Visualization tool for CDNs

Author: Daniel Gómez Bellido
Director: Luis Velasco, DAC
Date : 28-10-2020
Grau en Enginyeria Informatica
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

Index

Abstract

The aim of this project is to deploy a software stack for data-logging and monitoring using Grafana and Prometheus, with a focus on portability and performance

Resumen

El objetivo de este proyecto es desplegar un stack de software para data-logging y monitoring utilizando Grafana y Prometheus, haciendo hincapie en la portabilidad y el rendimiento

Resum

L'objectiu d'aquest projecte es desplegar un stack de software per data-logging y monitoring fent servir Grafana i Prometheus, posant l'accent en la portabilitat i rendiment

1. Introduction(GEP)

1.1 Context

-Introduction and motivation

A content delivery network or content distribution network (CDN) is a geographically distributed network of proxy servers and their data centers. The goal is to provide high availability and high performance by distributing the service spatially relative to end-users. CDNs serve a large portion of the Internet content today, including web objects (text, graphics and scripts), downloadable objects (media files, software, documents), applications (e-commerce, portals), live streaming media, on-demand streaming media, and social media sites.[1]

Live-TV and Video on Demand(VoD) distribution is in the portfolio of many telecom operators aiming at entering into competition with on-line, over-the-top broadcaster, such as Netflix. To this end, a CDN is being considered a suitable option to be deployed by telecom operators internally within their network infrastructure by placing cache nodes in distributed locations covering a territory.[2]

In a more general capacity, data monitoring is becoming more and more important in the current world due to the increasing capacity to log this data and store it, and to the great advantages it provides in many contexts. Data monitoring advantages include:

- Revealing anomalies in the data to solve problems in a proactive manner, for instance, detecting failures or performance issues on certain components early so action can be taken to mitigate the damage. Similarly, the same could be said for any system from which we can take enough performance metrics(Transportation systems, health, industrial processes…)

- Optimizing costs. Data monitoring can also allow us to refine the functioning of the process we are monitoring. In the context of CDNs, usage data about a specific node will help find the correct requirements needed to meet demand, thus making it harder to either overestimate or underestimate demand.

So data monitoring can allow, with good enough data analysis of course, to solve current problems, identify future ones before they happen to mitigate them, and improve processes.

The conclusion is that data logging and monitoring is essential and as data analytics improves it will be even more important, not just for data center performance and maintenance, but in everything that can be monitored.

- Formulation of the problem

For the reasons above, it's important to research the various solutions and software stacks used to effectively collect, visualize and analyze data in order to improve and optimize these networks and data centers.

In this project I will study the various solutions on the market for this purpose, and put together a full software stack capable of collecting and visualizing data such as system CPU usage, memory, disk, network metrics and I/O utilization over time

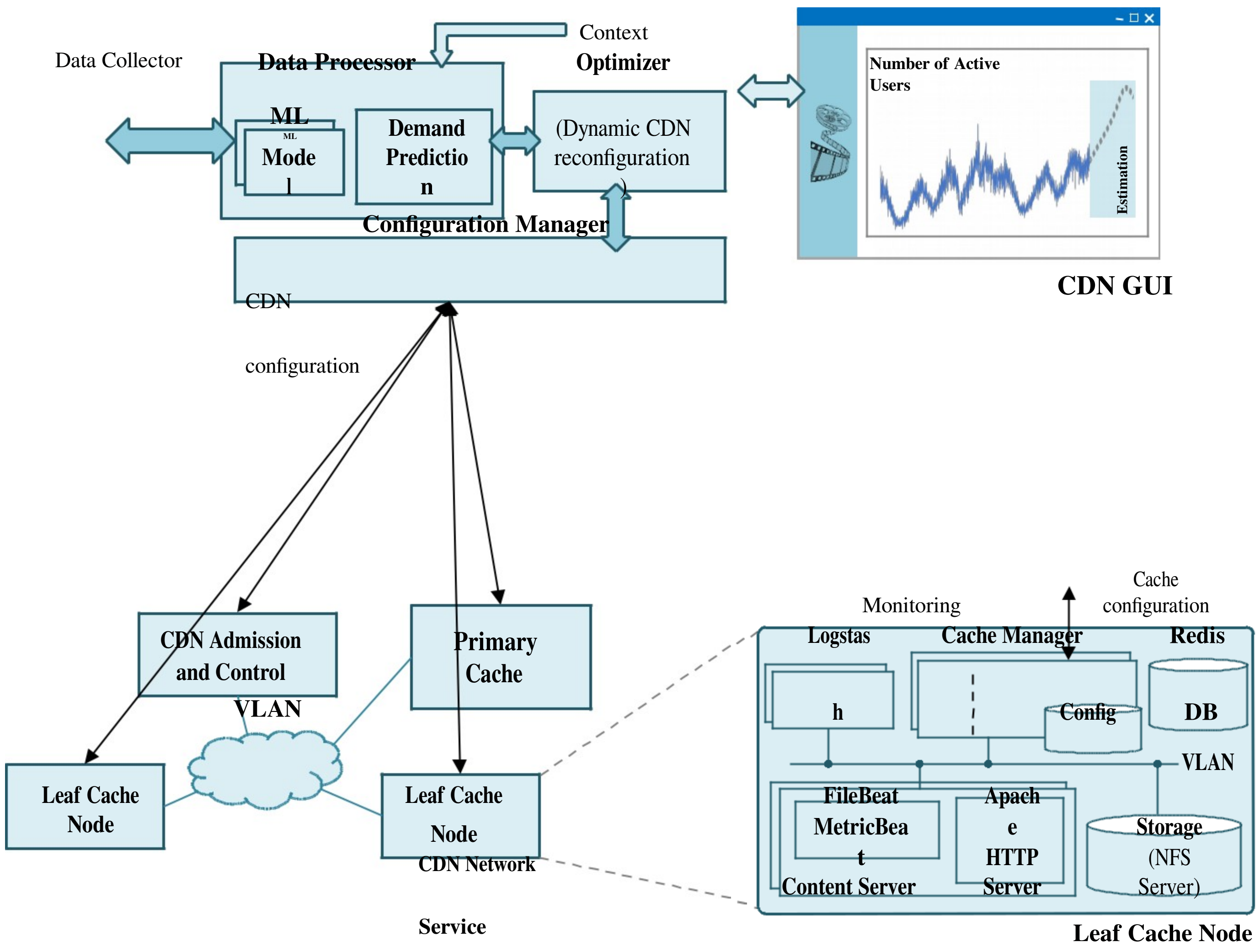


Figure 1 CDN Schematic

- State-of-the-art

Current solutions for data analytics rarely do all the functionality required by themselves, but rather function as a modular stack depending on what kind of data we want to collect, visualize and so on. So my analysis will be based on the most popular solutions and identifying trends amongst those.

- ELK
One of the most popular stacks is ELK (ElasticSearch+Logstash+Kibana) three separate services by the same developer. Logstash is a tool for collecting, parsing and sending logs.
ElasticSearch is a search engine which is highly scalable and can be used to store large quantities of data.
And finally, Kibana is a frontend based on web technologies that enables the visualization and analytics from the data stored on the ElasticSearch database.
This is one of the most flexible stacks, as it allows to store and visualize almost any kind of data, and also offers the advantge of having the same developer, thus ensuring compatibility between the modules.

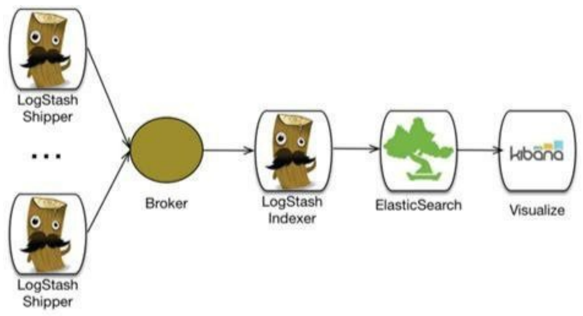


Figure 2. Example ELK stack

- Graylog
Another web based analytics platform, although a more standalone one, as it provides almost all the software needed to get it running. Data collecting is done through Graylog clients installed on each node[3]

- Grafana
Grafana is an open source, web-based monitoring and analytics platform that is completely modular, as it's compatible with most databases (53 different data sources) also it's compatible with previously commented ElasticSearch and Logstash. It is commonly used with Prometheus, a database designed to deal with time-series data logs.

It also offers the most customizability in terms of data visualization, thanks to the community support and plugins.[4] When compared to ELK, for instance, Grafana is a more general purpose data visualization tool, whereas ELK is more tailored towards log analysis.

The biggest trend we can observe amongst the various stacks is the lack of relational databases. This can be explained because other kinds of databases exist that are much more optimized at dealing with such data[5] such as Prometheus[6].

1.2 Scope

The main objective of this project will be putting together and deploying a full stack of software that enables monitoring and visualizing data from a distributed CDN such as CPU power, memory usage, disk, network and I/O utilization.
This data could be used later for improving the CDN or identifying problems and bottlenecks in the network.

-Requirements

-Small performance footprint, as it wouldn't make sense to worsen the performance when using a tool used to improve the CDN

-Portable and easy to deploy. The project should use Docker or a similar technology in order to have as much compatibility as possible and simplifying configurations, security etc.

-It should be able to actually visualize the data mentioned above in different visual presentations.

-Possible obstacles and solutions

-Compatibility problems
Considering that we are going to deploy several software packages working together, it's feasible that some compatibility/configuration problems might arise as the project advances.
My strategy is to work closely with my project tutor and/or even contacting the developers and community of said software.

-Optimization
It might be possible that the proposed solution doesn't meet the footprint requirements expected and consumes too much CPU power/memory. The best strategy against this is to keep the services hosted outside of the actual machines we monitor.

-Timeline
Because of the tight schedule required in this projects, getting ahead of the timetable will be critical to the success of the project.

1.3 Methodology

To ensure the project's success, a methodology has to be set in place in order to meet the requirements of the project in a reasonable time.
The method used for this project will be iterative. Recurring meetings will happen (weekly or bi-weekly) with the involved people in the project, in order to validate the progress made between meetings.

Additionally, to maintain the project git and gitlab will be used, to maintain the configuration and possible code, which also helps with the documentation associated with such files. Also, 2 additional copies will be saved, one locally, and another one in the remote virtual machine used for development.

1.4 Project planning

This section will go into detail in the description of the tasks in the project and later on, the dependencies between them and the order established to do them based on such dependencies, including a Gantt diagram, as well as the resources used. Also we'll try to account for the possible risks and deviations in the project.

- Tasks description

First I'll introduce all the tasks required for the project chronologically, describe the resources needed briefly, and the temporal estimation for each task

- GEP

All the work needed for the GEP coursework, with a total workload of 75 hours including possible classes, deliverables and the presentation:

- Module 1(ICT Tools to support team and project management): 4 hours
- Deliverable 1(Context and scope): 24.5 hours
- Deliverable 2(Project planning): 8.25 hours
- Deliverable 3(Budget and sustainability): 9.25 hours
- Module 3(Personal and professional skills for project and team management): 6.25 hours

- Deliverable 4(Oral presentation and final document): 18.25 hours

For this tasks the following resources will be used: A computer with an internet connection, LibreOffice, Gantter, and Dropbox.

- Prior analysis and design

When the GEP course is done, in this task we'll analyze the project to further define the main requirements and objectives that our tool must have, also defined by the previous task (Deliverable 1) and various meetings with the director of the project.

Basically a decision will be made on the functionality required to decide on general terms on the architecture and deployment of the application, without going into much detail, just in order to start working.

For this task, the following resources will be used: A computer with an internet connection, LibreOffice, Dropbox, vim and markdown for taking notes, and git for version control.

- Environment setup

This task will be about setting up the local development environment needed for all the following tasks.
This can include installing and configuring the operating system and all the software needed: Debian, git, text editors, browsers etc.

For this task, the following resources will be used: A computer with an internet connection, Debian operating system, git, Docker, DropBox, libreoffice, vim and markdown.

- Development cycle

This task depends on the prior analysis task and consists of the general development of the tool. Because it is such a big task, i've divided it into other tasks which establish dependencies between them:

- Virtualization environment setup: This task consists of setting up a remote testing environment in a virtual machine, simulating the theoretical real server the tool would be deployed on. This task involves setting up the operating system, users, and the software needed to run such as Docker, git etc

- Testing environment deploy and configuration: This task consists of putting together the software stack that we need. This includes a front-end, databases and various instances of data-logging services, simulating a real content delivery network, as well as the configuration needed for containerization, needed for easy deployment later on. This stage might be the most risky and time-consuming.
- Tooling evaluation and testing: When we have a working stack running and deployed on the virtualized environment, in this task we'll explore all the visualization options offered by the frontend/database and evaluate the most useful ones for the CDN use case.

- Real environment deploy test: In order to test the portability and reliability of the system, we'll deploy the containeraized stack onto another different environment.

For those tasks, the resources used in point 1.1.3 will be used, as well as the remote environment set up in the first sub-task of this task.

- Requirement evalutation

In this task, an analysis will be done to check if the development done meets the requirements expected, in terms of overhead, reliability and portability.

This task will use the same resources as the previous one, as well as the software developed in the previous task.

- Thesis report

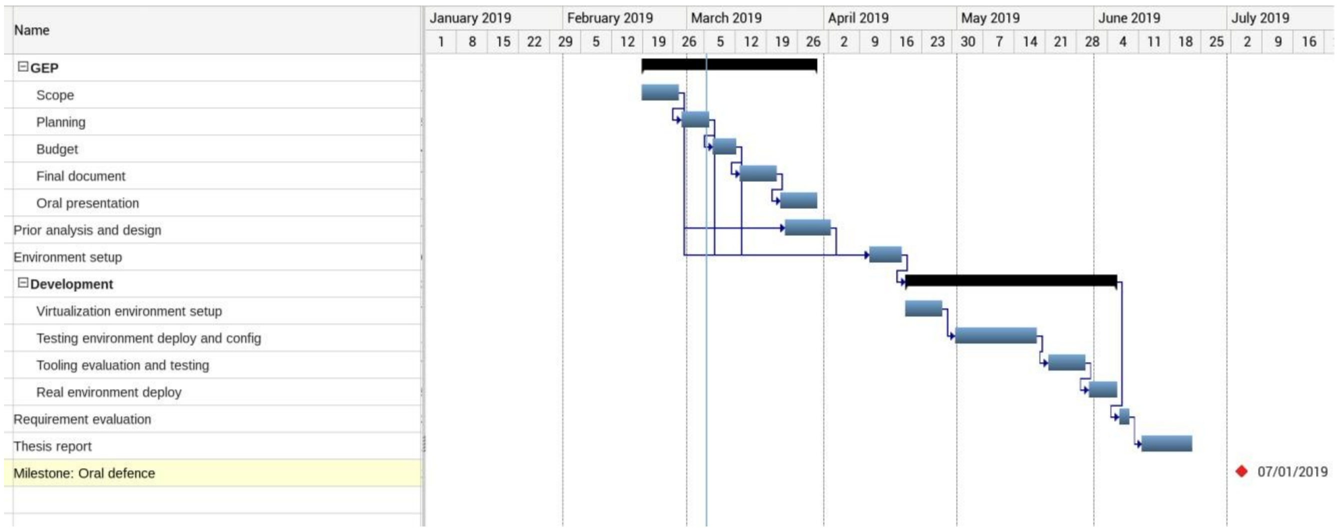
This final task will consist in the writing of the thesis report of this project, including getting ready all the material for the defence such as slides, live demostrations etc.

This task will need the same computer, libreoffice and the notes taken during the semester.

1.4.1 Estimated time

Stage	**Estimated time(hours)**
GEP	75
Prior analysis and design	50
Environment setup	20
Virtualization environment setup	20
Testing environment deploy and config.	120
Tooling evaluation and testing	60
Real environment deploy test	40
Requirement evaluation	20
Thesis report	70
TOTAL	445

1.4.2 Gantt chart



1.4.4 Resources

To develop this project, I’ll use the following resources:

- Hardware resources

- Desktop computer with internet connection, plus keyboard, mice and monitor.
- DigitalOcean virtual machine

- Software resources
- Debian 9 Linux distribution
- git
- Docker
- Dropbox
- LibreOffice
- vim + markdown

- Human resources

- Director of the project: Checking that the objectives are being accomplished in the project
- Developer: Responsible for all the tasks

1.4.5 Action plan

First of all, following the time estimation, the estimated time for the delivery of the thesis report is made one week before the required week. That gives us a small window for possible deviations in the project. Also, the meetings with the director will make sure that we are following a good direction and process in the project

Also, most of the risks are in the development process and as such, we planned accordingly by giving the most time to such tasks, so they would have to go really wrong in order to not meet the deadlines and also eat the week of margin that we allowed for this cases.

In that worst-case scenario, though, we would have to modify some tasks to make them easier to attain in that time, or even remove them completely, as some of them are not critical to the core of the project, such as deploying on a real environment. The estimated dedication will be of about 30hours/week.

Thanks to the measures and planning above, plus the director feedback and supervision of the project, the completion of the project in the time established is attainable.

1.5 Budget estimation

In this section we are going to do an estimation of the budget needed in order to make the project possible. The budget is divided in 2 sections, hardware and human resources. Also, there's an extra section to show the total budget obtained from combining the total cost.
To calculate the amortisation we are going to take into account two factors, the first one being the useful life and the second the fact that the project is going to last 5 months approximately.

Note that there is no "software resources" section, as all the software that will be used is free of cost, at this moment.

- Hardware resources

This table contains the costs of the hardware we are going to use in the project.

Product	**Price**	**Useful life**	**Amortisation**
PC	1700.00€	5 years	141.6€
DigitalOcean VM	50.00€*	Doesn't apply	50€
Laptop	200.00€	5 years	16.6€
Total	1950.00€		208.2€

*****The DigitalOcean VM has a fixed price of 10€/month, and the project will last for 5 months

- Human resources

I’m going to divide the tasks into two main roles:
- Developer: Takes care of the actual work of the project(Developing, testing, deploying…)
- Project manager: All the tasks related to the planning of the project.

This table contains the costs of human resources based on this division and the time planning,

Role	**Price per hour**	**Time**	**Cost**
Project manager	50€	75h	3750€
Developer	25€	370h	9250€
Total		445h	13000€

- Total budget

Using the data shown in the previous tables from points 1.1 and 1.2, we can estimate the total cost of the project

Concept	**Cost**
Hardware resources	208.2€
Human resources	13000€
Total	13208.2€

1.6 Budget control

Our budget could need a modification if the development of the project makes it impossible to follow the plan established.

It is unlikely that we need more hardware resources apart from the already budgeted in section 1.1, as long as the project stays the same and doesn't deviate too much.
Also it is unlikely that we need any software resource, as all of the software that is going to be used is free, and in the case of needing another application, the first option wouldn't be a paid one for sure.

However, it might happen that we need more time for some tasks, and if the tasks take more time that takes more budget into human resources, so we have to take into account that the current budget could grow if problems during the project become more complex and more time needs to be invested into them.

1.7 Sustainability

In this section we are going to evaluate the sustainability in three different aspects: Economic, social and environmental.

- Economic sustainability

In this document we can find a quantification of the costs for the project. The final cost could be the total cost in the project, and if completed succesfully, it shouldn't need any

further development, because it will be bundled with its own container, potentially meaning that it could work indefinitely as it is bundled with everything necessary for execution.
It would be hard to do the project with a lower cost, let alone if it can be done with the stated budget now, as it is stated in section 2 of this document, it could take even more time than now. There's no way to reduce software costs as they're already 0. Hardware cost could be reduced, but they might cost more convenience and time to the developing so it is counter-productive to try and reduce it.

This project improves other solutions in the sense that if developed correctly with the requirements stated, it will require almost no mainteinance or more developments, making it more competitive.

- Social sustainability

The application is focused towards CDNs, so it is going to be used mostly by big companies that house this big data centers, so there isn't a real "quality of life" improvement in any segment of society.

However, we could argue that indirectly, the quality of life of the consumers of these services would go up if they work better as a result of this application.
Also, no group of people would be indirectly or directly harmed as a result of the creation of this project.

- Environmental sustainability

During all the development of our project there will be a computer running. Also, the virtual machine will be running for the whole 5 months of the project. Some paper might also be used to print support material and/or documentation.

Knowing these factors we can estimate the energy spent developing the project. We can suppose that the computer+remote VM average 500W consumption when working on the project, which lasts 445 hours. The energy used is 222.5kWh, which is equivalent to 63kg of CO2. It is a high amount of energy but necessary to do the project. However it is a good point that spending less time on the project is better for the environment.

The project will also serve to make these data centers more efficient, which would make the energy consumption worth from both an environmental and economic perspective, as the energy savings generated by the tool would surpass the costs.

2. Configuring and installing the stack

-Why Grafana/Prometheus

After evaluating the options listed in the GEP document, Grafana+Prometheus was chosen for this project, for the following reasons:
- Even though they are closely integrated, they are independent projects, so if needed in the future any of the two could be easily replaced for a similar choice.
- For this project, they are the only option that met all the requirements specified:
Small performance footprint, modular stack, and more flexibility.

This chapter will focus on the actual initial setup of the stack, more in-depth justification for the choices can be found in chapters 4 and 6 respectively.

-VM setup

As stated in the introduction, I'll be using a DigitalOcean virtual machine for hosting with the following specifications, running Ubuntu Linux 18.04 :
2 vCPUs
4GB RAM
80GB SSD

The first thing I had to do was to create a new non-root user and install the needed software, which was Docker, docker-compose, ufw(firewall), and vim in order to edit scripts.
In this case docker version was 19.03.8, although it will most likely work in older versions. Firewall in this case is optional, as this is just a test environment.

-Prometheus/Grafana

We'll use Docker to run both prometheus and grafana. Docker is a platform that takes advantage of Linux's container technology(LXC)[7] to allow applications to run on a isolated environment without the disadvantages of virtualization hypervisors, and as such it's a great option for an application to be both portable, lightweight and isolated for security.

Docker compose is a tool in the docker project in which we can create a YAML configuration file to run,configure and start all the services we need from there.
With a docker-compose file, we define the execution environment for the application so it can be reproduced anywhere, given that the docker-compose version is compatible with the YAML version of the file.

In this section I'll give a rundown of the config file and explain the reasoning behind it.
For prometheus, first we need to make a custom Dockerfile that bakes in the config file into the Docker image, so we can run our own configuration and scrape targets. That's trivial, with just two lines

Dockerfile

```
FROM prom/prometheus
COPY ./prometheus.yml /etc/prometheus/prometheus.yml
```

This just copies our prometheus.yml configuration file into the container filesystem.
With that out of the way, this is the basic docker-compose.yml file

docker-compose.yml

```
#docker-compose.yml
version: '2'
services:
   prometheus:
      build: ./prometheus
      volumes:
            - prometheus_data:/prometheus
      command:
            - '--config.file=/etc/prometheus/prometheus.yml'
      ports:
            - '9090:9090'
   grafana:
      image: grafana/grafana
      environment:
            - GF_SECURITY_ADMIN_PASSWORD=pass
      depends_on:
            - prometheus
      ports:
            - "3000:3000"
      volumes:
            - grafana_data:/var/lib/grafana
volumes:
      prometheus_data: {}
      grafana_data: {}
```

First of all, for persistent storage there has to be defined a volume for each service, in this case prometheus_data and grafana_data. This volume binds to the docker filesystem on startup, but we have to define on each service the lines “volumes:” on both grafana and prometheus, and define the directory where the data directories will be located.
The volumes are created if they don’t exist and will be recovered on startup if they have already been created.
It’s also possible to delete all persistent storage easily with the command
docker-compose rm

For prometheus, the only difference is that it is specified via the command option where is our configuration file, as defined in the Dockerfile.

For grafana, we also specify that it is dependen on prometheus, simply because prometheus needs to be running first for grafana to be able to detect it correctly.

In grafana, the password is defined in the environment setting, which is not good to be run as it is now unless it’s for testing purposes, also, it isnt configured with ssl yet and
the prometheus web basic frontend is exposed without authentication.

-SSL and authentication (Optional for testing)

The solution to the problems above is to add a SSL service in the docker-compose.yml settings, at the same level as the other grafana and prometheus services

docker-compose.yml

```
ssl:
        image: smashwilson/lets-nginx:1.3
        environments: -EMAIL=yourmailhere
                      -DOMAIN=domainhere
                      -UPSTREAM=grafana:3000
                      -STAGING=1
        depends_on:
            - grafana
        ports:
            - "443:443"
        volumes:
            - letsencrypt:/etc/letsencrypt
            - letsencrypt_backups:/var/lib/letsencrypt
            - dhparam_cache:/cache
```

For SSL the easiest way to do it is to find a dockerised service that allows to run it easily in our docker-compose file, since it's already running on docker anyway.
In this case I'm using lets-nginx image[8] which by its specifications claims that it can add SSL to any http service in docker, using letsencrypt[9], a service that offers free SSL certificates.
Following the documentation I ended up with this service configuration:
-The environments setting needs the mail used for letsencrypt, to fetch the certificates, a domain name and the name of the service we want to add SSL to. The STAGING option uses the letsencrypt STAGING server, for debugging, since letsencrypt only issues 5 certificater per week.
The upstream option is the name of our backend container, in this case grafana, that the ssl container will communicate with internally and expose it through port 443 with SSL.
-This service needs to depend on grafana since it exposes grafana through a secure port
-In the volumes docker-compose option it is needed to add the 3 new volumes needed by this service.

Take into account that this doesn't remove the open ports in the previous grafana and prometheus services, so those subsections in the original docker-compose file would have to be also removed. This means that with this new SSL configuration, only 443 port can be accesed, and that would be the grafana service with SSL enabled.
This also means that we can't access prometheus directly, just the grafana frontend
For the purpose of the project, SSL wasn't enabled since I needed to access the prometheus endpoint directly, but in a production environment it wouldn't be necessary at all.

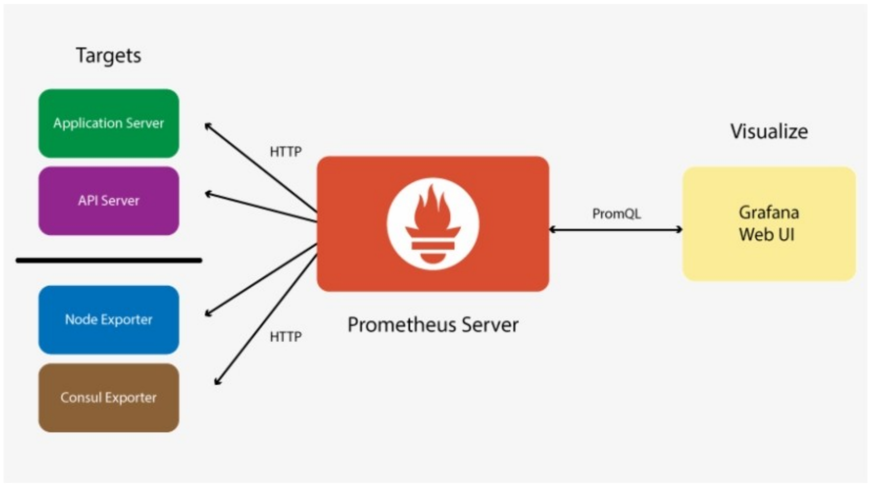
3. Prometheus

-Introduction and general architecture

Prometheus is an open-source time series database developed by SoundCloud, that also serves as a basic monitoring system and alerting. In this project we'll use it as our database for storing metrics.
The typical architecture with Prometheus data-logging includes:
-Exporters, applications that run on the monitored hosts that export local metrics, which prometheus scrapes via http. Prometheus also provides many client libraries for different languages so it can be integrated in any application
-Of course, Prometheus to store the metrics, with its language PromQL for prometheus querys
-Grafana to make dashboards and visualize the data, although Prometheus already includes a rudimentary one

Prometheus stores data in metrics, each metric has a identifying name that's used for the queries. This metrics are stored in the form of time series.
Basically Prometheus queries each exporter(data source) at the specified frequency. Then it stores the current value for that metric and aggregates it to the time series.

One of the main advantages of Prometheus is its interoperability, as it can be used as just a database that a frontend interacts with(like in this case), this was the main draw for this project, since if needed we can replace any part of the stack and keep the rest.



Example of typical Prometheus configurations

- Data types

Prometheus just offers four metric trypes, tailored to the kind of metric we want to log
-Counter
A counter that can only increase or reset to zero.
Good for metrics such as errors, requests,tasks, jobs completed…
-Gauge
A gauge is a value that can go both up and down, like memory usage, cpu usage…
-Histogram
Aggregates observations (Durations, sizes…) and counts themselves
-Summary
Similar to a Histogram, but it provides a total count of the observations it has done and a sum of the observed values, providing interesting options for querying quantiles.

-PromQL querying and exporters

With prometheus working, the first thing we should do is add a new source to query data from. This is very straightforward using the prometheus.yml file:

prometheus.yml

```
# prometheus.yml
global:
  scrape_interval: 5s
  external_labels:
    monitor: 'my-monitor'
scrape_configs:
  - job_name: 'node'
   static_configs:
       - targets: ['localhost:9090']
```

The scrape interval setting sets the frequency that prometheus will scrape the targets at, in this case every 5 seconds.
The important part is in the scrape_configs section. In this example we have one job with one target to scrape at localhost. By default, prometheus itself exposes a scraping target that monitors itself.
Jobs are groups that can have many targets, this is useful because we can query by entire groups, for instance, if we had a cluster, it would make sense to have it in the same job with all the targets inside that job, so it allows for more flexibility in the querying when we want to query a metric from a specific group of nodes at the same time.

-Alerting

One of the most interesting features of prometheus is its alerting system, using alert rules defined by the user, prometheus will take action when certain conditions are met.

To set this up we need to add an alerts text file, and adding it to prometheus image in the dockerfile, in the same way that the configuration file was added, and then add it to the prometheus.yml file

Alert file
ALERT high_load IF node_load1 > 0.7

And then appending this entry in prometheus.yml

```
rule_files:
    - ‘name of the alert file’
```

Note that we can call this file whatever we want

Alerts in themselves are also metrics that we can query, in this case, if node_load1 is higher than 0.7 the metric high_load will be 1, if not it will be 0.

Although we can setup this alerting system to send email, Grafana alerting system is much more complete so we will be focusing on that instead of prometheus one.

4. Exporters

-Introduction

As seen in the last section, Prometheus relies on http endpoints to scrape data from. This endpoints typically are exporters on remote machines that prometheus scrapes periodically. Prometheus provides several client libraries to instrument applications with, so in this chapter we'll explore the options that it offers for building custom exporters, and the node exporter, a custom exporter built by the prometheus project that exposes many generic performance metrics.

-Node exporter

Prometheus provides this exporter which exposes hardware and os metrics in *NIX kernels, written in Go. A really important thing to note is that generally any exporter that wants to expose OS metrics can't be containerized, as it wouldn't have access to the host system. The only way to run this exporter is by compiling it and running it natively with the instructions on its github [10]

-Custom exporter

Prometheus includes support for over 10 languages [11] with client libraries that allow instrumenting applications for metric collection with a few lines of code. In this section we'll see some examples using the python library client.
In this cases, the custom exporters could be containeraized as long as they don't use any hardware metrics.

The installation process required for the library is simple using pip

```
pip install prometheus_client
```

After that we can run python code using the library.

Let’s see a code example for this library

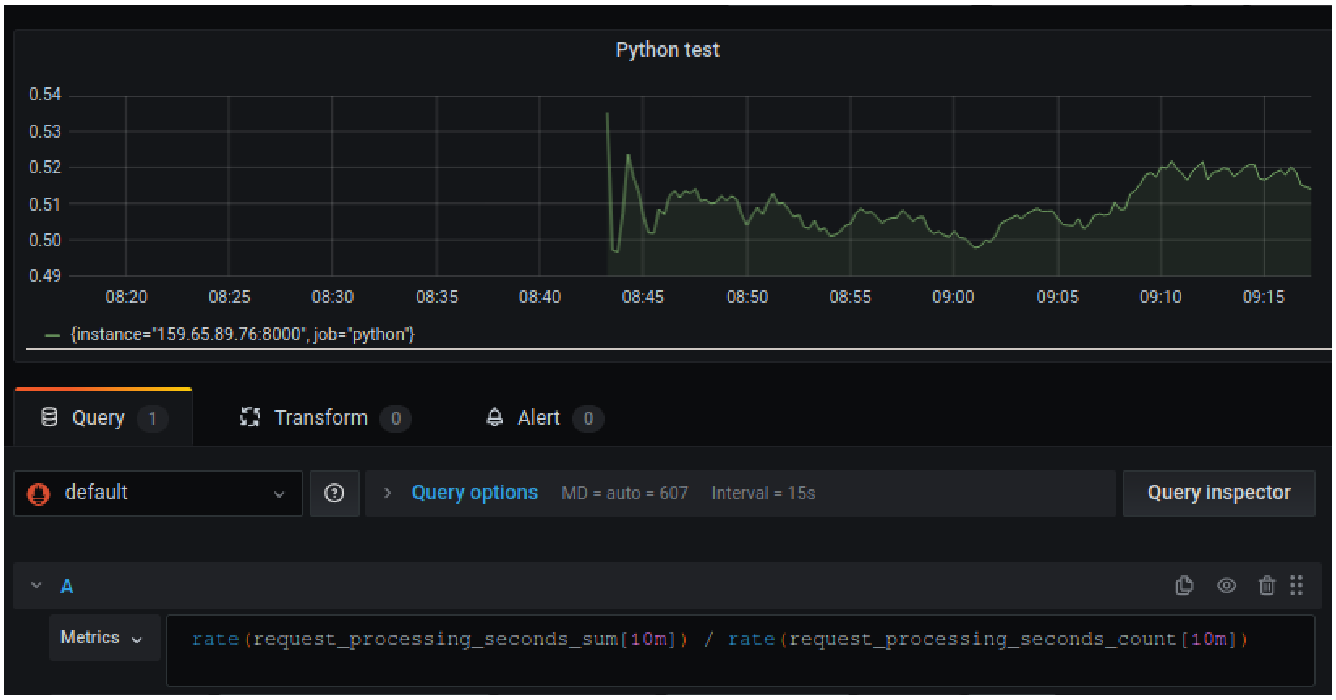
custom_exporter.py

```
from prometheus_client import start_http_server, Summary
import random
import time

REQUEST_TIME = Summary('request_processing_seconds', 'Time spent processing request')

@REQUEST_TIME.time()
def process_request(t):
   time.sleep(t)

if __name__ == '__main__':
       start_http_server(8000)
       while True:
          process_request(random.random())
```

The first line of code after importing libraries, creating the variable REQUEST_TIME, exposes a new Summary metric in the endpoint.
In the main loop of the program, it keeps calling a function that sleeps a random time between 0 and 1 seconds, and that time is a new entry in the summary metric created above. Then this summary metric will keep recording random numbers between 0 and 1.

Also note that a http server is started in port 8000 in order to expose the metrics to prometheus, and that the target should be added to the prometheus.yml file with al the jobs and targets to scrape.

After running this for a while, we can check in grafana with a PromQL query if this holds up



```
rate(request_processing_seconds_sum[10m]) / rate(request_processing_seconds_count[10m])
```

First of all, let’s observe the query made

```
rate(request_processing_seconds_sum[10m]) / rate(request_processing_seconds_coun[10m])
```

A summary tracks the number of observations, in this case, the number of times the function was called, and the total sum of the observations, in this case, the total time that was spent inside of that function. The function rate() just returns the average within that timeframe, so the query above, then, returns the average time spent in that function in the last 10 minutes.

As it hovers around 0.49-0.51, I can assume that it’s working as intended since it’s sleeping a random time between 0 and 1 seconds, so it will average at roughly that.

We can see more variance at the start of the execution, but that is because not even 10 minutes have gone by, so the result hasn’t stabilized yet at around 0.50

custom_exporter2.py

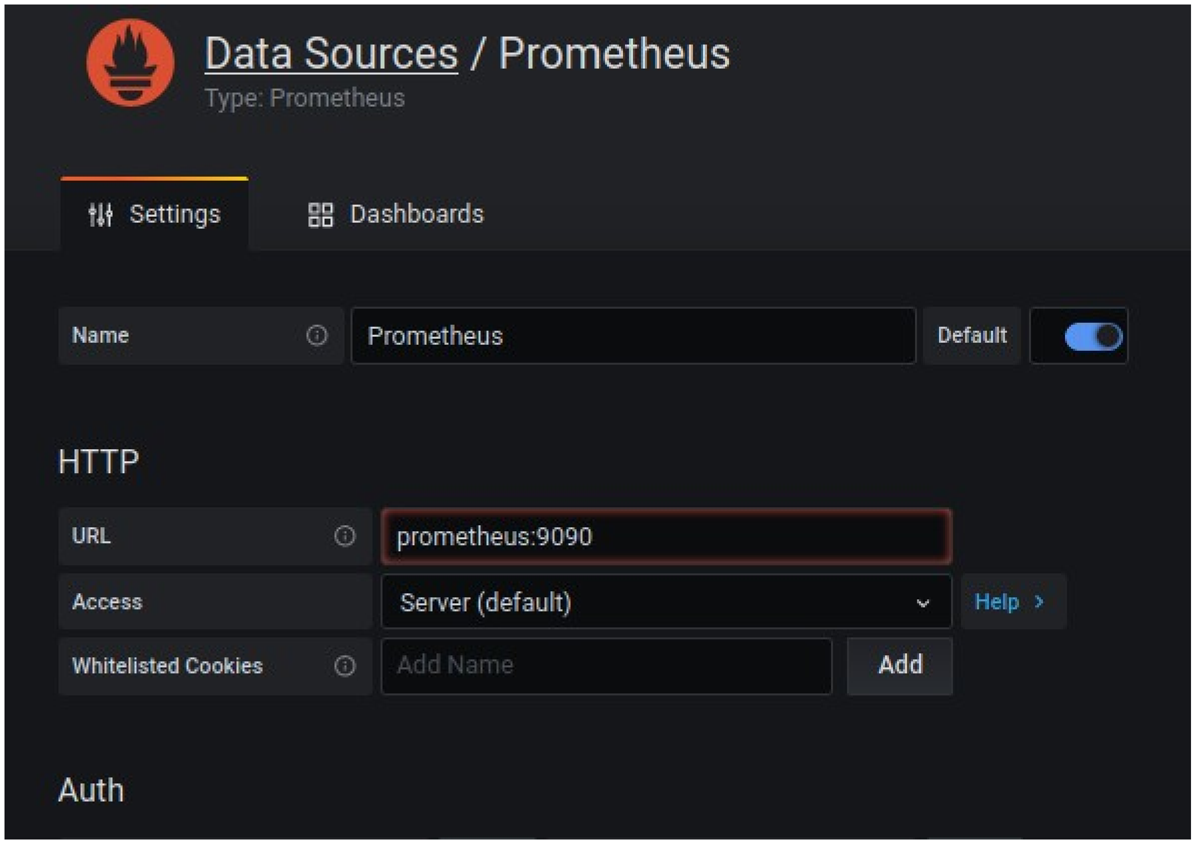
```
from prometheus_client import start_http_server, Counter
import random
import time

c = Counter('keyboard_counter', 'Keyboard counter')

def increment():
   c.inc()

if __name__ == '__main__':
       start_http_server(8001)
       while True:
          input()
          increment()
```

This second code is even simpler than the first, it takes a Counter metric (meaning it can only go up in prometheus) and increments it each time we press the enter key, exposing the metric at port 8001

5. Grafana

-Initial configuration

The only configuration I had to do for grafana is adding the prometheus data source from the web ui



Data Sources / Prometheus

Type: Prometheus

Name: Prometheus

Default: on

HTTP

URL: prometheus:9090

Access: Server (default)

Whitelisted Cookies: Add Name

Auth

After that it detects prometheus automatically, note that the url must be prometheus:9090 instead of localhost:9090

-Panels and dashboards

Grafana is organized in dashboards, which are workspaces that contain panels.

The panel is the basic visualization block in Grafana. A panel has a query editor that extracts the data from the source, in this case, from prometheus. With some notable exceptions, each panel represents one or more queries, that usually display data over time.

There is a lot of variety in terms of formatting options for each type of panel. They also can be rearranged or resized in the dashboard.

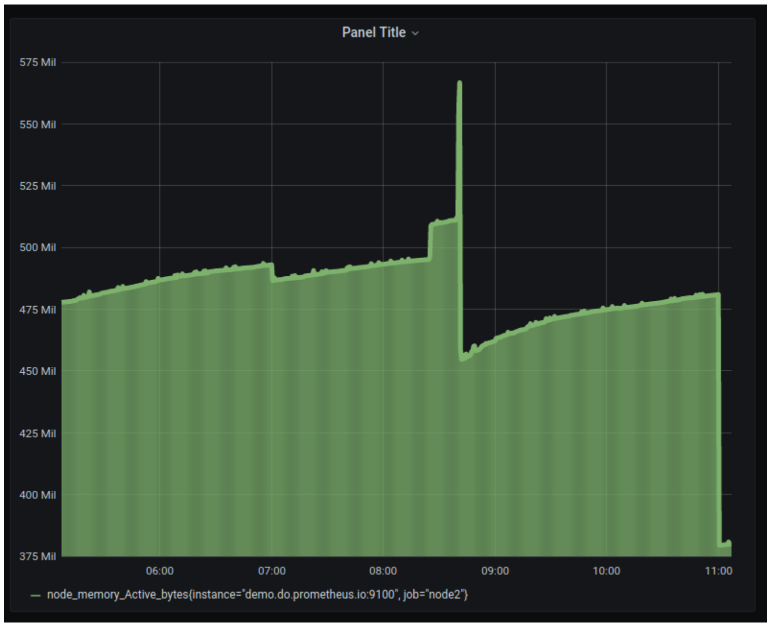
In this chapter I'll explore some of the visualization modes for panels and options

Grafana offers 5 visual representation for time-series data out of the box, with no plugins
For showing them all I will be using the same query, which is

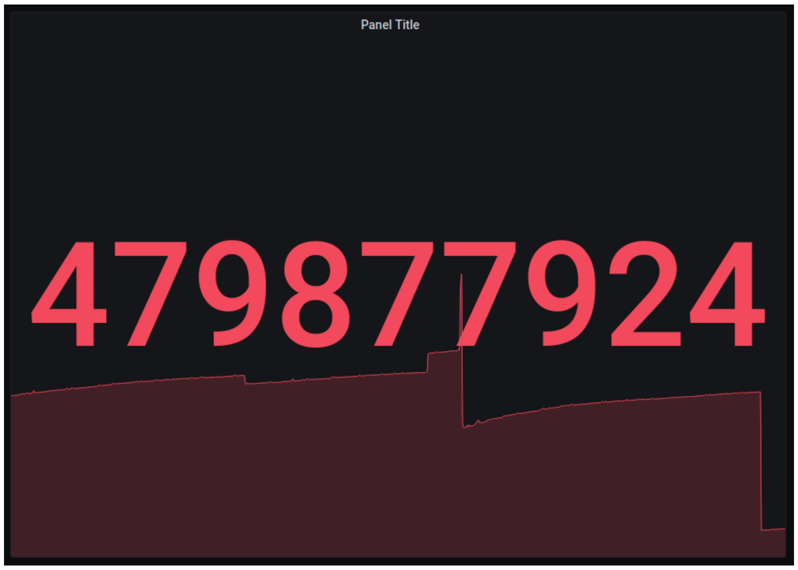
```
node_memory_Active_bytes{job="node2"}
```

-Graph

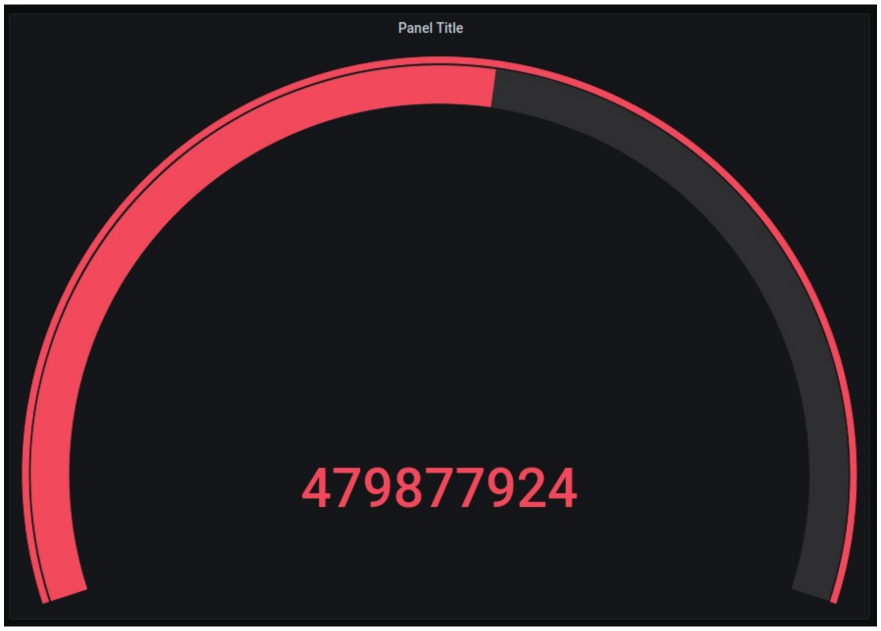
The most useful for most of the cases with prometheus, as seen before with the code example for the custom node exporter



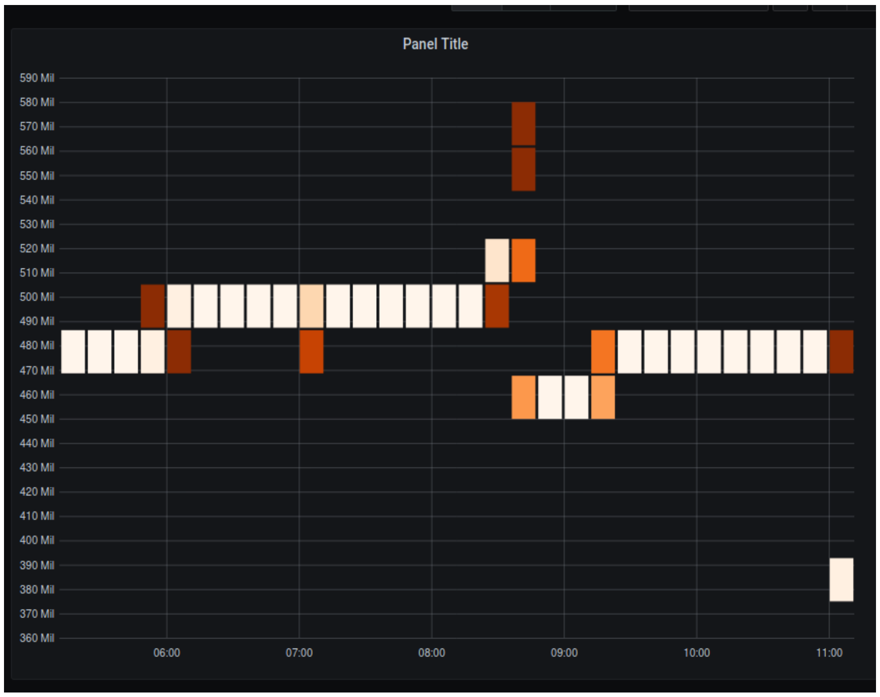
-Stat: Similar to the graph, but shows the actual graph without the scale and the current metric in a numeric form



-Gauge Specially designed for gauges, such as %CPU used, memory… but doesnt show the historic

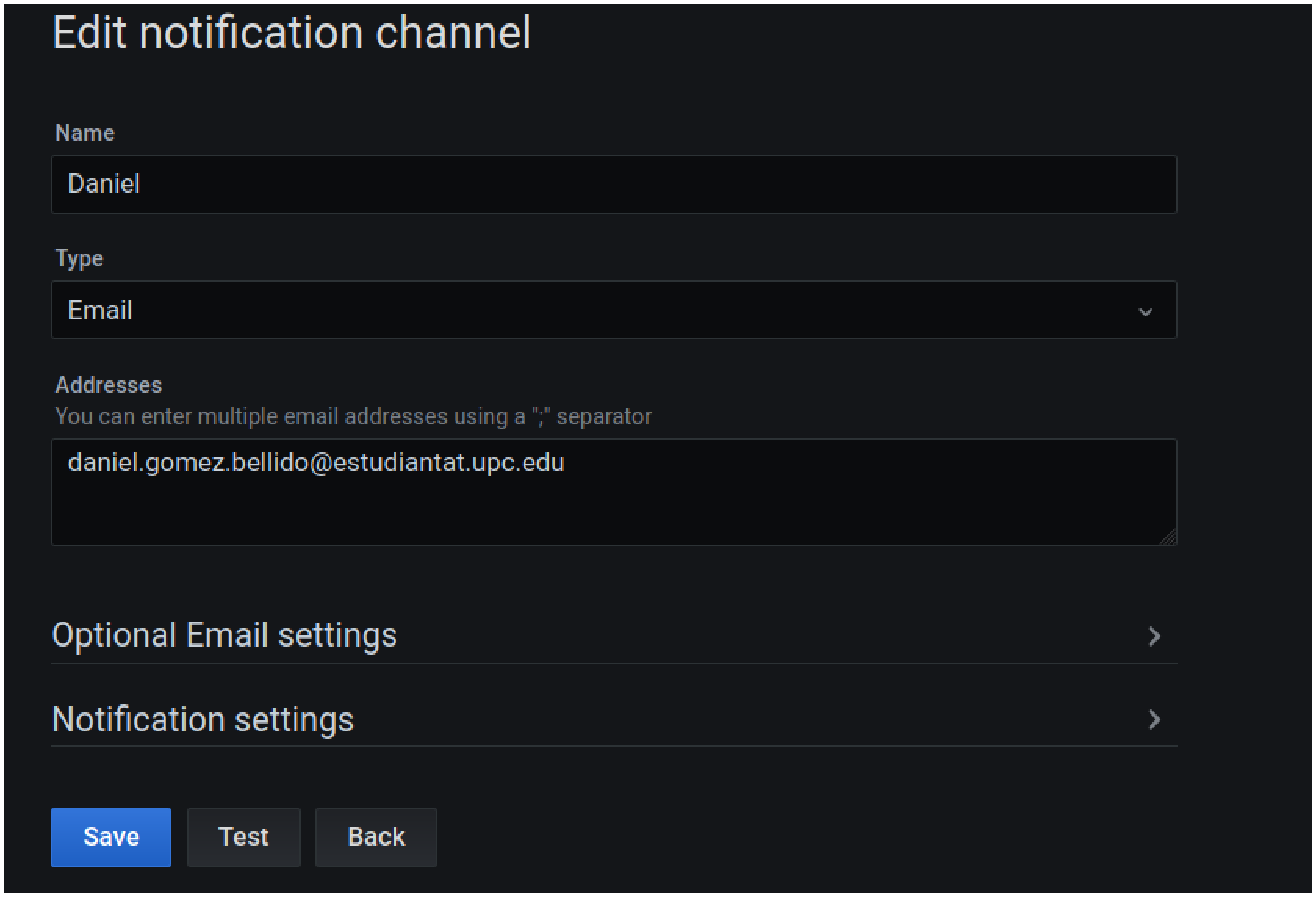


-Heatmap Shows the most common values in brighter colors in a map



-Alerting

Alerting in Grafana is made much easier than in Prometheus, as it is integrated in grafana out of the box and we can add notifications channels in the grafana control panel UI



Edit notification channel

Name: Daniel

Type: Email

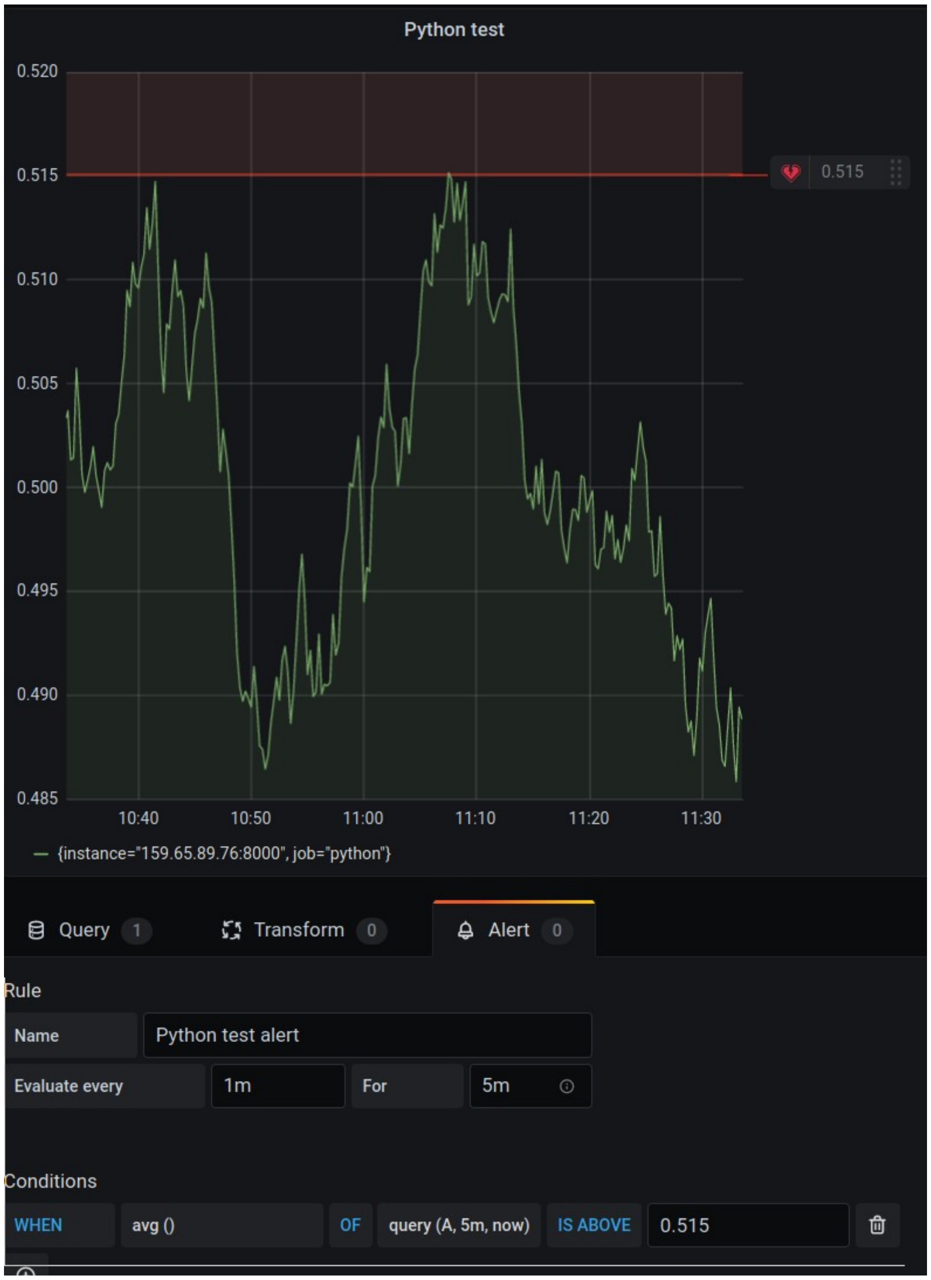
Addresses
You can enter multiple email addresses using a ";" separator

daniel.gomez.bellido@estudiantat.upc.edu

Here I added my email as an example, but there’s many more types of notifications.

Alert rules are also more straightforward to add, as they are added in the alert tab of any panel, letting you build and visualize an alert using already existing queries

For instance, I’ll create an alert for the code example in the node exporter, that notifies me via email if the query goes above 0.515



Rule

Name	Python test alert		
Evaluate every	1m	For	5m

Conditions

WHEN avg () OF query (A, 5m, now) IS ABOVE 0.515

The alert rule is straightforward to setup, with its own limited sentences, and also shows the threshold at which it will alert so you can compare it to the historic.

6. Performance analysis

The final requirement for the project was to have a small footprint, so for the tests I'll use a stress tester called Avalanche, that pulls a lot of metrics for prometheus, and then I'll monitor prometheus itself, specially the metric scrape_duration_seconds to have a target of 15 seconds, which is the default for prometheus, to consider it the capacity of the virtual machine.

This is the prometheus.yml file used for the test

prometheus.yml

```
# prometheus.yml
global:
   scrape_interval: 1s
   external_labels:
      monitor: 'my-monitor'
scrape_configs:
   - job_name: 'test'
     static_configs:
          - targets : ['localhost:9090']
   - job_name: 'avalanche'
     scrape_interval: 15s
     scrape_timeout: 10s
     metrics_path: /metrics
          - targets: ['159.65.89.76:9001']
```

The tests are run in relation to the number of series rendered to prometheus, which in avalanche is metric-count * series-count

Metric-count	Series-count	scrape_duration_seconds
200	200	0.5s
400	400	3.9s
600	600	5.6s
700	700	6.9s

At 800 metric count and series count, it crashed due to not having enough memory, so we can say that roughly 700*700 metrics is the cap in this machine, with the 4GB memory being the bottleneck, as it wasn't approaching the 15 second target yet. Which is in the range of approximately 500.000 metrics.

7. Sustainability Analysis

7.1 Project in production

-Environmental

The environmental costs of the project have gone down, since the computer I use now uses less power. However, it is very difficult to calculate the total power used since a virtual machine in the cloud was used for development mostly.

Having into account that, it would be almost impossible to calculate the gains of the new computer, because the rest of the costs have been the same and are well adjusted

-Economical

The economical prevision made in the initial feat was also adjusted correctly and there were no unexpected extra costs, however, there were probably some savings in the fact that some budget was reserver for unexpected costs, but it would be hard to calculate since it includes the hardware amortization

In conclusion, the cost was well adjusted and no more money was needed

-Social

For me personally the project made me think more about data and the importance of it, and the consequences that this kind of technology can have over our lifes. On an ethical level, I think data could be used for good but it will be very hard to not have some parties using it with only their own benefit in mind, which could be disastrous for society.

7.2 Life expectancy

-Environmental

If a similar software+hardware solution were to be deployed during a few years, it would definitely have a environmental impact. However if it was deployed like in the project, it would be almost impossible to calculate because we can’t know how much power a cloud virtual machine draws. Also, It could even improve the efficiency of small servers and data centes if used correctly, so depending on the use-case it would be benefitial.

-Economical

If the cost is the same as that with the project but with no development costs, it would cost just what the virtual machine costs per year.

So that would be about 120 euros per year, given that we want to run it on the same hardware of course.

-Social

As it was originally conceived, I see a positive social impact for this project, if it helps cutting costs for a company for example, it would be a positive change for society. As long as it’s not used to collect personal data of coursework

7.3 Risks

-Environmental

Environmentally, there’s the risk of expansion of the project, as It would have a bigger environmental footprint as it consumes more resources. However, if it got bigger it would also mean that it works in its original intention, which is collecting data for solving problems and ultimately improving the environment.

-Economical

Probably more sofisticated solutions exist or will exist that serve the same purpose, but that is a economic risk inherent to any technology, so I don’t think it’s significant

-Social

I don’t contemplate any social risk that harms any particular group

8. Conclusions

In conclusion, I think the project was completed with enough degree of success.
The software stack works and it can be instrumented with many languages for any custom application that it might need.
In terms of performance, with the hardware used it supports almost up to 500.000 metrics, with possibility of upgrading RAM to improve performance dramatically.

In terms of my personal conclusions, it has helped me develop previous knowledge to the project and realize the importance of data monitoring for solving problems.
Also to help me know more in detail some tools like docker that I didn't know that much about

References

1. Evi,, Nemeth, "Chapter 19, Web hosting, Content delivery networks". UNIX and Linux system administration handbook (Fifth ed.). Boston: Pearson Education. p. 690. ISBN 9780134277554. OCLC 1005898086, 2018.
2. M. Ruiz, M. Germán, L. M. Contreras, and L. Velasco, "Big Data-backed Video Distribution in the Telecom Cloud," Elsevier Computer Communications, vol. 84, pp. 1-11, 2016.
3. http://docs.graylog.org/en/3.0/pages/architecture.html
4. https://grafana.com/plugins?type=datasource
5. Namiot, Dmitry, "Time Series Databases" http://ceur-ws.org/Vol-1536/paper20.pdf
6. https://prometheus.io/docs/concepts/data_model/
7. https://linuxcontainers.org/lxc/introduction/
8. https://hub.docker.com/r/smashwilson/lets-nginx/
9. https://letsencrypt.org/es/
10. https://github.com/prometheus/node_exporter
11. https://prometheus.io/docs/instrumenting/clientlibs/