MiniTwit Project of DevOps 2021

DevOps, Software Evolution and Software Maintenance Copenhagen IT University Denmark

Einar Klarlund, eikl@itu.dj Malou Elmelund Landsgaard, ella@itu.dk
Daniel Guldberg Aaes, daaa@itu.dk Emil Dichmann, emdi@itu.dk
Jonas Aagaard, joaa@itu.dk

17 May 2021

Contents

System Design

Minitwit is a social media application that provides basic Twitter-like services. It consists of a web app and API services that are publicly available on the internet. Both services allow the user to register a profile, log in, create messages (tweets), follow and unfollow users. Basic authentication is required when creating messages, following, or unfollowing. Most of the application is written in Python since our web app and API uses the Django framework.

One of the first things that our application needed was a database to store user info and posts. We chose to use a containerized PostgreSQL server.

Minitwit also consists of monitoring tools, which the web app and API communicate with when specific metrics are updated. These tools include Prometheus and Grafana, which allow for the collection and displaying of metrics, respectively. The monitoring tools are handy for us developers since they help us to maintain the system properly.

For logging features, we have implemented an EFK stack that includes Elastic Search, Filebeat, and Kibana. Filebeat is responsible for harvesting the data that we want to log, while Elastic Search is used to store that data in a database. With the logging features implemented, it is much easier for developers to diagnose and debug problems with the system. The logging is absolute and allows perfect reproduction of requests from users to the front-end and backend due to the entire request body and the most significant features of the request header being logged.

Architecture

Minitwit is hosted on multiple Digital Ocean droplets, which form a Docker Swarm. Our logging and production database are each containerized on their separate Droplets and separated from the swarm to isolated these two systems from the systems exposed to the end-user. It allows us to easily horizontally scale everything besides our persistent data, which should not be horizontally scaled in this scenario. However, if this were supposed to run in a production environment, it would make sense to let Digital Ocean take care of the horizontal scaling of our database instead of maintaining it ourselves and set up a High Availability and Load Balancing for our db, which includes maintenance of the Master DB and all of the Slaves.

With a project that requires a web app, database, and API, it is normal to have the web app communicate with the database through the API. However, Django is designed such that direct communication with a database backend is much simpler to implement than communication with a custom backend server.

For this reason, our web app and API don't communicate with one another, and therefore don't form a frontend/backend structure. Instead, our database is our backend, and our web app/API servers are our frontends.

Django has an integrated Database API allowing both querying, creating, updating and deleting sql entries with connected database instances. Defining the tables within the Django Framework allows Django to both create and manipulate these tables. Additionally it also allows Django to export it's basic tables required for authentication management along side other basic functionalities, instead of storing it in the RAM. It also simplifies the required

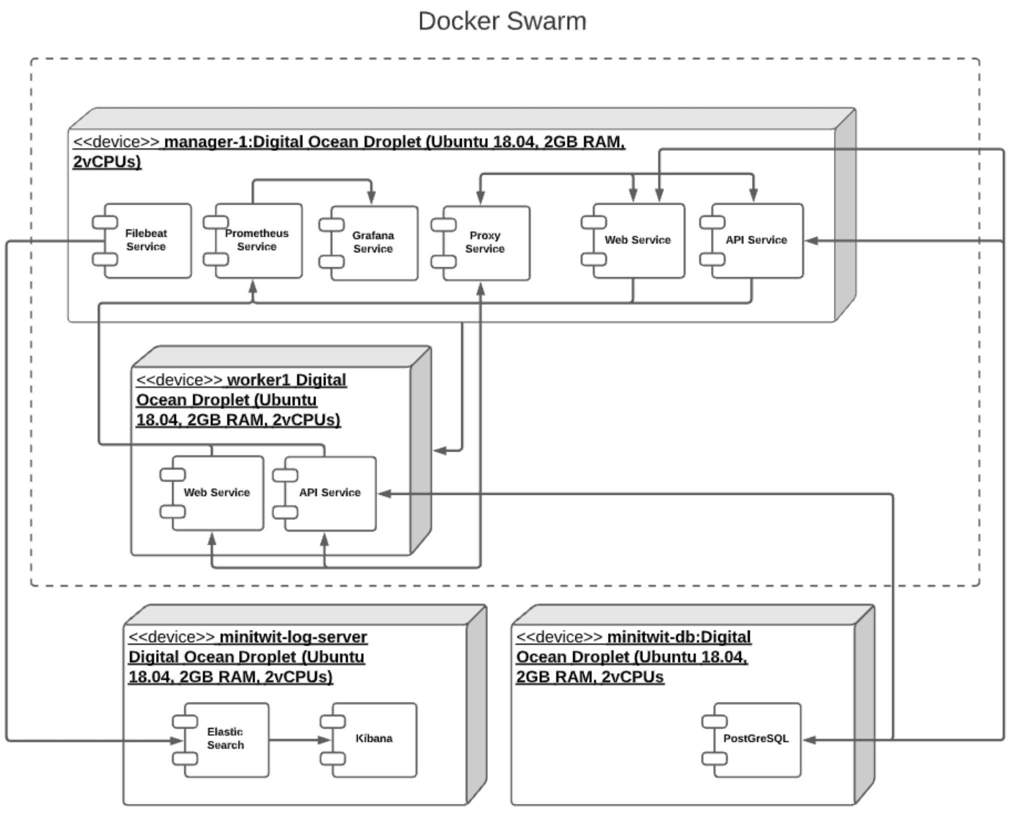
Docker Swarm
<<device>> manager-1:Digital Ocean Droplet (Ubuntu 18.04, 2GB RAM, 2vCPUs)
Filebeat Service
Prometheus Service
Grafana Service
Proxy Service
Web Service
API Service
<<device>> worker1 Digital Ocean Droplet (Ubuntu 18.04, 2GB RAM, 2vCPUs)
Web Service
API Service
<<device>> minitwit-log-server Digital Ocean Droplet (Ubuntu 18.04, 2GB RAM, 2vCPUs)
Elastic Search
Kibana
<<device>> minitwit-db:Digital Ocean Droplet (Ubuntu 18.04, 2GB RAM, 2vCPUs
PostGreSQL

Figure 1: Deployment diagram of docker swarm setup

syntax to both query and add to the tables themselves resulting in much more simple code. However doing it like this where we had to separate the API and the web application introduced some unnecessary complexity where we needed one of the application to take care of database migrations and both of the systems must have the same models for the database to not overwrite the tables in the database.

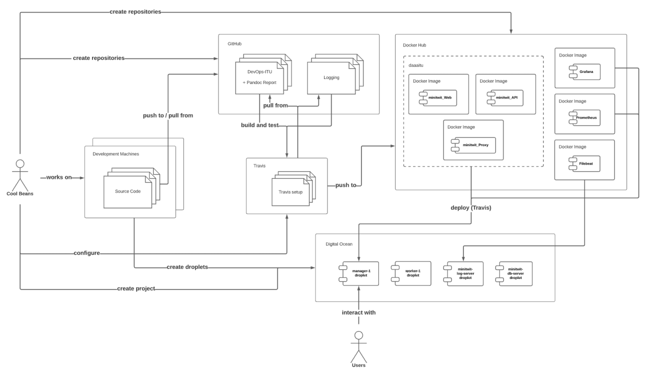

Figure 2: Subsystem diagram

We have a reverse proxy server in front of the web application and the api which routes the incoming traffic from `minitwititu.xyz` and `api.minitwititu.xyz`. The reverse proxy also acts as a loadbalancer where it will redirect a new connection to the module with the least current connections. In this way we can add ass many `worker-(n+1)` nodes to the system and `nginx` will take care of the load-balancing.

Our logging is accomplished by our Filebeat service, along with a logging database that hosts an Elastic Search instance. Filebeat scrapes the swarm manager's output, including all standard output for all services in the stack, and then logs relevant data in the logging database (Elastic Search). Kibana uses this logging database to display our log information in a neat and readable website.

Our monitoring is accomplished by Prometheus, which exposes our metrics on minitwitwitu.xyz/metrics. Our web app and API both make calls to update specific Prometheus metrics, and Prometheus gathers other performance-related metrics from both of them. The /metrics route is also checked by our Grafana service, which hosts a webpage where the metrics can be monitored through customizable dashboards.

As seen in the deployment diagram `prometheus`,`grafana`,`filebeat` and our `reverse proxy/load balancer` are only running on the `manager-node`, and each `node` in the swarm are limited to a single replica of the API and Web. So to upscale the system we need to configure the `Vagrant` file in our repository and increase the amount of `worker-nodes`, increase the amount of replicas in the `remote_files/stack.yml` which will trigger a new deployment through travis if a pull request to main is made. The manual process here is that we need to manually shh to the new worker and connect it to the swarm, this can be done with a simple shell script much like this

```bash
#!/bin/bash
# N is the number of workers.

for ((i = 1; i <= N; ++i)); do
```

```
    vagrant ssh "worker-$(i)" -c "docker swarm join \
    --token $(SWARM_WORKER_TOKEN) \
      $(SWARM_MANAGER_IP):2377"
done
```

Dependencies

Our dependencies are split into direct dependencies and tools

Tools

- **Docker** - Cloud computing services
- **Digital Ocean** - Cloud infrastructure provider
- **Travis** - Hosted continuous integration service
- **ElasticSearch** - Distributed RESTful search and analytics engine
- **Kibana** - Data visualization dashboard software for ElasticSearch
- **Filebeat** - File harvester
- **PostGreSQL** - Database Manangement System
- **NGINX** - (Web Server used for reverse proxy)
- **Flake8** - Python style consistency
- **Black** - Python code formatter
- **SonarQube** - Code quality inspector, bugs, vulnerabilities
- **Code Climate** - Test coverage
- **Better Code Hub** - Quality improvements

Application Dependencies

Web App dependencies are as follows:

- **asgiref 3.3.1** - Includes pytest a framework that makes it easy to write small tests
- **django 3.1.8** - Python Web Framework
- **django-prometheus 2.1.0** - Export django monitoring metrics for Prometheus
- **django-rest-framework 3.12.2** - Web APIs for Django

- **prometheus 0.9.0** - Prometheus instrumentation library
- **psutil 5.8.0** - Python system monitoring
- **psycopg2 2.8.6** - PostgreSQL database adapter for Python
- **pycodestyle 2.7.0** - Python style checker
- **pytz 2021.1** - Cross platform timezone calculations
- **requests 2.25.1** - HTTP library
- **sqlparse 0.4.1** - SQL query parser / transformer
- **uWSGI 2.0.18** - 2.1 - Web service gateway

API dependecies are as follows:

- **asgiref 3.3.1** - Includes pytest a framework that makes it easy to write small tests
- **django 3.1.8** | Python Web Framework
- **django-prometheus 2.1.0** - Export django monitoring metrics for Prometheus
- **markupsafe 1.1.1** - Safely add untrusted strings to HTML/XML markup
- **psutil 5.8.0** - Python system monitoring
- **psycopg2 2.8.6** - PostgreSQL database adapter for Python
- **pytz 2021.1** - Cross platform timezone calculations
- **requests 2.25.1** - HTTP library
- **sqlparse 0.4.1** - SQL query parser / transformer
- **toml 0.10.2** - Python library for TOML
- **uWSGI 2.0.18 - 2.1** - Web service gateway
- **wrapt 1.12.1** - A Python module for decorators, wrappers and monkey patching.

Current state

We had many timeout errors that were due to the large size of the public timeline. The frontend server would crash since it tried to load all messages on the public timeline page.Pagination would have solved that however we set a fixed number of messages which would be displayed.

For static analysis of the software, we've primarily used SonarCloud. As of now their report on the project is as follows:

Reliability: 0 bugs **Security**: 0 vulnerabilities with 3 security hotspots **Maintainability**: 2 hours technical debt based on 16 code smells, where hereof 8 are critical or major. **Duplications**: 5.0 % duplications and 6 dublicated blocks

The duplications are only test and settings files.

The encryption choosen for the user passwords is a weak one, however we could not migrate to a new one since we already had a lot of users using the wrong encryption methods.

The current state of the logging system is not ideal since we have to make some complicated search queries in order to extract useful information from them.

There are also some undiagnosed bugs in the follow and add_message functions, since they sometimes still return a 500 status code.

License

We collected all the licenses and copyright notices for every dependency to formulate our product's license. Here, we met the GNU GPL v2 for psycopg2. The GPL series are all copyleft licenses where any derivative work must be distributed under the same or equivalent license terms. Therefore, to cover the product, we chose to go with the GNU General Public License v3.0, which can be found in the licence document. We also collected all copyright notices for the dependencies; these are all placed in the Notice document.

The Team

The team is organized via Discord. We build a server to support all communication through that. Here we meet up, discuss meetings, solutions, problems and more. We also created a Github webhook to get a notification on the Discord server whenever there are changes to the project.

We use discord servers to meet up every Monday during and after our lecture. We usually work on the given task most of that day. The work that we did not finish is generally completed at the weekend because of the group members incompatible time schedules during the week.

CI/CD pipeline

The CI/CD Pipeline starts when a developer commits a change to the local git repository, it will first run Black to format the code, thereafter it will run [flake83(https://gitlab.com/pycqa/flake8) to ensure a codestyle in order to keep the source code clean an easy maintaiable.

Once the changes get to a branch which isn't the main branch travis will get activated and ensure that the branch can be build and unit test's will be ran. If and only if a pull request are made to the main branch, travis will be started again, a deployment to the production server will be performed id the two former steps are sucessfull and a release to github will be made. All of this is described in the following subsections

Static code analysis

We use 3 static analysis tools as software quality gates into your CI/CD pipeline.

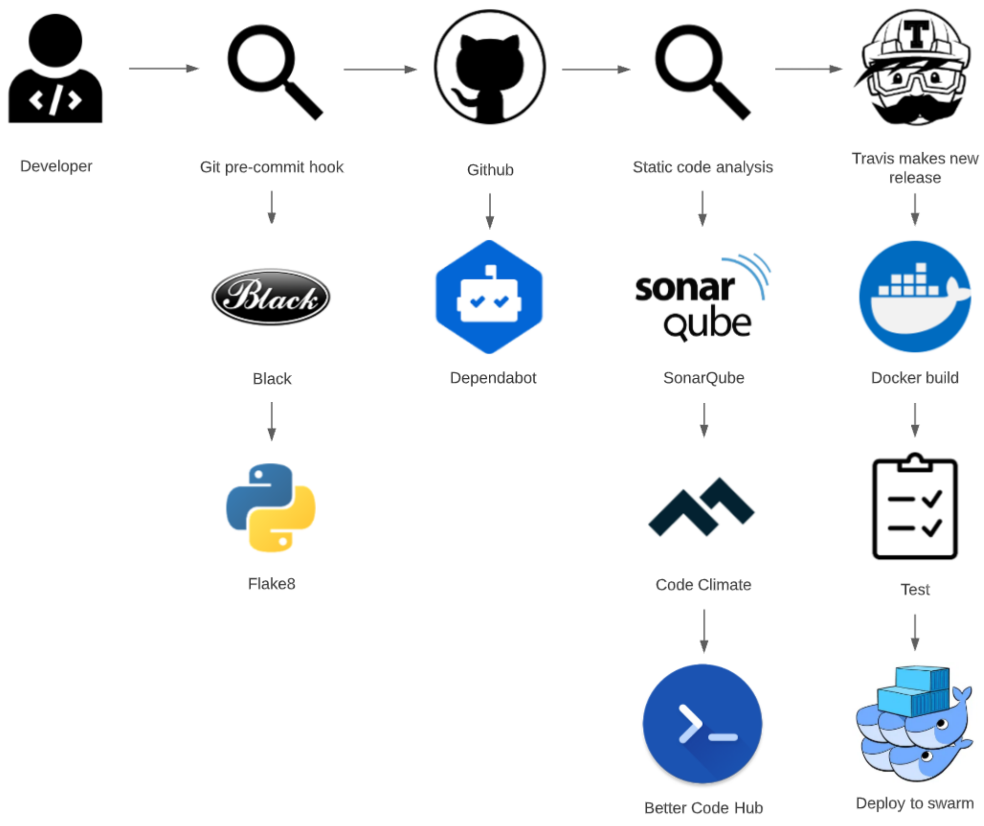

Figure 3: CI/CD pipeline

We use **SonarQube** for continuous inspection of code quality. It performs automatic reviews with static analysis of the code to detect bugs, code smells, and security vulnerabilities in our project.

We use **Code Climate** for test coverage.

We use **Better Code Hub** for quality improvements.

Travis

Our Travis setup consists of 3 jobs; build, test, deploy.

The first job in our Travis setup is the build job. Before this job, we start with getting access to Travis by using our ssh keys. Afterwards, the build job has three stages; log in to docker, build the three docker images web, API and proxy, and push the three images.

The second job is testing. Here we start by migrating. Migrations are Django's way of propagating changes we make to our models (adding a field, deleting a model, etc.) into our database schema. We only test the frontend with Travis. When we ran the backend test in Travis, it tried to start up the system.

The third job is deploy, and we only deploy when we commit to the main branch, which is how we make a new release. Before deploying, we set up the git user and tag the commit. The final step is pulling the images and deploying them to the swarm.

Repository

The repository is currently a mono repository since Django allows us to work with database models tightly incorporated into the framework.

We split logging into a separate repository because we wanted to be ready for the docker swarm, and we didn't want the logging system to take up all the resources from the web and API.

Branching strategy

Our branching strategy utilizes the Gitflow Workflow. We will have two main branches and several feature branches:

- Main branch. It is the main branch and contains the production code.
- Developer branch. It is the development branch containing the development code. This code is merged and pushed into main at the end of each completed weekly assignment.
- Feature branches. These branches are used to develop specific features relating to a specific assignment and is merged and pushed into the development branch when it is completed.

Branches should always be branched from the develop branch.

Always pull the newest development branch before creating a feature branch.

Experimenting is preferably done in branches. When working on a specific task, it might make sense to branch out from the feature branch, eg.

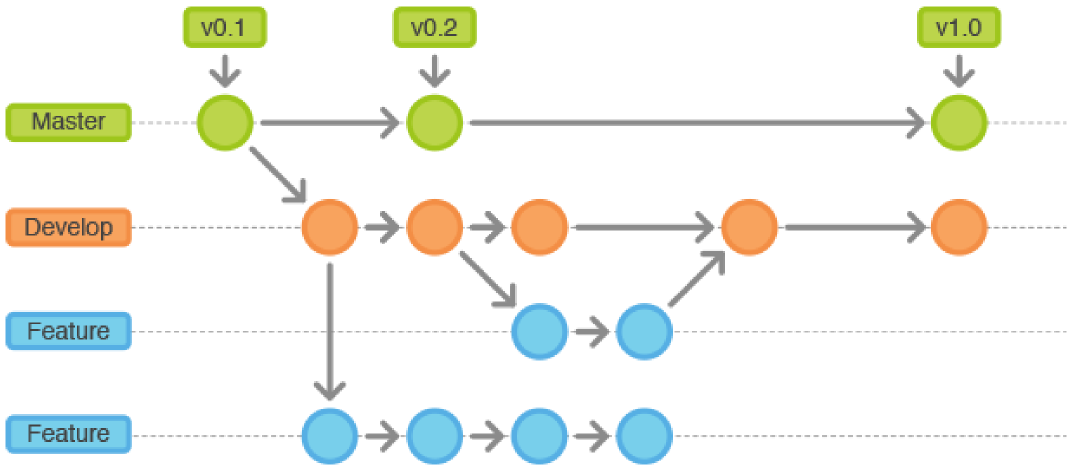

Figure 4: Branch strategy

Development process

For this project, we worked with an agile development process. Agile is all about moving fast, releasing often, and responding to the needs of your users, even if it goes against what's in your initial plan. Every week, we plan a new implementation, work on the implementation in smaller bids, test it, and release it for feedback session the following lecture. If we had any bugs or backlog from the last week, we would split up and work on tasks in smaller groups.

The workflow was distributed because the team would create a set of tasks in the project management tool git provides. Once that was done, the team members would distribute the tasks between each team member. Once a task is completed, a PR had to be made, and once it had gone through a review process, it would get merged into the dev branch. When all tasks for a given week had been completed, a new PR would be made to merge the current stage of the dev branch into the main branch, and a new release would be created with an appropriate dectrion of the recent changes.

Every single team member is responsible for integration and reviews. It is up to the group and the individual team member to decide if they can do the review or integration of a new feature.

Monitoring

We use the monitoring service Prometheus, to monitor our application. To display the data that we get from Prometheus, we use the web-based graph interface, Grafana. We split our monitoring into 2 Grafana dashboards; Business Monitoring (e.g. images/Business Monitoring), which displays our PostgreSQL Query data, and Infrastructure Monitoring (e.g. images/Infrastructure Monitoring), which displays our Prometheus metrics.

The Business Monitoring dashboard contains amount of: Users, Messages and Followers. We used these data to monitor the correctness of successful requests.

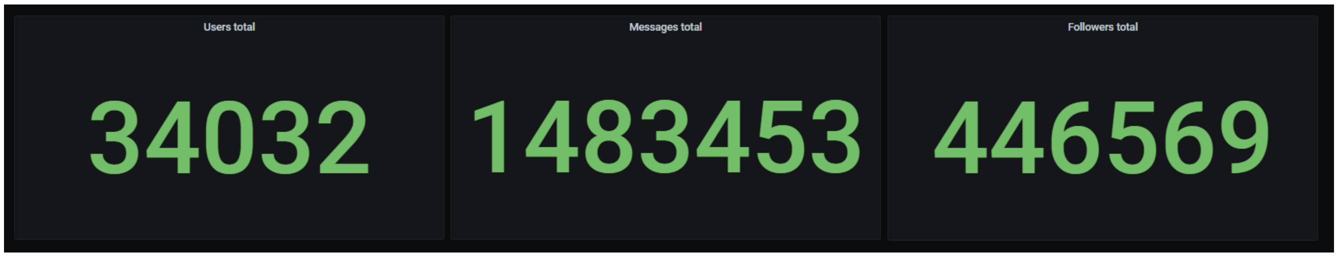

Figure 5: Business Monitoring

The Infrastructure Monitoring dashboard contains CPU Load percent, used for up-time calculation as well as strain on the system and HTTP Responses (Frontend / Backend), used to monitor system failure as well as performance in regards to correct response.

Logging

Logging in Django can be implemented as middleware allowing logging of both current views and future views without tailoring the logging system in every view. It results in a simplistic all-sided logging system without hindering the developer from adding new features or functionality.

Our EFK stack is set up to report each transaction's user, IP address, request type, content, page redirect, and response status. This data is collected by Filebeat when Django invokes its middleware. They are logged as soon as the API/web response is ready to be sent back to the user, and thus both include the initial request and the result of the request. When Filebeat logs the data, it is sent to the Elastic Search database hosted on a separate droplet, such that it is accessible on Kibana.

Security

The security assessment helped us to identify which issues to prioritize. We performed 2 vulnerability scans overall, and the second scan was done after we had fixed various vulnerabilities that were identified in the first scan.

After using WMAP to scan our system we identified 4 vulnerabilities and noticed that our 2 of most urgent issues were related to our project being unprotected by a firewall. We also tried to run SQL injections and cross site scripting on our site, which did not reveal any vulnerabilities.

After we created the firewall for our project, we ran a second vulnerability scan using OWASP ZAP. We found just one issue that wasn't picked up by WMAP.

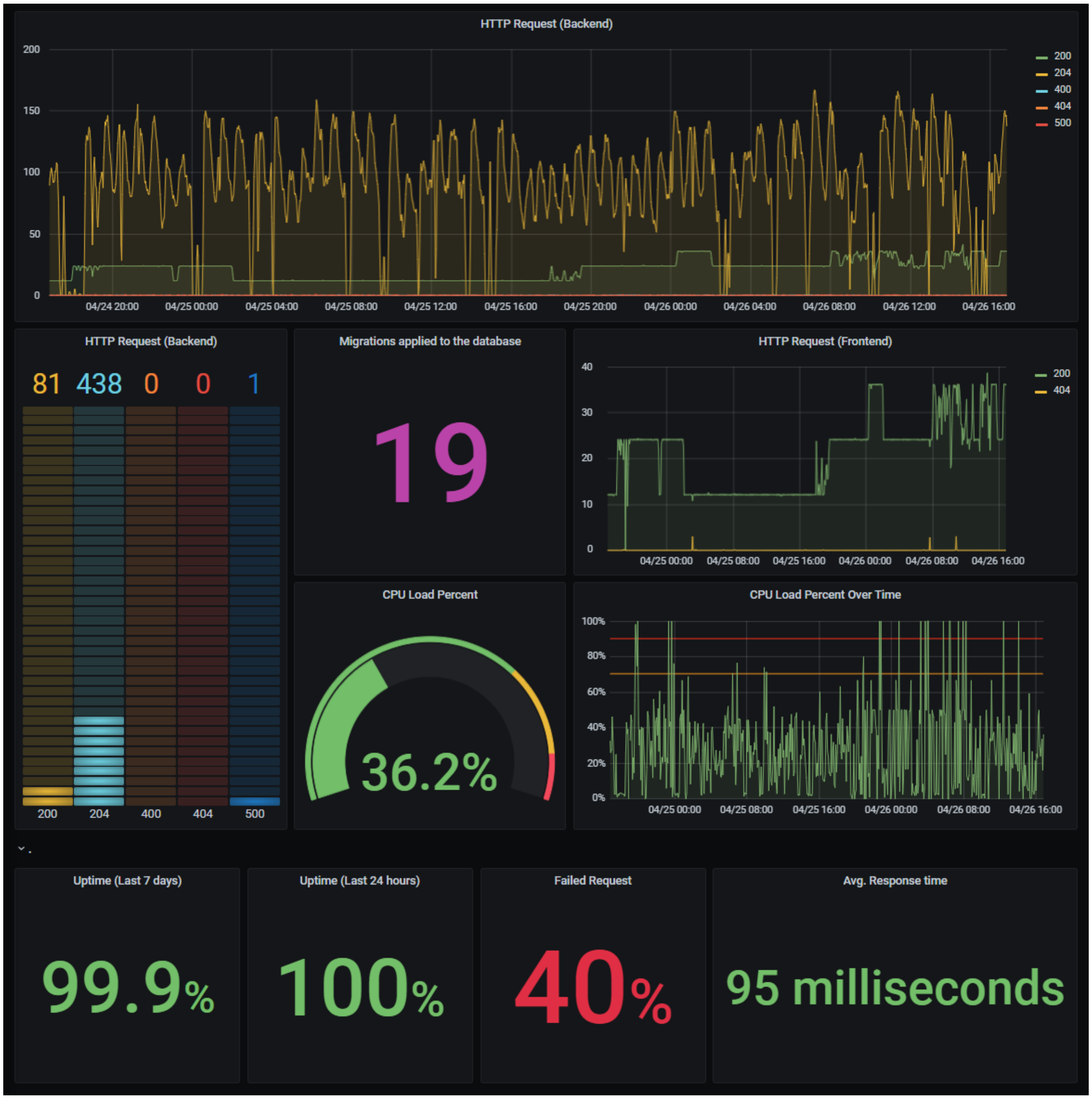

Figure 6: Infrastructure Monitoring

Lessons Learned

Communications

The team had a pretty rough start going into the project. We didn't know each other, our schedules, skills or experience levels. It turned out to be quite a roadblock throughout the entire course. We experienced a lot of miscommunication and times with no communication at all, which would result in tasks being completed twice, tasks not being met at all and group members working alone on a big task.

Because of the lack of communication, we created quite a backlog throughout the course. We did, at one point, create a technical debt on the API, which meant that a group member had to work on the CI/CD and docker alone.

We should have started by getting to know each other and match our expectations for the project. We had many different backgrounds; three were writing their bachelor, and four had a job taking up time. It meant that we should have focused a lot more time discussing schedules and meetings, not meeting odd hours every week. By focusing more time on this, we could have gone into the project with a better foundation and possibly a better end product. It would probably have been a good idea to give each group member a role and a responsibility and then actively use a kanban board, so everybody knows what is going on and what is missing.

Tool Choice

With the lack of communication came some bad choices. We chose to go with a framework that only one group member knew, which meant that the rest of the group had to use a lot of time getting to know the structure and functionalities.

Django is an extensive scale framework. It has a large amount of implemented functionality; easy database management, easy implementation of views and pages, authentications tools and security features. It is all beneficial when you use it as a large scale API framework. We ran into the problem here because our project didn't take advantage of these features the way it was meant. Django turned out to be unmanageable for inexperienced users due to the high complexity of the innate features.

We should have started discussing several choices, including prices in the long run, for the whole tech stack and the team members prior work with these.

Technical Debt

The team didn't use enough time going through and testing the individual pull request, which meant that we spent a lot of time looking into minor errors that.

When the simulator started, we already had a backlog, resulting in our product not being ready for the simulator. It meant that some register request already failed the first day, resulting in a future error on message and follow request. The team should have focused a lot more time on getting the project ready for the simulator and avoiding future mistakes.

Later in the project, we decided to try and implement docker swarm. It took a lot of time and looking back. We probably shouldn't have chosen the more complex implementation when we already had quite a significant backlog. We did learn a lot from working with docker swarm, and in the end, it was worth it to get it up and run for the learning experience.

Our technical debt included a couple of errors from former tasks. It took a lot of time to find these and fix them. We should have focused more time on getting the logging system up and running so that we had a tool to help us in these cases.

Database

So a significant roadblock has been to migrate the database two times to a new host with minimal downtime and minimal data loss. We could have made a dump of the minitwit database with `pg_dump`; however, then we would manually have to download the dump, transfer it to a new host, write the `INSERT and UPDATE` queries and so on, which would be time-consuming and result in a more extensive loss of data. But we ended up using the (postgres_fwd)[https://www.postgresql.org/docs/9.5/postgres-fdw.html], which is a foreign-data wrapper that can make a real-time/live connection to another database. The only problem was that we had to write an `INSERT` query that would get the data and make sure that the relationship between the entities in the different tables wouldn't be broken. To do so, we took every id from the "old" database and made them negative, so when a new entity is created in the "new" database, the id will not conflict with the "old" id's.

However, this created a new issue when we had to migrate the database once again. We could not use the same strategy again. We could probably just have done it with the same strategy and added the length of the table to the ID, but that is a hacky fix that would hit us hard in the future. After a lot of thinking, we decided to take a snapshot of the droplet and recreating it in another droplet which was by far t easiest solution, and we lost some data since the DB was down for about 40 minutes. With that knowledge now, we would have chosen to use Digital Ocean's Managed Database Cluster, saving us **a lot** of trouble in this project.