DevOps, Software Evolution and Software Maintenance (BSDSEMSM1KU)

Group H - Neutrals

Alekxander Baxwill (abax@itu.dk)

Jacob Sjöblom (jsjo@itu.dk)

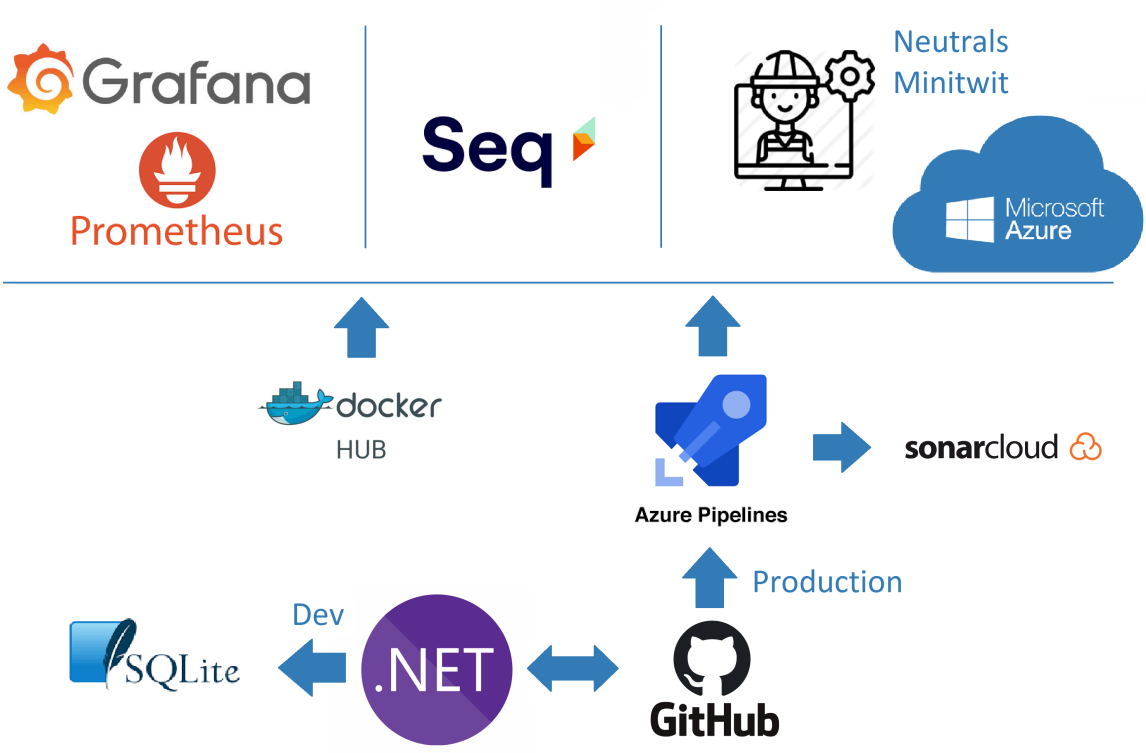
Frederik Peter Volkers (frvo@itu.dk)

Albert Bethlowsky Rovsing (arov@itu.dk)

Rasmus Andreas de Neergaard (rade@itu.dk)

Repository: https://github.com/albertbethlowsky/DevOpsGroupH

May 19, 2021



Contents

1 System's Perspective

1.1 Design and Architecture of the system

In the beginning we decided on rebuilding the solution with the language C#. As an alternative Java was also discussed, since everyone in the group have previous experience with it. We decided on C# over Java to give ourselves 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 a good addition to our future job aspects.

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

For the backend we decided on ASP.NET Core MVC which uses the Model-View-Controller architectural pattern. For the object relational mapper (ORM), Microsoft's Entity Framework Core (EF Core) was used. Other common options are NHibernate or Linq2Db. EF Core has great documentation, was simple to integrate and is popular with .NET web applications. In relation to choosing the DBMS, MySQL was considered due to it being open source and popular in general. We however opted for Azure SQL Server (db as a service) because it's easy to scale, setup and comes with neat features such as AI optimised queries etc.

For an overview of the architecture and design of the MiniTwit application, the 3+1 approach is used.

Module View

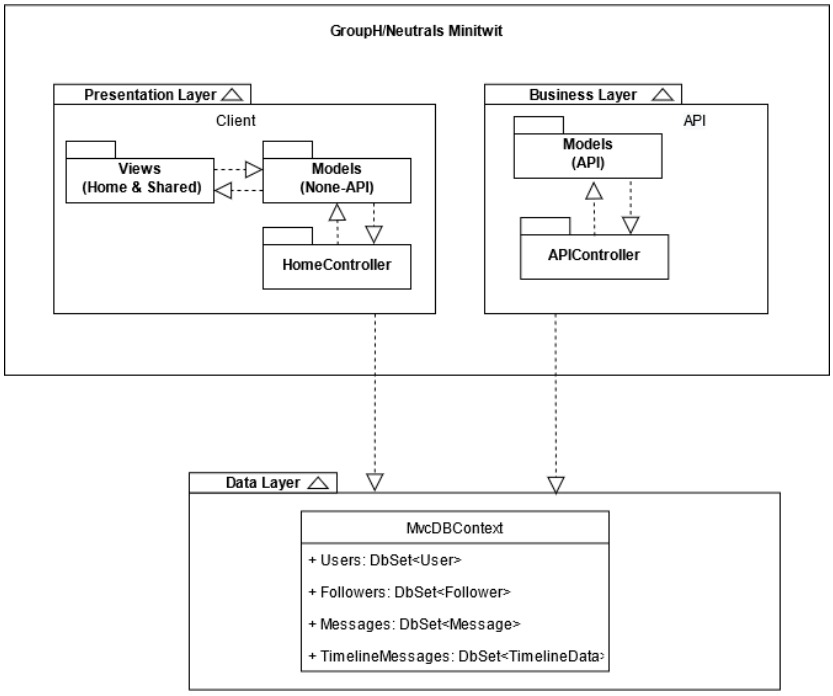


Figure 1: Module View

Fig. **??** illustrates the relations between the packages at design time. Presentation Layer highlights the flow of information via the MVC model based on user interaction. User input in any view is translated into a model object, fed into the HomeController. The targeted action is processed by the controller and output is distributed back to the view via a model object, providing a dynamic user interface, which is not included in the Module view. The Business layer is built on the same principle, but does not include a view. For data retention, model objects are translated to SQL and executed by EF Core on the db.

Component & Connector View

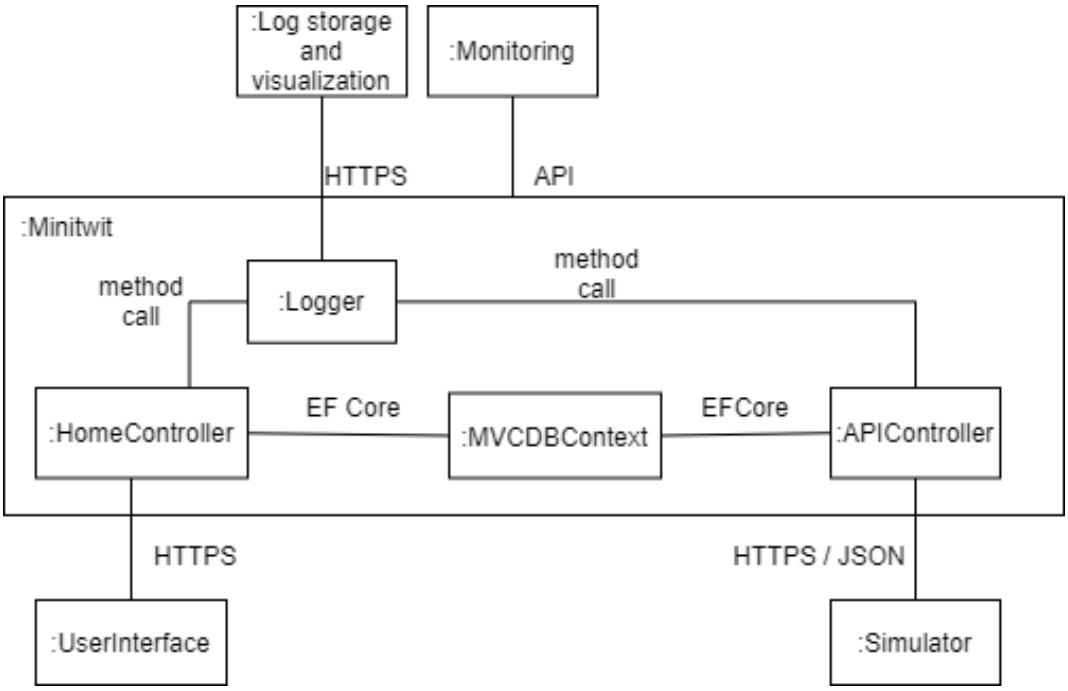


Figure 2: Connector and Component View

Fig. **??** illustrates how the system interacts between its components at runtime. It is worth mentioning that Logger is a default extension of the framework where Serilog is injected on top at startup.

Deployment View

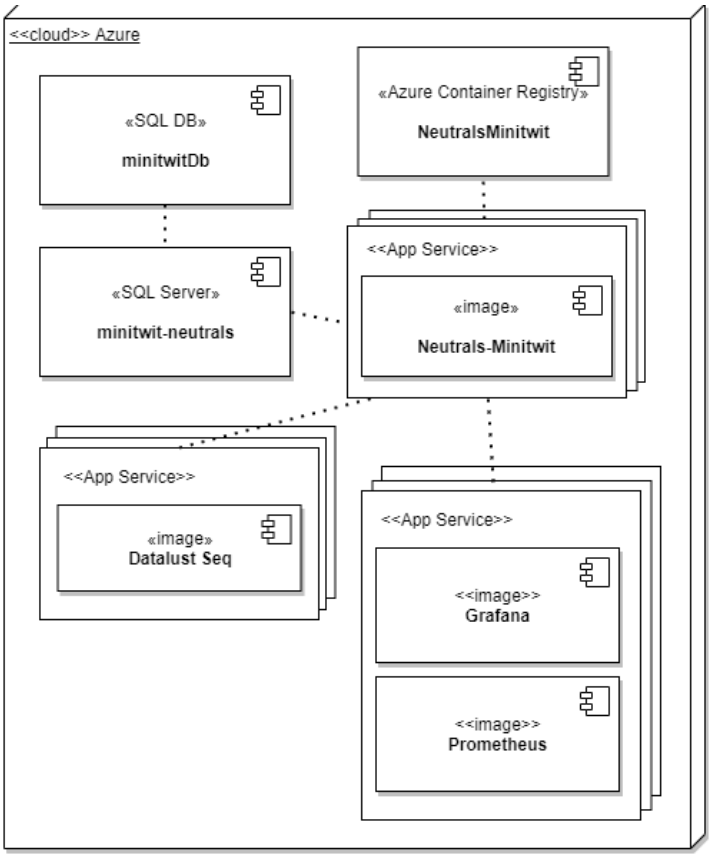


Figure 3: Deployment View

Fig. **??** illustrates the system at it's deployed state.

- ≪cloud≫: Refers to the hosting of our services through Azure.
- ≪App Service≫: Service hosting the images
- ≪Azure Container Registry≫: Repository for the NeutralsMiniTwit images
- Neutrals-Minitwit image: The front- and back-end (API and server) of MiniTwit.
- Grafana image: Collects PromQL data from Prometheus, enabling visualization of metrics data.
- Prometheus image: Collects the exposed metrics of MiniTwit, and translates it into PromQL readable by Grafana.
- DataLust Seq image: Aggregates and stores logs sent from MiniTwit hosted in an App Service.
- ≪SQL Server≫ and ≪SQL DB≫: Azure hosted SQL Server with underlying database, used for persisting data used by the system.

NB: Worth noting, the App Services Neutrals-MiniTwit, Datalust Seq and (Grafana and Prometheus) are part of two separate service plans which are balanced between 3 VM's each.

1.2 Dependencies

1.2.1 Software dependencies

The dependencies of the system are derived via GitHub from the `mvc-minitwit.csproj` and `HomeControllerTests.csproj` project files, as seen in fig. **??**. These correspond to the MiniTwit and testing project respectively [**?**].

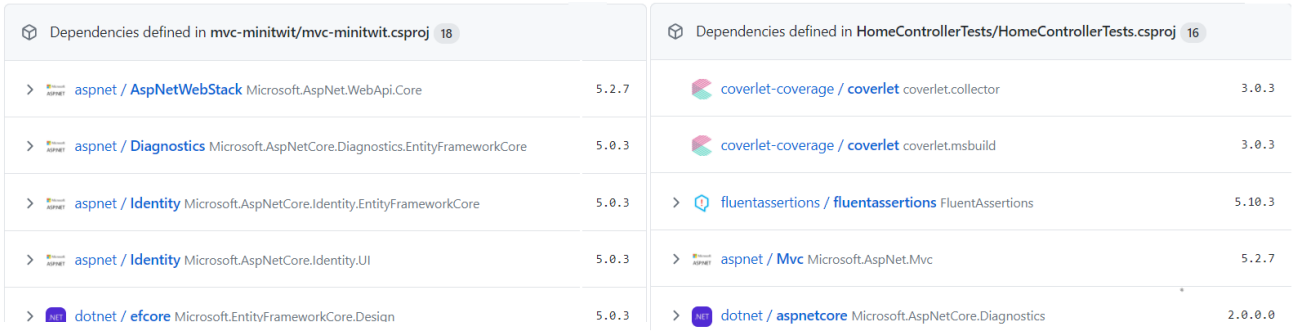


Figure 4: Snippet of GitHub dependency graph

Additionally, using NDepend, we can provide a more detailed visualization of the dependencies. It is a static analysis tool which generates 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) [**?**]. This matrix does not include test dependencies. See GitHub repos for full image.

Figure 5: Snippet of Ndepend - Dependency Matrix

1.2.2 Cloud dependencies

These non-repository 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 stated in following sections.

SonarCloud

As observed from Fig. **??**, SonarCloud 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.

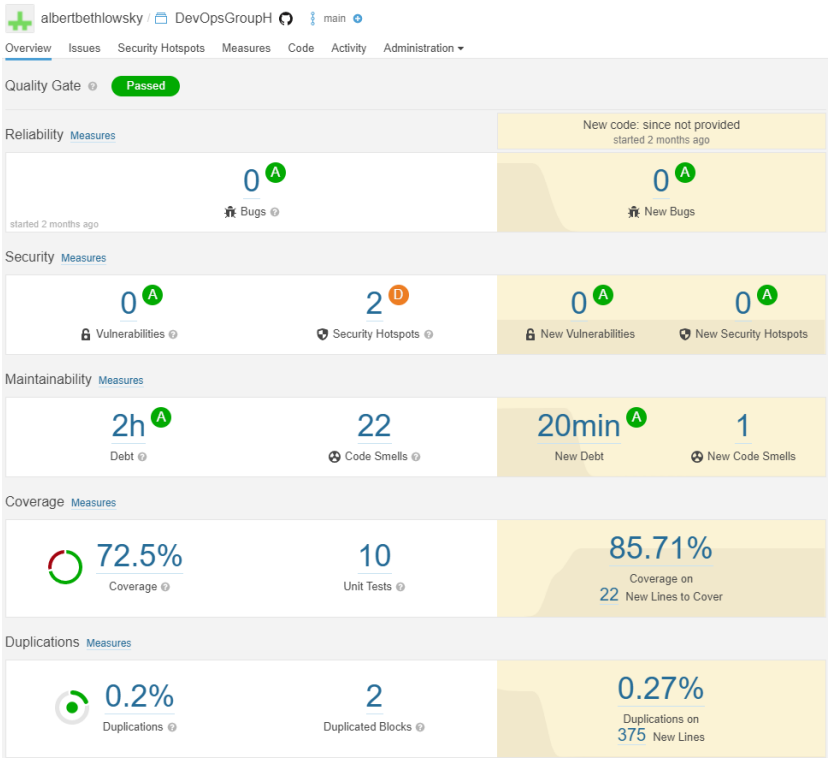


Figure 6: Results from SonarCloud

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 estimates from the other tools.

Code Climate

The repository has a maintainability grade of "B" as seen in Fig. **??**, 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"*. Arguably, this could be related to the debate of "clean OOP" approach, which is harder to implement in a server/API application.

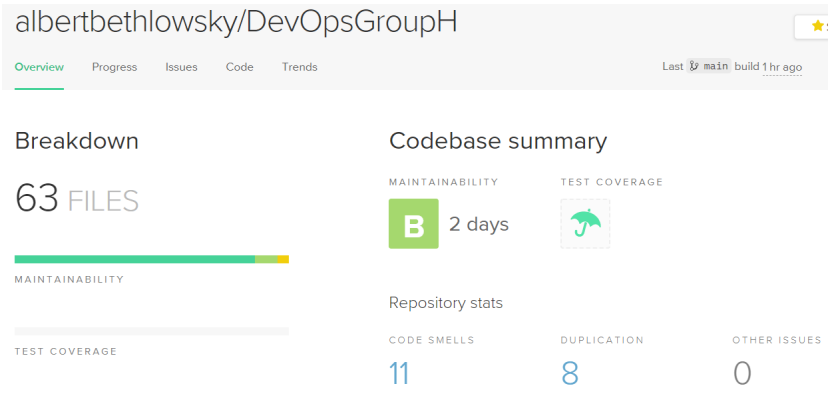


Figure 7: Code Climate summary and overview

Better Code Hub

We can observe from Fig. **??**, that our compliance is 7/10 and the code smells are similar to previous. We acknowledge that our lack of experience with C# has some effects on our coding style.

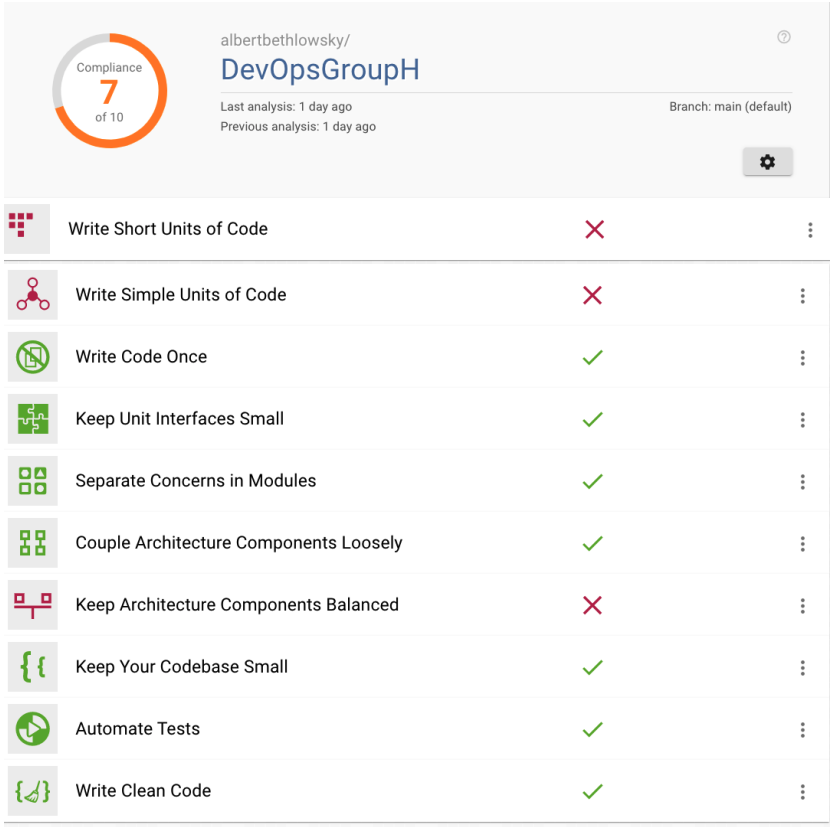


Figure 8: Better Code Hub results

It's worth noting 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 [**?**], the MIT license [**?**] and the BSD-3 clause [**?**]. All of these licenses fall within the category of permissive licenses. The following table **??** shows 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		

All licenses impose no restrictions for us as we do not modify the source code of the dependencies. We

have chosen Apache 2.0 as our license, since all licenses fall into the same category, and Apache 2.0 being most used.

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 fully. We strove towards having two weekly meetings on Mondays and Wednesdays. The communication and meetings were held online exclusive on Teams, and in each meeting we planned and presented progress. We assigned ourselves to tasks using GitHub Project Management. Often times team members would collaborate and do pair/group programming via VSCode Live Share.

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 hosting on Azure. As an example, certain services (Azure Kubernetes Services, Azure App Service for deployment, etc.) allowed for direct yaml-template injection via the marketplace on Azure DevOps.

There are two pipelines; the CI pipeline used for testing, code analysis, and building as seen in Fig **??**. And the CD pipeline for deploying as seen in Fig. **??**.

Continuous Integration

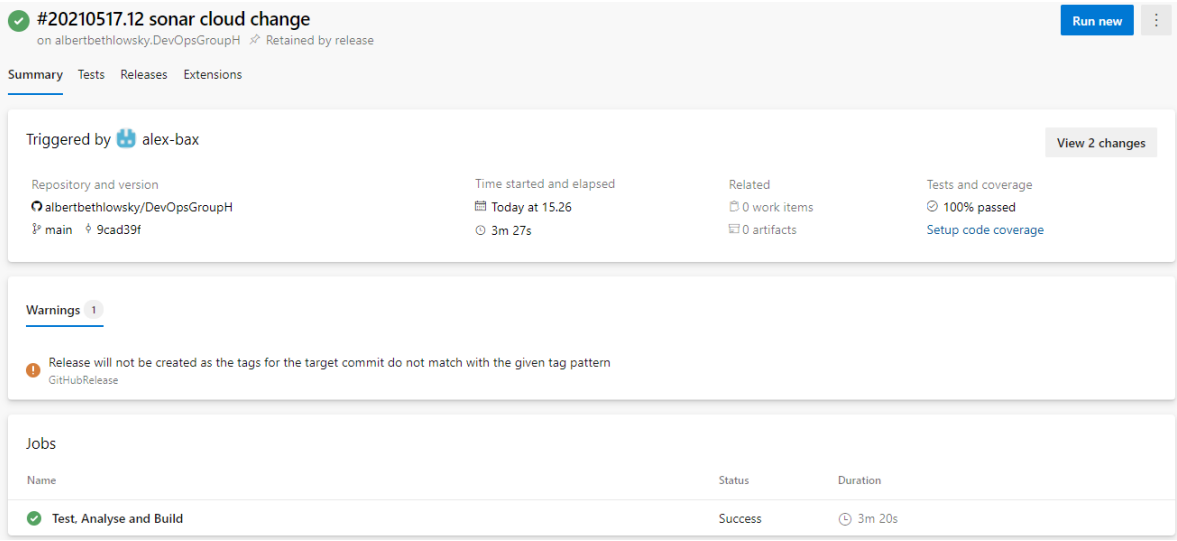


Figure 9: Result of the CI chain from `azure-pipelines.yaml`

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

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. No automatic git branch-merging tasks have been added.

Continuous Delivery

The CD pipeline, as seen in Fig. **??**, 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 and fits for a future implementation of Kubernetes.

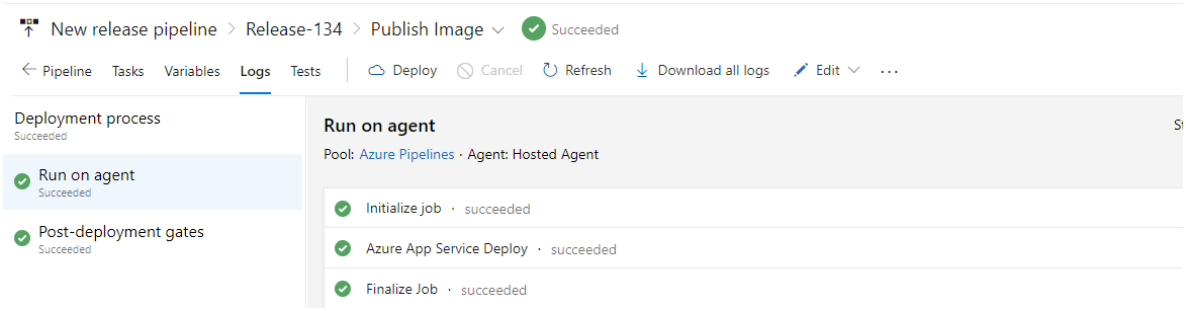


Figure 10: CD pipeline

After having pulled, a post deployment gate is checked, in this CD chain the status of a SonarCloud Quality Gate as seen in Fig. **??**. 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.

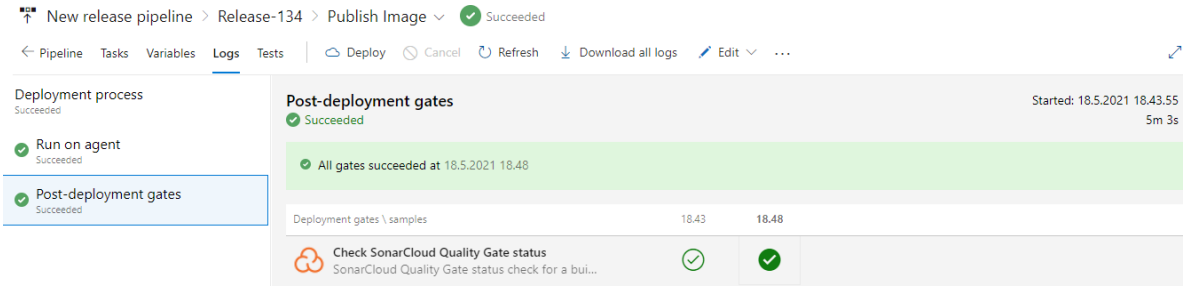


Figure 11: Passing SonarCloud Quality gate on the CD pipeline

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 Applied Branching Strategy

Throughout the project a "Long-Running Branches" strategy has been used. The *main* branch has primarily been used for code releases as pull requests from *development* or `.yml` pipeline changes. Smaller feature branches were merged into a *development* branch, that in turn was merged into *main* at the next stable release.

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 deleted when done.

Pull requests where inspected as a team or by another developer. Releases were often done manually, since that way we could a provide a description/documentation to it. This had its drawbacks, one being that it's somewhat against the DevOps idea, but also that we didn't release as often and consistently as we would have liked.

2.4 Applied development process and related tools

Mono-Repository

We make use of a 'mono-repository', meaning that there's a unified source-code 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. First, 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.

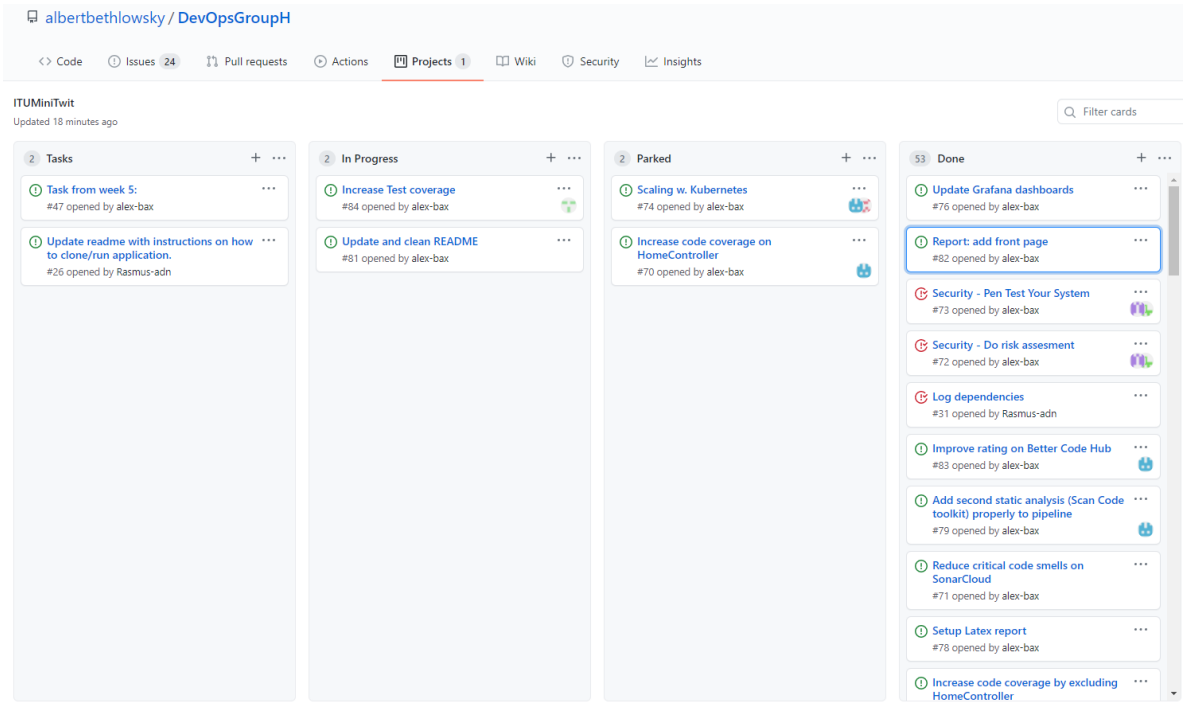


Figure 12: Github Project - Kanban Board

We decided to use Github Projects since we wanted to centralize our tasks in one place. Other mediums were also possible, such as monday.com, jira.com, or temgantt.com, DevOps, etc. However, we were concerned that utilizing an external solution from Github would result in poor task management.

2.5 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 images are deployed in a container 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 also have been chosen.

To supplement Grafana, 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 as seen in Fig. **??**. These stats are especially useful to determine the general load and whether or not to scale the server to meet a demand.

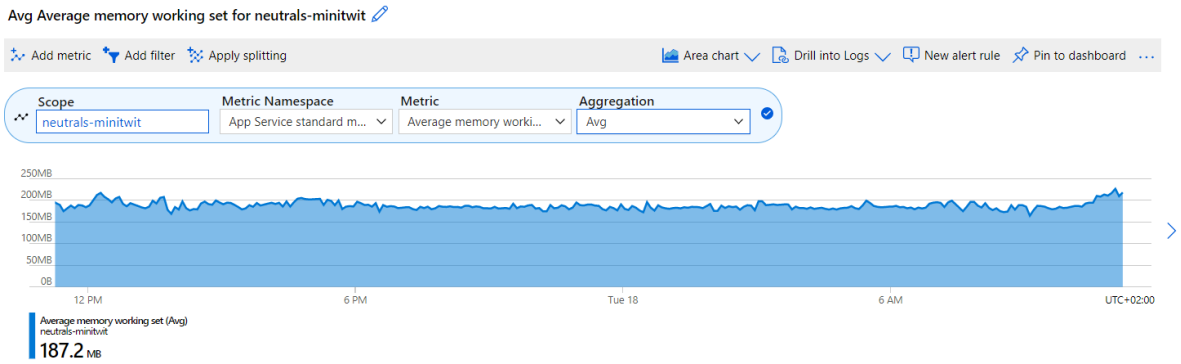


Figure 13: Memory usage of MiniTwit container from the Azure App Service

2.5.1 Dashboards 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 Fig. 4, 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*.

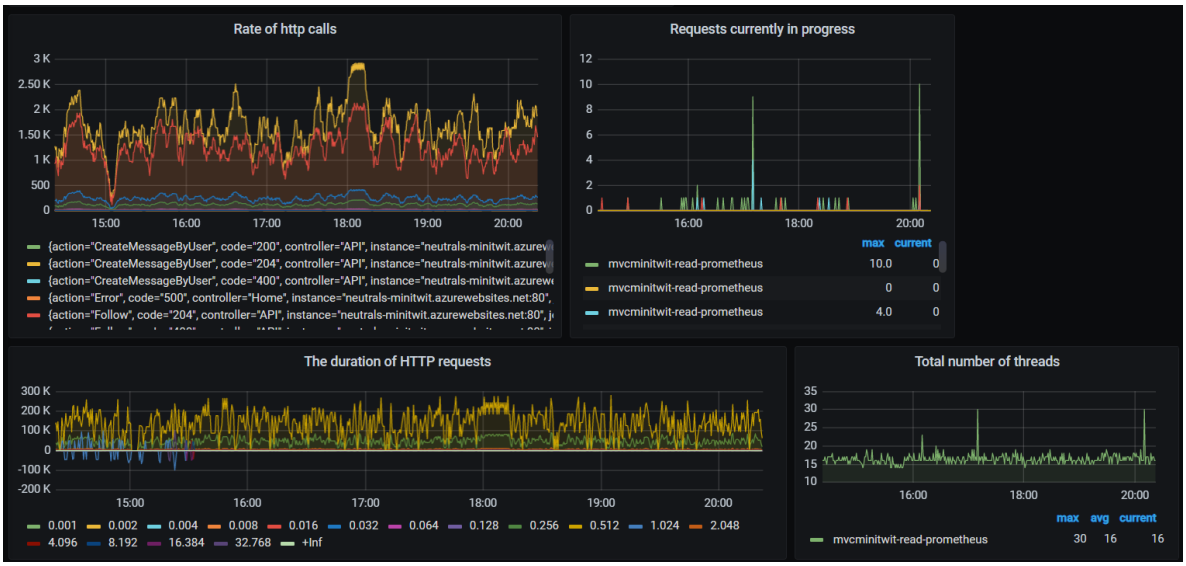


Figure 14: Grafana dashboards

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.

2.6 Logging and log aggregation

2.6.1 Selecting a Solution

For purposes of logging, multiple different solutions were initially discussed, one being the ELK stack introduced during the course. Even though it's very advanced, it seemed difficult to implement into our system in comparison to its benefits. As an alternative we investigated Promtail together with Loki. These solutions apply 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.

Testing this proved more difficult than expected, since working with Azure as our host gave issues with opening and forwarding ports required to utilize these programs. In the end we decided on Serilog with Seq, further described below.

2.6.2 Serilog

Serilog, shown in Fig. **??** is injected on top of .NET's native logging infrastructure and provides simple enrichment of the log context in addition to enabling logging of custom data points captured or generated within the system. Serilog uses a sink to continuously send log data to Datalust Seq, which is the aggregation and visualization tool used. The "Events" tab enables 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.

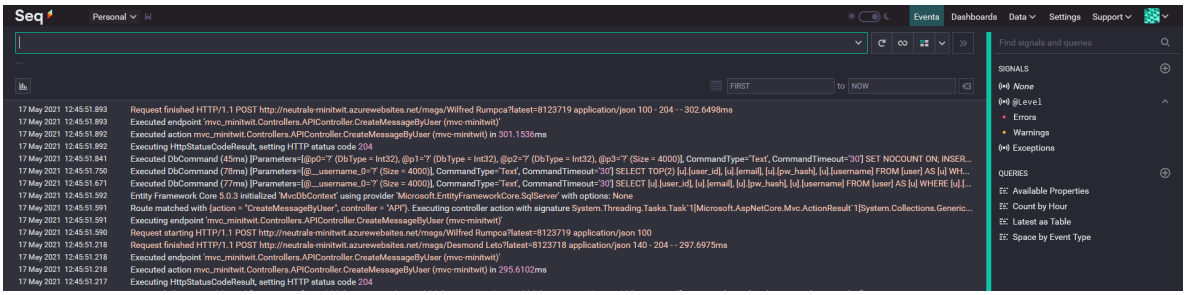


Figure 15: Datalust Seq log

The logging we perform is primarily centered around the controllers in the system. All calls generates information in regards to actions performed, in addition user defined log entries we created to provide additional information within each call for better error-handling.

Figure 16: Datalust Seq dashboard

2.7 Brief security assessment

While dotnet provides useful classes and services to write more secure code, a known security issue is that we use MD5 for hashing users passwords. MD5 is no longer considered secure as it has been cryptographically broken. Therefore using an alternative such as bcrypt or Argon2 would be safer. The most secure hashing algorithm for passwords that is currently natively supported in dotnet is PBKDF2. To solve this security concern, we could make a check when the user sign in and then rehash their password with the new algorithm.

Furthermore, we use jQuery and bootstrap, that have known security vulnerabilities. To eliminate any risks, they would have to be upgraded to the the newest version available. We consider this risk as low. While we use LINQ for our queries with our Entity Framework, the system is not vulnerable to SQL injection [**?**]. The Entity Framework provides standards that have 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 [**?**] and Hostedscan [**?**] to pen test our system. Our security assessment can be found in 'SecurityAssesment.md'.

2.8 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 according to Microsoft [**?**]. It enables our services to run on multiple VMs (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 this 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 occurred 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. Some decisions did however result in unforeseen issues, most notable described below.

3.1 Evolution and refactoring

3.1.1 Language and Framework

Early on in the process, we identified that going with .NET held advantages in their offering of templates and wast majority of guidance online on multiple topics. 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 the system.

3.1.2 Database Management

Though a shift from SQLite into another DBMS was clear from the start, it became obvious, not having this transition in place in a timely manner can cause great complications as we experienced the hard way. During the early stages of the simulator things looked to be running as intended, we however quickly observed that this was unfortunately not the case. We identified an issue with the API unsuccessfully posting certain messages, together with following and unfollowing of users. Upon fixing the issue, 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 many 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. Hereafter, it was clear 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 many of the weekly assignments due to a steep learning curve that had to be conquered. With lacking knowledge of some of the technologies we needed to implement, most viable solutions where not always chosen, which occasionally lead to future 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.

3.2.2 Azure Limitations

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, despite 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 of our Azure resource group through Terraform together with a working Azure DevOps CD/CI pipelines, as seen in Fig. **??**. 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.

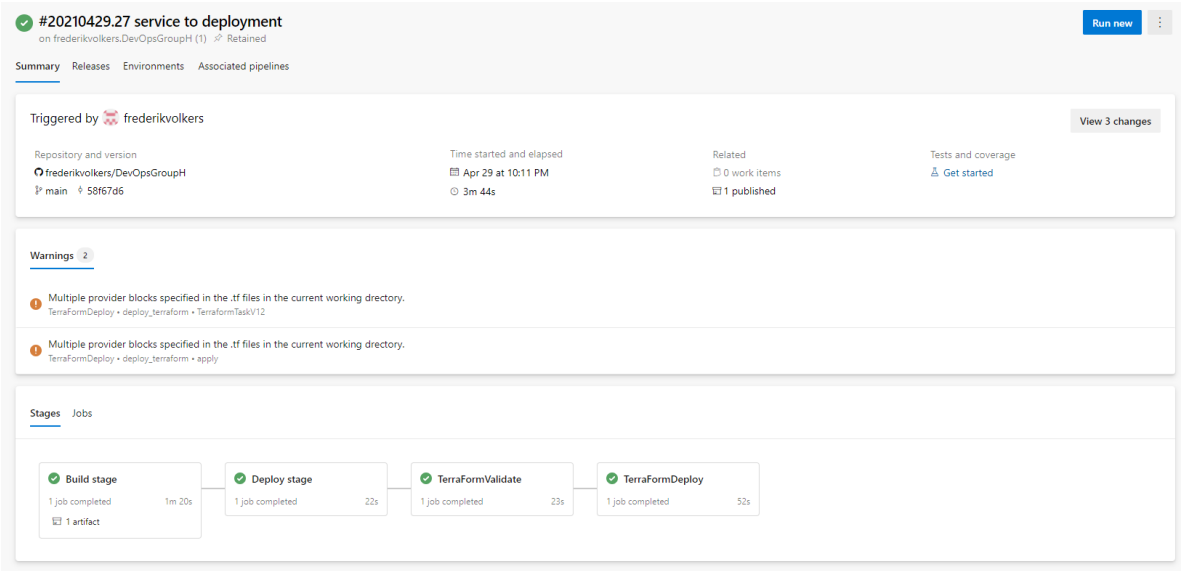


Figure 17: Terraform Azure DevOps CD Pipeline

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 various static analysis tools e.g. 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 send the correct status codes for the simulator. As seen in fig **??**, we were the group with the largest amount of http errors.

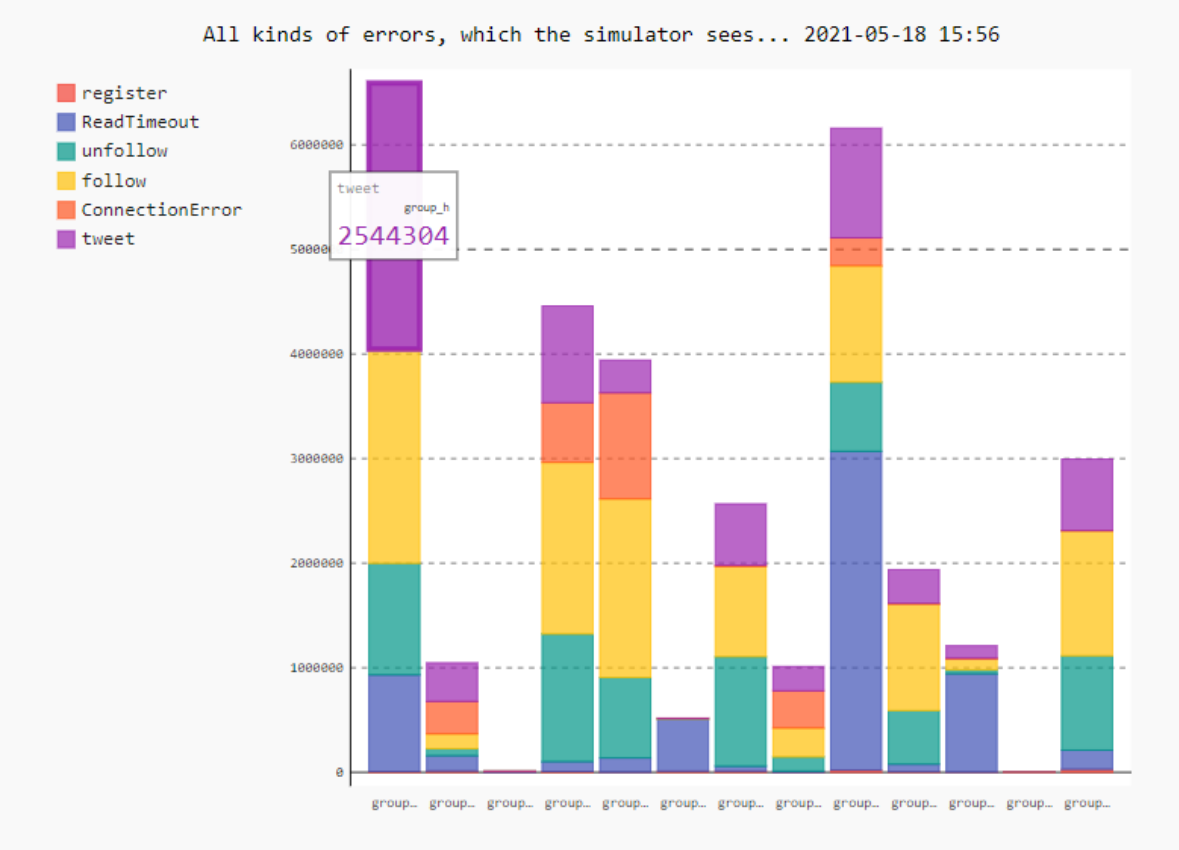


Figure 18: Simulator Errors

One reason for the large amount of http errors being created was, because of the limitations of our basic pricing tier in Azure. We solved it by vertically scaling up our database tier allowing for more Database Transaction Units (DTU's). Other reasons included: responding with the wrong response code for some time, and our API controller functioning very poorly in handling http requests. We solved the latter, by optimizing queries.