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 3

Architecture 3

Dependencies 6

Tools . . . 6

Application Dependencies . . . 6

The Team 8

CI/CD pipeline 8

Static code analysis . . . 9

Travis . . . 9

Repository 9

Branching strategy 10

Development process 10

Monitoring 11

Logging 11

Security 13

Scaling and load balancing 13

Leasons learned 13

Commications . . . 13

Tool choice . . . 13

Technical debt . . . 14

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 are using 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 certain metrics are updated. These tools include Prometheus and Grafana, which allow for collection and displaying of metrics respectively. The monitoring tools are very useful 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 thus also allows perfect reproduction of requests from users to both the frontend and backend. This is 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 own separate Droplets and separated from the swarm in order to isolated theese two systems from the systems which are exposed to the end-user. This allows us to easily horizontally scale everything besides our persistent data, which should not be horizontally scaled in this scenario. However if this was supposed to run in a production enviroment it would make sense to let Digital Ocean take care of the horizontal scaling of our database instead of maintain it ourself and setup a High Availability and Load Balancing for our db which includes maintaince 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

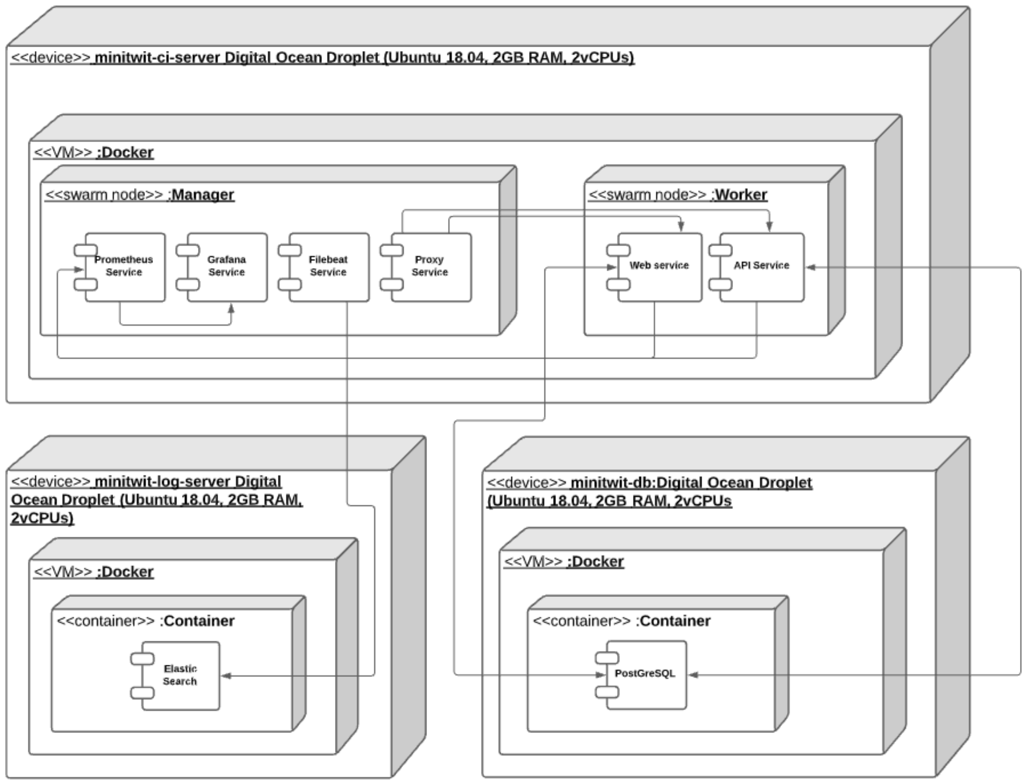

Figure 1: Deployment diagram of docker swarm setup

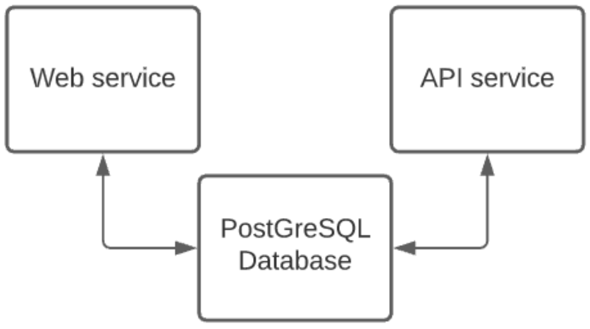

Figure 2: Diagram of project structure

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 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.

more detailed diagram

We have a proxy service that uses nginx to route traffic from minitwititu.xyz to our web app server's IP address, and from api.minitwititu.xyz to our API server's address. It only exposes those two IPs, so all of the logging and monitoring related IPs are not exposed.

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). This logging database is used by Kibana 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 certain 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 in which metrics can be monitored through customizeable 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.

jjkkkkkk # Current state

Do we know any bugs in our system or ?

We have some timeout errors, we used nginx to … This is mainly due to the large size of the `public timeline` and rendering that in the frontend will crash the server, pagination would have solved that however we set a fixed number of messages which would be displayed.

Some error with follower.

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 (23 with tests??) **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 (including tests/settings)

Mono repo caused the settings fields to be duplicated.

We chose a bad encryptions method # License

We collected all the license for every dependency we have to form the license our product. Here we met the GNU GPL v2 for psycopg2, The GPL series are all copyleft licenses, which means that any derivative work must be distributed under the same or equivalent license terms. To cover the product we therefore chose to go with the GNU General Public License v3.0; which can be found in the licence document. In this process we also collected all copyright noticies 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, so that everyone gets a notification on the Discord server, whenever there are made 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 are usually completed in the weekend, because of the group members incompatible time schedules during the week.

CI/CD pipeline

Our CI/CD chain is run everytime we commit to any branch.

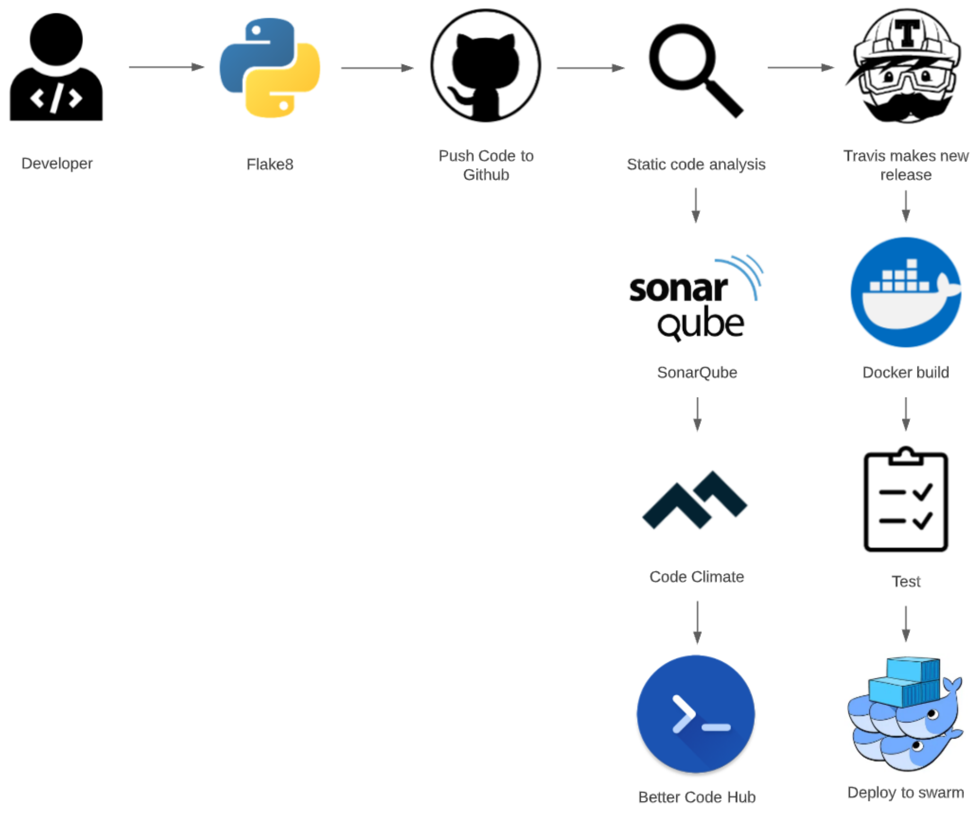

Figure 3: CI/CD pipeline

Before commiting, we have a git hook that runs **Flake8** and **Black** to enforce ... and style consistency across our Python project.

The developer code is then pushed to Github ...

Static code analysis

We use 3 static analysis tools as software quality gates into your 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 consist of 3 jobs; build, test, deploy.

The first job in our Travis setup is build. Before build we start out with getting access to Travis by using our ssh keys. Afterwards, build has 3 stages; login to docker, build the 3 docker images web, api and proxy and lastly push the 3 images.

The second job is testing. Here we start out 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 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 deplying to the swarm. - deploy token?

Repository

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

We split logging into a separate repository because we wanted to be ready for docker swarm and we didnt want the logging system to take up all the ressources from the web and api.

Branching strategy

Our branching strategy utilizes the Gitflow Workflow. Basically we will have 2 main branches and a number of feature branches:

- main branch - This is the main branch and contains the production code.
- developer branch - This 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 - 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 develop

Always pull the newest development branch before creating a feature branch

Experimenting is preferably done in branches. In some cases when working on a specific task it might make sense to further branch out from the feature branch eg.

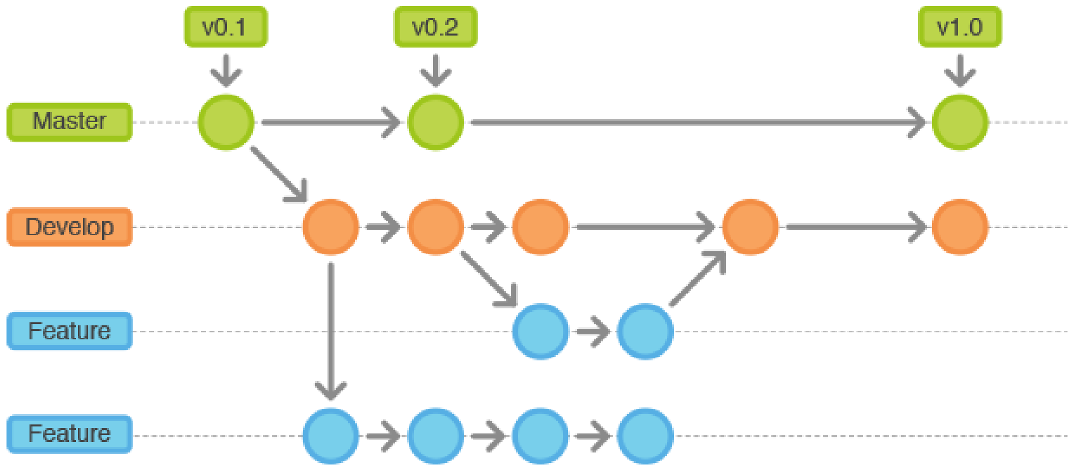

Figure 4: Branch strategy

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 would plan a new implementation, work on the implementation in smaller bids, test it and release it for feedback session the next lecture. If we had any bugs or backlog from the last week we would split up and work on tasks in smaller groups.

The way the workflow was distributed was that 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 a 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 in order 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 new changes.

Every single team member is responsible for integration and reviews. It is up to the group to and to the idividual team member to decide if he or she has the competence to 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.

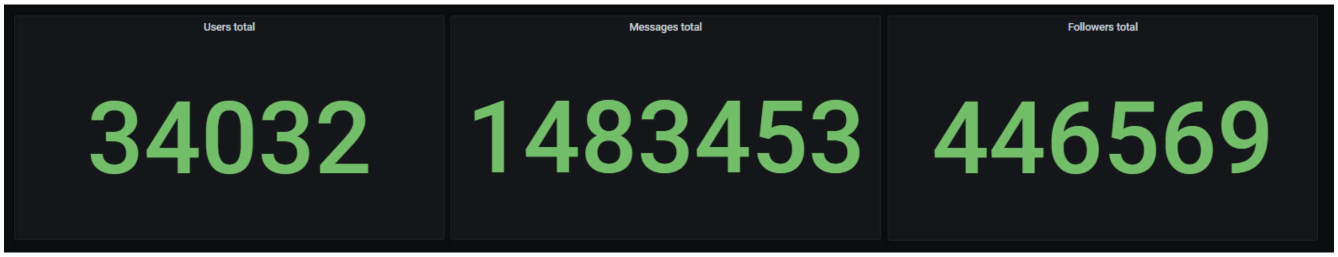

Figure 5: Business Monitoring

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

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 intial request as well as the result of the request. When Filebeat logs the data, it is sent to the Elastic Search database that is hosted on a separate droplet, such that it is accessible on Kibana.

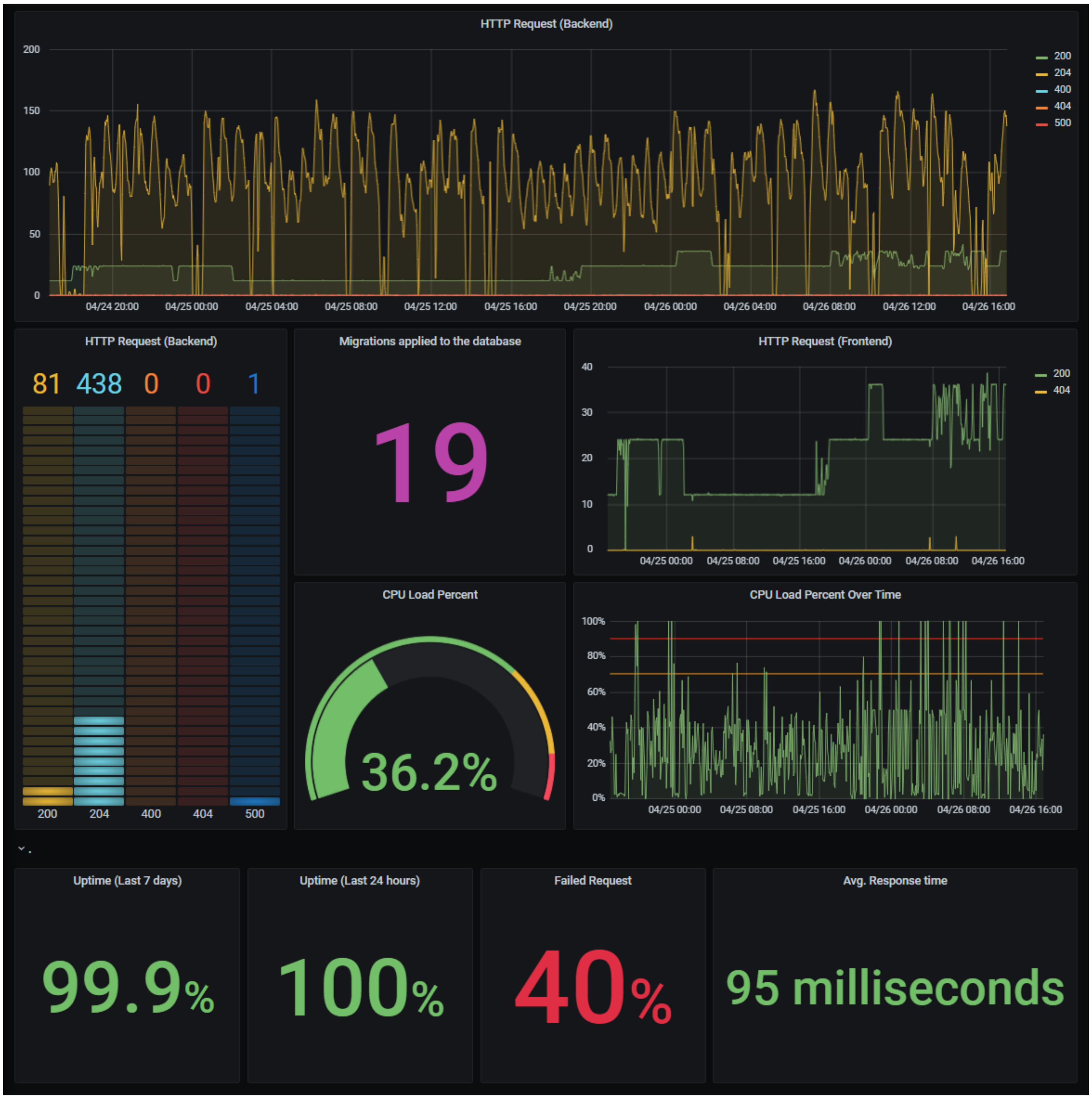

Figure 6: Infrastructure Monitoring

Security

Brief results of the security assessment.

What tools did why use? Known risks? Did we get firewall on database?

We do not have anything like fail2ban setup on the database server, and we only use basic authentication which provides very little security and are therefore vulnerable to brute force and dictionary attacks.

Django protects us from host-header injections. It also provides CSRF token support, which helps to protect our users from CSRF attacks.

Scaling and load balancing

Our scaling and load balancing is done through a single Docker Swarm that holds all of Minitwit's non-database services.

Leasons learned

CICD

Logging in Django can be implemeted as middleware allowing logging of both existing views and future views withour tailoring the logging system in each and every view. This both results in a very simplistic an all sided logging system, without hindering the developer in adding new features or functionality.

Commications

Importance of communicating in a devops team (different skills / experience) Get to know each other Match your expectations

Organizing

technical debt on api, (Daniel alone on CICD)

Tool choice

Large scale framework - much implemeted functionality - easy database management - easy implementation of simplistic views and pages - large scale api framework - authentication tools - security features by default - simplistic custom implementation - many resources for help (large docs, stack overflow, widely used)

disadvantages: - much implemeted functionality - unmanageable for inexperienced users due to the high complexity of the innate features

. . . use tools we already know (% django) % python latest, use db instead of global variable in python.

Technical debt

testing implementations (vagrant; mac) we werent ready for the simulator (already behind) when you have a enormous backlog dont choose the harder implementation (docker swarm) get loggin up and running for use.