the `mvc-minitwit.csproj` and `HomeControllerTests.csproj` project files. These correspond to the MiniTwit and testing project respectively. The dependency graph can be found on https://github.com/albertbethlowsky/DevOpsGroupH/network/dependencies. A small snippet of the current graph for both projects can be seen below:

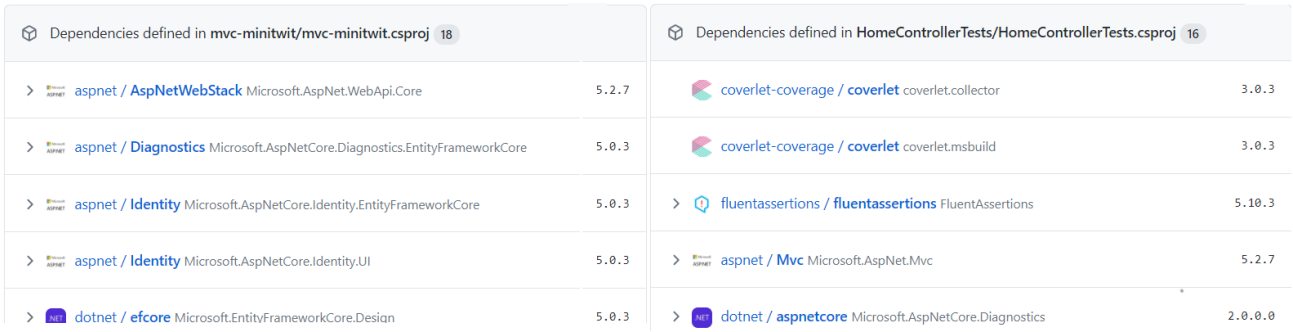


Figure 3: Snippet of dependency graphs from GitHub of both `mvc-minitwit.csproj` and

`HomeControllerTests.csproj`

With the use of NDepend we can provide a more detailed visualization, showing the concrete package dependencies. It is a static analysis tool which generate a dependency matrix for the backend dependencies. From the snippet of the matrix **??**, we can observe that the *'mvc-minitwit'* package (row)

is used by 20 code elements (blue) in *'mvc-minitwit.Views'* (column) and *'mvc-minitwit'* (column) is using 5 code elements (green) from *'mvc-minitwit.Views'* (row) [**?**].

Figure 4: Snippet of Ndepend - Dependency Matrix

1.2.2 Cloud dependencies

These dependencies are related to the services hosted on cloud-based domains.

Name	Service	Provider	Description
neutrals-minitwit	App Service	Microsoft Azure	development, rollout and scaling of web apps (.NET application)
minitwit-neutrals	App Service	Microsoft Azure	development, rollout and scaling of web apps (prometheus, grafana)
neutralsseq	App Service	Microsoft Azure	development, rollout and scaling of web apps (Datalust - Seq)
minitwit-neutrals	SQL Server	Microsoft Azure	Hosting of SQL database
minitwitDb	SQL database	Microsoft Azure	SQL database
ASP-NeutralsRG	App Service Plan	Microsoft Azure	Hosting of Web Services (.NET application and Datalust - Seq)
ASP-mvcminitwit	App Service Plan	Microsoft Azure	Hosting of Web Services (Grafana and Prometheus)

1.3 Current state of the system

With the help of different static analysis tools, it's possible to get a sense of the state of the system, including an estimation of the technical debt. Their respective results are briefly presented:

SonarCloud

Based on the results from SonarCloud the systems seems to be in a satisfactory state, however there are a few caveats to point out. The auto-generated contents of `Migrations/`, and the `wwwroot/lib` folder has been excluded from all analysis since the latter contains deprecated jQuery libraries. These were given from the beginning, and were thus deemed out of scope to fix with time available.

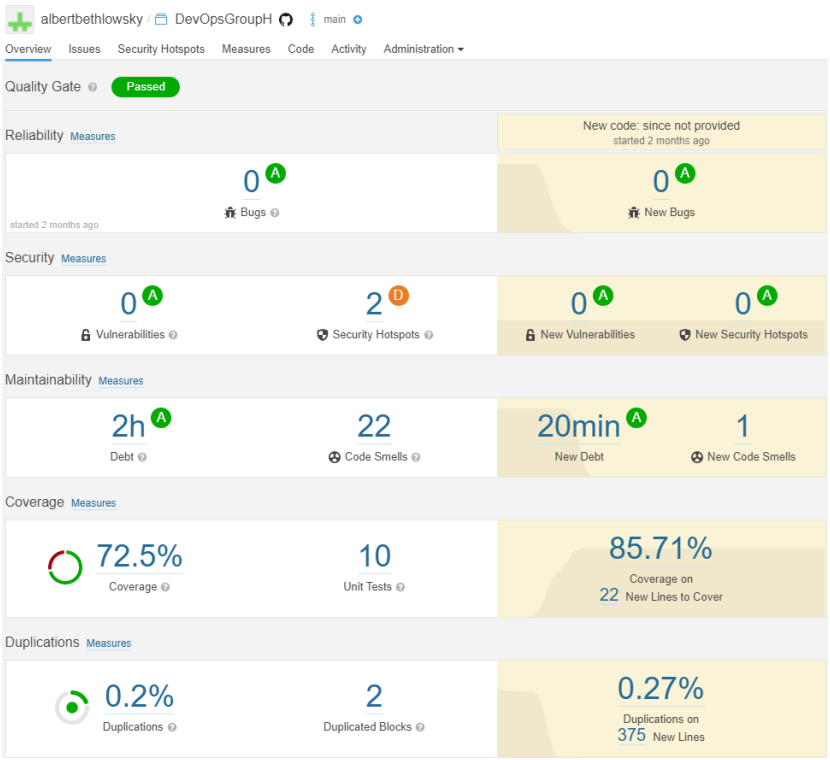


Figure 5: Results from SonarCloud and its different categories, with the Quality Gate passed

The boiler plate, initialization files `Program.cs`, `Startup.cs` are excluded from test coverage analysis. `HomeController.cs` is also excluded from test coverage due to project scope and challenges with mocking cookies. From the maintainability category, there's "2 hours" of debt, but it's worth mentioning that it's a relative estimate based on "Code smells", uncovered and duplicated lines. This estimate should be compared with the estimates from the other tools.

Code Climate

The repository has a maintainability grade of "B", which with their estimates would be about 2 days of technical debt. There are 19 issues in total, split into 8 of duplication and 11 code smells. The file with the lowest maintainability score is `HomeController.cs` and has a "C", which would be the file to prioritize first for refactoring. The same files and folders have been excluded as with SonarCloud.

As a side note a big source of the issues in both `HomeController.cs` and `APIController.cs` comes in the shape of *"Avoid too many return statements"*. This is an issue that can be taken with a grain of salt, since it relates to the debate of "clean OOP" approach, which is harder to implement in server/API application.

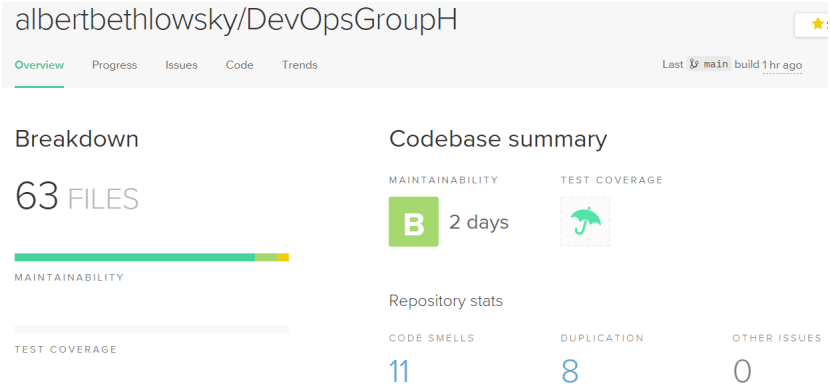


Figure 6: Code Climate summary and overview

Better Code Hub

Better code hub rates our compliance 7/10 and gives us many of the same code smells as the other service scans have given us. We acknowledge that our lack of experience with C# has some effects of on our coding style.

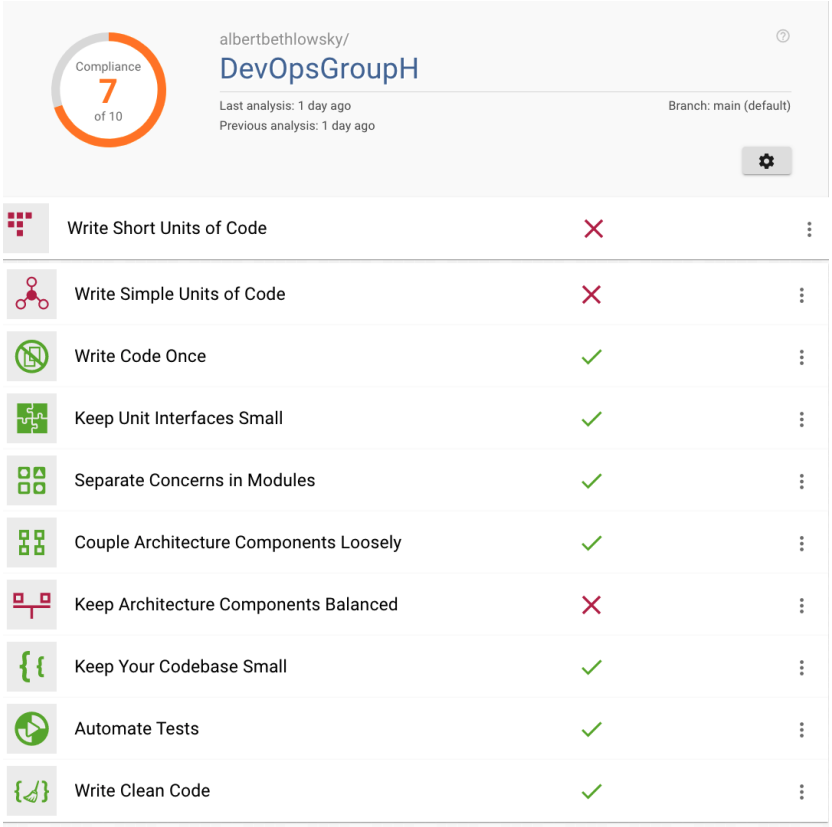


Figure 7: Better Code Hub results

It's worth mentioning that technical debt is a relative term, and as seen between the tools, can vary quite a lot (e.g. 2 days to 2 hours).

1.4 Licenses and compatibility

The dependencies in our project are licensed under three licenses. The Apache 2.0 license [1], the MIT license [2] and the BSD-3 clause [3]. All of these licenses fall within the category of permissive licenses. Below is a table showing how the dependencies are distributed on these different licenses.

Apache 2 License	MIT License	BSD 3
Aspnet/AspNetWebStack	Coverlet-coverage/coverlet	Moq4/moq4
Aspnet/Diagnostics	Microsoft/vstest	
Aspnet/hosting	Newtonsoft.Json	
Aspnet/Identity	prometheus-net/prometheus-net	
Aspnet/Logging	Swashbuckle.AspNetCore	
Aspnet/Mvc	Swashbuckle.AspNetCore	
Dotnet/efcore		
Donet/scaffolding		
Serilog/serilog-aspnetcorer		
Serilog/serilog-filters-expressions		
Serilog/serilog-settings-configuration		
Serilog/serilog-sinks-file		
Serilog/serilog-sinks-seq		
Fluentassertions/fluentassertions		
Xunit/xunit		
Xunit/visualstudio.xunit		

As all licenses are permissive, they impose no restrictions for us as we do not modify the source code of the dependencies. As the majority of our dependencies are licensed under Apache 2.0, we have chosen to also license under Apache 2.0.

2 Process perspective

2.1 Interactions between developers and organization

As everyone in the team have different schedules it was difficult to find time to always meet as a team. We strove towards having two weekly meetings on Mondays and Wednesdays. The meetings were held online on Teams, and in each meeting we presented what we were currently working on and what we had finished since last meeting. When we had to make big decisions we did it all together but when working on the project we often delegated the task between us. We assigned ourselves to tasks using Github Project Management. Often times team members would collaborate and do pair/group programming and work on tasks together. The communication of the team was exclusively through Teams to keep everything centralized.

2.2 Description of CI/CD pipelines, stages and tools

Azure DevOps is used to manage the CI/CD pipelines for the project. Azure DevOps was picked since it was easy to use and integrated well with the hosting on Azure. As an example, certain services (Azure Kubernetes Services, Azure App Service for deployment, etc.) for Azure could be used from the market place on Azure DevOps.

There are two pipelines, the CI pipeline used for testing, code analysis and building and the CD pipeline for deploying.

Continuous Integration

Whenever the main branch is receives commits, the CI pipeline is triggered. The chain consists of one stage with the jobs executed in the order of:

[1] https://github.com/dotnet/aspnetcore/blob/main/LICENSE.txt
[2] https://opensource.org/licenses/MIT
[3] https://opensource.org/licenses/BSD-3-Clause

1. Test and Sonar Cloud Setup and Analysis
2. Build Image and Upload to Azure Container Registry
3. Make auto GitHub release (if tag added in commit)

If testing fails the pipeline stops. The result of running the CI pipeline looks as follows:

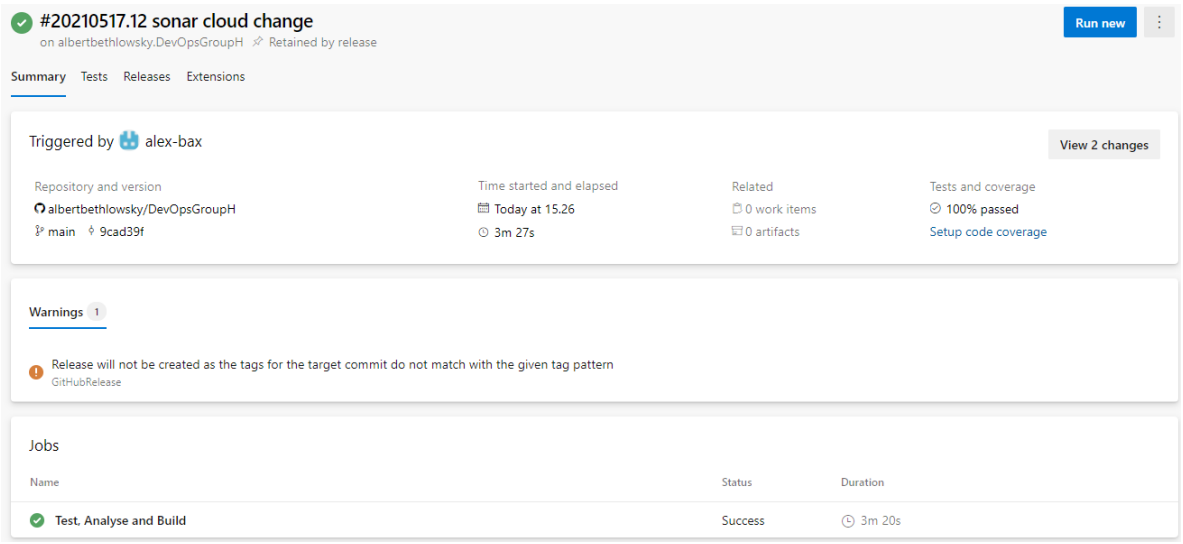


Figure 8: Result of the CI chain from `azure-pipelines.yaml`

No automatic git branch-merging tasks have been added, which is reflected on in Lessons Learned section.

Continuous Delivery

The CD pipeline is responsible for pulling the latest image on ACR, and is thus triggered when a new image is pushed. ACR was chosen as the container registry for the MiniTwit image since it integrates more easily with Azure Kubernetes Services and Azure. After having pulled, a post deployment gate is checked, in this CD chain the status of a SonarCloud Quality Gate. Based on the results of the SonarCloud analysis from the CI pipeline the gate is updated, and if it passes the image is deployed onto Azure. Otherwise the release will timeout after 5 minutes and fail.

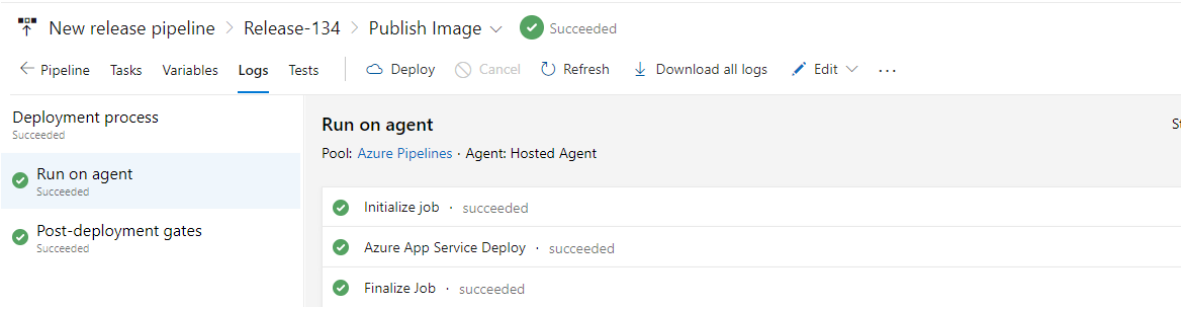


Figure 9: CD pipeline

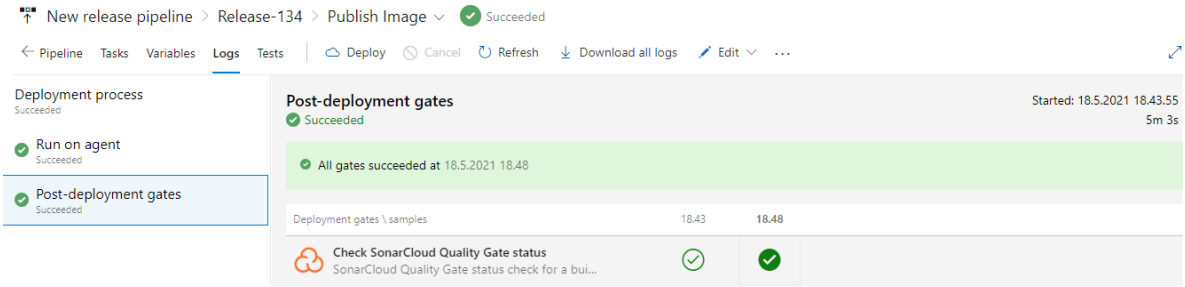


Figure 10: Passing SonarCloud Quality gate on the CD pipeline

See `azure-pipelines.yml` for more details on the stages and tools used in the CI pipeline. Terraform will be applied to the pipeline in future iterations such that it will deploy all the services to Azure.

2.3 Repository arrangement/organization

2.4 Applied Branching Strategy

Throughout the project a "Long-Running Branches" strategy has been used. The *master* branch has primarily been used for code releases as pull requests from *development* or `.yml` pipeline changes. The *development* branch was used to merge all the feature branches back into, so that it in turn becomes can be merged into *master* as the next stable release.

Finally the individual feature branches correspond to the issues on the task board on GitHub. Each feature branch is named after their issue number, e.g. `fea34APIController`. Feature branches are short lived, and are closed when done.

Pull requests where inspected as a team or by another developer. Automatic releases can be triggered by the CI pipeline, by adding a specific tag to a commit when merging into *master*. Releases were however often done manually, since that way we could a provide a description/documentation to it. This had its drawbacks, one being that's it's somewhat against the DevOps idea, but also that we didn't release as often as we would have liked.

2.5 Applied development process and related tools

Distributed workflow

A "Centralized Workflow" was used, meaning that there's one shared repository, that every developer can collaborate on. This seemed fitting for the size of the code base and team.

Simple Kanban board

We used Github's Projects for task identification, specification, and planning. The board consists of the traditional Kanban setup with 'tasks', 'in progress', 'parked', and 'done' as seen the fig **??**. Each week we would receive a list of tasks as an assignment for the course. Firstly, we listed them all within the 'tasks' lane. Secondly, we converted the tasks into issues and assigned group members to them. Thirdly, the tasks would be inserted into its retrospective lane. Then each Sunday we would gather up on the tasks completed on the assignment and review the code before a merge to the main branch was completed.

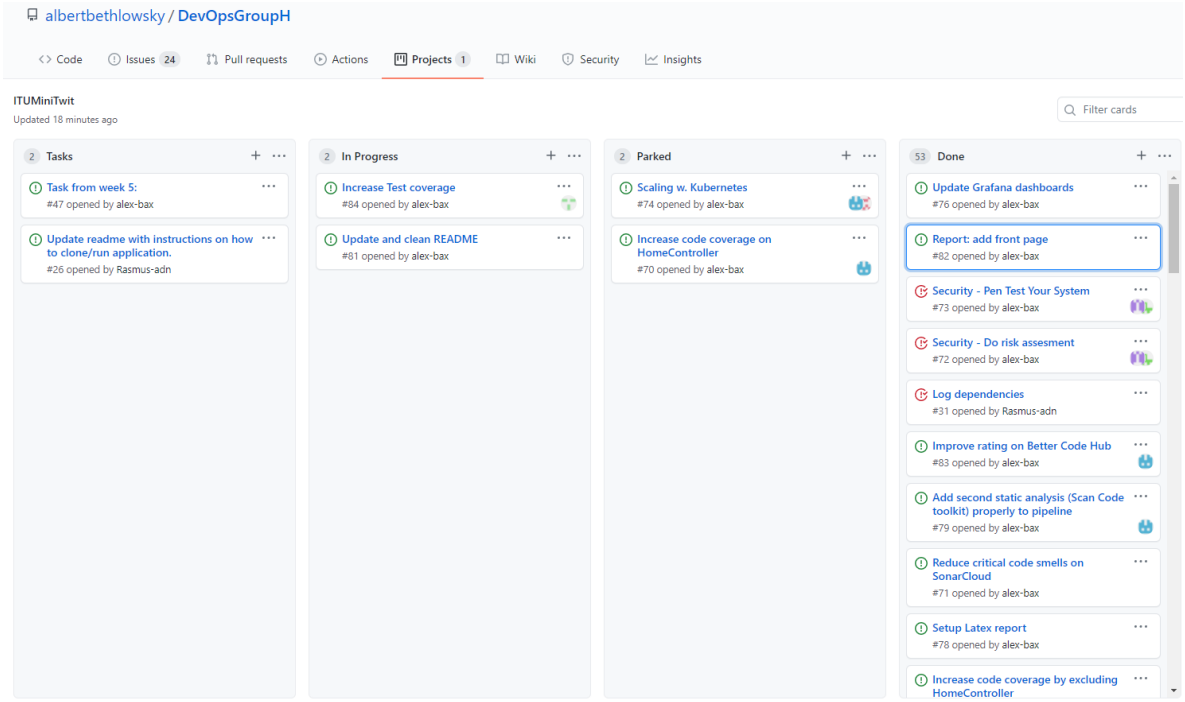


Figure 11: Github Project - Kanban Board

We decided to use Github Projects since we wanted to centralize as much of our work to one medium. Other mediums were also possible to use such as a simple excel sheet with a product backlog. Or online solutions such as monday.com, jira.com, or temgantt.com. However, we were concerned that utilizing an external solution from Github would result in less task management.

2.6 Monitoring

For monitoring Prometheus was chosen to scrape the data from the MiniTwit container, and Grafana to visualize it. More precisely Grafana pulls data from the Prometheus container regularly, which in turn pulls data from the application through the /metrics access point of the website. The Prometheus and Grafana are deployed on containers on their own App Service, separate from the App Service hosting MiniTwit.

From quick research, these tools were chosen since they have good compatibility and documentation, but other alternatives to Prometheus such as DataDog could have been chosen as well.

To supplement Grafana, for example to get insights on metrics such as RAM usage, in and out-going data bandwidth usage, thread count, etc. we used the monitoring tools available on Azure for the services hosting MiniTwit and the SQL db. These stats are especially useful to determine the general load and whether or not to scale the server to meet demand.

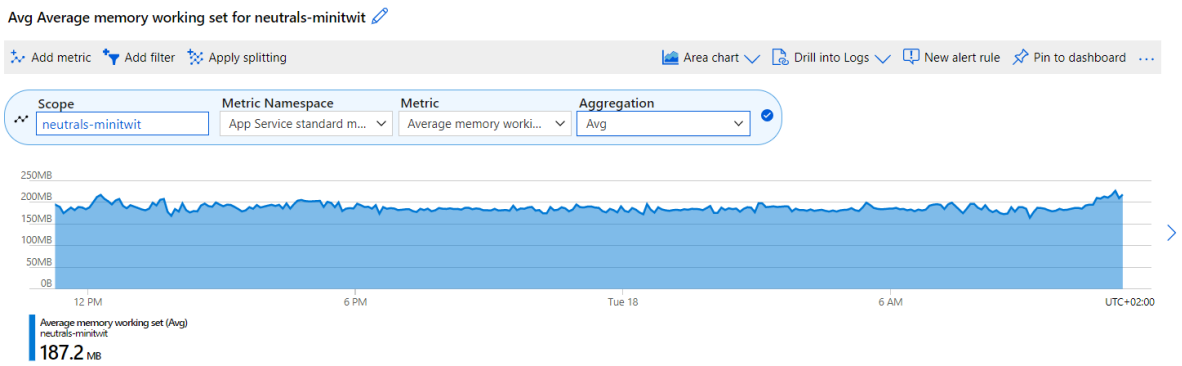


Figure 12: Memory usage of MiniTwit container from the Azure App Service

2.6.1 Dashboards on Grafana

On Grafana the *Rate of http calls* dashboard shows the number of different types of HTTP responses emitted from traffic on the endpoints over time.

For example, from inspecting figure 10: at 18.00 there was about 2.5 thousand 204 status codes sent from the server on the endpoint of the form `https://neutrals-minitwit.azurewebsites.net/msgs/Joe` triggering method `CreateMessageByUser()` in the API.

This includes the most common HTTP status codes 2xx, 3xx, 4xx and 5xx. It gives a general overview of how the server responds to the traffic, and has been useful to indicate error trends.

Other dashboards used are: *The duration of HTTP requests*, *Total no. threads* and *Request currently in progress*.

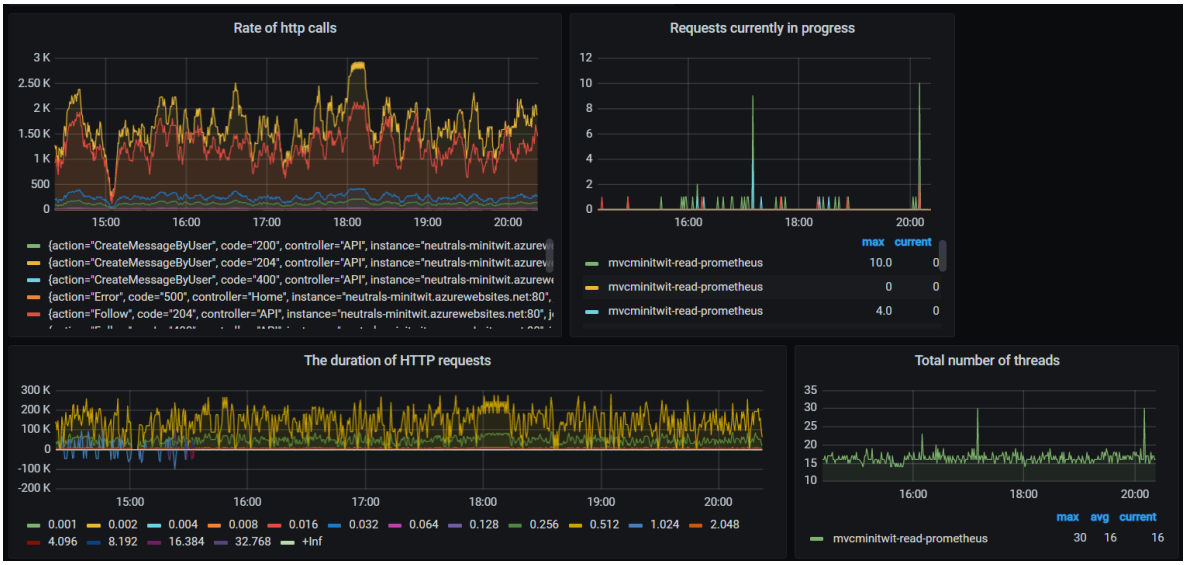


Figure 13: Grafana dashboards

In future iterations alerts from either Azure or Grafana could be considered. Also more in depth db monitoring could be implemented, since it was only shortly experimented with, tracking no. users and no. messages.

2.7 Logging and log aggregation

For purposes of logging, multiple different solutions were initially discussed within the group. The ELK stack introduced to us during the course, even though being very advanced, seemed difficult

to implement into our system in comparison to its benefits. As an alternative we investigated the possibility of using Promtail together with Loki. These softwares offers the possibilities of applying the same methodology as Prometheus, but in contrast captures logging information, diverting it to Loki which in turn can aggregate the log information into Grafana. This strategy proved more difficult than we expected, since working with Azure as our host displayed a lot of issues with opening and forwarding ports required to utilize these programs. The experimentation with these solutions however did lead us to the use of Serilog, which ended up being our chosen solution for logging. Serilog is injected on top of .NET's native logging infrastructure and provides easy enhancements to the log context in addition to log custom data points captured or generated within the system. Serilog uses a sink to continuously send log data to Datalust Seq, which is the aggreation and visualization tool we opted to use. Seq enables us to monitor multiple levels of the logged information within various timeframes that can be specified by the user. In addition to this, the "Events" tab allows us to search and filter on all the aggregated logs, providing a simple approach to identifying overall trends and specific errors which might have occurred in the system.

Working with the MVC framework, the logging we perform are primarily centered around the controller section of the framework. All calls passing through the controller is logged with elevation "Information" or higher giving information in regards to call initiation and completion. In addition to this user defined log entries are created to provide additional information within each call for better errorhandling.

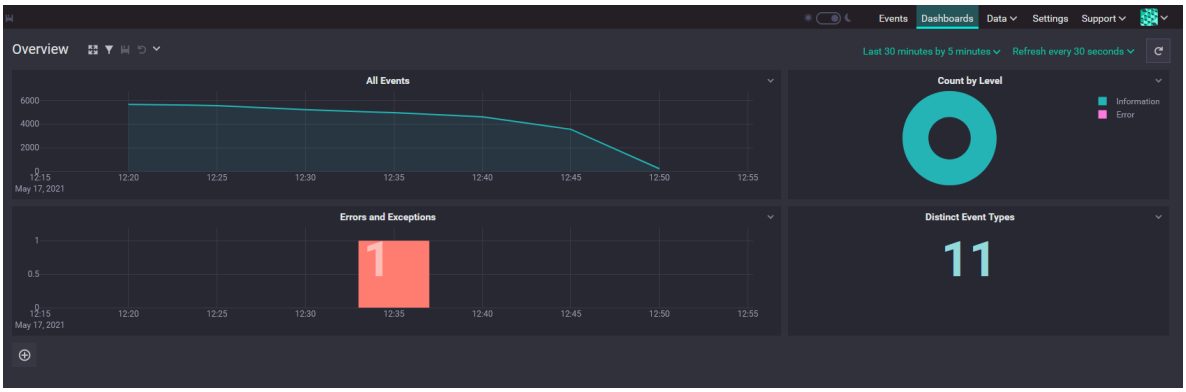


Figure 14: Datalust Seq dashboard

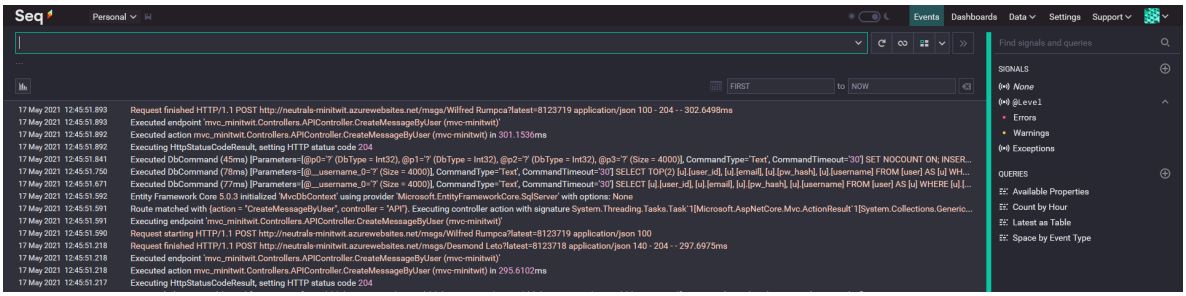


Figure 15: Datalust Seq log

2.8 Brief security assessment

While dotnet provides useful classes and services to write more secure code, a known security issue in our code is that we use 'Message-Digest algorithm 5' (MD5) for hashing the users' passwords. We are aware that MD5 is no longer considered secure as it has been cryptographically broken. Therefore using an alternative such as bcrypt or Argon2 would be a better alternative. The most secure hashing algorithm for passwords that is currently natively supported in dotnet is PBKDF2. Therefore we would have to be cautious if we were to use bcrypt or Argon2 as anyone can author a NuGet package. We consider this risk to be high. A way of overcoming the problem of the users having a password

stored with the old hashing algorithm, we would have to make a check when the user logs in and then rehash their password with the new algorithm.

Furthermore we use versions of the JavaScript Libraries, jquery and bootstrap, that have known security vulnerabilities. To eliminate any risks we would have to upgrade the libraries to the newest version available. We consider this risk to be low. While we use use LINQ for our queries with our Entity Framework the system is not vulnerable to SQL injection.[4] The Entity Framework provides standards that has generally helped us eliminate different vulnerabilities. We have prevented XSS, X-frame options and sniffing by setting security headers with the Use method directly in our Configure method in the StartUp.cs file. We have used ScanTitan[5] and Hostedscan[6] to pen test our system.

2.9 Applied scaling and load balancing strategy

We implemented azure's 'scale out' service which is a built-in feature that helps applications perform their best when demand changes. It enables our services to run on multiple Virtual Machines (three in our case) simultaneously and where the load will be balanced automatically. We applied this scale-out to multiple services running; Web App Services and Service Plans [**?**].

Moreover, we did try to establish Kubernetes services and incorporate it into our pipeline. However, we met multiple errors, both pricing tiers with azure and also authentication problems with the active directory. Simply put, ITU's active directory did not allow us to administrate Kubernetes on Azure. We also investigated the possibility of using docker swarm, but they are in a vanilla stage with azure and hence major errors occured when trying to implement. For example, this docker swarm template for azure is being linked by Microsoft and it fails almost all of their tests [**?**].

3 Lessons learned perspective

Throughout the lifetime of the project, numerous decisions have been made in good faith when considering the understanding of the overall project and it's technologies at those points in time. However as the project evolved from birth to it's current state, some of these prior decisions, showed to pose bigger obstacles for the project than anticipated at the time, as we will discuss in the following sections.

3.1 Evolution and refactoring

3.1.1 Language and Framework

Early on in the process, we quickly identified that going with .NET held major advantages in their offering of templates and wast majority of guidance online on multiple topics. The benefit of having information readily available when needing assistance, however also proved to serve as a disadvantage at certain times. The .NET framework spans across multiple iterations and versions like .NETCore, .NETFramework, ASP.NET among others. Though NuGet packages are obtainable for most of the different versions, solutions to various problems were not always applicable to our approach and desired outcome. This of course was not expected at any given time, but with ongoing research and due diligence we improved at identifying which resources were relevant and which were outdated. This experience helped clearly showcase that during a refactoring process of a "legacy" software, choice of language is only one part of rebuilding the software into a more modern solution.

3.1.2 Database Management

Though a shift from SQLite into another DBMS was clear from the start, it became clear though, not having this transition in place in a timely manner can cause great complications as we had to experience the hard way. During the early stages of the simulator things looked to be running as intended, we however quickly after that, realized this was unfortunately not the case. We identified

[4] https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/security-considerations#:~:text=The%20Entity%20Framework%20does%20not%20enforce%20any%20security%20permissions%20and,store%20and%20by%20your%20application.

[5] https://scantitan.com

[6] https://hostedscan.com

an issue with the API not posting certain messages and successfully following and unfollowing other users. Upon fixing the issue it occurred to us, we had mistakenly locked the database into the container image, thereby not being able to access any of the data stored from outside the system. After countless of hours spent trying to mitigate this error, we had to admit defeat and perform our migration into another DBMS with backup data from another group. This allowed us to carry through with the necessary fixes for the API and move on with the project. It was blatantly clear after this incident that decoupling of the software and the database is a high priority, or as a minimum ensuring proper access to it's content before going live. Since we are simulating a real life environment, this action resulted in the deletion of all users and messages which is far from ideal, since a backup from a third party under normal circumstances is highly unrealistic.

3.2 Operation

3.2.1 Foreshadowing

We define operations as the completion of the weekly assignments. Many struggles occurred at almost each weekly assignment due to steep learning curve. Many of us have not used many of the technologies suggested and therefore many obstacles had to be conquered. In addition to lacking required knowledge of the technologies we needed to implement, not taking into consideration the implementations of future features, occasionally lead to discrepancies in our system. Initially certain functionality was implemented on standalone basis, which worked for certain aspects of the system, but in time we got better at foreshadowing, basing technology and implementation decisions not only on our current tasks, but also tasks to come. Working agile holds huge advantages to being flexible during development, but without proper organization of items can also lead to mismanagement of time and resources. Overall having a filled out Kanban board or similar, with all development items to come, could have helped in guiding our decision making in the early stages of the project.

3.2.2 Azure

Azure as a service provider comes with countless baked in solutions supporting deployment, monitoring, health checks, logging and more, providing easy setup and maintenance of simple solutions. In regards to customization of our deployment however, Azure did prove to be very difficult to work with. Azure is comprised of many tiers of pricing, granting access to various possibilities within their services. Being on a budget and on an "Azure for Students" license, made of some of the weekly requirements for the course, very difficult to complete. One of the most noticeable obstacles worth to mention is the inability to access certain ports. Azure as a default provides www.xxx.azurewebsites.net as access points, not enabling port specification when accessing the website. This caused a lot of issues in instances where multiple technologies needed to be embedded into a docker compose file. Ultimately in rendered us unable to run our program together with Prometheus and Grafana as an example. With some tinkering we did discover the possibility of layering images on top of each other in situations where only one image needed to be displayed for usage. In the example of Prometheus and Grafana, putting Prometheus as the first image of the compose file allowed us to access the interface and confirm that traffic was flowing through correctly. Afterwards we rearranged the compose file putting Grafana as the first image, granting access to our monitoring service, while still having Prometheus functioning as intended in the background, though not accessible. These kinds of tweaks were required on multiple occasions, causing a lot of headaches during development and testing of different aspects in the system. Having to make these workarounds, considering the excitement of discovering the possibility to do so, in the end proved to cause major setbacks throughout the entire project. Trying to scale our tier to overcome these issues rapidly bleed out our funds and was therefore not deemed a suitable option. As a conclusion, Azure would probably not be a desirable choice on other development projects of this scale and complexity, where multiple technologies needs to be accessible, ideally through the same website / server.

3.2.3 Infrastructure as Code (IaC

We tried to implement Terraform as part of our CD/CI, but we met multiple issues. We did manage to establish a working replica our Azure resource group with resources through Terraform together with a working Azure DevOps CD/CI pipelines. However, we didn't manage to secure our various

passwords and connection strings and we experienced issues with the Terraform statefile. A small typo error in the Terraform file resulted in an error which then lead to our the statefile, which is stored in a blob storage on azure, to become desynchronized, forcing us to import every single resource into a new statefile. Due to these obstacles, we decided to postpone the implementation of integrating IaC as part of our CD/CI.

3.3 Maintenance

As earlier mentioned, logging and monitoring was used to monitor the behavior of our application from where we applied maintenance accordingly. We fixed a long list of code bugs and smells with the help of the static analysis tool SonarCloud. We also increased the coverage of our code with tests of our home and API controllers. As mentioned earlier, one of our biggest struggles was to get the simulator to work on our application. As seen in fig **??**, we were the group with the largest amount of http errors.

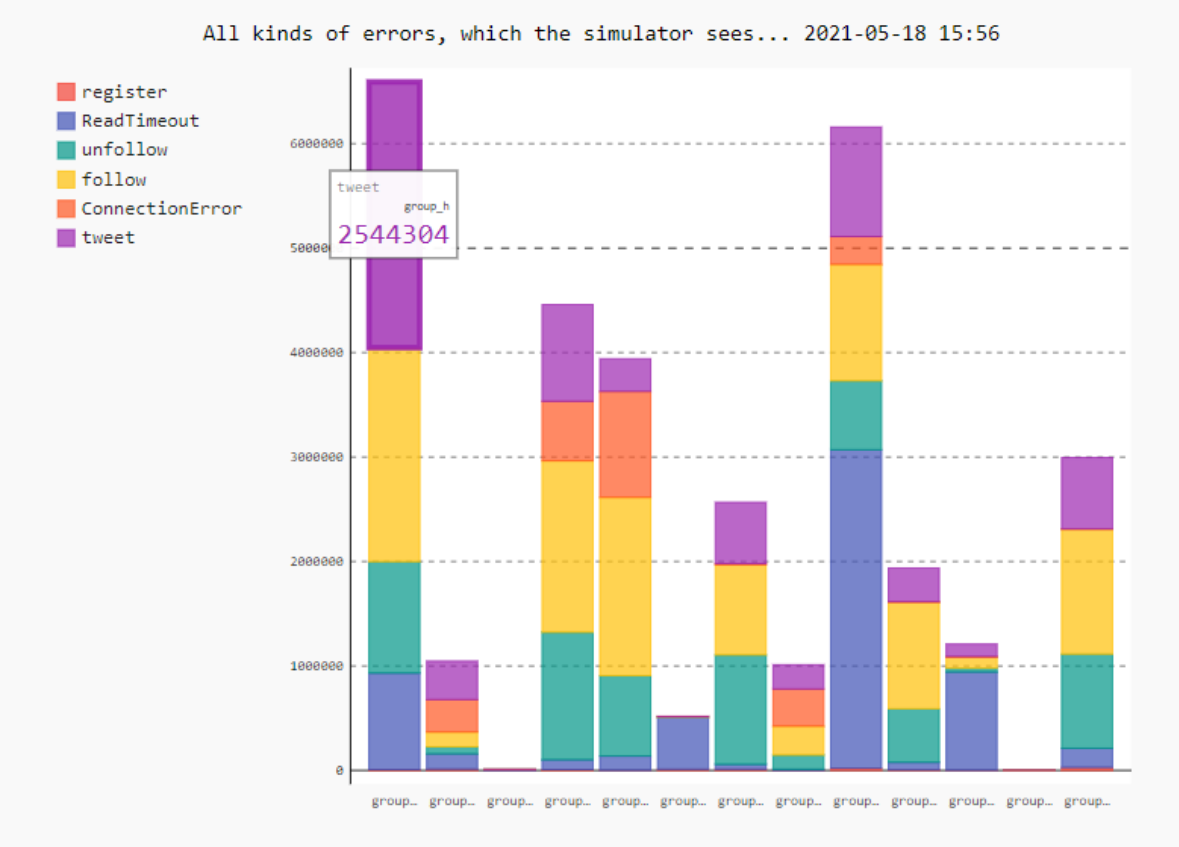


Figure 16: Simulator Errors

One reason to the large amount of http errors being created was, because of our pricing tier in Azure. We solved it by vertically scaling up our database tier allowing for more Database Transaction Units (DTU's). Another reason was responding with the wrong response code for quite some time. Another reason was that our API controller functioned very poorly in handling http requests. To solve this the http requests were divided up into simpler decoupled queries to avoid duplication of database calls, which ultimately optimized the performance.