Master's degree in Data Analysis, Cybersecurity and Cloud Computing

POPBL - 2

Ander Bolumburu Casado
Álvaro Huarte Albas
Haritz Saiz Sierra
Onintza Ugarte Iguiñiz
Xabier Etxezarreta Argarate
Xabier Gandiaga Iriondo

June 2020

Abstract

Este documento describe el segundo proyecto POPBL de la universidad de Mondragon para el máster en Análisis de datos, Ciberseguridad y Cloud Computing. Este proyecto pretende combinar las ramas del análisis de datos, ciberseguridad y computación en la nube un gran proyecto común. El objetivo principal es crear una aplicación segura a partir de un modelo de machine Learning. El diseño e implementación de la aplicación tienen que ser seguros, así como realizar tests para la búsqueda de vulnerabilidades. También, la integración y el despliegue de las diferentes versiones de la aplicación serán gestionadas de manera automática.

This document describes the second POPBL project of the University of Mondragon for the master's degree in Data Analysis, Cybersecurity and Cloud Computing. This project aims to combine the branches of data analysis, cybersecurity and cloud computing into one large common project. The main objective is to create a secure application from a machine learning model. The design and implementation of the application has to be secure and testing for vulnerabilities must be carried out. Also, the integration and deployment of the different versions of the application will be managed automatically.

Dokumentu honetan Mondragon Unibertsitateko Datu analisia, zibersegurtasuna eta hodei konputazioko masterreko POPBL bigarren proiektua deskribatzen da. Lan honen xedea datu analisia, zibersegurtasuna eta hodei konputazioko zenbait kontzeptuen konbinazioan datza. Helburu nagusia aplikazio seguru bat garatzea da machine learning modelo batetik. Aplikazioaren inplementazioa eta diseinua seguruak izan behar dira baita. Gainera, aplikazioaren bertsio ezberdinen integrazioa eta hedatzea automatikoki kudeatuko beharko da.

Index

Illustrations

Tables

1. Introduction

The purpose of this document is to give a detailed description of the project developed in the second semester of the Master in Data Analytics, Cybersecurity and Cloud Computing. The project has consisted of merging the knowledge obtained during the semester by putting in practice the intelligence acquired in each lecture coursed. Therefore, the work that has been carried out has included many Computer Science fields.

The context of the work developed takes place in the United States elections of the year 2020. The two presidential candidates, Donald Trump of the Republican Party and Joe Biden of the Democratic Party, are engaged in a battle to become the next president of the United States. Social media are one of the most important tools to attract the crowds and gain popularity. The free time that candidates have is very limited during the election campaign, having little time to devote to social networks. This is where ITAPP company comes in. ITAPP offers a tool for the automatic generation of personal texts simulating the writing of both candidates. This way, they don't have to waste time thinking about what to write about.

The automated management of the different versions of the application and security have been essential in order to develop the elements fitting best the needs of the project. Data intensive applications have also been deployed for data transport and storage. Machine learning has also played an important role in the development of the product. The application has been developed to make the value provided by the model to be accessible in an easy way.

1.1 Objective

ITAPP is a startup that wants to bring to its customers the knowledge of its workers in the areas of artificial intelligence, cyber security and CICD.

Its first developed application called ITAPP, aims to offer an automatic text generation service based on a person's writing style. This service will be offered in a web application making it accessible to anyone with internet access.

To this end, the following objectives have been defined:

- Create a secure application from a machine learning model.
    - The development will follow a secure design and implementation as well as testing for vulnerabilities.
    - Continuous integration and deployment of the different versions of the application
- Create or collect the data locally from different sources.
- Data will be sent in streaming to two cloud providers.
    - Data will be sent securely and validated at both ends.
    - Cloud provider 1
        - It will be used safely for the evaluation/execution of the model.
        - It will have to be the developed application (and not another one from third parties) who collects the data coming from streaming.
    - Cloud provider 2
        - Data will be safely stored for use in the creation of the ML model
- The machine learning model will need a computer cluster for training, not for execution.
    - The training of the model will be done in a cloud provider. The data necessary for the training will be stored in this provider.
    - The testing of the ML model and the execution will be carried out at a second cloud provider. In this provider the data will arrive in streaming and will be discarded once the result is obtained.
- The machine learning model will be trained or prepared to train every X time.
    - Each version will be a different application
    - Is necessary to check that the new version improves the result of the previous version before putting it into production.
    - It will have to be checked that it works well before it goes into production.
- The result of the model execution will be displayed locally.
    - The visualization will be independent from the creation of the data.

2. Proposed solution

This section will detail the process followed to design the solution. The solution considers all the previously presented requirements. Available data obtained directly from three

different data sources will be used: Twitter API, Tumblr API and a dataset formed by political tweets. The project emulates the following scenario:

Data (tweets) have been stored during the last days consuming directly from both APIs. Using the currently available data, a model has been created that is able to generate tweets with sense related with the topic that has been written by the client thought the application (ITAPP).

New data is generated from each the data sources steadily. Our solution is ready to capture the new data and process it with different validation and sanitization methods that "clean" the data before stored it. Our previously trained model is fed with this new data in order to generate tweets with more sense.

Finally, the results are published via twitter in our official PBL account.

It should be noted that the CI/CD methodology has been used to carry out an automation of the different stages, highlighting the following:

- **Test model:** stage where the model is validated, if the results obtained are better than the previously stored the model is updated.
- **Test app*:*** stage where SonarQube tool and black library are used to realise code static analysis of the developed application.
- **Deploy-staging*:*** Blue Green architecture have been selected to create a Zero-Downtime deployment of the application.

The whole process has been developed using the agile methodology that has allowed us to work in an organized way during all the time of the project.

2.1 Presenting ITAPP

Once the project requirements have been identified and the solution to the problem has been correctly defined, was necessary to develop a web application that allows the client to write the topic that will later be used to generate the tweet, in this case ITAPP.

The community managers are the ones in charge of managing and directing the social networks of the companies they work for or managing the social network profiles of public figures such as politicians. In this way they have a direct relationship with the

public and their objective is to try to capture their attention with interesting tweets about the company's theme or a current situation in the country such as an election.

ITAPP has been designed due to the proximity of the USA 2020 elections and allows the Republican party's press chief to generate tweets adopting the style of President Donald Trump, thus being able to capture the attention of his followers.

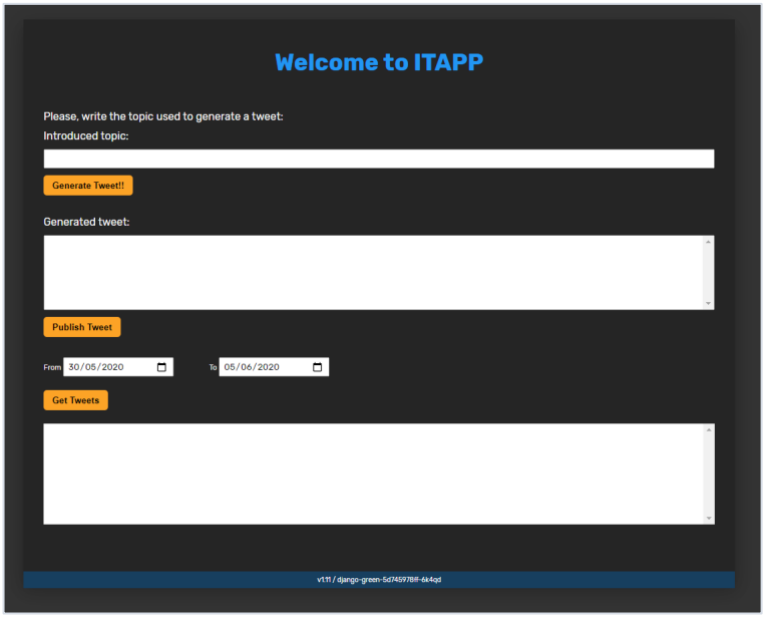

Illustration 1. Itapp application

The image above refers to IPAPP website interface, which will be accessible to customers.

It is divided into three parts:

- **Introduce Topic:** section where the client introduces the topic which will be used to generate the tweet such as: police, USA, antifa...etc
- **Generate Tweet**: section where the client can visualize the tweet that has been generated and publish it later in the official ITAPP account.
- **Get tweets**: option that allows the client to visualize the tweets related with the written topic that have been generated within a certain time range.

2.1.1 Data sources

The most crucial aspect of a big data project is the treatment of the data, capture, transfer, process…etc. Sometimes data does not come in a suitable way, and processing systems have to be applied in order to adjust the data to a "own standard" or a format that allows for an easier data analysis.

For this project, a large amount of data was needed to train and generate the model, which had to come from three different sources:

- **Twitter API**: As first source Twitter have been selected, this is one of the most famous social networks of the planet, which generates millions of tweets daily that contains huge amount of information. This data is accessible to companies, developers, and users through its API. It is a way to capture a large amount of data in a short time in a simple way.
- **Tumblr API**: As second source Tumblr have been selected, is another social network not as famous as twitter but it fit our requirements. In this case, instead of publishing tweets as Twitter stories, photos, links, or videos are shared and published. Tumblr has its own API to access its public content.
- **Huge Dataset**: The attempt was made to use another social network (reddit) as a third source of data, however this was not possible because no answer from the social network was received so finally a dataset whose contents tweets was selected as third source.

2.2 Task plan

For this type of project where many members are involved and there are a considerable number of tasks, a mandatory practice is use a tool that allows to manage and control all the tasks and the time required for each one of them.

In this case the Gantt tool has been used. The Gantt chart is a tool for planning and scheduling tasks over a given period. Thanks to an easy and comfortable visualization of the planned actions, it allows to follow up and control the progress of each of the stages of a project.

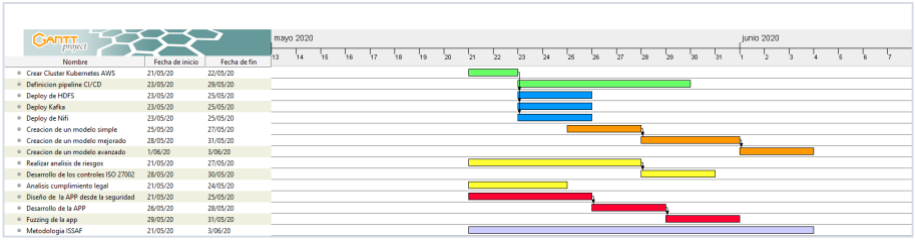

Illustration 2. Gant

3. Development

Once the requirements and scope of the project are clear, it is time to think about the development of the project.

For this purpose, a concept that has been on the rise in recent years, the Big Data has been used. The objective is to achieve an efficient and fast handling of the data, allowing its capture, processing, cleaning, transfer, and storage in a distributed and simple way.

Finally, unite the concept of Big Data with Machine Learning, thus allowing to train a model that can generate meaningful tweets.

3.1 Big data infrastructure

The core element of the application is the machine learning model that is in charge of generating new twits using a pre-defined topic. In order to train the model, some data must be provided to it. For this reason, the team has designed an architecture a three-level architecture known as cloud, gateway and input.

Although the gateway infrastructure as well as the input infrastructure should be deployed on premise, due to external circumstances they also have been deployed on the cloud. The idea is that the architecture is presented in those two levels is valid for local devices non hosted on the cloud.

All the infrastructure has been deployed into one cloud provider, Amazon web services to avoid higher maintenance cost due to data roaming between different providers.

Basic architecture

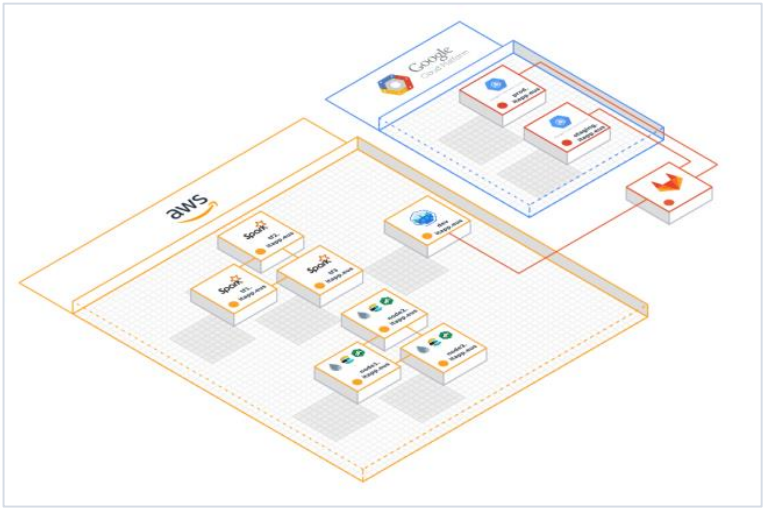

Illustration 3. Basic architecture

The aim of this section is to provide a brief overview of the infrastructure needed for the project. In later sections, each element of the architecture will be furtherly detailed and explained.

ITAPP web application is deployed into three different stages. One for the developers deployed in AWS, another one for staging and the last one, the production environment that is available to the general public. These two last environments are hosted in another cloud provider, Google Cloud.

The application's core is a machine learning model that needs from data to be trained. The process of recollecting, enriching, pre-processing and storing data is dived among three different areas.

The first two should've been deployed locally but ended up also being deployed on AWS although all configuration has been designed to be used by non-hosted cloud machines. These two areas are called Input and Gateway. The purpose of the Input is to obtain streaming data, enrich it with some extra data and to send it to the gateway. The gateway's job, on the other hand, is to forward data to the cloud.

The last area is called Cloud, were a three-node multipurpose cluster has been set up with many technologies such as Elasticsearch, Hadoop, Nifi or Spark that receive streaming data, enriches them, and finally, it is stored.

With all the stored data, a distributed pre-processing system using Spark is set up to transform the raw data into specific input data that our neural network models admit.

3.1.1 Data capture (Twitter, Tumblr, Script)

The data that is used to train the deep learning models on is captured through Apache NiFi which is an integrated platform for processing and logistics of data in real time, to automate the movement of data between different systems in a fast, easy and secure way. Data is treated on the go passing through different filters, scripts and processor that validate and sanitize it. Once it reaches the end of the Apache NiFi workflows it is directed to HDFS in order to be read by the Jupyter Notebooks that create the deep learning model.

This transfer flow has been split in three different points: Input, Gateway, and Cloud. Each of them has specific tasks and locations within the flow that allows to follow a consolidated chain of work.

All of the inputs that are obtained by Apache NiFi are JSON files of some kind following a known JSON schema.

3.1.2 Data storage (ES, Hadoop)

The data storage used by Itapp can be divided into two main parts, those being the Hadoop cluster and the machine that hosts Elastic Search. Both of them are hosted in AWS and their finality is to save all the data used by Itapp and generated by Itapp.

3.1.2.1 Hadoop

The Hadoop cluster offers an HDFS (Hadoop Distributed File System) were files that contain posts that are going to be read by the Spark are placed. These files will be JSON files and will be placed there either by Apache NiFi or the web application.

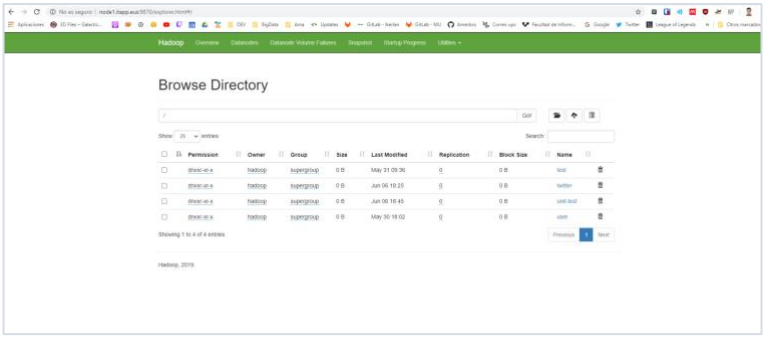

Illustration 4. Browse directory

The main reason for using Apache Hadoop is Itapp's structure does not make any complicated queries whilst reading data for the deep learning training steps, which is also the only moment when something is read from HDFS in Itapp's entire workflow. Thus, nothing more advanced than a simple filesystem or NoSQL documental database was needed.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
   <property>
      <name>dfs.namenode.name.dir</name>
      <value>/home/hadoop/data/nameNode</value>
   </property>
   <property>
      <name>dfs.datanode.data.dir</name>
      <value>/home/hadoop/data/dataNode</value>
   </property>
   <property>
      <name>dfs.replication</name>
      <value>2</value>
   </property>
   <property>
      <name>dfs.webhdfs.enabled</name>
      <value>true</value>
   </property>
</configuration>
```

Illustration 5. replication in Hadoop

The files present in HDFS are mirrored between all the nodes of the cluster. This is called "replication" in Hadoop.

3.1.2.2 Elastic Search

Illustration 6. ElasticSearch

Elastic Search, on the other hand, exists to gather individual tweets that are generated by the web application. Once a user submits a new tweet this tweet will be saved in Elastic Search with a timestamp as its index value.

This allows the web application to query for tweets that were generated in the past and the users to limit the range of time from which tweets have to be pulled from. Generating and executing this timestamp-based query is computationally very expensive in something like a simple filesystem and will also add a lot of delay to the response times.

Elastic Search helps Itapp bypass these problems.

The queries to the elastic search engine are made through HTTP request forged by the elastic-dsl library for python.

3.1.3 Data transfer

3.1.3.1 Input

The first machine, the one that takes care of data capture from the three different data sources, pre-processing and enrichment is Input. It is a single machine housed in AWS.

In the following paragraphs all the processors involved in Input will be explained, a division of them will be made according to the tasks they perform, facilitating the understanding of the long and complex flows we employ.

As discussed above there are three different data sources, therefore there are three data streams that share most of the infrastructure except for the first processor to capture the data, thus, the specific processor for each source will be explained individually, finishing with the common processors of the flows.

Twitter Source

To capture data from twitter the "Get Twitter" processor has been used:

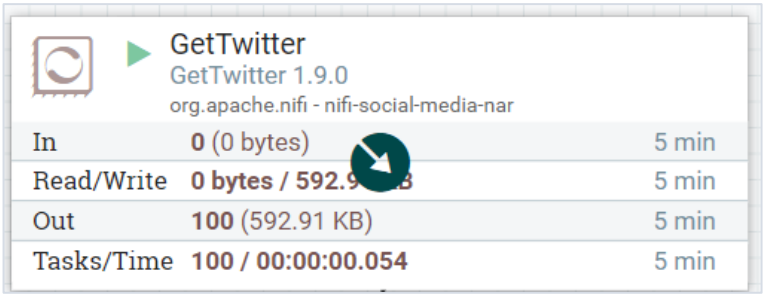

Illustration 7. Twitter source

This processor gets data from the twitter API with the following configuration:

- Terms to filter on: the key word that will be used to get tweets related with that word. (two processor have been created, one for Trump and one for Biden).
- Languages: it filters tweets based on their language tag, in our case that is "en" meaning English.

Tumblr Source

There is no specific processor for capture data directly from Tumblr so a common "Invoke Http" processor was used:

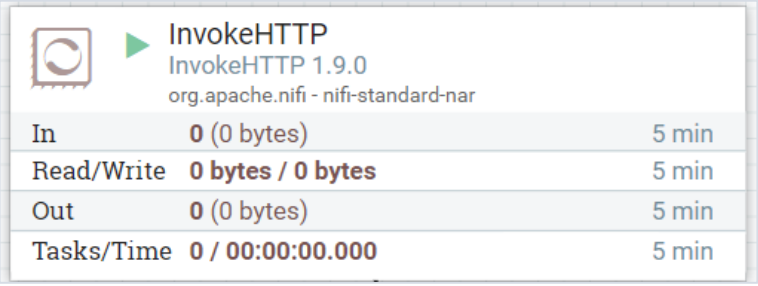

Illustration 8. tumblr source

This processor gets data from the Tumblr API with the following configuration:

- **Remote URL:** direction of the Tumblr API where is necessary to specify the tag as Trump (key word for the tweets), and the type of data required (only text in json format).
- **Additional settings were used were we specify:**
  - Limit: The number of posts to return.
  - Tag: Return posts tagged with the following keyword.

Dataset Source

The third data source consists of a dataset containing tweets about the USA's politics, so a script has been generated that will simulate data capture and store it on another file simulating new generated data.

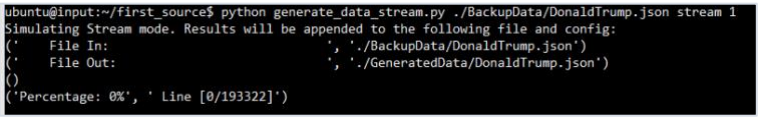

Illustration 9. Dataset source

The "Tail file" processor has been used for this purpose which will capture new data generated in a specific file:

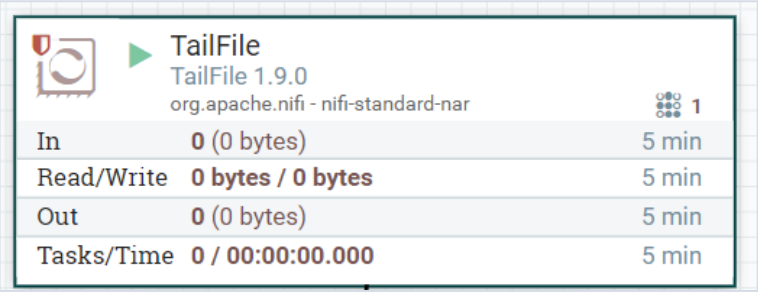

Illustration 10. Tail file processor

- **File to tail:** full path of the file where data will be generated through the homemade script.

Once the specific processors for each data sources have been explained, the next step is to show in detail the common flow of data these three data sources share before they are sent to the next point/machine in the flow.

Due to the length of the flow that follows we have decided to split it in two parts. The first one consists of the pre-processing of the data, which includes filtering, sanitization and enrichment of the data, and the second one that includes the validation and transference of it to the Gateway.

Pre-processing - Sanitization and Enrichment

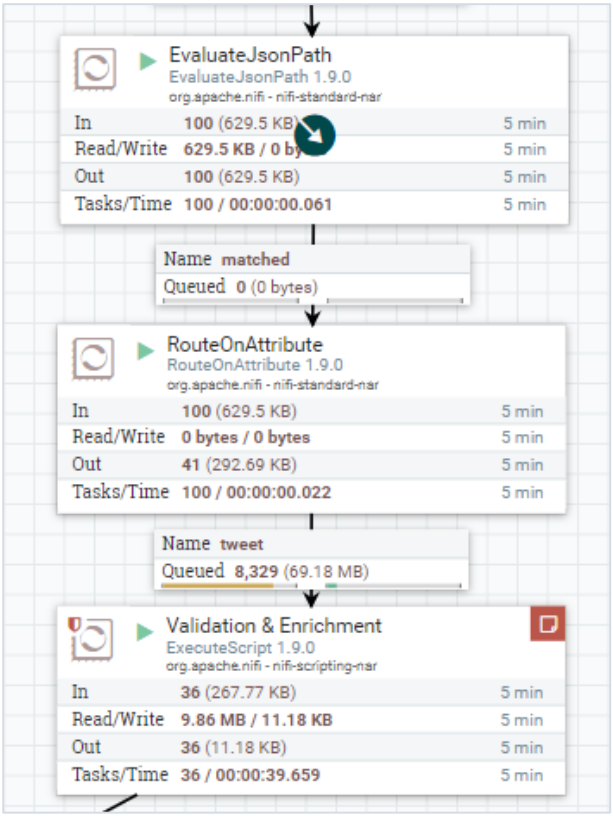

Illustration 11. Pre-processing

Three processors are involved in the first part, each of them has a specific task as explained here:

- **EvaluateJsonPath**: proves that the value you get is a Json. Otherwise, the flow will fail. It also adds attributes to the flowfile so processors further down the chain can access data on the flowfile.
- **RouteOnAttribute**: checks if the incoming data has the key/values we want. Also, if the data comes from twitter it checks that the language is correct, in this case English. Otherwise, the flow will fail.

- **ExecuteScript**: processor that executes a script which eliminates all those characters that are not considered as valid. The incoming data goes through an enrichment adding:
    - A new column which is the data source from which the data comes (Twitter, Tumblr or Dataset).
    - A tag that explains the topic from which we are consuming this JSON file (If we are consuming Trump's tweets this tag will say "Trump").

On the ExecuteScript processor

The sanitization and filtering of data happens through a Python script that is executed in a Java Process through NiFi. This meant instead of using Python it technically speaking uses Python, which is a Java binary that runs Python code. It has a very important downside, which is you cannot install new Python libraries on it easily.

Thus, the script we developed uses only libraries that are embedded in Python.

```
def process(self, inputStream, outputStream):
    text = IOUtils.toString(inputStream, StandardCharsets.UTF_8)

    # Tratamiento

    printable = set(string.printable)
    no_emojis = ''.join(filter(lambda x: x in printable, text))

    cleanr = re.compile(
        '<.*?>|&([a-z0-9]+|#[0-9]{1,6}|#x[0-9a-f]{1,6})')
    clean_text = re.sub(cleanr, '', no_emojis)
    clean_text = clean_text.replace("\\n", "")
clean_text = clean_text.replace("\\u201c", "\"")
clean_text = clean_text.replace("\\u2019", "\'")

    text = clean_text

    # Cambio de JSON
    obj = json.loads(text)
    newObj = {
        "Source": "Twitter",
        "Tag":"Trump",
        "Text": obj['retweeted_status']['extended_tweet']['full_text']

    }
    outputStream.write(
        bytearray(json.dumps(newObj, indent=4).encode('utf-8')))
```

Illustration 12. sanitization and filtering

The explanation of the script is pretty straightforward:

1. The data from the flowfile is read in a UTF8 encoding to the "text" variable.
2. That string is escaped to only include printable characters.
3. After that, a regex pattern deletes:
    a. HTML tags

b. Encoding artefacts that start with & or #x

4. We manually replace the characters we want to maintain (new line, quotemark, single quotemark) as they tend to be linked to other words thus dodging the regex pattern.
5. We reshape the JSON file so it has the key/values we want whilst adding the enrichment.
6. The new JSON file is written to a new flowfile.

This is the processor that consumes most of the horsepower the Input machine has, as it needs to sanitize the entire JSON file before reshaping it, thus, it is the only processor we have set up to have more than one execution thread.

Validation and Data Transfer

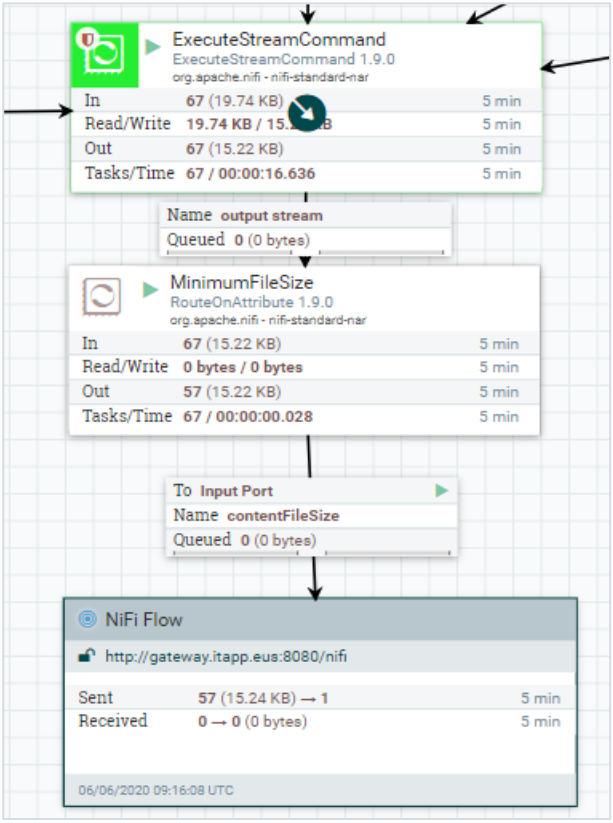

Illustration 13. Validation and data transfer

- **ExecuteStreamCommand:** this processor uses a locally saved python script which validates incoming data to check the JSON flowfiles against a JSON schema:
  - **Source:** check if the source key's value is string type and if it corresponds with one of these three options: Twitter, Tumblr, and Dataset. Otherwise, the flow will fail.

  - **Text**: check if the text pattern is in string type and has a length between 280 and 3. Otherwise, the flow will fail.
  - **Tag:** check if the tag pattern is in string format. Otherwise, the flow will fail.
- **MiniumFileSize**: check that the incoming data is not empty.
- NiFi Flow is a remote process group, which oversees sending the data from the input to the next point, the Gateway.

NiFi Flow is a remote process group, which oversees sending the data from the input to the next point, the Gateway.

On the ExecuteStreamCommand processor

The ExecuteStreamCommand processor takes care of the validation of the JSON files against a JSON schema.

The only way in which we could check a JSON file against a JSON schema in Python was using a non-embedded library for Python, thus, we could not use the same processor as before, ExecuteScript. Installing Jython libraries on NiFi is pretty tough.

We tried to make this validation check through Clojure and Golang scripts, to bypass this problem, which had no results at all. We also tried to look for a custom NiFi processor that would check a JSON flowfile against a JSON schema, but, it seems those code lines were taken out from the NiFi Issues on their official Jira and Github accounts. These issues are still open and no functional JSON schema validators for NiFi exist.

The approach we went for in the end was to use a ExecuteStreamCommand processor, which executes a Python script with a Python environment that is installed in the machine instead of using Jython.

This had a problem however, which was we could not get access to the Apache Commons binaries that Jython uses to access flowfiles. Thus we had to manually read and write the FlowFiles with a buffer. This created a empty flowfile problem, which is detailed in the fuzzing section of this document.

```
#!/usr/bin/python3
import sys
import json
from jsonschema import validate

schema = {
            "type": "object",
            "required": ["Source", "Text", "Tag"],
            "properties": {
                "Source": {"type": "string", "pattern": "Twitter|Tumblr|Reddit"},
                "Text": {"type": "string", "maxLength": 280, "minLength": 3},
                "Tag":{"type": "string"}
            }
        }

wb = sys.stdin.buffer.read()
sys_json = wb.decode("utf-8")
sys_json = json.loads(sys_json)

try:
  validate(instance=sys_json, schema=schema)
except:
  valid = False
else:
  valid = True

if valid:
  sys.stdout.write(json.dumps(sys_json))
else:
  sys.stdout.write("")
```

Illustration 14. ExecuteStreamCommand

The script runs as follows:

1. Select the python environment to run this script on.
2. Write the JSON Schema.
3. Read the flowfile from a buffer as a UTF8 text.
4. Load a json from it.
5. Validate the JSON.
6. Route it according to the schema check.
    a. The problem mentioned earlier has to do with the second stdout. If this stdout does not exist the processor crashes. If it exists it writes empty flowfiles on checks of the schema that fail.

On the MinimunFileSize processor

This processor was created after the fuzzing tests revealed the problems with empty flowfiles. It checks for 0Kb and NULL flowfiles with a NiFi expression and routes only non-empty flowfiles.

Required field

Property	Value
Routing Strategy	Route to Property name
contentFileSize	${fileSize:gt(0)}

Illustration 15. MinimunFileSize processor

3.1.3.2 Gateway

Once data arrives to the gateway, it is simply forwarded to the corresponding NiFi cloud instance. No processing is required at this point.

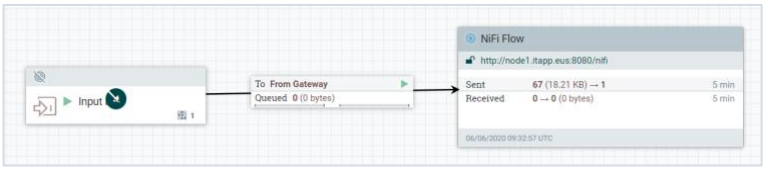

Illustration 16. Gateway

3.1.3.3 Cloud

Finally, when the data arrives to the Cloud it passes through a series of processors before being stored in HDFS.

- **ExecuteScript**: processor that uses a script to enrich the incoming data, in this case the corresponding time-stamp is added when data arrives.
    - We will not get into more detail as it is almost a mirror copy of the enrichment detailed in the Input section.
- **MergeContent**: each incoming flowfile contains a json. This processor merges ten individual jsons into one, generating new data that contains ten rows per file.
- **UpdateAttribute**: processor that adds the .json extension at the end of each incoming data file.
- **PutHDFS**: last data transfer processor, which is responsible for storing all the incoming data in Hadoop.

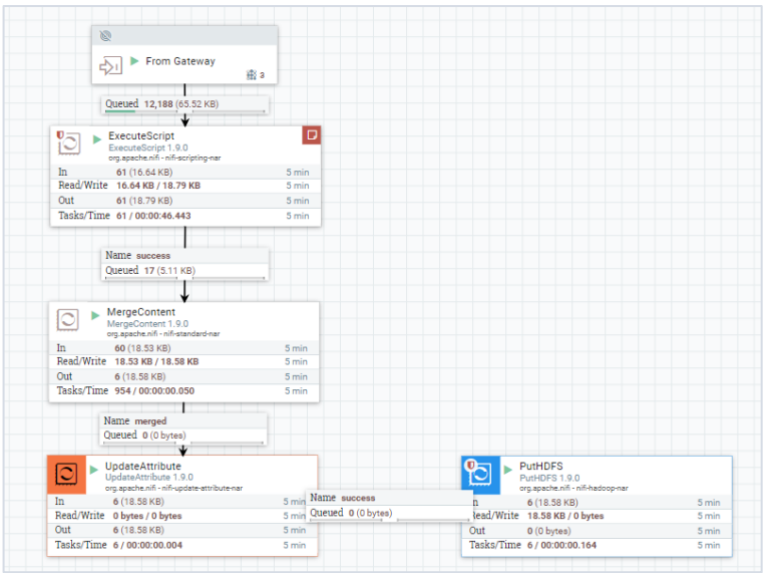

Illustration 17. Cloud

3.1.4 Deep learning problem modelling

Text Generation is a type of Language Modelling problem. Language Modelling is the core problem for a number of natural language processing tasks such as speech to text, conversational system, and text summarization. Language models can be operated at character level, word level, sentence level or even paragraph level.

Our work has focused on generating models that are capable of generating text. By modelling the way, a person writes, we wanted to be able to generate text as if we were that person. Thinking about the United States elections in 2020, we thought it was a good idea to collect tweets from the two candidates for the presidency: Donald Trump for the Republican Party and Joe Biden for the Democratic Party. First, we will generate a model with Donald Trump's data and then based on this trained model, we will do a transfer learning with Joe Biden's data and be able to generate text like him.

Due to the data we have available, short texts that are independent of each other, we have analysed two types of systems to provide a solution to our problem: character-level and word-level systems.

Character-level language models are often quicker to train, requiring less memory and having faster inference than word-based models. This is because the "vocabulary" (the number of training features) for the model is likely to be much smaller overall, limited to some hundreds of characters rather than hundreds of thousands of words. Character-based models also perform well when translating words between languages because they capture the characters which make up words, rather than trying to capture the semantic qualities of words.

In contrasts, word-level language models tend to display higher accuracy than character-level language models. This is because they can form shorter representations of sentences and preserve the context between words easier than character-level language models. However, large corpuses are needed to sufficiently train word-level language models, and one-hot encoding isn't very feasible for word level models.

These two methods are used in conjunction with recurrent neural networks. These types of networks can be used also as generative models. This means that apart from being predictive models, they can learn the sequences of a problem and generate new

sequences. For the generation of text we will start with a seed defined by the user. Starting from this introduced text, we will be able to generate text of any length.

3.1.4.1 Character-level language modeling

Character-level models consist on entering a sequence of characters and predicting probabilities of the next character. We have focused on the methodology proposed by Andrej Karpathy for character by character text generation. In his blog he proposes the generation of a multilayer recurrent model. He talks about the effectiveness of recurrent neural networks to model sequences, how to generate a character level model and shows results achieved with his method.

The following details our implementation of the model following the Karpathy methodology and the universal flow of machine learning.

3.1.4.1.1 Metric election

As we want to obtain the probability of each character to be the next, in this last layer we have used the '*softmax*' activation function. Is given by:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Illustration 18. softmax

As a result of the selection of this activation function, categorical cross entropy has been used as a loss function. This function is what we use to measure the error at a *softmax* layer. Is given by:

$$L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N}\sum_{n=1}^{N} \left[y_n \log \hat{y}_n + (1-y_n) \log (1-\hat{y}_n) \right]$$

Illustration 19. crossentropy

3.1.4.1.2 Evaluation protocol

For the evaluation of the model and to ensure that the performance is as expected, we have divided the data into 2 blocks, train (90%) and test (10%). The training data has been used to train the model while the test data has been used to evaluate it. When training the model, Keras allows you to split the data and have a validation block. We have defined that from the train data, 10% is validation.

As a summary, 80% of the data have been used for training, 10% of the training data for validation and 10% of the data to evaluate the model once trained.

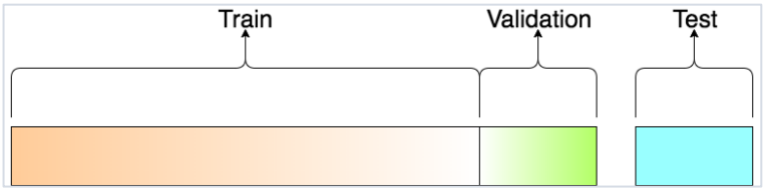

Illustration 20. tren/validation/test split

The validation of the model with the test data has been integrated into the continuous integration and deployment pipeline of our application. In this way we are able to automate the evaluation of the model and control that our application does not deploy a worst model (with a higher loss) than it already was.

3.1.4.1.3 Pre-processing (distributed - SPARK)

The pre-processing of the data or the adaptation to train the model has been done in a distributed way with Apache Spark. As the data is stored in Apache Hadoop, the first step is to make a reading and get our data in a DataFrame. First, we generate a Spark context with the connection configuration to our Spark cluster. From that context we get the Spark session.

```
conf = SparkConf() \
    .setAppName('ITAPP') \
    .setMaster('spark://tf2:7077') \
    .set("spark.executor.memory", "12g") \
    .set("spark.driver.memory", "12g")

sc = SparkContext(conf=conf)
spark = SparkSession(sc)
```

Illustration 21. pre-processing spark

With the session we make a request to obtain all the data. They are stored in json format. We filter the dataframe to have only Donald Trump's data.

```
df = spark.read.option("inferSchema", "true").json('/home/ubuntu/export/*.json')
df = df.select("Text").where("Tag == 'Trump'")
df.show()
```

Illustration 22. read spark

If we plot the dataframe, we have a single column dataframe called 'Text'. Each row represents a tweet written by Donald Trump.

```
+--------------------+
|                Text|
+--------------------+
|64-0.That's the r...|
|The rebuke of Tru...|
|Vote Red America!...|
|Breaking via The ...|
|Trump acts tough,...|
|Asked what messag...|
|Esper spineless a...|
|Just had a Trump ...|
|Before y'all star...|
|When it comes to ...|
|You are witnessin...|
|.@PressSec won't ...|
|Trump now claims ...|
|BREAKING: The Tru...|
|The horrific murd...|
|They are going to...|
|Y'all gonna be SO...|
|Fun Fact: Leaders...|
|@drawandstrike If...|
|Mitch McConnell h...|
+--------------------+
only showing top 20 rows
```

Illustration 23. Show tweets

With the data already available, the pre-processing followed has consisted of the following steps.

- **Data cleansing:** This step consists of removing the characters we are not interested in from the text. A function has been developed that receives as input a text, makes a cleaning using regular expressions. With this function and a UDF (User Defined Function) we can clean the text in a distributed way. The following changes have been made to the text.
  - Transform the characters to lowercase.
  - Remove quotes.
  - Remove URLs.
  - Remove line breaks.
  - Remove multiple whitespaces.
  - Replace '&amp' with 'and'.
  - Replace abbreviations.
  - Remove twitter mentions (@menion)
  - Remove twitter tags (#tag)

- Remove special characters.

```
import re

def clean_text(t):
    # to lower
    t = t.lower()
    # remove quotes
    t = re.sub(r'"@.*', '', t)
    t = re.sub(r'^".*"$', '', t)
    # remove URLs
    t = re.sub(r'https*:\/\/\S*', '', t)
    t = re.sub(r'pic\.twitter\.com\/\S*', '', t)
    # remove \n
    t = re.sub('\n', ' ', t)
    # remove extra whitespaces
    t = re.sub(r'\s+', ' ', t)
    # replace '&amp' with 'and'
    t = re.sub('&amp;', 'and', t)
    # replace abbreviations
    t = re.sub("'ll", ' will', t)
    t = re.sub("won't", 'will not', t)
    t = re.sub("n't", ' not', t)
    # remove @mention
    t = re.sub(r'@[A-Za-z0-9_]+', '', t)
    # remove #tag
    t = re.sub(r'#[A-Za-z0-9_]+', '', t)
    # remove special characters
    t = re.sub(r'[^a-zA-Z ]', '', t)
    # remove multiple spaces
    t = re.sub("\s\s+", " ", t)
    return t

cleaner = udf(lambda x : clean_text(x))
df = df.withColumn("Text", cleaner(df.Text))
```

Illustration 24. clean tweets

- **Text tokenization with RegexTokenizer:** This step consists of transforming the text we have to have an array of characters of the text ('hello' → ['h', 'e', 'l', 'l', 'o']). For this we have used the Spark library function called RegexTokenizer. This function extracts tokens either by using the provided regex pattern to split the text.

```
tokenizer = RegexTokenizer(inputCol="Text", outputCol="chars", pattern="")
tokenized = tokenizer.transform(df)
tokenized.show(5, False, True)
```

Illustration 25. tokenizer

```
 Text  | trump acts tough talking about thugs looting and shooting domination but donnie who turned off the lights and hid in t
he basement
 chars | [t, r, u, m, p,  , a, c, t, s,  , t, o, u, g, h,  , t, a, l, k, i, n, g,  , a, b, o, u, t,  , t, h, u, g, s,  , l, o,
o, t, i, n, g,  , a, n, d,  , s, h, o, o, t, i, n, g,  , d, o, m, i, n, a, t, i, o, n,  , b, u, t,  , d, o, n, n, i, e,  , w,
h, o,  , t, u, r, n, e, d,  , o, f, f,  , t, h, e,  , l, i, g, h, t, s,  , a, n, d,  , h, i, d,  , i, n,  , t, h, e,  , b, a,
s, e, m, e, n, t,  ]
```

Illustration 26. tokenizer results

- **Text vectorization with CountVectorizer:** This step consists of creating a mapping of characters to integers, assigning to each character a number (['a', 'a', 'b', 'b', 'c'] → ['1', '1', '2', '2', '3']). To both predict and understand the predictions, we have to have defined a mapping of characters to integers and a reverse mapping of integers to characters.

```
vectorizer = CountVectorizer(inputCol="chars", outputCol="features").fit(tokenized)
vocab = vectorizer.vocabulary

def vocab_to_dict(vocab):
    dict_v = {}
    for i in range(len(vocab)):
        dict_v[vocab[i]] = i
    return dict_v

def encode_arr(arr, dict_v):
    res = []
    for i in arr:
        res.append(dict_v[i])
    return res

dict_v = vocab_to_dict(vectorizer.vocabulary)

labelEncoder = udf(lambda x : encode_arr(x, dict_v))
encoded = tokenized.withColumn("chars", labelEncoder(tokenized.chars))
encoded.show(5, False, True)
```

Illustration 27. vectorizer

```
 Text  | trump acts tough talking about thugs looting and shooting domination but donnie who turned off the lights and hid in t
he basement
 chars | [2, 8, 12, 15, 13, 0, 3, 14, 2, 6, 0, 2, 4, 12, 16, 9, 0, 2, 3, 10, 22, 5, 7, 16, 0, 3, 20, 4, 12, 2, 0, 2, 9, 12, 16,
6, 0, 10, 4, 4, 2, 5, 7, 16, 0, 3, 7, 11, 0, 6, 9, 4, 4, 2, 5, 7, 16, 0, 11, 4, 15, 5, 7, 3, 2, 5, 4, 7, 0, 20, 12, 2, 0, 11,
4, 7, 7, 5, 1, 0, 18, 9, 4, 0, 2, 12, 8, 7, 1, 11, 0, 4, 17, 17, 0, 2, 9, 1, 0, 10, 5, 16, 9, 2, 6, 0, 3, 7, 11, 0, 9, 5, 11,
0, 5, 7, 0, 2, 9, 1, 0, 20, 3, 6, 1, 15, 1, 7, 2, 0]
```

Illustration 28. vectorizer results

- **Sequences creation:** Our recurrent model, as input needs to have the text divided into sequences. Each sequence will have a target that represents the next character in the sequence. We have organized our data in sequences of 100 characters (X) and our target (y) will be the next character, character number 101. With a step of 6, a total of 1856961 sequences have been generated.

```
# prepare the dataset of input to output pairs encoded as integers
seq_length = 100
dataX = []
dataY = []
for i in range(0, n_chars - seq_length, 6):
    seq_in = encoded_texts[i:i + seq_length]
    seq_out = encoded_texts[i + seq_length]
    dataX.append(seq_in)
    dataY.append(seq_out)
n_patterns = len(dataX)
print("Total Patterns: ", n_patterns)
```

Illustration 29. sequence creation

- **Data reshape, scaling and target (y) encoding:** The last step is to reshape and scale the data. First, the data is reshaped to the format necessary to use it in a recurrent neural network [samples, time steps, features]. After reshaping, data is scaled dividing with the number of unique characters, moving to a scale [0, 1]. Finally, in order to calculate the probability of each single character to be the next, we have to transform the target (y) to be an array of probabilities. To do this we've used the *to_categorical* function. This function converts a class vector (integers) to binary class matrix ([0, 1, 2, 3] → [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]).

```
# reshape X to be [samples, time steps, features]
X = numpy.reshape(dataX, (n_patterns, seq_length, 1))
# normalize
X = X / float(n_vocab)
# one hot encode the output variable
y = to_categorical(dataY)
```

Illustration 30. label transformation

3.1.4.1.4 Model generation, regularization and scaling up

The first test performed was with a single LSTM layer with 16 neurons. Seeing that the results were not satisfactory and that the text generated was always the same word over and over again in a loop, we decided to scale up our network, increasing its size progressively and as far as our computing power allowed. Seeing that our network ended up overfitting without any kind of regularization, we developed two networks of the same size but with different regularization techniques.

The first consists of 3 LSTM layers of 256 neurons and a dense layer with an amount of output neurons equal to the number of unique characters in our text. A dropout of 0.2 has been used as a regularization technique. The term "dropout" refers to dropping out units in a neural network. In this case, with a value of 0.2, the 20% of the units will be dropped out. With this we manage to avoid overfitting during training and achieve a very stable training and validation loss reduction. As we want to obtain the probability of each character to be the next, in this last layer we have used the *'softmax'* activation function.

Something to highlight in this network is how the input shape has been defined. As the user's seed can be of any size, the dimension that defines the size of the input is defined as None. This way we are able to start generating text from a sentence of any size.

```
# define the LSTM model
model = Sequential([
    LSTM(256, dropout=0.2, input_shape=(None, X.shape[2]), return_sequences=True),
    LSTM(256, dropout=0.2, return_sequences=True),
    LSTM(256, dropout=0.2,),
    Dense(y.shape[1], activation='softmax')
])

# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

Illustration 31. LSTM model

The second model developed differs from the first in the regularization techniques used. Apart from the dropout, the recurrent dropout has also been used. Recurrent dropout drops the connections between the recurrent units. With this regularization technique

we still overfitting, achieving a constant loss reduction in training, but the training time increases significantly.

```
# define the LSTM model
model = Sequential([
    LSTM(256, dropout=0.2, recurrent_dropout=0.2, input_shape=(None, X.shape[2]), return_sequences=True),
    LSTM(256, dropout=0.2, recurrent_dropout=0.2, return_sequences=True),
    LSTM(256, dropout=0.2, recurrent_dropout=0.2),
    Dense(y.shape[1], activation='softmax')
])

# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

Illustration 32. second model

The training duration of the model with only dropout was 3.3 hours. The training duration of the model with dropout and recurrent dropout was 4.8 hours. When training the models, the configuration used has been the same in both cases:

- **Callbacks:**
  - **EarlyStopping:** Is a form of regularization used to avoid overfitting when training a learner with an iterative method. With a patience of 3, if the validation loss has not been improved in 3 epochs, the training is stopped.
  - **ModelCheckpoint:** If the validation loss improves in the new epoch, this callback exports the model in H5 format. If the loss gets worse, it does not export.This way we get the best model of the training process
- **Epochs:** 100
- **Batch size:** 128
- **Validation split:** 0.1

```
early_stop = EarlyStopping(monitor='val_loss', min_delta=0, patience=3, verbose=0, mode='auto')
model_save = ModelCheckpoint('models/trump_model.h5', save_best_only=True, monitor='val_loss', mode='min')

# fit the model
model.fit(X, y, epochs=100, batch_size=128, callbacks=[early_stop, model_save], validation_split=0.1)
```

Illustration 33. fit model

Below are two plots of the training and validation loss of the two trained models.

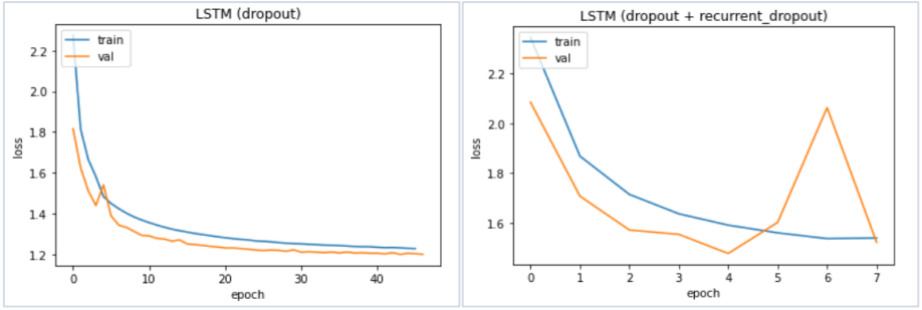

Illustration 34. Deep Learning results

3.1.4.1.5 Results

The model with which a lower loss was achieved was the one with which only the dropout regularization technique was used. Below are some 140-character texts generated by our model.

SEED	GENERATED TEXT (140 chars)
make america great again	I will be interviewed on at am enjoy thank you for your support it is totally really bad thank you
hillary clinton	Is a total disaster and the people of our country is a major big crowd will be interviewed on the t
obamacare	and make america is the worst new hampshired the people with the state of the us senator to make am
united states	and a major speech thank you for the nice words than bernie sanders is a total lie the state of the
bernie sanders	is a total lie the debate and make america got the people of the debate and make america got the pe

Table 1. DL results

3.1.4.1.6 Hyperparameter tuning

Because of the training times of our recurrent model, it has been unfeasible to use hyperparameter tuning automation tools such as hyperas or grid-search. To reach these results the road has been long and we have tested with many variants. As an example, in relation to the dropout value, we started with a value of 0.5 and ended up lowering it to 0.2. Also, we have tested with different amounts of layers and neurons, starting with 16 neurons and ending with a 3-layer neural network of 256 neurons each.

Even so, the tests we have carried out by changing the parameters have been totally manual, without using tools that automate the process.

3.1.4.1.7 Transfer learning (From Trump to Biden)

Once we have a model that writes like Donald Trump, we want to retrain the model to write like Joe Biden, Donald Trump's political adversary. We will use the technique of transfer learning. Transfer learning is the reuse of a pre-trained model on a new problem.

We basically try to exploit what has been learned in one task to improve generalization in another. We transfer the weights that a network has learned at "task A" to a new "task B."

As we have far fewer tweets from Joe Biden compared to Donald Trump, it's a good idea to train the Joe Biden model based on Donald Trump's model. The processing of the data has been the same as that of Donald Trump, there is only difference in the way of training.

- Load Donald Trump weights.
- Define EarlyStopping and ModelCheckpoint callbacks.
- Fit the model with Joe Biden data.

```
# Load Donald Trump's model
model = load_model('models/trump_model.h5')

# define callbacks
early_stop = EarlyStopping(monitor='val_loss', min_delta=0, patience=3, verbose=0, mode='auto')
model_save = ModelCheckpoint('models/biden_model.h5', save_best_only=True, monitor='val_loss', mode='min')

# fit the model
model.fit(X, y, epochs=100, batch_size=128, callbacks=[early_stop, model_save], validation_split=0.1)
```

Illustration 35. Transfer learning

Below a plot of the training and validation loss during training.

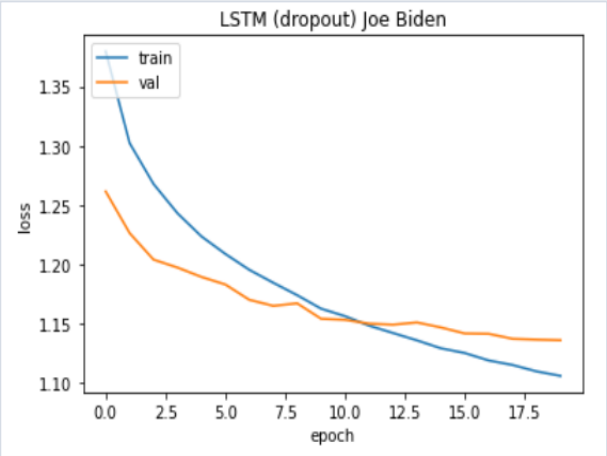

Illustration 36. Transfer learning results

Below are some 140-character texts generated by Joe Biden's model.

SEED	GENERATED TEXT (140 chars)
make america great again	st the stage we will beat Donald Trump is a democratic care they deserve to the progress we need a president who will be a president who wil
hillary clinton	is a democratic community to the progress we need a

	president who will be the soul of this nation we can do it again the stage we cant do t
obamacare	and i will be a part of the world today i will be a part of the world today i will be a part of the world today i will be a part of the wor
united states	to the president trump is the soul of this crisis is a disgrace the stage we need a president will be the soul of this crisis is a disgrace
bernie sanders	for the soul of this nation we can do it again in the white house the president trump is the powerful for the soul of this nation we can do
donald trump	and the stage we cant do the state of the world to stand up to the president i will be a president trump is the soul of this country we cal

Table 2. Generated text

4.1.4.2 Word-level language modeling

Word-level models consist on entering a sequence of words and predicting probabilities of the next word. Our char-rnn model is capable of generating words but is not capable of building sentences with an understandable context. A good option to generate phrases with a proper context may be to use a word level text generation model.

The following details our implementation of the word-level model following the universal flow of machine learning.

3.1.4.2.1 Metric election

As we want to obtain the probability of each character to be the next, in this last layer we have used the *'softmax'* activation function.

As a result of the selection of this activation function, categorical cross entropy has been used as a loss function. This function is what we use to measure the error at a *softmax* layer.

3.1.4.2.2 Evaluation protocol

Regarding the evaluation method, the data has been divided into 2 blocks: train (90%) and validation (10%). The word-level model was developed to see if with our data we were able to generate text word by word. The intention was not to introduce the validation of the model in the pipeline, so data has not been saved for the test phase.

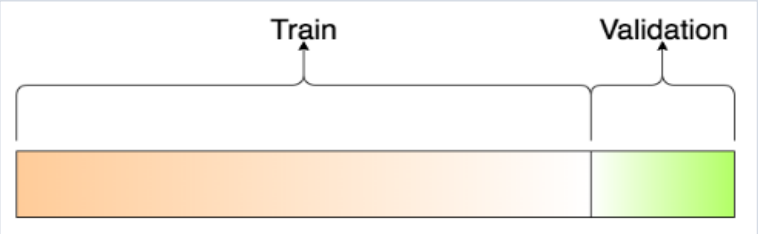

Illustration 37. Train/Validation split

The division between train and validation is done automatically by Keras, indicating the percentage of the split in the fit function.

3.1.4.2.3 Pre-processing (local - not distributed)

The pre-processing of the data or the adaptation to train the model has been done in a not distributed way locally. The data has been read from a local file. When reading all the text it has been stored in a single string.

With the data already available, the pre-processing followed has consisted of the following steps.

- **Data cleansing:** This step consists of removing the characters we are not interested in from the text. A function has been developed that receives as input a text, makes a cleaning using regular expressions. At the end of the cleaning, we make a split of the text, dividing the string into an array of words.
    - Transform the characters to lowercase.
    - Remove quotes.
    - Remove URLs.
    - Remove line breaks.
    - Remove multiple whitespaces.
    - Replace '&amp' with 'and'.
    - Replace abbreviations.
    - Remove twitter mentions (@menion)
    - Remove twitter tags (#tag)

- Remove special characters.
- Text split.

```
# clean text
def clean_text(t):
    # to lower
    t = t.lower()
    # remove quotes
    t = re.sub(r'"@.*', '', t)
    t = re.sub(r'^“.*”$', '', t)
    # remove URLs
    t = re.sub(r'https*:\/\/\S*', '', t)
    t = re.sub(r'pic\.twitter\.com\/\S*', '', t)
    # remove \n
    t = re.sub('\n', ' ', t)
    # remove extra whitespaces
    t = re.sub(r'\s+', ' ', t)
    # replace '&amp' with 'and'
    t = re.sub('&amp;', 'and', t)
    # replace abbreviations
    t = re.sub("'ll", ' will', t)
    t = re.sub("won't", 'will not', t)
    t = re.sub("n't", ' not', t)
    # remove @mention
    t = re.sub(r'@[A-Za-z0-9_]+', '', t)
    # remove #tag
    t = re.sub(r'#[A-Za-z0-9_]+', '', t)
    # remove special characters
    t = re.sub(r'[^a-zA-Z ]', '', t)
    # remove multiple spaces
    t = re.sub("\s\s+", " ", t)
    # split into tokens by white space
    words = t.split()
    return words
```

Illustration 38. clean text

```
['great', 'to', 'be', 'with', 'our', 'wonderful', 'men', 'and', 'women', 'of', 'the', 'what', 'a', 'job', 'they', 'are', 'doin
g', 'the', 'problem', 'is', 'not', 'the', 'very', 'talented', 'lowflying', 'helicopter', 'pilots', 'wanting', 'to', 'save', 'ou
r', 'city', 'the', 'problem', 'is', 'the', 'arsonists', 'looters', 'criminals', 'and', 'anarchists', 'wanting', 'to', 'destro
y', 'it', 'and', 'our', 'country', 'michael', 'is', 'tough', 'smart', 'and', 'loves', 'our', 'country', 'this', 'has', 'been',
'a', 'big', 'battle', 'in', 'congress', 'for', 'years', 'thank', 'you', 'to', 'our', 'great', 'republican', 'senate', 'congratu
lations', 'to', 'michael', 'pack', 'nobody', 'has', 'any', 'idea', 'what', 'a', 'big', 'victory', 'this', 'is', 'for', 'americ
a', 'why', 'because', 'he', 'is', 'going', 'to', 'be', 'running', 'the', 'voice', 'of', 'america', 'and', 'everything', 'associ
ated', 'with', 'it', 'to', 'the', 'united', 'states', 'we', 'have', 'now', 'brought', 'more', 'than', 'american', 'hostages',
'and', 'detainees', 'back', 'home', 'since', 'i', 'took', 'office', 'thank', 'you', 'to', 'iran', 'it', 'shows', 'a', 'deal',
'is', 'possible', 'i', 'just', 'got', 'off', 'the', 'phone', 'with', 'former', 'american', 'hostage', 'michael', 'white', 'wh
o', 'is', 'now', 'in', 'zurich', 'after', 'being', 'released', 'from', 'iran', 'he', 'will', 'be', 'on', 'a', 'us', 'plane', 's
hortly', 'and', 'is', 'coming', 'home', 'we', 'need', 'healing', 'but', 'we', 'also', 'need', 'strength', 'its', 'time', 'for',
'all', 'of', 'americas', 'governors', 'and', 'mayors', 'to', 'restore', 'law', 'and', 'order', 'president', 'provided', 'perman
ent', 'funding', 'to', 'historically', 'black', 'colleges']
```

Illustration 39. show clean text

- **Word sequences creation:** Our recurrent model, as input needs to have the text divided into sequences. Each sequence will have a target that represents the next word in the sequence. We have organized our data in sequences of 10 words (X) and our target (y) will be the next word. With a step of 1, a total of 164562 sequences have been generated.

```
# organize into sequences of tokens
length = 10
sequences = list()
for i in range(length, len(words)):
    # select sequence of tokens
    seq = words[i-length:i]
    # convert into a line
    line = ' '.join(seq)
    # store
    sequences.append(line)
print('Total Sequences: %d' % len(sequences))
```

Illustration 40. organize into sequences of tokens

- **Encode sequences:** The word embedding layer expects input sequences to be comprised of integers. We can map each word in our vocabulary to a unique integer and encode our input sequences. The Keras Tokenizer function has been used for this.

```
tokenizer = Tokenizer().fit_on_texts(lines)
sequences = tokenizer.texts_to_sequences(lines)
```

Illustration 41. Encode sequences

- **Separate into in/out and one hot encode the target**: From every sequence, the last word is used as a target. In order to calculate the probability of each single word to be the next, we have to transform the target (y) to be an array of probabilities. To do this we've used the to_categorical function. This function converts a class vector (integers) to binary class matrix ([0, 1, 2, 3] → [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]).

```
X, y = sequences[:,:-1], sequences[:,-1]
y = to_categorical(y, num_classes=vocab_size)
```

Illustration 42. hot encode

3.1.4.2.4 Model generation, regularization and scaling up

The model consists of an Embedding layer, 2 LSTM layers of 256 neurons and 2 dense layers. A dropout of 0.2 and a recurrent dropout of 0.2 has been used as a regularization technique.

```
model = Sequential([
    Embedding(vocab_size, 50, input_length=seq_length),
    LSTM(256, dropout=0.2, recurrent_dropout=0.2, return_sequences=True),
    LSTM(256, dropout=0.2, recurrent_dropout=0.2),
    Dense(256, activation='relu'),
    Dense(vocab_size, activation='softmax')
])

model.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

Illustration 43. Model generation

The training duration of the model with only dropout was 10 minutes. The training configuration was as follows.

- **Callbacks:**
  - **EarlyStopping:** Is a form of regularization used to avoid overfitting when training a learner with an iterative method. With a patience of 3, if the validation loss has not been improved in 3 epochs, the training is stopped.
  - **ModelCheckpoint:** If the validation loss improves in the new epoch, this callback exports the model in H5 format. If the loss gets worse, it does not export.This way we get the best model of the training process
- **Epochs:** 100
- **Batch size:** 128
- **Validation split:** 0.1

```
# callbacks
early_stop = EarlyStopping(monitor='val_loss', min_delta=0, patience=3, verbose=0, mode='auto')
model_save = ModelCheckpoint('models/trump_model.h5', save_best_only=True, monitor='val_loss', mode='min')

# fit model
history = model.fit(X, y, batch_size=128, epochs=100, callbacks=[early_stop, model_save], validation_split=0.1)
```

Illustration 44. fit model

Below a plot of the training and validation loss of the model.

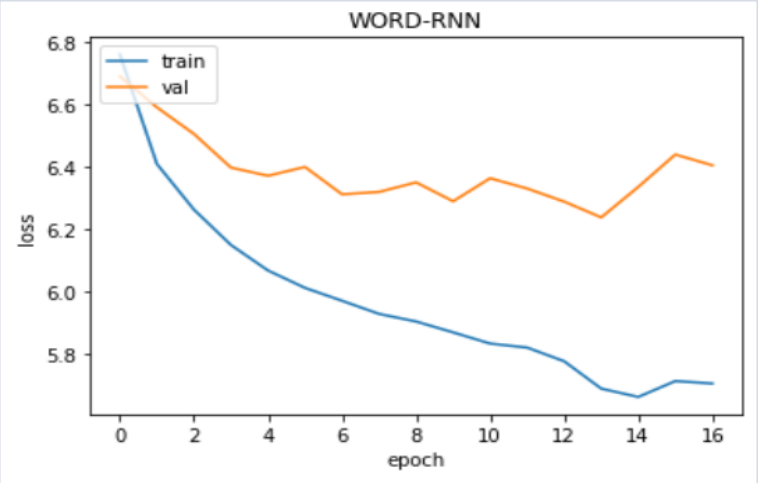

Illustration 45. Word RNN plot

3.1.4.2.5 Results

The results when generating text have been quite bad compared to the char-rnn model. Below a table with some examples.

SEED	GENERATED TEXT (140 chars)
make america great again	thank you for your support thank you for your support thank you for your support thank you
hillary clinton	is a great job and the democrats are a great job in the united states and the united states and the democrats
obamacare	is a great job and the democrats are a great job in the united states and the unit d states and the democrats
united states	are a great job and the democrats are a great job in the united states and the united states and the
bernie sanders	is a great job in the united states and the united states and the democrats are a great job and the united

Table 3. Results model and conclusion

3.2 ITAPP development

3.2.1 The application architecture

ITAPP has been developed using a python web framework called Django. This framework is based on a popular software pater design called Model, View and Controller or MVC for short. The idea behind this concept is to divide the app into three main areas that focus each one of them in just one task.

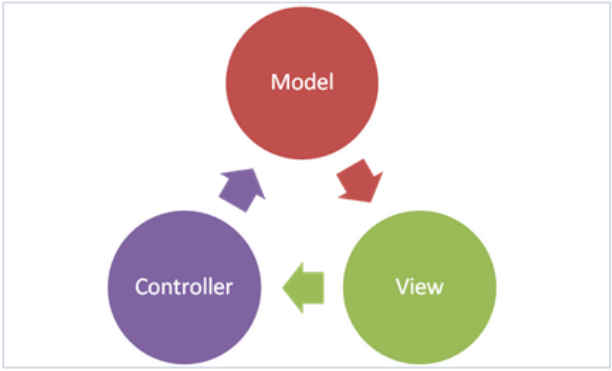

Illustration 46. Application architecture

The view service is in charge of rendering the UI and to provide interaction with the app's users. In Django the view element is represented by a set of HTML files called Templates. This set of templates are rendered depending of the URL the user is requesting. Itapp only has one template.

The controller service takes care of processing any form request or business logic related stuff and updates the UI. In Django, the controller (ironically) is coded on the views file. When users submit a form, for instance when they request a twit generation, a function takes care of that request.

Well, that has potential security concerns. Users may introduce nonstandard text erroneously. In order to avoid having issues related to inputs, the application has a module named Input, that has two main methods. The first one called sanitize, transforms the text to ascii encoding dropping characters that cannot be represented by the ascii codification. It also replaces new line characters and quotations marks by the empty character (“”).

```
def sanizate(self, text):
    try:
        text = ascii(text)
        text = text.replace("\\n", " ")
        text = text.replace("'", "")
        text = text.replace('"', "")
        text = str(text)
        text = text.lower()
        return text
    except Exception:
        return ""
```

Illustration 47. Input sanitizer

The methods called validate_tweet and validate_topic use a regular expression that consist of any character sequence containing alphanumerical characters, blanks, new lines, commas, single quotes, dots, and semicolons between 3 and 50 for the topic validator method and between 3 and 280 for the twit validator. Any exception during this process will result in an invalid input.

```
def validate_topic(self, text):
    val = re.compile(r"^[a-zA-Z0-9\s\n,'.;]{3,50}$")
    try:
        result = val.search(text)
        if result is None:
            return False
        else:
            return True
    except Exception:
        return False
```

Illustration 48. Input validator

The service on the MVC pattern is the model. Itapp defines a simple model object that is used to store or to retrieve data from Elasticsearch. Perhaps this is the less interesting aspect to cover due to its simplicity and due to a higher importance of other elements such as the controller.

```
class Tweet(Document):
    tweet = Text()
    date = Date()

    class Index:
        name = "itapp"
        settings = {
            "number_of_shards": 2,
        }

    def save(self, **kwargs):
        return super(Tweet, self).save(**kwargs)

    def is_published(self):
        return datetime.now() > self.published_from
```

Illustration 49. model definition

3.2.1.1 Inputs of the application

Itapp's structure is compartmentalized into various parts, like puzzle pieces that come together to form a bigger picture. Each one of those parts receives data, validating and sanitizing it if it comes from outside the structure and once it finishes working with it, sends it to another part.

Itapp only validates data that comes from outside the structure Itapp runs on because during the development this was stablished as the trust boundary. Anything coming from the outside or that must be sent outwards must be validated to avoid security flaws.

This approach allows to plan which inputs should reach which parts of the structure, what those data should look like and how they must be treated.

It also adds the possibility of making each of the parts redundant, as a set of input data can be sent to any mirror copies a part has if it is ensured the storage engines will reach a consistency state down the line and the service does not get compromised.

This chapter describes the inputs each main part that composes Itapp has.

3.2.1.1.1 User inputs

The web application where the user can interact with Itapp has the following inputs:

Introduced topic

This input is the topic from which the model generates a tweet, a seed from which to grow a tweet. If a user submits “great”, the end result will include it in it.

The user can input any text to it. Said text will be sent to the backend of the web application were, if the validation and sanitization checks pass, it will be redirected as an input for the model embedded in the backend.

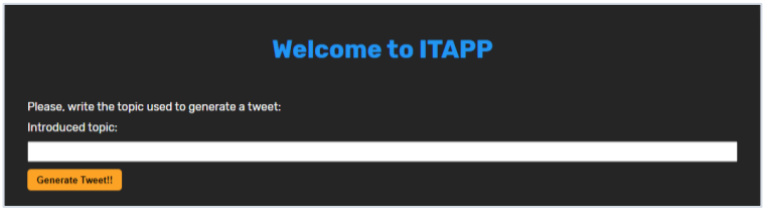

Illustration 50. Introduce topic

Validation

```
def validate_topic(self, text):
    val = re.compile(r"^[a-zA-Z\s\n,'.;]{3,50}$")
    try:
        result = val.search(text)
        if result is None:
            return False
        else:
            return True
    except Exception:
        return False
```

Illustration 51. Validation

The validation process for the topic checks for:

- Length between 3 and 50 characters.
- Only A-Z, a-z, spaces, new lines and some special characters are allowed.

If the validation fails, the web application returns an alert telling the user the inputted topic is not a valid one.

Sanitization

```
def sanizate(self, text):
    try:
        text = ascii(text)
        text = text.replace("\\n", " ")
        text = text.replace("'", "")
        text = text.replace('"', "")
        text = str(text)
        text = text.lower()
        return text
    except Exception:
        return ""
```

Illustration 52. Sanizate

The sanitization process:

- Encodes the text as ASCII.
- Replaces/escapes some key characters

The sanitization process is identical for the other inputs that appear on the web application and thus will not be further discussed.

Generated tweet

Illustration 53. Generate tweet

The generated tweet is the tweet the model creates after a seed is sent to it. It allows the user to change the generated tweet before it is sent anywhere else.

Once a user presses the publish tweet button the generated tweet is sanitized, validated and sent to:

- The Twitter API
    - The tweets are automatically published in a Twitter account Itapp has set up.
- HDFS
    - The generated tweets are stored to be used as training fodder down the line.

- They follow the same JSON schema as the tweets on the NiFi flows' endpoint.

- Elastic Search
  - The interface allows searching for previous tweets. To ease that tasks Elastic Search is another of the sinks tweets are directed into.

```
def insert_hdfs_tweet(tweet, folder):
    ts = time.time()

    milliseconds = int(round(time.time() * 1000))
    tweet = {
        "Text": tweet,
        "Source": "Itapp",
        "TimeStamp": datetime.datetime.fromtimestamp(ts).strftime("%Y-%m-%d %H:%M:%S"),
    }

    r1 = requests.put(
        "http://node1.itapp.eus:9870/webhdfs/v1/"
        + folder
        + "/generated-twit-"
        + str(milliseconds)
        + ".json?user.name=hadoop&op=CREATE&noredirect=true"
    )
    data = r1.json()
    url = data["Location"]
    url = url[7:]
    idx = url.index(":")
    url = "http://" + url[:idx] + ".itapp.eus" + url[idx:]
    requests.put(url, data=json.dumps(tweet))
    return "generated-twit-" + str(milliseconds)
```

Illustration 54. insert hdfs tweet

```
def insert_tweet(tweet_content):
    try:
        # create conection
        connections.create_connection(hosts=["node1.itapp.eus:9200"])
        # create the mappings in elasticsearch
        Tweet.init()
        # create and save the tweet
        tweet = Tweet(tweet=tweet_content, date=datetime.now())
        tweet.save()
    except Exception:
        pass
```

Illustration 55. insert tweet

Validation

```
def validate_tweet(self, text):
    val = re.compile(r"^[a-zA-Z0-9\s\n,'.;]{3,280}$")
    try:
        result = val.search(text)
        if result is None:
            return False
        else:
            return True
    except Exception:
        return False
```

Illustration 56. validation tweet

The validation of the generated tweets is identical to the validation of the topics apart from the lengths a tweet is allowed to have. Extended tweets in Twitter allow up to 280 characters and so does Itapp for edited tweets.

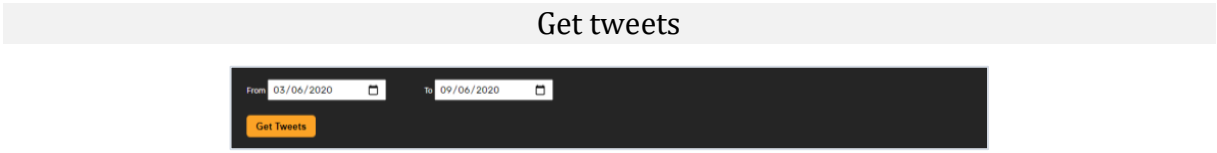

Illustration 57. get tweets

A user can query Elastic Search for old tweets using HTML5's datetimepickers as the query parameters. These two inputs are thoroughly tested by HTML5's developers, and, according to the internet, neither they or the validation they have by default contain any flaws or errors that could compromise Itapp's services.

Once a "Get tweet" is submitted an HTTP query through the Elastic_dsl python library queries the Elastic Search engine, returning the tweets as strings that can be read by a simple for loop.

```
def get_tweet_list(request):
    global tweet_list
    filtered_tweets = query_timestamp_range(request.POST["from"], request.POST["to"])
    tweet_list = []
    for hit in filtered_tweets:
        print(hit.tweet)
        tw = TweetModel(hit.tweet, str(hit.date).split("T")[0])
        tweet_list.append(tw)
```

Illustration 58. get tweets list

3.2.1.1.2 Inputs to the model

The model is trained in Spark based Jupyter Notebook files that are executed on a Spark cluster. For data to reach that cluster they must be read from HDFS, located in the Hadoop cluster. For this to happen:

- The process that runs Sparks must be owned by a user that has read permissions on the HDFS directories were data is located.
- Connections from HDFS to Spark must be enabled in firewalls and other roadblocks.

Once that is set up, the following piece of code will be able to read data from HDFS into Spark:

```
df = spark.read.option("inferSchema", "true").json('/home/ubuntu/export/*.json')
df = df.select("Text").where("Tag == 'Trump'")
df.show()
```

Illustration 59. read spark

The spark read options allow the developers to specify the format in which the files are read, JSON in this case, and to filter which texts of those read files should be used.

There is no validation process for this input as it is an internal read, internal meaning it is a read we locate inside our trust boundary. The JSON files are located in Itapp's Hadoop cluster and are read into Itapp's Spark cluster.

3.2.1.1.3 ElasticSearch input

The only input that is unique to Elastic Search are the dates the users pick on the datetimepicker present in the web interface. Those dates are validated by HTML5's own library and sent to Elastic Search by elastic_dsl. This exchange of information also happens inside our trust boundary but the HTML5 library does execute a validation of the dates.

Elastic Search, HDFS and Twitter receive the exact same output, that being the tweets user edit and submit on the web application. (Publishing on Twitter is still under construction)

HDFS has inputs that are unique to it, those being the end JSON files NiFi generates which are created and validated by NiFi before their storage in HDFS. They are further discussed on the 4.1.1 Data Capture section.

3.2.1.2 Fuzzing

Fuzzing is the software security technique that consists of trying purposefully malicious, wrongly constructed or randomly generated data for the inputs a piece of software has. This is done so errors that could have gone undetected by the development team come to the surface, as, trying different inputs will trigger different responses by the software, including crashes, exploit executions and so on.

This is a very costly testing method as fuzzing needs to try a large variety of data with each and every one of the input fields present on the software, resetting the state of the runtime environment to one previous to the interaction after every input that is tried. This may take miliseconds for a computer, but, the completion time for the complete

testing scales exponentially by the number of input fields and the type of data that will be tested.

Fuzzing can be divided into three main categories depending on the method used for the generation of data:

- **Blackbox fuzzing:** generates data to be tested randomly.
- **Grammar based fuzzing**: data is generated according to user defined metrics, be them schemas some data formats must follow or instructions on how to expand previous data.
- **Whitebox fuzzing**: data is generated from an interpretation the fuzzer does of the code that composes a software piece, generating data that could break it whilst keeping track of the execution logic.

The Itapp development team has used fuzzing to further secure the whole architecture Itapp runs on from external attacks. This fuzz testing has been conducted on the two main external access points to the application, those being:

- The data Itapp gathers from the internet.
- The user input the web application answers to.

Two main approaches were followed to fuzz test those.

Data gathered from the internet

The APIs Itapp calls to gather data from the internet send their responses from which data is extracted in JSON files, following set schemas. The entire data capture for these JSON files already takes care of filtering responses that are not JSON files or JSONs that do not contain information Itapp wants. (Refer to 4.1.1 to obtain a more in depth explanation of the data capture).

However, the capture process in NiFi did not take purposefully malicious JSON files into account, thus, a custom made JSON file that contained the information key/values we checked for could end up inserting anything to the database.

Thus, the approach for the fuzz testing of this input was to create a script that would expand/edit a valid JSON file with the goal of crashing our NiFi processors. This could be done in two ways:

- Blackbox based, meaning, an ever expanding JSON file.
- Grammar based, meaning, a set number of rules on how to generate a JSON file.

Pyjfuzz

For those purposes we ended up using the pyjfuzz library for python. Pyjfuzz is a custom-made library that enables, with a few lines of code, to generate malicious JSON files from a valid JSON file.

```
import json
from argparse import Namespace
from pyjfuzz.lib import *

config = PJFConfiguration(Namespace(json = {"text": "RT @BernieSanders: I don't often agree with Defense Secretary Esper,
config.techniques = [2, 5, 6, 7, 8, 9, 10, 11, 13]
config.parameters = ["full_text"]
fuzzer = PJFFactory(config)

print(type(fuzzer.fuzzed))


with open('fuzzed.json', 'a') as f:
    json_str = json.loads(fuzzer.fuzzed)
    json.dump(json_str, f)
```

Illustration 60. Pyjfuzz

The JSONs NiFi consumes are not read by anything apart from the NiFi processor until they get to HDFS, thus, the vulnerabilities the fuzzing had to look for were:

- Vulnerabilities that could crash the NiFi processors.
- Vulnerabilities that could affect the Linux servers that use the JSON files.

That is why the “techniques” parameter for the pyjfuzz fuzzer, “techniques” meaning attacks/vulnerabilities to check for includes:

- File Inclusion attacks
- RCE (Remote Command Execution) injection
- A list of malicious JSON schemas
- Randomly generated data (black box fuzzing)

Generating the JSONs themselves was not a tricky task, feeding them to NiFi was. Pyjfuzz has a module that allows to setup an HTML server that responds with an input for the fuzzing if you invoke the URL. This feature was not working properly, however.

Creating a test file that had several JSONs was not an option either, as pyjfuzz does not generate all the possible inputs for each of the “techniques”, neither does generate them in order. Does, if we generated a file to be read by NiFi we could be skipping some of the inputs for the fuzzing.

In order for NiFi to receive the inputs continuously we created a python script that wrote a complete JSON in a new row of a file in every execution of a loop and we placed a TailFile NiFi processor that read this input stream.

This processor is connected to a copy of the NiFi flow the real inputs follow, thus, enabling us to test its robustness. If any of the processors crash or the flowfile at the end has data we do not want, we edit the processor to bypass those errors.

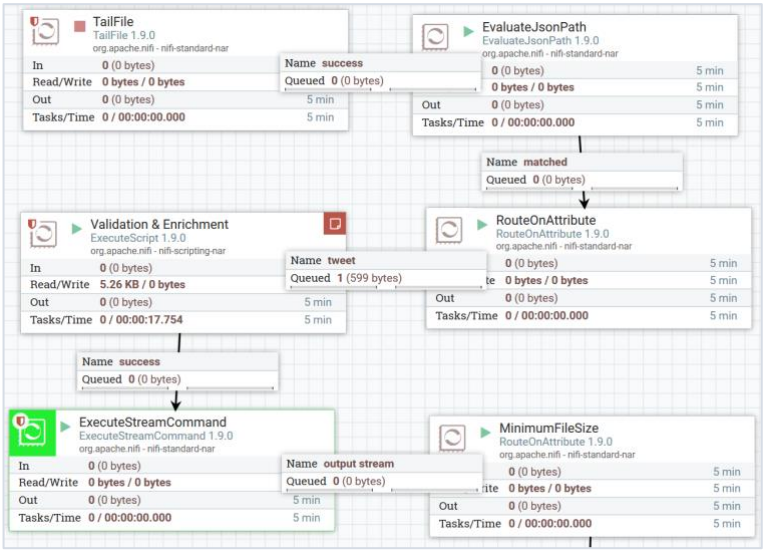

Illustration 61. Nifi flow

Problems detected with pyjfuzz

1. Empty inputs (Fixed)

One of the inputs we did not consider whilst building the NiFi flow for the APIs was that a call to the API would ever return a value for the "text" key that was empty. The logic behind an API call is that you will always receive something, elsewise, that data would not appear on the API.

If the text value were empty or marked as NULL, the python scripts that tried to change the encoding of the text would crash and leave a zombie flowfile on the queue. Gather enough of these and the queue would become full, blocking the NiFi processors.

This was fixed changing the RouteOnAttribute processors logic. Instead of checking if a key for "text" exists on the JSON the API call returns now it checks for the value of this key to be notEmpty. This check ensures the key exists and that it has some form of content in it.

Illustration 62. Empty inputs

2. Inputs that break the JSON transformation (Bypassed)

The JSON files we treat have user made posts in them, that is the main data we want to consume. In order to shape these data into ones that we can later use we filter out some characters and encoding/decoding artifacts from them. This includes stuff like Unicode emojis, HTML tags and quote marks that have been transformed into "\u2029" whilst changing the encoding from Unicode to UTF8.

Usually this character escaping does not cause any problems but, it seems JSON files can be made in such a way that the character escaping itself breaks them, deleting characters that form the JSON:

- quote marks that delimit the JSON fields
- the colons that separate keys from values

Once the JSON is broken the method we use to write the flowfile out from the Jython script towards NiFi generates a NULL or empty flowfile. These do not linger as zombie flowfiles but they did interact with the MergeContent processors down the line, adding empty string lines to the JSON files that were being saved in HDFS. These empty lines were then read by the Jupyter Notebook processes we use for the deep learning as empty data rows.

To avoid this from happening we had 2 choices:

- Completely change the character escaping we know works for 99% of the scenarios.
- Filter out empty flowfiles before they are sent to the cloud processors.

The new regexes and checks we created for the character escaping were, on the one hand, more computationally expensive than the filtering of empty flowfiles, and two, we could not pin down all the JSON characters sequences that created these empty flowfiles. Thus, we decided to create a processor that only routed flowfiles that were not empty.

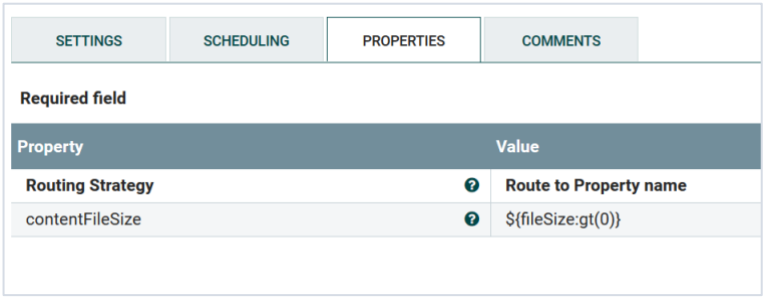

Illustration 63. 2. Inputs that break the JSON transformation

3. Knowledge if whether exploits in JSONs affect our system (Obtained)

Pyjfuzz has generated some JSON files that contain exploits that, for what we understand, should harm our system once they are read. (RCE Injection) We tried to execute them by reading them with Jupyter Notebook but none of them seemed to do nothing. Probably because the escaping deleting characters core to the exploit and the encoding change.

Nevertheless, we have ensured that, at least for now, no exploits are viable with this input flow as their entry point.

User input

The web application that is offered as the main interface for Itapp accepts user input, mainly in text form, but also with date pickers. These input fields interact with the deep learning model, embedded in the web application, HDFS, ElasticSearch and the twitter accounts that can be set up along with an instance of Itapp.

This meant these inputs had great potential of breaking the application itself or parts of the infrastructure if not treated correctly.

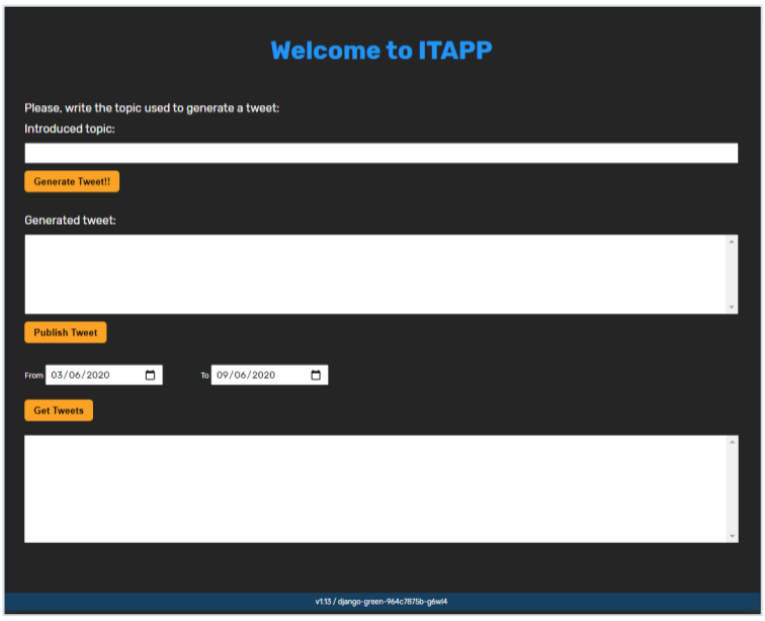

Illustration 64. Itapp application

As explained before in **3.1.1.1** every input field in this web application goes through a validation and sanitization process. This process was not always as it is now though. It has been edited according to the errors the fuzz testing found.

The approach for the fuzz testing has been completely based on grammar-based fuzzing, as the validation process included checks about whether the input was a text of a known length from the very beginning. All attacks that we wanted to test against were related to breaking:

- The reload Django does of the input the user tried to input once the backend processes are carried out.
- The validation and sanitization processes.
- The model's execution.
- The communication with the storage and the Twitter API.

To simulate a real deployment scenario, the fuzz testing has been carried out in the staging deployment of Itapp. This is because the canary/pre-production and production deployments are supposed to be in use by users and, if by chance, the fuzzing finds a vulnerability that breaks the application once it is exploited, we would bring the user facing services down.

Staging itself is a mirror image of those two deployments so, in principle, carrying the testing out in staging is a valid method to strengthen the security of Itapp.

All the fuzzing attempts, or at least most of them, are included as appendices to this document as csv files. INSERT

OWASP Zap

To carry the fuzzing out we have used OWASP Zap's integrated fuzzer, which is an implementation of the old fuzzing software OWASP recommended, jbrofuzz. Zap alternatively offers community made fuzzers and attack lists that can be installed through the extension market, which we have also made use of.

Zap itself is an all-on-one web application vulnerability tester that includes modules to act as a proxy web browser, a fuzzer, includes spiders, web scrappers and so on. Everything in a tightly packed interface that eases its use.

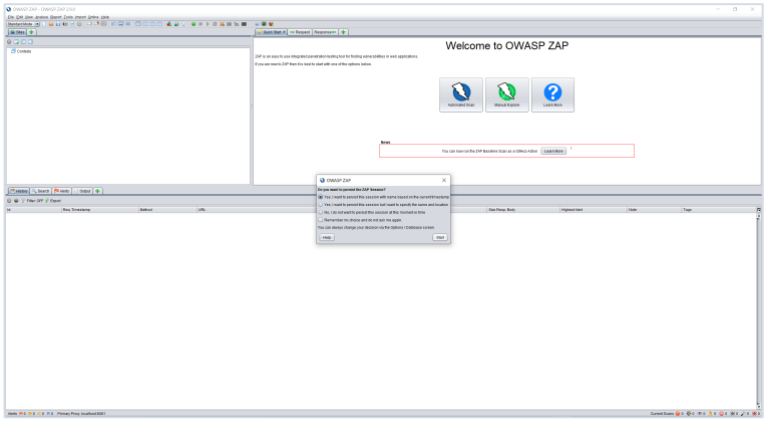

Illustration 65. OWASP zap

The steps for using it are straightforward:

1. Open the web browser proxy Zap offers.
2. Navigate to the target URL.
3. Input something on a field that wants to be tested.
4. Open that request on Zap:
    a. Look for the "history" tab on the interface.
5. Select the input that was sent on the request with the mouse, right-click and fuzz:
    a. Zap automatically picks up
        i. GET variables
        ii. POST variables
        iii. Session variables
        iv. Cookies
6. Select the payloads/inputs that are to be tested.

7. Edit the following parameters if needed:
    a. Concurrent threads.
    b. Timeouts.
    c. Delays.
    d. What to look for once requests are sent.

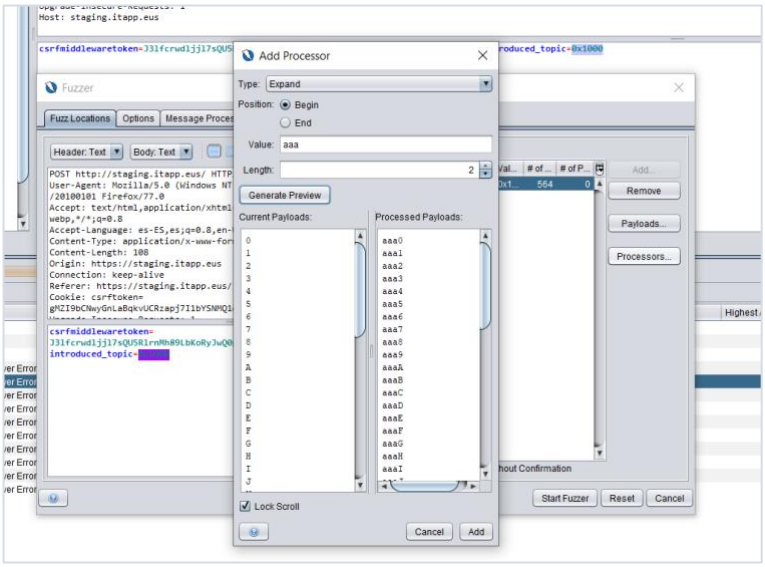

Illustration 66. using straightforward

On that las note, Zap, by default, looks for inputs that are “reflected” on the responses it gets from the web application, “reflected” meaning an input send to the web application is presented on the HTML body of the response.

You can configure Zap to look for specific texts on the responses HTML or to follow redirects the website wants to carry out. Different tags can be assigned to these findings in order to improve the readability of the fuzzing results.

In our case the input is always reflected, as, once the backend processes are terminated, the HTML reloads whilst maintaining the values for each of the input field. On the other hand, if the validation and sanitization processes “break” an alert is sent to the user informing the input he/she has tried is not valid.

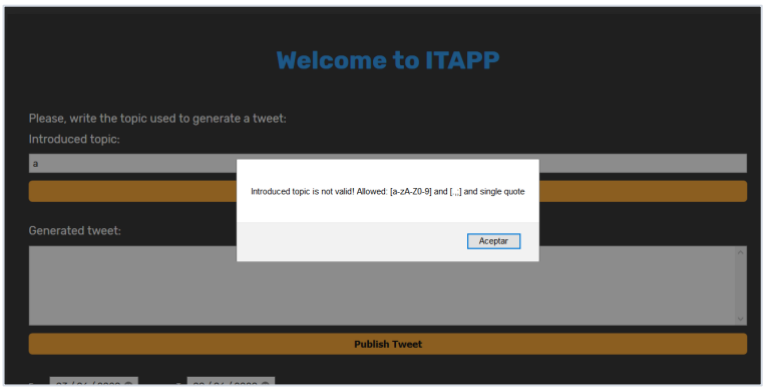

Illustration 67. break alert

We configured a "Blocked" tag for this event, as any of the fuzzed inputs that broke the validation and sanitization processes would trigger this alert.

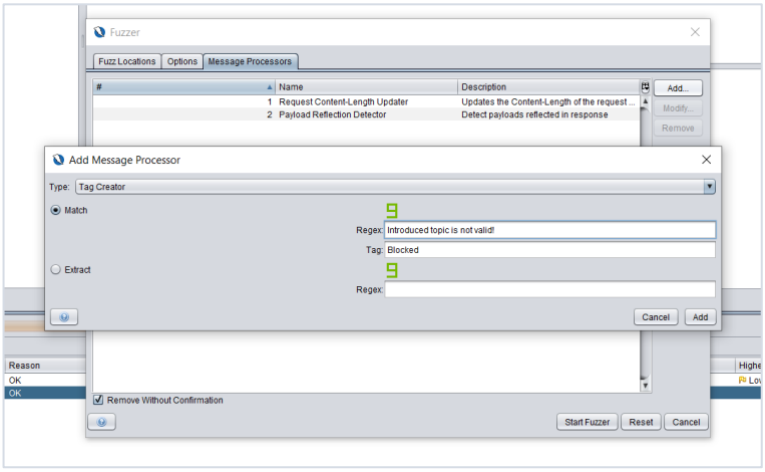

Illustration 68. blocked configuration

Furthermore, if any web application vulnerabilities are detected on the input field treatment by Zap, it raises flags alerting the testers.

Problems detected with Zap

1. Changes in the model break the web app (Fixed)

The web application was developed along with the deep learning model. When both of them reached a functional milestone, the model was embedded into the web application's backend. This is more or less the time at which the validation and sanitization were added to the web application. These processes were adjusted so they would let all the inputs that the model could gobble up pass.

At some point during the development of Itapp the model was changed in a way the inputs it allowed were shortened. To be more specific, a char_to_int() function was added which

needed a filtering of numerical characters in the input, meaning no strings that included numbers could be directed towards the model.

For some reason or another, this was not changed on the regex patterns the sanitization process had on the web application pattern, meaning the character escaping did not delete numerical values, thus, breaking the applications flow each time they were inputted to the "tweet_topic" input field by a user.

This error resurfaced while checking for integer overflows with Zap and was quickly fixed.

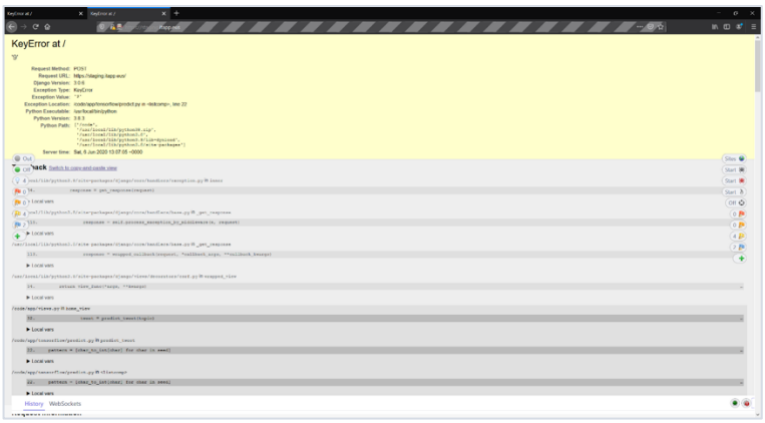

Illustration 69. Changes in the model break the web app

2. Knowledge if whether endpoints can be exploited (Obtained)

All the inputs that sink into the storage modules of Itapp, those being HDFS and Elastic search, and the calls to the Twitter API, have been thoroughly tested. This included no-sql injection tests, malformed HTML headers and so on.

None of them seem to escape our sanitization process.

3. Maximum number of users served by Itapp (Not in this project's scope)

Even if it is not directly related to the fuzzing of inputs itself, Zap has more or less laid out the number of concurrent users our current infrastructure can provide a service to. Zap has an option of limiting the number of concurrent threads that test the fuzzed inputs and delays can be set up between tests.

We have seen that around 6 concurrent executions of the model are the limit for the architecture Itapp is running on. This can be easily upgraded, and it is good news knowing we can use the fuzzing application to test those limits too.

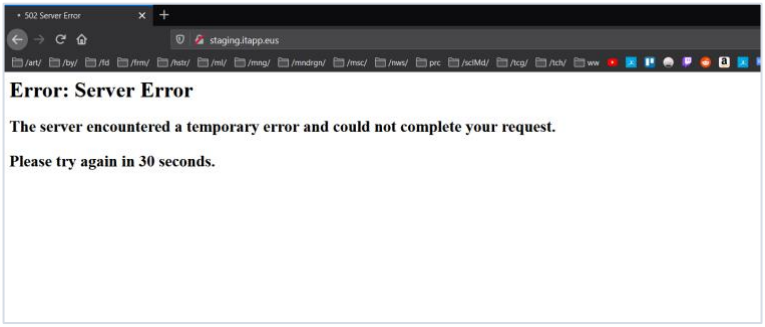

Illustration 70. Server error

Before this problem was identified other fuzzed inputs were suspected of causing internal server errors but, luckily, this was the root cause and not an exploit.

3.2.2 Software security requirements

For a reasoned definition of the software security requirements, a document consisting of two parts has been defined. On the one hand, the security measures taken in the infrastructure services have been defined, as well as a detailed identification of the inputs and outputs of the services. On the other hand, the good security practices that have been carried out during the design and development of the application are included. All this can be seen gathered in the following document.

3.2.3 DevOps

Production level software requires a special crafted development cycle to avoid deploying unstable software into production servers. That's when software engineering comes into place and tries to minimize the risk of introducing bugs or error prone code.

Traditional waterfall software model, as seen on figure X has a set of fixed stages that a program faces while developing. This methodology was introduced back in 1956 and was able to capture most of the needs that developers faced. In fact, this schema is quite used in today even though it has been adapted to be agile, meaning that those phases are repeated over time.

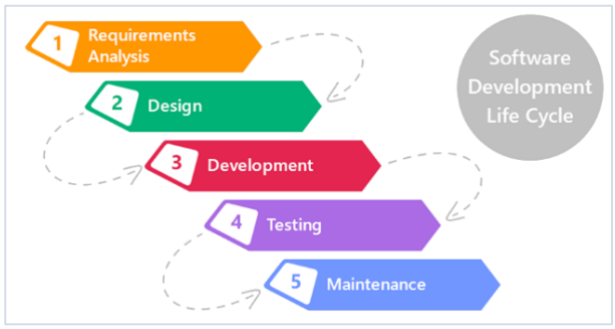

Illustration 71. Software Development Life Cycle

The DevOps concept tries to automate the software lifecycle at its core, and it expands some extra handy features that the traditional software life cycle was not able to capture. Such as deploying or monitoring the application infrastructure. Figure X contains all the stages that today's development cycle cover.

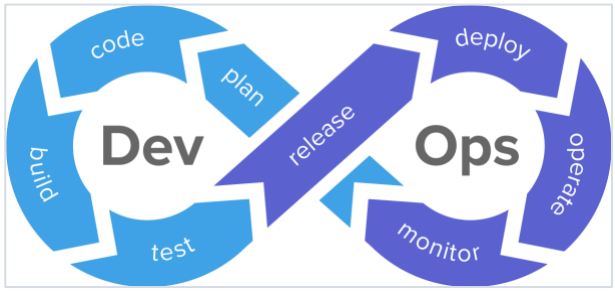

Illustration 72. DevOps

For this project, the DevOps concept has been of vital importance while developing the ITAPP web application.

The monitoring DevOps stage has not been implemented as it was out of the scope of the project. The already provided monitorization provided by our cloud providers has been enough for debugging purposes.

3.2.2.1 Planning

The first task carried out by our team was to plan the whole project. In order to do so, we've used GitLab's issue tracker to identify all the tasks needed to develop in order to fulfil the project's objectives. That includes not only code level issues but also project related issues.

Once a new need was identified, a new issue was created assigning a set of labels, it was referred to a specific milestone as well as selecting the responsible that should take care of said issue and an approximated due date was assigned too.

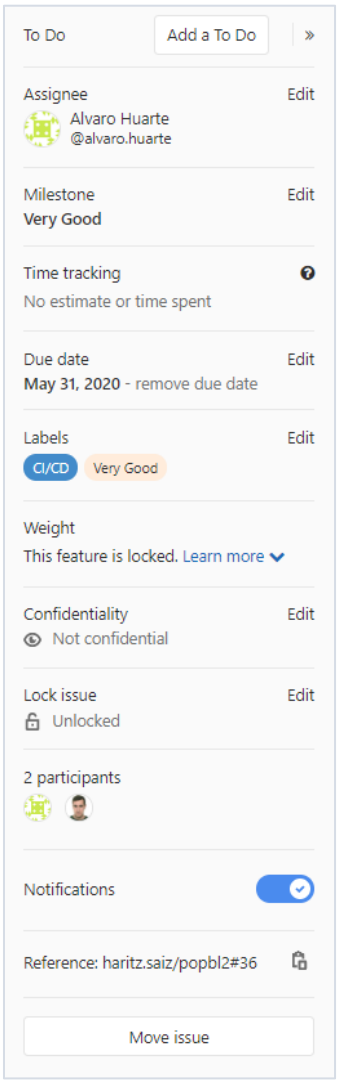

Illustration 73. Planning

To get an overview of the project, a total of 54 issues were created for this project in the span of three weeks.

Earlier it was mentioned that each issue had a set of labels assigned to. For this project three sets of labels were created. The first set consists of two labels that track the state of each task: Todo, Doing.

The second set of labels track the importance of each issue. This also corresponds to the three milestones that we have defined for our project: Basic, Good and Very good. This meaning of each label corresponds to the three levels that each subject has defined on the grading system. A task with a basic label (and milestone) is more important than an issue that has the Good label.

Finally, the las set of labels are used to identify an issue within a subject. For example, the issue on figure X has been labelled with the continuous integration and deployment subject.

The following image illustrates all the eleven labels present in our project:

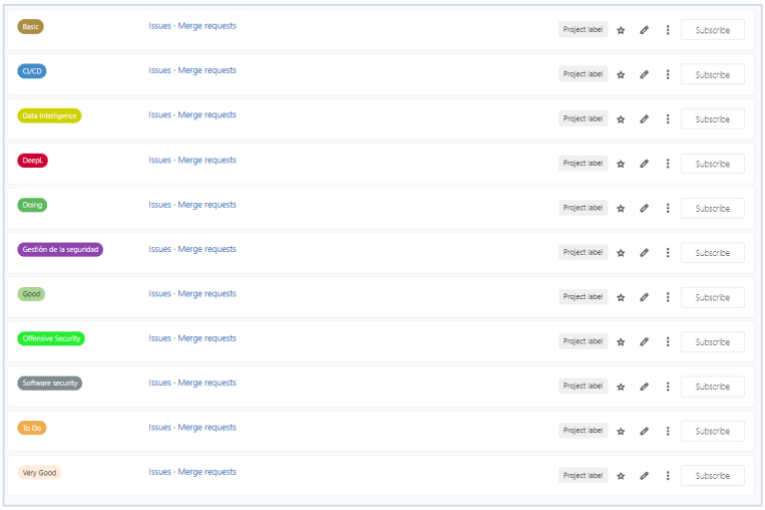

Illustration 74. Eleven labels

Earlier we also mentioned that each subject was labelled using their milestone name. That was done to provide an easier visual identification of each issue and their milestone.

The decision of going with three milestones was done to provide fix schedule of tasks that needed to be developed in order to track the project status. If the basic milestone had a lot of open issues, that meant that the project was not ready to achieve the minimum of each rubric. On the contrary, when the project had the basics covered, the other two milestones started to improve the project quality and grade. The following figure illustrates the status project as per the 1st of June:

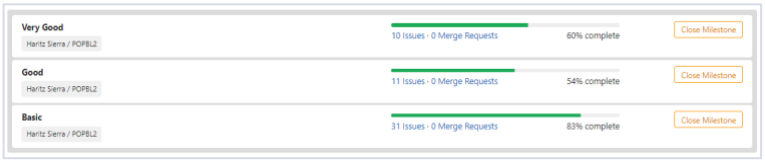

Illustration 75. Milestone

Finally, in order to manage all the project's issues, two boards were created from two points of views. The first one allowed to track all the current issues that were being developed as well as to see all the unopened or to do tasks but also the already finished ones:

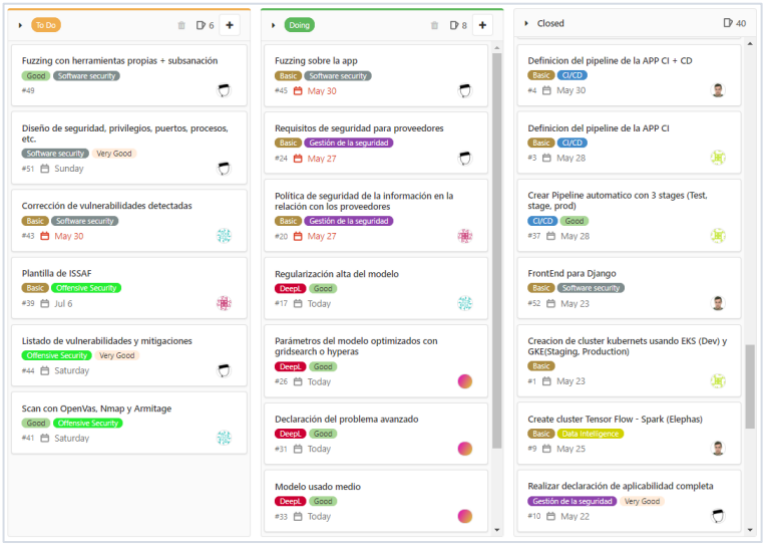

Illustration 76. Board

The second board was designed to have an easier overview of the tasks that were not closed per subject. In fact, it was the board that was primarily used due to the ease of visual identification of each task within the subject. That allowed an easier management of our resources and if a subject had many unfinished issues, the team would allocate more people towards that subject. The following figure illustrates all the tasks within their subject:

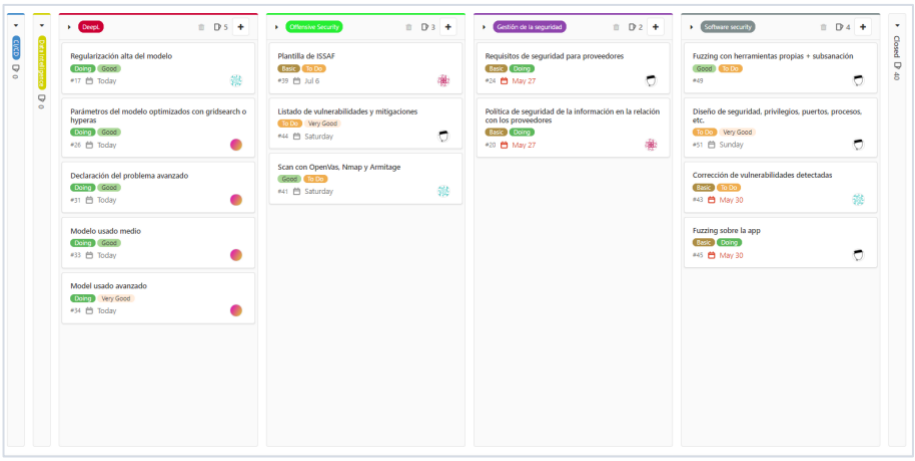

Illustration 77. Board add labels

3.2.2.2 Testing the application

In order to assess the correct behavior of our application, the application counts with a set of tests developed using Django's testing modules. The application has a total of seven-unit tests defined organized in four tests files.

The first two tests are coded in the *web/app/app/tests/test_input.py* are in charge of assuring that the form controllers are correctly coded and return the expected values. This are the most basic test that have been coded:

```
class TopicFormTest(TestCase):
    def test_form(self):
        form_data = {"introduced_topic": "Ander"}
        form = TopicForm(data=form_data)
        self.assertTrue(form.is_valid())


class TweetFormTest(TestCase):
    def test_form(self):
        form_data = {"generated_tweet": "Hola me llamo Ander"}
        form = TweetForm(data=form_data)
        self.assertTrue(form.is_valid())
```

Illustration 78. Testing the application

The second two sets of test focus around input validation and sanitation methods. They can be found in *web/app/app/tests/test_input.py.* Using a predefined list of non-sanitized strings that may result in allowed and not allowed inputs we can provided assurance that those two functions do their job as expected:

```
class InputTest(TestCase):
    def test_allowed_input_validator(self):
        inp = Input()
        with open("app/tests/allowed.json", encoding="utf-8") as json_file:
            data = json.load(json_file)
            for elem in data:
                san = inp.sanizate(elem)
                res_topic = inp.validate_topic(san)
                res_tweet = inp.validate_tweet(san)
                assert res_topic == True
                assert res_tweet == True

    def test_not_allowed_input_validator(self):
        inp = Input()
        with open("app/tests/not_allowed.json", encoding="utf-8") as json_file:
            data = json.load(json_file)
            for elem in data:
                san = inp.sanizate(elem)
                res_topic = inp.validate_topic(san)
                assert res_topic == False
```

Illustration 79. two sets of test

The not allowed json file contains a list of 437 prohibited strings once they have been sanitized. The following figure collects some of them:

```
"2.2250738585072011e-308",
",./;'[]\\-=",
"<>?:\"{}|_+",
"!@#$%^&*()`~",
"\ufeff",
```

Illustration 80. sanitization

The third tests located in *web/app/app/tests/test_hdfs.py* focuses on assuring that the tweets that should be published are stored into HDFS. This test makes use of our *insert_hdfs_tweet* to upload a new json file into Hadoop and to read it back and assure that the content of the file is the same as the uploaded once:

```
class HDFSRequestTest(TestCase):
    def test_file_insert(self):
        text = "awesome twit"
        filename = insert_hdfs_tweet(text, "unit-test")
        r1 = requests.get(
            "http://node1.itapp.eus:9870/webhdfs/v1/unit-test/"
            + filename
            + ".json?user.name=hadoop&op=OPEN&noredirect=true"
        )
        data = r1.json()
        url = data["Location"]
        url = url[7:]
        idx = url.index(":")
        url = "http://" + url[:idx] + ".itapp.eus" + url[idx:]
        response = requests.get(url)
        assert response.status_code == 200
        decoded = response.content.decode("utf-8")
        assert text in decoded
```

Illustration 81. HDFS request

It is worth mentioning that this test is not mucked, meaning that some data is persisted into HDFS. In normal production like application development, testing should not be performed with the production database, or in this case, the distributed file system cluster. But duplicating the HDFS cluster for testing is quite expensive. For this reason, the testing is performed using the production cluster but using a specific directory only used for testing. Even though it's not a perfect testing practice, we achieve partial isolation by using different directories.

Finally, we have created a test that mocks a query to ElasticSearch to retrieve published tweets between two dates. This test can be found in

web/app/app/tests/test_elasticsearch.py. The mock simulates that query and returns two elements.

```
def mock_execute():
    list_res = []
    list_res.append(hit("Text-test1", "date1"))
    list_res.append(hit("Text-test2", "date2"))
    return list_res


@mock.patch("elasticsearch_dsl.Search.execute", side_effect=mock_execute)
class ESQueryTestMock(TestCase):
    def test_query(self, mock_execute):

        search = query_timestamp_range("2020-05-20", "2020-05-29")
        print("mock res", search)
        count = 0
        for hit in search:
            print(hit.tweet)
            if count == 0:
                assert hit.tweet == "Text-test1"
                assert hit.date == "date1"
            else:
                assert hit.tweet == "Text-test2"
                assert hit.date == "date2"
            count = count + 1
        assert count == 2

    def test_tweet_model(self, mock_execute):
        tm = TweetModel("text", "date")
        assert tm.tweet == "text"
        assert tm.date == "date"
```

Illustration 82. Mocking

With the test we assure that the functionality of searching published tweets between two dates works as expected using our function.

3.2.2.3 Deployment stages

Applications need to be available for many groups of people. From the point of view of the user experience and security, the application available for the general public should have as few bugs that can cause trouble not only the application (and the whole infrastructure as the database) but also how it can affect the user's perception of the application.

In order to minimize and reduce the chance that an unidentified bug is present on the application that users use, developers have different environments where the application is deployed, each of them with different characteristics and different target audience.

For this project, there are three different environments also known as stages. Each of them fulfils a role to ensure that the final version deployed to the public is risk free, secure

and with no unintended behavior. These three stages are called *development stage*, *staging* and *production stage.*

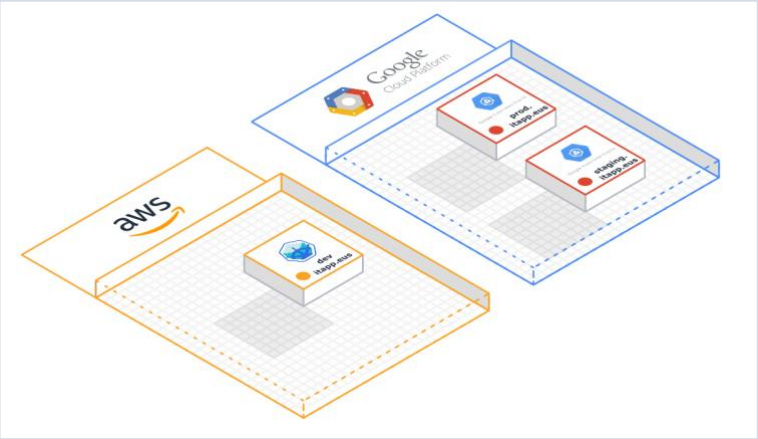

Illustration 83. Deployment staging

The first one, the development stage is only available by the programmers and developer, and as a consequence, it is not available to the public. It is used to see the effects that each new feature, bug correction, infrastructure change affects the whole application ecosystem. In this project, this stage was deployed on one of our cloud providers. To be more precise, the development stage was deployed in AWS dude to the already existent big data infrastructure. That way, data transit between the application and the storage systems is minimized from the point of view of the latency, but also the cost due to the data transfer because the data was always on the same virtual private network, and no exchange with the outside was required.

Initially, the team decided to use the AWS service called Elastic Kubernetes Service or EKS for shorter, but due to the classroom limitations, we were not able to create Kubernetes clusters using their own service. We analyzed other alternatives such as Kubernetes Operations but due to a higher complexity to set up the cluster and maintenance of this option, other options were explored.

Finally, the easiest option was selected for the development stage. Minikube was set up to run our Kubernetes clusters. This first cluster only had one node. But for development purposes is enough.

The next two stages, the staging and production one, the Google Kubernetes Engine service was used. For these scenarios, instead of just using only one node for each Kubernetes cluster, two were used. We could've used more, but to minimize the

maintenance cost of having 24/7 active machines, the decision was taken to have just two nodes. It is worth mentioning that each staging had its own GKE cluster to isolate from each other.

3.2.2.4 Deployment composition

Up next, a description is given of how each deployment stage is structured using Kubernetes resources. The following list of resources is replicated in each Kubernetes cluster except the last one (the ingress resource) because it is not created in the development cluster.

The first step to deploy the application into kubernetes is to define the docker file of the Django application. We used a python docker image and the required dependencies were installed. The following image sums up the content of said docker file. We called it the base image.

```
# pull official base image
FROM python:slim-buster

# set environment varibles
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# install requirements
RUN pip install --upgrade pip && \
    pip install \
        gunicorn==20.0.4 \
        Django==3.0.6 \
        tensorflow==2.2.0
```

Illustration 84. Deployment composition

As seen, there is no actual Django code or anything in particular that is worth mentioning. The reason of creating this docker file is fairly simple. It diminishes the time to build new images from the ground up, if the core and immutable (or static) dependencies are already downloaded.

The actual docker file that virtualizes the Django application, also called, production docker file has the following content:

```
# pull official base image
FROM registry.gitlab.com/haritz.saiz/popbl2:base

# set environment varibles
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# set working directory
RUN mkdir /code
WORKDIR /code

# install requirements
COPY web/requirements.txt /code/
RUN pip install --upgrade pip && pip install -r requirements.txt

# copy code
COPY web/app/ /code/
CMD [ "python",  "./manage.py", "runserver", "0.0.0.0:8000" ]
```

Illustration 85. Dockerfile that virtualizes the Djuango

This docker file uses the previous image and installs extra, smaller, dependencies and starts a web server that hosts the Django application using port 8000.

Once all the images were built and uploaded to the project's container repository on GitLab, the actual Kubernetes deployment can start.

The first resource that must be created in order to deploy the app into Kubernetes is to create the corresponding Kubernetes secret that contains the credentials to access GitLab to pull the latest version of the Django docker file. To create such secret, it is only necessary to run the following command:

```
$ kubectl create secret docker-registry regcred --docker-server=registry.gitlab.com
--docker-username=${GITLAB_USERNAME}  --docker-password  ${GITLAB_PASS}  --docker-
email ${GITLAB_EMAIL}
```

The secret is named *regcred* and it is used by the next Kubernetes resource: the deployment. This resource is fairly simple. It just declares the image to use, the number of replicas to create (set to 1), and the uses some extra metadata and selectors. Due to the zero-downtime strategy that will be described later on, two deployments were created, each of them with different selectors. Here is an example of the blue deployment yaml file:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: django-blue
spec:
  replicas: 1
  selector:
    matchLabels:
      app: django-blue
      deployment: blue
  template:
    metadata:
      labels:
        app: django-blue
        deployment: blue
    spec:
      containers:
        - name: web
          image: registry.gitlab.com/haritz.saiz/popbl2:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8000

      imagePullSecrets:
        - name: regcred
```

Illustration 86. Secret

It is also worth mentioning, that this deployment only makes use of one container per pod, and that the only exposed port per pod is the 8000 port.

The next resource create in the three clusters is the service one. This service defines how the actual deployments are reached from the outside (or the inside of the cluster). The following image sums up the service used:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: django
spec:
  type: NodePort
  ports:
    - name: django-port
      protocol: TCP
      port: 80
      targetPort: 8000
  selector:
    app: django-blue
    deployment: blue
```

Illustration 87. Service

The interesting part of this service is that it doesn't expose any port directly to the public as the kind of service used is NodePort. Well that's not actually true, the service is exposed, well kind of. We will use an Ingress to make it available.

Finally, the last Kubernetes resource that must be created for both the staging and production stages is an Ingress. This ingress will redirect all the incoming traffic to the service. The reason behind using the ingress was taken because it is way easier to securely communicate clients to the cluster using a tls secret. This secret can be generated with the following command:

```
$ kubectl create secret tls ${CERT_NAME} --key ${KEY_FILE} --cert ${CERT_FILE}
```

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: prod-ingress
  annotations:
    kubernetes.io/ingress.allow-http: "false"
spec:
  tls:
  - secretName: prod-secret
  rules:
  - host: prod.itapp.eus
    http:
      paths:
      - backend:
          serviceName: django
          servicePort: 80
```

Illustration 88. Ingress

So, in order to access the development Kubernetes cluster an SSH with port forwarding must be set up by the dev team.

3.2.2.5 Branch strategy

Our Gitlab repository for this project has been structured into 4 branches. Each of these branches has a porpoise. First, let's name them: Dev app, Dev model, Pre-production and master.

The first mentioned is Dev-app. The aim of the branch is to provide an environment where developers can upload the latest updates of our Django application. As it can be seen on image 89, the branch is not closed. In fact, no branch in our repository is closed at any point. The reason is very simple. We need previously developed code to continue with the development and there is no need to fork other branches, simple updates are enough.

The second branch, called Dev model, is where different model versions are uploaded. Once they have a candidate model, a merge request is performed to the dev-app branch. Once the dev-app branch has a candidate application ready to ship it to pre-production,

a merge request is performed, and so, the app is deployed to a staging environment where a selected number of users can test the application.

Once we are sure that the application is ready for prime time, a new merge request is performed from pre-production to the master branch. The new app version is shipped to the production server and thus, all users having access to the updated app.

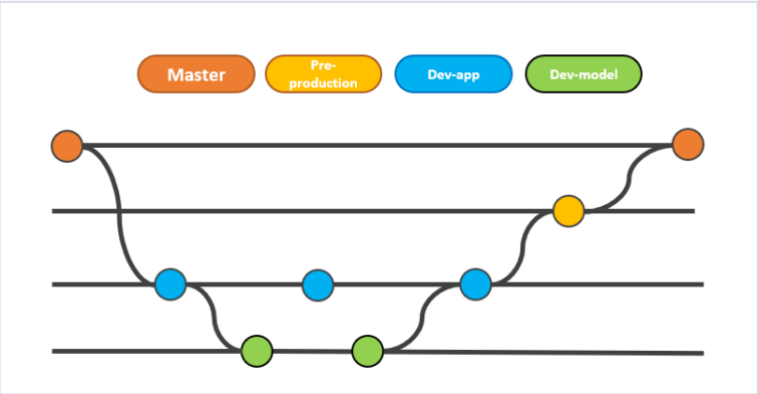

Illustration 89. Branch strategy

All explained so far is the desired workflow to manage the repository. But did the developing team respect this strategy? To find the answers, a simple look to GitLab repository is enough. The following figure corresponds to the last sets of commits and merge request done to our repository

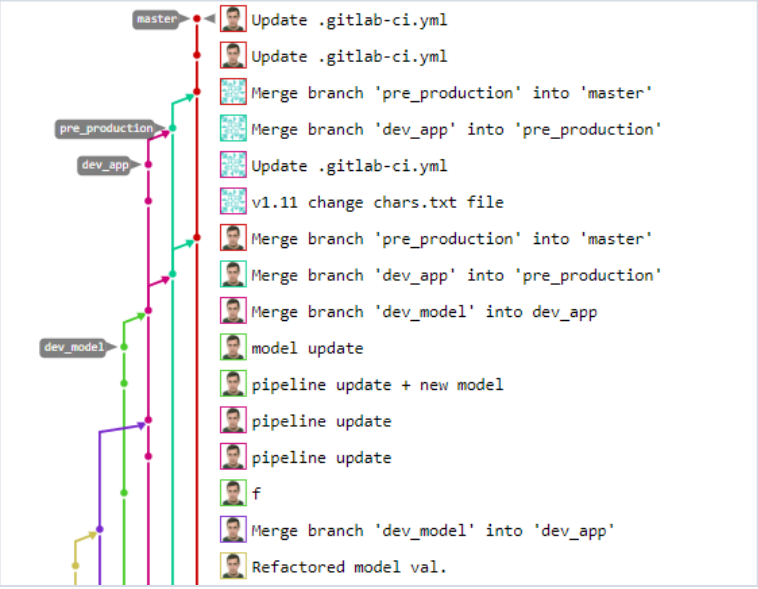

Illustration 90. Graph

As seen on the image the merge request flow has been followed pretty well. There are a couple of “branches” a yellow and purple one that we are not able to identify. GitLab

doesn't label them either, but leaving those two aside, the branch strategy proposed, has also been executed.

3.2.2.6 Pipeline

So far, the CI/CD fundamentals have been set. The app has a set of unit tests, we have our three deployment stages ready to deploy different app versions, but there is no automation so far. That's when our pipeline comes in. A total of 9 stages have been defined in the pipeline, each of them oversees different aspects. Image X Let's begin by analyzing closely the first stage:

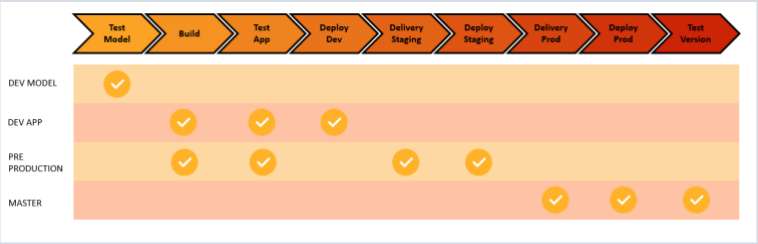

Illustration 91. Pipeline

Test Model

- **Branches**: Dev model
- **Porpoise**: The aim of this stage is to assure that only new models that decrease the model's loss pass the pipeline. If a model improves, means that the application will generate better text, and thus, among other things, users will have to edit less text, or the generated text will have more overall sense.
- **Stage definition:** There is only one job for this stage, and its definition is as follows:

```
test-model:
  stage: test-model
  image: registry.gitlab.com/haritz.saiz/popbl2/model-validator
  script:
      - apt-get update && apt-get install curl -y
      - python ml/model_validation.py web/app/app/tensorflow/model.h5 web/app/app/tensorflow/chars.txt
  only:
    - dev_model
```

Illustration 92. Test model

Well, this job uses a docker image that has TensorFlow installed on top of a python3 image and has a validation dataset located at */data/val_data.txt* of the docker image. When the job starts, it installs curl as it will be used by the python validator script. Said script can be found in *ml/model_validator.py* and takes two arguments, the first one is the Keras model itself, and the second one is a text file used to decode integers to chars and

vice versa. The following image, shows how the main method of the validation script works:

```python
def main():
    model = load_model(sys.argv[1])
    X, y = get_val_data()
    loss = model.evaluate(X, y)
    print("Old loss:", os.environ.get("MODEL_LOSS"), "  ->   New loss:", loss)
    if float(os.environ.get("MODEL_LOSS")) <= loss:
        print("Model not improoved")
        exit(-1)
    else:
        print("Model improoved")
        os.environ["MODEL_LOSS"] = str(loss)

    curl = (
        "curl --request PUT --header 'PRIVATE-TOKEN:"
        + os.environ.get("TOKEN")
        + "'  'https://gitlab.com/api/v4/projects/"
        + os.environ.get("CI_PROJECT_ID")
        + "/variables/MODEL_LOSS' --form 'value="
        + str(loss)
        + "' "
    )
    os.system(curl)
```

Illustration 93. main method of the validation

This script loads the given model and evaluates it with the validation data present in the docker image. In order to determine if the new model is better than the previous one, a GitLab environment variable is used called MODEL_LOSS. This environment variable can be accessed within a python script by using the `os` library. By simply calling `os.environ.get("MODEL_LOSS")` the GitLab variable can be read and used by the validator process. Only if the new model loss decreases, the job succeeds, otherwise, the job fails by calling an `exit(-1)`.

When the model loss decreases are time to update the GitLab environment variable. Updating the variable using the previously used library didn't work but calling GitLab's API did. That's why a CURL is performed. In order to do so the following elements have to be passed to the request: The project ID of the repository hosting the GitLab env variable available in the pipeline using the predefined env variable `CI_PROJECT_ID`, the name of the env variable to update, the new value of said variable and a private access token with at least API permissions as the following one:

Name	Created	Expires	Scopes	
GanttLab	May 18, 2020	In 12 days	api	Revoke
Pipeline	Jun 2, 2020	In 24 days	api, read_user, read_api, read_repository, write_repository, read_registry, write_registry	Revoke
haritz	Mar 21, 2020	In about 2 months	api, read_user, read_repository, write_repository, read_registry	Revoke

Illustration 94. Repository

- **Stage output:** The following image sums the result of a passing job due to a loss decrease from 100 to 1.9

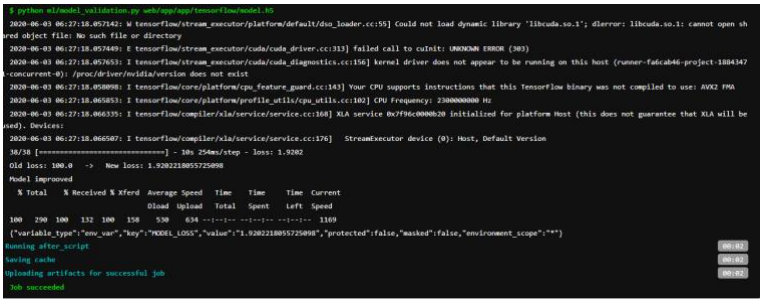

Illustration 95. Stage output

In order to check that the update worked, a quick lock to GitLab's env variable is enough:

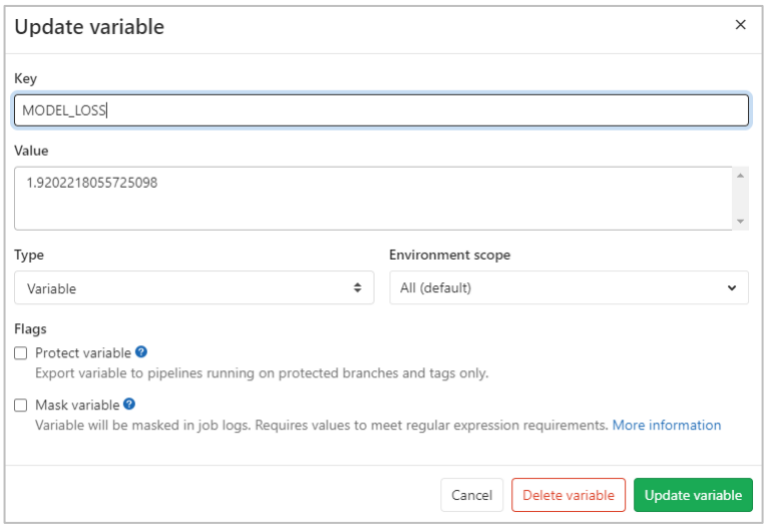

Illustration 96. Update variable

But the model fails if the new loss is the same as the previous loss:

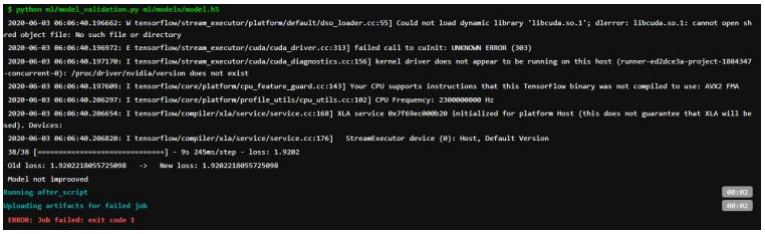

Illustration 97. model fails

Build

- **Branches**: Dev app, Pre-production
- **Porpoise:** The aim of this branch is to generate a new docker image that contains the new source code for the Django application.
- **Stage definition**: There is only one job for this stage, and its definition is as follows:

```
build:
  stage: build
  image: docker:stable
  services:
    - docker:dind
  variables:
    DOCKER_DRIVER: overlay2
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_JOB_TOKEN $CI_REGISTRY
    - docker pull $IMAGE_LATEST || true
    - docker build
          --cache-from $IMAGE_LATEST
          --tag $IMAGE_TEST
          --file ./web/Dockerfile.prod
          "."
    - docker push $IMAGE_TEST
  only:
    - dev_app
    - pre_production
```

Illustration 98. build

There is nothing fancy or complex to understand. First, the job logs in to GitLab container registry using three env variables available in the pipeline `CI_REGISTRY_USER` for the username `CI_JOB_TOKEN` for the password `CI_REGISTRY` for the exact registry name. Then, it tries to pull the latest available Itapp image and uses it to build a new docker image using the previous one to cache it from. Once the image is built, it is pushed back to the container repository using a tag containing the keyword test plus the commit SHA.

- **Stage output**: The results are quite simple to see. If all the above worked, it should appear on the container registry

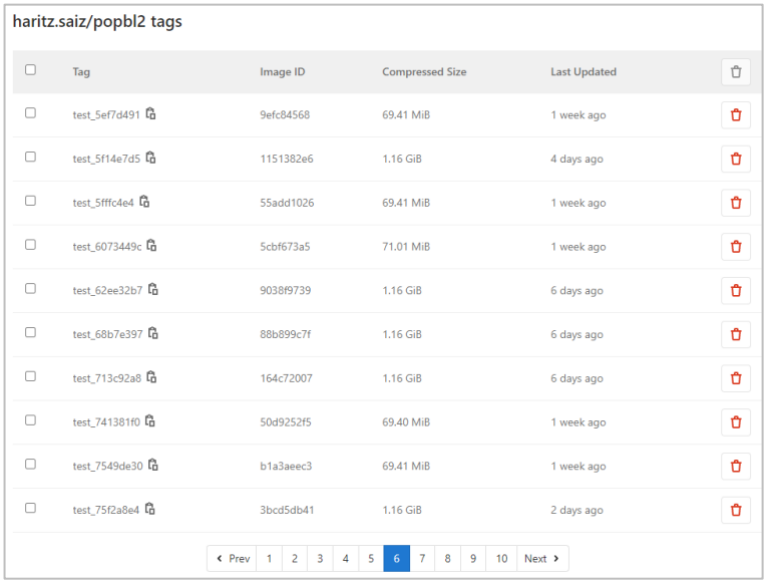

Illustration 99. container registry

Test App

- **Branches:** Dev app, Pre-production
- **Porpoise**: Assure that the application has no known bugs and vulnerabilities. In order to do so not only unit testing is executed, but also static code analysis.
- **Stage definition:** There are two jobs for this stage, and its definition is as follows: The first one, the testing code job executes the set of unit test that were explained earlier, checks that the code is well formatted and reports back to GitLab the coverage percentage of the tested code. The following image corresponds to this job:

```
test-code:
  stage: test-app
  image: $IMAGE_TEST
  script:
      - cd /code
      - black . --check
      - coverage run --source=app manage.py test
      - coverage report
  coverage: '/TOTAL.+ ([0-9]{1,3}%)/'
  only:
    - dev_app
    - pre_production
```

Illustration 100. Test app

The second job called SonarQube runs a scan to the Django app code to find security vulnerabilities, code smells and asses how much time it would take to fix all those issues. Those scan results are reported back to our sonar cube server for manual revision.

```yaml
sonarqube:
  stage: test-app
  image: emeraldsquad/sonar-scanner:1.0.2
  variables:
    SONAR_PROJECT: popbl2
    SONAR_HOST: http://34.67.2.152:9000/
  script:
    - cd web/app
    - sonar-scanner
          -Dsonar.projectKey=$SONAR_PROJECT
          -Dsonar.host.url=$SONAR_HOST
          -Dsonar.login=$SONARQUBE_AUTH
  only:
    - dev_app
    - pre_production
```

Illustration 101. Sonarqube

- **Stage output**: First let's analyze the results of the test-code job after it runs the unit tests. Well the coverage code tested is 69%. The results could've improved if the py files generated by the Django structure were not included.

```
$ coverage run --source=app manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
[<app.tests.test_elasticsearch.hit object at 0x7efe82d48ac0>, <app.tests.test_elasticsearch.hit object at 0x7efe82d48b50>]
<elasticsearch_dsl.search.Search object at 0x7efe82d48910>
mock res <elasticsearch_dsl.search.Search object at 0x7efe82d48910>
Text-test1
Text-test2
.......
----------------------------------------------------------------------
Ran 7 tests in 0.464s
OK
Destroying test database for alias 'default'...
$ coverage report
Name                              Stmts   Miss  Cover
-----------------------------------------------------
app/__init__.py                       0      0   100%
app/asgi.py                           4      4     0%
app/forms.py                          5      0   100%
app/hdfs.py                          17      0   100%
app/input.py                         34      7    79%
app/models/elastic_model.py          42     14    67%
app/settings.py                      19      0   100%
app/tensorflow/predict.py            27     22    19%
app/tests/__init__.py                 0      0   100%
app/tests/test_elasticsearch.py      32      0   100%
app/tests/test_forms.py              12      0   100%
app/tests/test_hdfs.py               17      0   100%
app/tests/test_input.py              23      0   100%
app/urls.py                           4      0   100%
app/views.py                         54     41    24%
app/wsgi.py                           4      4     0%
-----------------------------------------------------
TOTAL                               294     92    69%
Running after_script
Saving cache
Uploading artifacts for successful job
Job succeeded
```

Illustration 102. test code analyze

It is worth noting that earlier unit testing had a less covered code. A total of 59% was covered (https://gitlab.com/haritz.saiz/popbl2/-/jobs/570770845) and it even had less developed code. So, it is safe to say that the team improved the app's code coverage.

To see the results of the second job, it will be enough to analyze the SonarQube server results:

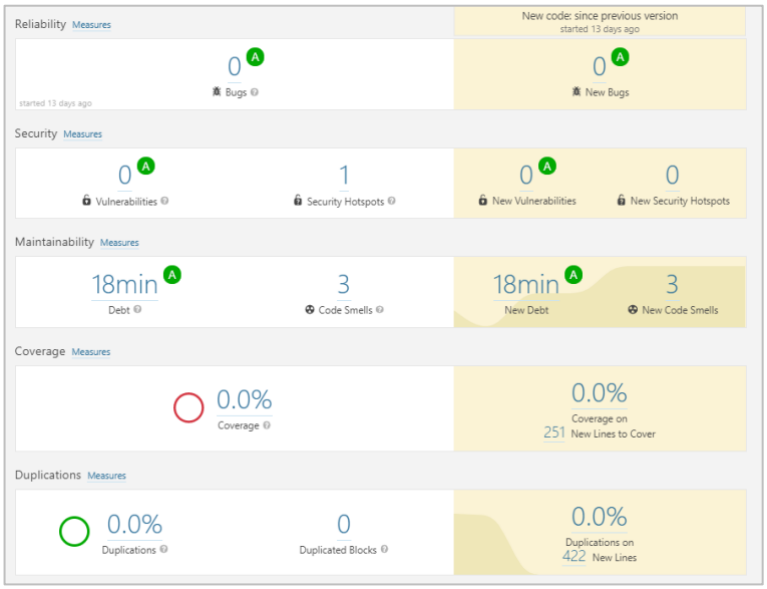

Illustration 103. results

The static analysis reveals that we don't have any code having known vulnerabilities but one security hotspot. A security hotspot differs from vulnerabilities due to the nature of detection. In other words, it is impossible to determine if it is a threat for the application or not in an automatic way. It needs of human code review to determine if it has any impact at all.

The detected security hotspot refers to the use of using command lines arguments. To run the server Django server. A security problem may happen if an attacker was able to change those arguments to list the process list of the OS, or even worse if the Django app runs using root privileges.

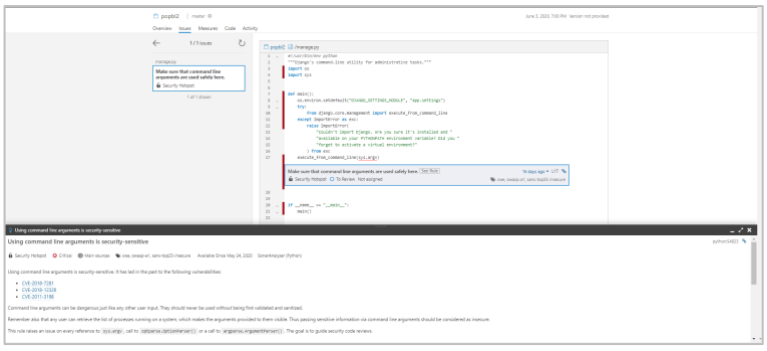

Illustration 104. Django app run

From the point of view of the coding practices we have the following alerts:

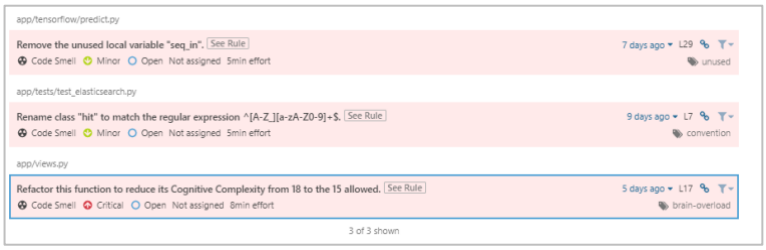

Illustration 105. Alerts

They all are easy to fix so in future versions they will be fixed.

Deploy Dev

- **Branches**: Dev app
- **Porpoise:** The aim is to deploy the tested application into the dev environment.
- **Stage definition:** There is only one job for this stage, and its definition is as follows:

```
deploy-dev:
  stage: deploy-dev
  image: debian:10
  before_script:
    - apt-get update -qq
    # Setup SSH deploy keys
    - 'which ssh-agent || ( apt-get install -qq openssh-client )'
    - eval $(ssh-agent -s)
    - ssh-add <(cat $SSH_DEV)
    - mkdir -p ~/.ssh
    - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrictHostKeyChecking no\n\n" > ~/.ssh/config'
  script:
    - |
      SSH_CMD=$(cat << EOF
        /home/ubuntu/k8s-configure.sh dev
        kubectl set image deploy django-green web=$IMAGE_TEST
        kubectl set image deploy django-blue web=$IMAGE_TEST
      EOF
      )
    - ssh ubuntu@dev.itapp.eus -i $SSH_DEV "$SSH_CMD"
  only:
    - dev_app
```

Illustration 106. deploy-dev

What it is done its quite straight forward. The job logs in via SSH to the dev machine using the GitLab env variable `SSH_DEV` hosting the pem ssh file to access said machine. It switches the Kubernetes context to use minikube's and updates both the blue and green django's deployment images.

Delivery Staging & Delivery Production

- **Branches:** Pre-production (Delivery staging), Master (Delivery production)
- **Porpoise:** Tag the Docker testing docker image with a new one.
- **Stage definition:** There is only one job for this stage, and its definition is as follows:

The jobs logs in again to GitLab's container registry and pulls the latest docker image tagged by testing (for the delivery staging job) or the latest staging image(for the delivery production job).

Then, it tags with the correspond name and pushes them back to GitLab's container registry.

```
delivery-staging:
  image: docker:stable
  stage: delivery-staging
  services:
    - docker:dind
  variables:
      DOCKER_DRIVER: overlay2
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_JOB_TOKEN $CI_REGISTRY
    - docker pull $IMAGE_TEST
    - docker tag $IMAGE_TEST $IMAGE_RELEASE_STAGING
    - docker push $IMAGE_RELEASE_STAGING
    - docker tag $IMAGE_RELEASE_STAGING $IMAGE_LATEST_STAGING
    - docker push $IMAGE_LATEST_STAGING
  only:
    - pre_production
```

Illustration 107. delivery staging

```
delivery-prod:
  image: docker:stable
  stage: delivery-prod
  services:
    - docker:dind
  variables:
      DOCKER_DRIVER: overlay2
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_JOB_TOKEN $CI_REGISTRY
    - docker pull $IMAGE_LATEST_STAGING
    - docker tag $IMAGE_LATEST_STAGING $IMAGE_RELEASE_PROD
    - docker push $IMAGE_RELEASE_PROD
    - docker tag $IMAGE_RELEASE_PROD $IMAGE_LATEST_PROD
    - docker push $IMAGE_LATEST_PROD
  only:
    - master
```

Illustration 108. delivery-prod

- **Stage output**: The new pushed images appear on GitLab's container registry for both jobs:

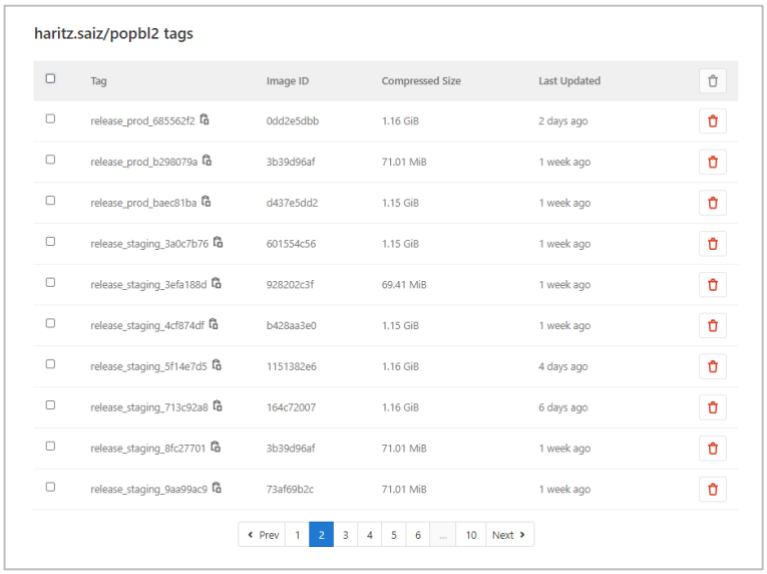

Illustration 109. container registry

Deploy Staging & Deploy Production

- **Branches**: Pre-production (Deploy staging), Master (Deploy production)
- **Porpoise**: Update the Kubernetes deploys to use the newest docker images containing the latest betted source code with zero downtime during the updating process.
- **Stage definition:** There is only one job for this stage, and its definition is as follows:

```
deploy-staging:
  stage: deploy-staging
  image: google/cloud-sdk:latest
  services:
    - docker:dind
  script:
    - gcloud auth activate-service-account --key-file $GOOGLE_CREDENTIALS
    - gcloud config set project $PROJECT_ID
    - gcloud container clusters get-credentials $CLUSTER_STAGING_NAME --region us-east1-d
    - |
      res=$(kubectl get svc django -o custom-columns="SELECTOR:.spec.selector.app" | tail -n 1); if [ $res == "django-blue" ];
```

Illustration 110. Deploy staging & deploy production

In order to update the Kubernetes images, it is important remembering that both staging, and production clusters are hosted using GKE services. So, this time there is no need to SSH to the machines hosting the cluster. Google cloud SDK allows for an easy access to our clusters by providing a JSON file that has among other things, the credentials needed to authenticate.

Once authenticated by the SDK, a project context must be configured as well as selecting the cluster to run the Kubernetes commands with.

Now is where the complex stuff begins. The aim is to update the images used by the deployments without having any downtime. On top of that, instead of going for an easier

zero-downtime strategy as it would be using a canary approach, it is intended to use the blue-green zero-downtime strategy.

The next image corresponds to the formatted code in charge to deploy using the blue-green strategy:

```
res=$(kubectl get svc django -o custom-columns="SELECTOR:.spec.selector.app" | tail -n 1);
if [ $res == "django-blue" ];
    then kubectl set image deploy/django-green web=$IMAGE_RELEASE_STAGING;
    kubectl scale --replicas=2 deploy.apps/django-green; kubectl rollout status deployment django-green -w ;
     dcolor='kubectl patch service django -p '"'"'{"spec":{"selector":{"app": "django-green", "deployment": "green" }}}'"'"'';
     kubectl scale --replicas=0 deploy.apps/django-blue ;

else kubectl set image deploy/django-blue web=$IMAGE_RELEASE_STAGING;
     kubectl scale --replicas=2 deploy.apps/django-blue;
     ubectl rollout status deployment django-blue -w;
     dcolor='kubectl patch service django -p '"'"'{"spec":{"selector":{"app": "django-blue", "deployment": "blue" }}}'"'"'';
     kubectl scale --replicas=0 deploy.apps/django-green;
fi;
eval $dcolor;
kubectl get svc -o wide
```

Illustration 111. blue-green strategy

- **Stage output**: Let's analyze the previous script line by line. The first line gets the "color" or the selector that the Kubernetes service is pointing at currently:

```
res=$(kubectl get svc django -o custom-columns="SELECTOR:.spec.selector.app" | tail -n 1);
```

```
ubuntu@node1:~$ res=$(kubectl get svc django -o custom-columns="SELECTOR:.spec.selector.app" | tail -n 1);
ubuntu@node1:~$ echo $res
django-blue
ubuntu@node1:~$
```

Illustration 112. django-blue

So now that the current service selector is obtained, in this case the blue one, indicates which deployment has to be updated (the one not being pointed by the selector, in this case, the green one). The green deployment images used by the "web" container is updated to use the latest image:

```
$ kubectl set image deploy/django-green web=registry.gitlab.com/haritz.saiz/popbl2:latest_staging
```

Now it is time to increase the number of pods that will be used by the green deployment to two instead of zero:

```
$ kubectl scale --replicas=2 deploy.apps/django-green;
```

Finally, it is time to update the service selector to point to the green deployment instead of the blue but after the health checks of the green deployments (and pods) are OK not after, otherwise there would be a downtime while the pods are being booted. That's what this line is used for:

```
$ kubectl rollout status deployment django-green -w;
```

This will block the command execution until the pods are up and running. Then we update the Kubernetes service selector with this command:

```
$ dcolor='kubectl patch service django -p '"'"'{"spec":{"selector":{"app": "django-green", "deployment": "green" }}}'"'"';

eval $dcolor;
```

And finally, the blue deployment is scaled to zero because the service won't be redirecting any traffic to that deployment:

```
$ kubectl scale --replicas=0 deploy.apps/django-blue ;
```

If run all the command at once, the output is as follows:

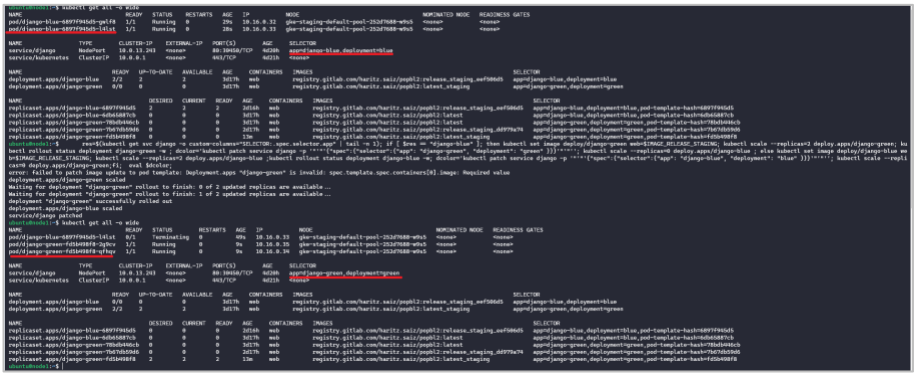

Illustration 113. output commands

Test version

- **Branches**: Master
- **Porpoise:** Check that the web app version number has updated when the production deployment has finished to the newer version. This way, developers can test that the deployment didn't have any undesired side effects, bugs or errors.
- **Stage definition:** There is only one job for this stage, and its definition is as follows:

```
test-deploy-version:
  stage: test-deploy-version
  image: debian:10
  script:
      - apt-get update && apt-get install libxml2-utils -y && apt-get install wget -y
      - chmod u+x version-checker.sh
      - ./version-checker.sh
  only:
    - master
```

Illustration 114. Test version

This job installs a library that will be in charge of finding the HTML DOM element that hosts the version number using XPath queries. In order to run said command, the HTML file must be obtained. That's why it is also required to install wget.

After that the version checker script is executed. The content of said script is as follows:

```
wget https://prod.itapp.eus --no-check-certificate
xmllint --html --xpath '//div[@class="footer"]' index.html > tmp
stringarray=($(cat tmp))
res1=`eval echo ${stringarray[2]}`

xmllint --html --xpath '//div[@class="footer"]' web/app/app/templates/index.html > tmp
stringarray=($(cat tmp))
res2=`eval echo ${stringarray[2]}`

echo $res1
echo $res2

if [ $res1 == $res2 ]
then
  echo "Version match"
  exit 0
else
  echo "Version missmatch"
  exit -1
fi
```

Illustration 115. version checker script

- **Stage output:** First, the served HTML file is downloaded, after that the XPath query finds the div element containing the version identifier:

```
ubuntu@node1:~$ xmllint --html --xpath '//div[@class="footer"]' index.html
<div class="footer"> v1.11 / django-green-5d745978ff-6k4qd</div>ubuntu@node1:~$
```

Illustration 116. Xpath query

If the both version match (source code and production code) then the job succeeds as it can be seen on the next image:

```
$ chmod u+x version-checker.sh
$ ./version-checker.sh
--2020-06-04 06:41:49--  https://prod.itapp.eus/
Resolving prod.itapp.eus (prod.itapp.eus)... 34.120.57.230
Connecting to prod.itapp.eus (prod.itapp.eus)|34.120.57.230|:443... connected.
WARNING: The certificate of 'prod.itapp.eus' is not trusted.
WARNING: The certificate of 'prod.itapp.eus' doesn't have a known issuer.
HTTP request sent, awaiting response... 200 OK
Length: 5336 (5.2K) [text/html]
Saving to: 'index.html'
     0K .....                                                 100% 8.84M=0.001s
2020-06-04 06:41:49 (8.84 MB/s) - 'index.html' saved [5336/5336]
v1.11
v1.11
Version match
Running after_script
Saving cache
Uploading artifacts for successful job
Job succeeded
```

Illustration 117. job succeeds

3.3 Security assessment – ISAAF based

A security breach can be compared to a robbery. When a burglar forces the door and enters a house with access to everything inside, it is considered a security breach. An unauthorized person who breaks into someone else's system is violating privacy, and this is increasingly common, which can happen to both large companies and one like ours, ITAPP.

A cybercriminal or malware tries to avoid the protection mechanisms of a computer system to access restricted areas and steal data or access information of interest.

Warren Buffett said something that every business should take very seriously, and that is that it takes "a lifetime to build a good reputation and only 5 minutes to destroy it".

To avoid this, a pentesting have been done.

3.3.1 What is a pentesting?

Pentesting is a simulation of a cyber-attack whose purpose is to compromise a computer system to identify and classify its flaws, vulnerabilities, and other security errors, and to see how big they are. Its objective is to locate the gaps to prevent external attacks, make the business see what it is facing and make the risks visible.

A penetration test is a way of doing hacking but in a permissive and legal way. It is ethical hacking since it has the full consent of the owners of the computers on which the test is to be performed. Likewise, the damage caused is totally controlled.

Once the problems have been analysed, sufficient knowledge is obtained to determine what the system's defences are: what are the chances of success of a cyber-attack, what is the organization's response capacity? Pentests answer all these questions.

3.3.2 Benefits of a penetration test

Today, there is a certain fear of carrying out an attack that is so intrusive. To try to extinguish that feeling, let's take a look at the advantages of conducting a security check on your system:

1. This test is done with the consent and measurement of the tested organization and under a contract that delimits scope and responsibilities.
2. Mitigate potential threats to protect the integrity of the network.
3. To have more information to manage and combat potential vulnerabilities efficiently.
4. Test the capacity of the entity's cyber security.
5. To uncover the fragilities of the system before a cybercriminal does.
6. Comply with data protection regulations and mitigate sanctions.
7. Reduce costs associated with downtime caused by such incidents.
8. Protect the company's reputation and ensure business continuity

To check some of the aspects contemplated in this section, a pentesting to the implemented system has been made which you can observe here.

3.4 Security management

Considering that we live in a globalized and competitive world in which companies are faced with new challenges daily, it is increasingly important to manage information security in the company and thus avoid the loss of its most valuable asset today: data.

Both the Information Security Management Systems and the work networks of any organization are constantly affected by security threats, cyber-attacks, and computer fraud. In addition, they are constantly faced with sabotage or viruses with the consequent risk of elimination and loss of information.

The key is for the organization to invest resources in applying tools that improve security.

As far as information security is concerned, some of the fundamental aspects that must be analysed and measured are:

- The availability of data
- The confidentiality of documents
- Information integrity

So, how can incidents be minimized? Among other things, by preparing an inventory of assets, in which all the information available in the company is identified and located.

On the other hand, these assets must be valued to avoid possible failures and finally, the risks that these assets may face must be analysed and a protocol to face those risks must be established.

Finally, through different documents developed we will try to cover different aspects of security management explained in the ISO 27000 certification which brings together good practices for the establishment, implementation, maintenance and improvement of Information Security Management Systems.

3.4.1 Risk analysis & SOA

In this section, each of the risks must be identified by establishing the appropriate controls to manage the risk in a systematic way so that we consider all the important aspects for the company.

For this purpose, an analysis has been made considering all the assets, vulnerabilities and risks of the company, and the following document has gone into detail.

In addition to the analysis, some controls have been developed to contemplate different aspects of the development, such as the following.

3.4.2 Information Systems Requirements Specification

Before starting any development, it is necessary to consider what are the quality requirements to which the software has to respond. That is why a specification of the requirements has been made, identifying the systems/subsystems to be implemented, as well as the requirements.

3.4.3 Safe development policy

In order to have a safe development and to contemplate the security in each one of the stages of the development it has been proceeded to develop a control that defines different aspects of the development as they can be the points of verification of the security or the management of vulnerabilities.

3.4.4 Control procedures for system changes

Another aspect that we have tried to cover is the change control process. Change is that moment when both the client and a member of the development team want to modify the development in some way. This change usually involves a modification of the development and may even vary the delivery date or scope. To make this change as documented and clear as possible, a process has been defined to follow each time you want to make one.

3.4.5 Information security policy and requirements in the relationship with suppliers

As at the time of a complex development it is normal to work with different providers that provide different functionalities such as in our case, access to social networks in order to collect data, it is important to define a security policy.

3.4.6 Legal compliance analysis

As in our case we work with data that can be very intimate, we have proceeded both to identify legal requirements, as well as to record the activities of the treatments of these data, as well as the evaluation of the impact of these data in case of any breach.

All this can be seen gathered in the following document .

4. Challenges faced

This section explains in detail each of the problems that have been encountered throughout the project with a short explanation of how were solved.

4.1 Nifi cluster setup problems

NiFi is one of the pillars of the project due to basically the entire data transfer process was carried out with this tool, with the specific feature that in the cloud infrastructure a cluster mode deployment was necessary, which generate the following problems:

- **Flow in Nodes:** to carry out the first tests only the master node was used and once the flow was working correctly the whole cluster was deployed. This generated some problems in the workers because at the moment they joined to the master node the flow was different and must be the same between all the members of the cluster. As a solution it was necessary to transfer the file "flow.gz.xml" to the other workers.
- **Amount of logs**: NiFi by default does not have a limit of generation and storage of logs so after several days using the machines the amount of logs stored were GiGas of data, something that saturated the machine and prevented the correct functioning of it. As a solution the logs were removed from all the machines where NiFi was installed and the configuration of NiFi was changed in order to generate a maximum of three log files of 50 MB each of them.

4.2 Node red integration with Nifi

As mentioned above, the third source of data consisted in using a python script to simulate the generation of data from a dataset hosted on the same machine locally.

From the first moment it has been known that this is a bad practice because no tool is used to send the data to NiFi in Input, as a consequence the use if NodeRed was tried as an intermediate tool between data and NiFi, without getting a positive result due to the following problems.

For the deployment of NodeRed, the docker architecture has been used, deploying the application through a docker-compose:

```
version: '3'
services:
  nodered:
    container_name: node-red
    image: nodered/node-red-docker
    volumes:
      - /home/ubuntu/nodered/data:/data
    ports:
      - "1880:1880"
    restart: always
```

Illustration 118. Nodered

In the same way, a volume has been mounted inside the application that will store the file to be sent from NodeRed to NiFi.

Once NodeRed has been correctly deployed, the UI is located at port 1880 and create the flow that will be formed by the following "processors":

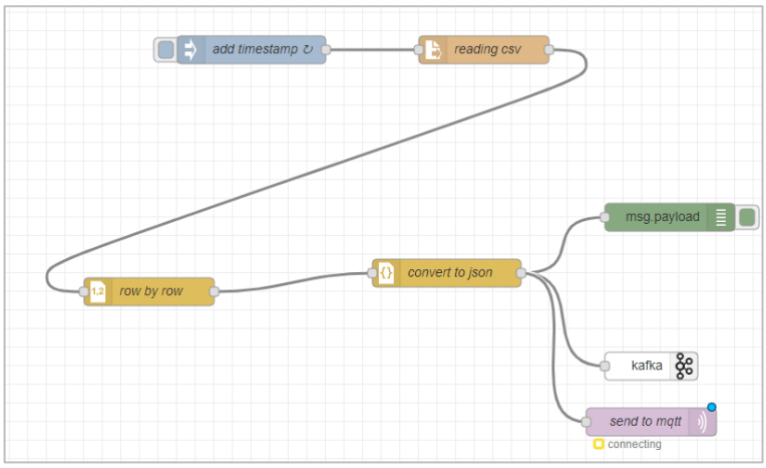

Illustration 119. Nodered flow

- Timestamp: period between every execution of the flow.
- Reading csv: read the dataset that emulate the third source.
- Row by Row: pass the file read line by line
- Convert To Json: convert each line of the csv file into a json that can be received correctly by NiFi.
- Output: two brokers were tried to send the data generated in NodeRed to Input point:

- MQTT: on one hand MQTT was selected and the connection was successfully established but there were some problems with the NiFi processor "consumeMQTT" related with the TCP Ip of the machine.
- Kafka: the connection was established in a satisfactory way, however, the data that arrived to NiFi was in an inadequate and strange format, due to the short time available this option had to be discarded as well.

Finally, it was decided to continue with the method initially proposed.

4.3 Kubernetes taint problems (no disk space)

Sometime during the development of the project, during a routinely deployment of Itapp to the development environment, an anomalous behavior was detected where pods would not successfully run when they should. In fact, running Kubernetes command such as getting all deployments, pods or services wouldn't work either.

After investigating the issue, it was found that the Kubernetes cluster nodes indicated the following: KubernetesHasNoDiskPressure. Later on, it was also found that the running pods, that at that time did use persistent volumes and persistent volume claims were not enough to host the docker image size. The reason was that the last docker build, did change one of its pip requirements. TensorFlow was installed. That dependency made that Itapp build size was 50MB or so, sky rocked to more than 1GB.

The first decision was to increase the persistent volumes and persistent volume claims to make room for the brand new docker images. And it did work. The minikube cluster gained back its disk pressure.

Later on, the Itapp deployment did no longer required to install nginex on each pod, and during this refactoring process of the deployment, it was also decided to not make use of volumes, and instead, let each pod use as much space as required to let them room for growth during development.

4.4 Data sources problems

Problems about data sources are explained in this section of the document.

5. Conclusions

One of the biggest strengths of the project is that the continuous integration and deployment set up has sped many phases that would've been done manually otherwise. Even though it requires that set up phase than may take some time, it ends up being worth it due to the amount of time that is not wasted.

The Kubernetes technology has a lot of potential due to the amount of things than can be achieved with it. Perhaps this project has worked with the typo of the iceberg of such enormous tool. Never the less, the experience gained will be extremely helpful in our professional life.

From the point of view of the big data architecture, it is worth noting that nifi gave us a lot of freedom to create custom scripts to validate incoming data. But it also came with its problems. Debugging the nifi processors was a time-consuming task that gave us a couple of headaches.

The infrastructure is ready for replication on the non-cloud levels if more data was to be consumed. The Hadoop architecture (if scaled with more nodes and capacity) could store the huge amount of incoming data and therefore no data would be dropped.

6. Future lines

- Itapp should be able to scale automatically thus responding to the demand the web application has, meaning, the more users it has the more resources it should use and consume.
- The pipeline should stop if the static code analysis found the code does not comply with set software coding styles and rules. For now the static code analysis is something that is executed and checked by the developers afterwards.
- The tweet generation model should be a lot more generic, meaning, it should reach a point were it should be able to generate tweets about any new topic discussed in social media. This would mean the data consumption and model training should, at least up to a point, be automatized.
- The cluster based architecture, in the same manner as the demand based scaling, should be dynamically resizable and data mirroring should be set up to answer to real needs instead of being a simple mirror copy per node.
- The pentesting and fuzz tests should be deepened, either with new techniques or with guides on how to carry them out.
- The documentation regarding the 27001 and 27002 ISOs should be updated so Itapp can actually manage to obtain those certifications.

7. Bibliography

https://github.com/mseclab/PyJFuzz

https://karpathy.github.io/2015/05/21/rnn-effectiveness/

https://www.openvas.org/

https://www.g2.com/products/openvas/competitors/alternatives

https://community.greenbone.net/t/openvas-vulnerability-sweet32/1211/4

https://nmap.org/book/zenmap-scanning.html

https://www.kali.org/tutorials/configuring-and-tuning-openvas-in-kali-linux/

https://issues.apache.org/jira/projects/NIFI/summary

https://www.jython.org/

https://machinelearningmastery.com/how-to-develop-a-word-level-neural-language-model-in-keras/

8. Annexes

- Política de desarrollo here.
- RGPD here.
- Procedimiento de control de cambios here.
- Análisis de riesgos documentación here.
- Especificación de requisitos here.
- 15-1-1 Relación con proveedores here.
- 15-1-2 Controles Contrato Proveedores here.
- Análisis de riesgos excel here.
- Pentesting here
- Software security practice here
- Fuzzing Evidences (Documents/Fuzzing Evidences)