Serverless Tweet Analysis Implementing Cloud Patterns

Tchanjou Njomou Aquilas

April 2020

Abstract

Various types of patterns, particularly design patterns, are famous and helpful ideas in software engineering. They are constantly utilized in structuring and building up software with different architectures. However, adaptability and reusability of the software sometime accompanies the cost of diminished effectiveness. Simultaneously, performance is frequently a key quality characteristic of cloud applications. It is therefore valuable to investigate whether design patterns can impact cloud applications efficiency. This paper examines the impact of two design patterns on a cloud application interacting with a database: : Proxy pattern and sharding pattern. The results shows that the patterns has a significant impact on the performance of the application, and that the sharding patterns outperforms the proxy pattern.

1 Introduction

One of the most common and basic software quality attributes is efficiency. Performance is a non-functional requirement which is an essential factor to consider in enterprise cloud applications to achieve high application quality. A design pattern in object-oriented software is generally thought of as a collection of reusable solution to a frequently occurring design problem. When Patterns are used, they often come with some extra indirect layer and creates characteristics of increased complexity in the program. That can have a positive or negative impact on results. Design pattern offers consistency in developing or refactoring a better software architecture but can provide no assurance of software quality output [1]

For this assignment, we were assigned the list A. In this paper we analyzed the effect of the proxy and sharding pattern on the performance of a cloud sentiment analysis application. We used some Amazon Web Services (AWS) features such as Lambda functions and Amazon Relational Database Service (RDS) to implement the different patterns. Since this work is aimed at assessing the efficiency of cloud applications, we are focused in storing the majority of data information with the least amount of resources. Since distribution of resources to run software systems is on average the same, more productive pattern will typically complete the same process in less time [8]

To assess this, we built a machine learning application that determine the sentiment of tweets, store the results in RDS instances using the proxy and sharding pattern, then retrieve the stored results. section 2 of this paper describes our sentiment analysis app and the library used. Section 3 presents the architecture built to implement each pattern, section 4 presents the pattern implementation, section 5 presents the evaluation methods and the results of our work and section 6 describes how to reproduce these results.

2 Building the sentiment analysis application

This section describes the process of building the application for sentiment analysis. The goal is to detect the sentiment of a tweet provided through a REST Application on Amazon. To build the model, we followed a set the process described in the previous work [2].

The field of Sentiment Analysis attempts to use computational algorithms in order to decode and quantify the emotion contained in media such as text, audio, and video. Sentiment Analysis also known as 'Opinion Mining' is used for the analysis of a text (word or sentence or a document) as a 'positive' , 'negative' or 'neutral'. The specific methods come down to two approaches: the lexical method and the machine learning approach [5]

Lexical approaches aim to map words to sentiment by building a lexicon or a 'dictionary of sentiment.' we are able to use this dictionary to assess the sentiment of phrases and sentences, without the requirement of observing anything. Lexical approaches observe the sentiment category or score of every word within the sentence and judge what the sentiment category or score of the full sentence is. the ability of lexical approaches lies within the fact that we don't have to train a model using labeled data, since everything needed to assess the sentiment of sentences lies within the dictionary of emotions.

Machine learning approaches, on the opposite hand, observe previously labeled data so as to work out the sentiment of never-before-seen sentences. The machine learning approach involves training a model using previously seen text to predict/classify the sentiment of some new input text.

We used the VADER (Valence Aware Dictionary for sEntiment Reasoning) which used the lexical approach to build our sentiment analysis application [7].

2.1 VADER Sentiment Analysis tool

VADER (Valence Aware Dictionary for sEntiment Reasoning) is a model used for sentiment analysis that is sensitive to both polarity (positive/negative) and intensity (strength) of emotion. Introduced in 2014, VADER sentiment analysis uses a human-centric approach, combining qualitative analysis and empirical validation by using human raters

Primarily, VADER relies on a dictionary which maps lexical features to emotion intensities called valence scores. The valence score of a text can be obtained by summing up the intensity of each word in the text. lexical feature are anything that are used for textual communication. On social medias, there are many different types of lexical features such as words, emoticons like ":-)", acronyms like "LOL", and also slang terms like "meh". Given the complexity of social media texts, we choose VADER as it takes into account all of the lexical features used on social medias like Twitter, and map them to intensity values as well.

Emotion intensity or valence score is measured on a scale from -4 to +4, where -4 is the most negative and +4 is the most positive. The midpoint 0 represents a neutral sentiment. Sample entries in the dictionary are "horrible" and "okay," which get mapped to -2.5 and 0.9, respectively. In addition, the emoticons "/-:" and "0:-3" get mapped to -1.3 and 1.5. VADER returns a valence score in the range -1 to 1, from most negative to most positive. The valence score of a sentence is calculated by summing up the valence scores of each VADER-dictionary-listed word in the sentence, then normalize it to a value between -1 to 1.

VADER has a lot of advantages over traditional methods of Sentiment Analysis, including:

- It works exceedingly well on social media type text, yet readily generalizes to multiple domains [6],
- It does not require any training data but is constructed from a generalize, valence-based, human-curated sentiment lexicon,
- It is fast enough to be used online with streaming data,
- It does not severely suffer from a speed-performance trade-off.

The function to get the sentiment of a text is prolarity_score which output is a dictionary with keys neg, pos, neu and compound. Let's take the example of this code: polarity_scores("This assignment is super cool"). The output is {'neg': 0.0, 'neu': 0.326, 'pos': 0.674, 'compound': 0.7351}

- The Positive, Negative and Neutral scores represent the proportion of text that falls in these categories. This means our sentence was rated as 67% Positive, 33% Neutral and 0% Negative. Hence all these should add up to 1.
- The Compound score is a metric that calculates the sum of all the lexicon ratings which have been normalized between -1(most extreme negative) and +1 (most extreme positive). In the case above, lexicon ratings for super and cool are 2.9 and respectively 1.3. The compound score turns out to be 0.75 , denoting a very high positive sentiment. the compound score is used as a threshold value for the analysis of the text data.
- Positive sentiment: compound score $\geq$ 0.05

- Neutral sentiment: (compound score $>$ -0.05) and (compound score $<$ 0.05)
- Negative sentiment: compound score $\leq$ -0.05
- We used the absolute value of compound score as overall score for positive and negative sentiments, as the more the compound is close to 1 (or -1 for negative sentiments), the more accurate is the sentiment detected. However, for neutral sentiments, we used the 'neu' score, because the compound value for neutral sentiments lies between -0.05 and 0.05, it is not very relevant to the actual accuracy of the result.

2.2 Data acquisition and preprocessing

We used the dataset provided by the lab, having a certain number of tweets gathered from twitter. The data was in a csv format. We converted it in json format as stated in the assignment.

VADER is not a machine learning tool. It doesn't need any training and testing process. The library includes the code and the lexicons necessary for analysis. As the VADER tool is particularly suitable for social media text, the data preprocessing step is not mandatory. The main point to take into account was to make sure the tool gets a text he could analyse. This was done by removing any non ascii characters from each input text.

3 Tools and general architecture

Using the vader sentiment analysis tool, we built a cloud app to detect and save the sentiments of tweets. We took advantages of the aws platform and python library flask. The main aws tools used are: Lambda functions, Amazon EC2 instance, Amazon s3 and Amazon RDS.

3.1 Tools

3.1.1 The Flask library

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

We used flask to build a simple rest application that accept post request from the user. The body of the request contains the data that will further be analysed by the sentiment analysis tool.

3.1.2 Lambda functions

AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of Amazon Web Services. It is a computing service that runs code in response to events and automatically manages the computing resources required by that code. It was introduced in November 2014. The purpose of Lambda, is to simplify building smaller, on-demand applications that are responsive to events and new information. AWS targets starting a Lambda instance within milliseconds of an event. AWS Lambda supports securely running native Linux executables via calling out from a supported runtime such as python. AWS Lambda was designed for use cases such as image or object uploads to Amazon S3, updates to DynamoDB tables, responding to website clicks or reacting to sensor readings from an IoT connected device. AWS Lambda can also be used to automatically provision back-end services triggered by custom HTTP requests, and "spin down" such services when not in use, to save resources.

We built two lambda functions to handle the sentiment analysis part and the pattern implementation.

3.1.3 Amazon EC2

Amazon Elastic Compute Cloud (EC2) forms a central part of Amazon's cloud-computing platform, Amazon Web Services (AWS), by allowing users to rent virtual computers on which to run their own computer applications. EC2 encourages scalable deployment of applications by providing a web service through which a user can boot an Amazon Machine Image (AMI) to configure a virtual machine, which Amazon calls an "instance", containing any software desired. EC2 provides users with control over the geographical location of instances that allows for latency optimization and high levels of redundancy

3.1.4 Amazon RDS

Amazon Relational Database Service (or Amazon RDS) is a distributed relational database service by Amazon Web Services (AWS). It is a web service running in the cloud designed to simplify the setup, operation, and scaling of a relational database for use in applications. Administration processes like patching the database software, backing up databases and enabling point-in-time recovery are managed automatically. Scaling storage and compute resources can be performed by a single API call as AWS does not offer an ssh connection to RDS instances.

3.1.5 Amazon S3

Amazon S3 or Amazon Simple Storage Service is a service offered by Amazon Web Services (AWS) that provides object storage through a web service interface. Amazon S3 uses the same scalable storage infrastructure that Amazon.com uses to run its global e-commerce network. Amazon S3 can be employed to store any type of object which

allows for uses like storage for Internet applications, backup and recovery, disaster recovery, data archives, data lakes for analytics, and hybrid cloud storage.

3.2 General architecture

Figure 1 shows the overall architecture of the cloud application.

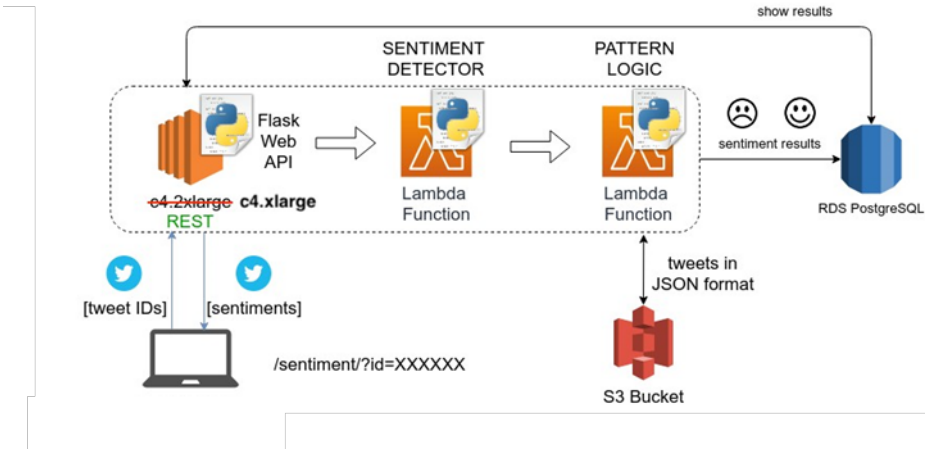


Figure 1: Design Architecture of the Sentiment Analysis Data Pipeline

This pipeline has four main steps: (1) Getting and preprocessing the input data through Flask REST API, (2) Analysis of the data to get the sentiments, (3) Saving the result data to s3 and RDS through the pattern implementation middleware and (4) Retrieving the data from RDS.

3.2.1 Data acquisition and preprocessing:

This is done through the flask app running on an EC2 instance in the AWS cloud. The app listens to port 5000 and accepts only POST request with the json data in the body. As the application is already deployed in the cloud, there is no need to implement a different web server to handle incoming public requests. We opened the adequate port on the ec2 instance to accept all incoming requests. The user just needs to make a POST request to the amazon instance's IP and the results will be provided to him through a JSON response. There are two possible url to call: /sentiment/sharding to use he sharding pattern, or /sentiment/proxy to use the proxy pattern. Once it receives data from the user's post request, the app preprocesses the data, removing non ASCII characters and deleting tweets without id or data. It then triggers the first lambda function and pass it the preprocessed data. An example of the input data from the user is shown below:

```
    data = {
         "1": {
            "id": 1246953034355298304,
            "text": "some sample tweet",
            "date": "2020-04-05 00:12:10"
           },
         "2": {
            "id":1246490999347707905,
            "text":"this is another sample tweet",
            "date": "2020 04-06 00:11:59"
            }
        }
```

After preprocessing the input data, the flask app calls the lambda_handler1 and passes new data with the chosen pattern. Let's say the user called /sentiment/proxy, the flask app will call the lambda_handler1 with the input in the json format shown below:

```
        {
         "pattern": "proxy",
         "data": data
        }
```

3.2.2 Sentiment analysis:

The sentiment analysis is deployed on a lambda function called lambda_handler1. This function is triggered via a boto3 call from the flask app in ec2. After the analysis is done, it triggers the second lambda function lambda_handler2 and passes the analysed data as input. The data passed to lambda_handler2 as argument will look like this:

```
    data = {
         "1": {
            "id": 1246953034355298304,
            "sentiment": "Positive",
            "score": 0.458
           },
         "2": {
            "id":1246490999347707905,
            "sentiment":"Negative",
            "score": 0.684
            }
        }
```

The input for lambda_handler will still looks the same as the one for lambda_handler1, except that the value of data will be different.

3.2.3 Saving to s3 and RDS through pattern implementation:

The lambda_handler2 has two main functions: first save the data it receives to amazon s3 bucket, then uses a specific pattern to save the data to RDS. data are saved to s3 bucket using boto3 library. This function also implement both patterns, and each pattern is selected via the input parameters received from the user.

3.2.4 Retrieving data from RDS:

After the data have been saved to s3 and RDS, the flask app then retrieves the data from RDS using psycopg2 and sends it as json response to the user. All the intermediate steps are transparent to the user, who gets just the output sent by the flask app. The output data will be the same as the data passed as input parameter to lambda_handler2, with the same keys.

4 Pattern implementation

The patterns were implemented in the lambda_handler2 serverless app. Each pattern is selected according to the original url called by the user: /sentiment/proxy or /sentiment/sharding.

4.1 Architecture

The architecture of the pattern implementation is described at Figure 2.

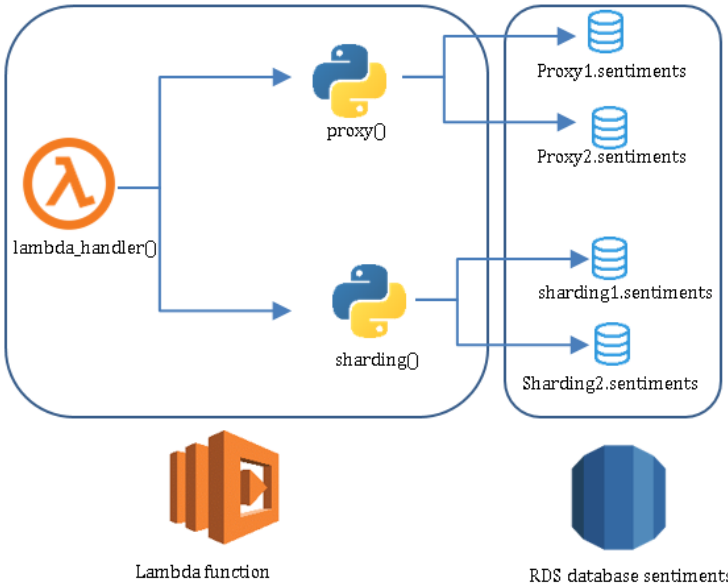


Figure 2: Architecture used for pattern implementation

The lambda_handler() function is the entry point. It handles the input data and determines which pattern should be executed according to the input data. It then calls the corresponding pattern function (sharding() for sharding pattern and proxy() for proxy pattern). Each pattern function acts as a load balancer which select the instance where any tweet will be stored.

For the RDS instances, we created a postgresql database called sentiments, and split it into schemas to simulate real instances. A schema is a named collection of tables. A schema can also contain views, indexes, sequences, data types, operators, and functions. Schema are analogous to directories at the operating system level, except that schema cannot be nested. All the schema and tables have the same characteristics, in order to avoid creating any bias during the evaluation of the patterns.

We used psycopg2 to communicate with the postgresql database. We created a function called exec() that takes an instance name and data as input, then saves the data to that instance. This way, the instance selection was left to the function implementing the pattern.

4.2 Proxy pattern

To implement the proxy pattern, we created two schema in the database "sentiments": proxy1 and proxy2. In each schema, we then created a table called "sentiments" to store the results of the sentiment analysis. The proxy pattern had two dedicated tables in two schemas: proxy1.sentiments and proxy2.sentiments. Note that the table name "sentiments" is different from the database name "sentiments". The table is included in a schema, which is included in the database.

The function proxy() as shown in figure 2 implements the proxy pattern. It takes as input the list of tweet sentiments and iterates over it. At every iteration, it randomly selects an instance using the choice() method from the random library:

```
1    instance = random.choice(["proxy1", "proxy2"])
```

It then saves the tweet sentiment to the database of that schema and returns the execution result message. The returned message is either "Everything was inserted" if the exec() function performed well, or the python generated error message if something didn't go as expected or if the database can't be reached due to any external or internal reason. The code for the proxy() function is shown below:

```
1    def proxy(data):  # Proxy pattern
2    for k, t in data.items():
3        #k is the key id of the tweet
4        #t is the tweet's sentiment as dict (with id, sentiment and
             score)
5        instance = random.choice(["proxy1", "proxy2"])  # Randomly
             selects the instance to save the data
6        res, message = exec(instance, k, t)
```

```
        if not res:
            break
    return message
```

4.3 Sharding pattern

To implement the sharding pattern, we followed the same process as the proxy pattern. The name of the schemas for sharding are sharding 1 and sharding2. We ended up with two tables for sharding, shardind1.sentiments and sharding2.sentiments, like those for the proxy pattern. These schemas were created in the same database and with the same characteristics as those for the proxy pattern. The function sharding() as shown in figure 2 implements the sharding pattern. It takes as input the list of tweets and get the number N of tweets in the list. It then randomly chooses a number k between 1 and N, and saves the k first tweets in the first instance (sharding1.sentiments) and the rest (N-k) in the second instance (sharding2.sentiments). It returns the execution result message. The code for the sharding() function is shown below:

```
    def sharding(data):  # sharding pattern
    N = len(list(data))
    k = random.choice(list(range(1,  N+1)))  # k first data goes to
       first instance
    i = 0
    for key, t in data.items():
        instance = "sharding1" if i < k else "sharding2"  # Specify the
           instance where the data will be saved
        res, message = exec(instance, key, t)
        if not res:
            break
        i += 1
    return message
```

5 Evaluation and results of the patterns

To evaluate the impact of each pattern on the performance of the cloud application, we measured the overall time the application takes to execute all the process: sentiment analysis, saving to rds and s3, then retrieve the data from rds. We used the dataset provided by the assignment. it has 287 tweets that were preprocessed, analysed, then saved to the database. To improve the accuracy of the results and asses the bias due to the network latency, We executed the app 10 times with the same data. Between each iteration, we cleared all data from instances, because the size of the database can affect its performance. [4]. To measure the execution time, we used the library

time from python. We saved the actual time at the beginning and at the end of each function, then calculated the difference. The algorithm is as below:

```
    start_time = time.time()
    #Execute the function or the process
    end_time = time.time()
    execution_time = end_time - start_time
```

This experiment was conducted for three cases: without any pattern (saving directly to one database), with the proxy pattern and with the sharding pattern. For each case, we measured:

- **lh1_time**: It's the time (in seconds) the lambda_handler1 function took to predict the sentiments of the list of tweets it got as input
- **lh2_time**: Time (in seconds) the lambda_handler2 function took to save the data to the RDS (Note that we didn't monitor the time taken to upload data to s3, and started monitoring after the data were uploaded.
- **time_to_predict_and_save**: The time it took to the function to execute all the cloud application but without retrieving the results (just from preprocess to saving to rds)
- **time_to_retrieve**: The time it took to the flask app to retrieve the saved data from rds
- **whole_time**: The overall time the cloud application took to execute all the processes.

5.1 Without any pattern

The first experiment was conducted when using no pattern. All the analysed data were saved to a single database. The results are shown in table 1 The results show that the app takes an average of 2.55 seconds to execute the whole process, with a low standard deviation (0.08). This is an average of 8ms per tweet, assuming the tweets have the same size and that all tweets are viable (that some of them didn't get deleted because they had no id or date). This shows that the database performs well and is quite stable. The average time for lambda_handler1 and lambda_handler2 are respectively 0.62 and 0.86 seconds. While the time to predict and save is higher (2.18 sec) and the time to retrieve is very low (0.37 sec). These values are more valuable when compared to the patterns, which is done in section 5.4

5.2 With proxy pattern

The results for the proxy pattern are shown in table 2. The results show that the app takes an average of 3.02 seconds to execute the whole process, with a standard devia-

Table 1: Time achieved whith no pattern

iteration	lh1_time	lh2_time	time_to_predict_and_save	time_to_retrieve	whole_time
1	0.624023914	0.960613918	2.258588743	0.36676693	2.625363541
2	0.620035648	0.879396868	2.208995533	0.360149622	2.5691535
3	0.636692762	0.893225861	2.201987696	0.390302658	2.592298222
4	0.636848688	0.791883183	2.123940659	0.374773502	2.498722029
5	0.609066248	0.84590168	2.163388681	0.371151209	2.534548712
6	0.611304522	0.891854715	2.32019825	0.35297513	2.673180771
7	0.624049187	0.842738342	2.162710142	0.373355389	2.536073637
8	0.63273859	0.744892311	2.017928314	0.387973309	2.405910206
9	0.648338079	0.805005026	2.100226355	0.401647568	2.50188179
10	0.636467457	0.987281752	2.308615875	0.353138924	2.661764336
avg	0.62795651	0.864279366	2.186658025	0.373223424	2.559889674
stddev	0.012444138	0.074570213	0.094046844	0.016158578	0.082098666
max	0.648338079	0.987281752	2.32019825	0.401647568	2.673180771
min	0.609066248	0.744892311	2.017928314	0.35297513	2.405910206

Table 2: Time achieved whith proxy pattern

iteration	lh1_time	lh2_time	time_to_predict_and_save	time_to_retrieve	whole_time
1	0.617187977	1.317959785	2.79123354	0.373896599	3.165137768
2	0.656997919	1.417725325	2.713817596	0.368520737	3.082345486
3	0.649387598	1.249205351	2.631650448	0.35793829	2.989597321
4	0.661391258	1.202753305	2.528459072	0.355886698	2.884354115
5	0.657817125	1.252223015	2.694923401	0.387187481	3.08211875
6	0.641163588	1.18662405	2.502055645	0.505064487	3.007128477
7	0.63362813	1.194458723	2.485582829	0.378605843	2.864196777
8	0.657580614	1.263959169	2.648019791	0.492594481	3.140621662
9	0.62950325	1.267178297	2.550128937	0.473023653	3.023169279
10	0.681364536	1.352118969	2.662436485	0.353573322	3.016017914
avg	0.6486021996	1.2704205990	2.6208307743	0.4046291590	3.0254687548
stddev	0.0185734022	0.0736641872	0.1009353651	0.0604568318	0.0982352809
max	0.6813645363	1.4177253246	2.7912335396	0.5050644875	3.1651377678
min	0.6171879768	1.1866240501	2.4855828285	0.3535733223	2.8641967773

tion of 0.09 seconds. The average time for lambda_handler1 and lambda_handler2 are respectively 0.64 and 1.27 seconds. While the time to predict and save is 2.53 sec and the time to retrieve is 0.4 sec).

5.3 With sharding pattern

The results for the sharding pattern are shown in table 3. The app takes an average of 2.85 seconds to execute the whole process, with a standard deviation of 0.08 seconds. The average time for lambda_handler1 and lambda_handler2 are respectively 0.62 and 1.16 seconds. While the time to predict and save is 2.48 sec and the time to retrieve is 0.37 sec).

Table 3: Time achieved whith sharding pattern

iteration	lh1_time	lh2_time	time_to_predict_and_save	time_to_retrieve	whole_time
1	0.6240239143	1.2606139183	2.5585887432	0.3667669296	2.9253635406
2	0.6200356483	1.1793968678	2.5089955330	0.3601496220	2.8691534996
3	0.6366927624	1.1932258606	2.5019876957	0.3903026581	2.8922982216
4	0.6368486881	1.0918831825	2.4239406586	0.3747735023	2.7987220287
5	0.6090662479	1.1459016800	2.4633886814	0.3711512089	2.8345487118
6	0.6113045216	1.1918547153	2.6201982498	0.3529751301	2.9731807709
7	0.6240491867	1.1427383423	2.4627101421	0.3733553886	2.8360736370
8	0.6327385902	1.0448923111	2.3179283142	0.3879733086	2.7059102058
9	0.6483380795	1.1050050259	2.4002263546	0.4016475677	2.8018817902
10	0.6364674568	1.2872817516	2.6086158752	0.3531389236	2.9617643356
avg	0.6279565096	1.1642793655	2.4866580248	0.3732234240	2.8598896742
stddev	0.0124441377	0.0745702129	0.0940468445	0.0161585777	0.0820986662
max	0.6483380795	1.2872817516	2.6201982498	0.4016475677	2.9731807709
min	0.6090662479	1.0448923111	2.3179283142	0.3529751301	2.7059102058

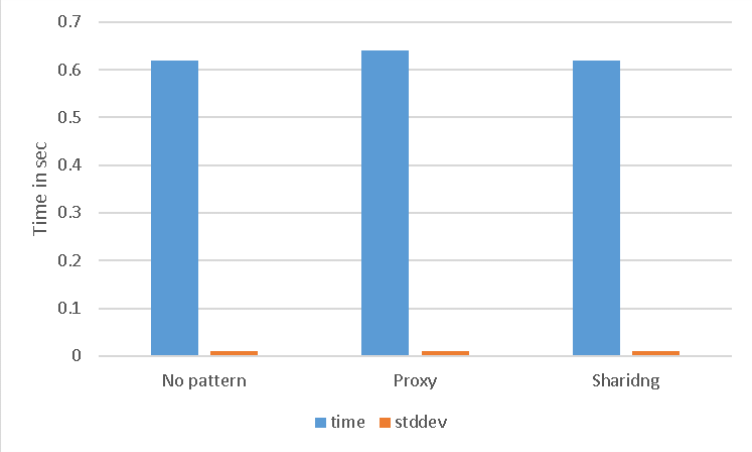


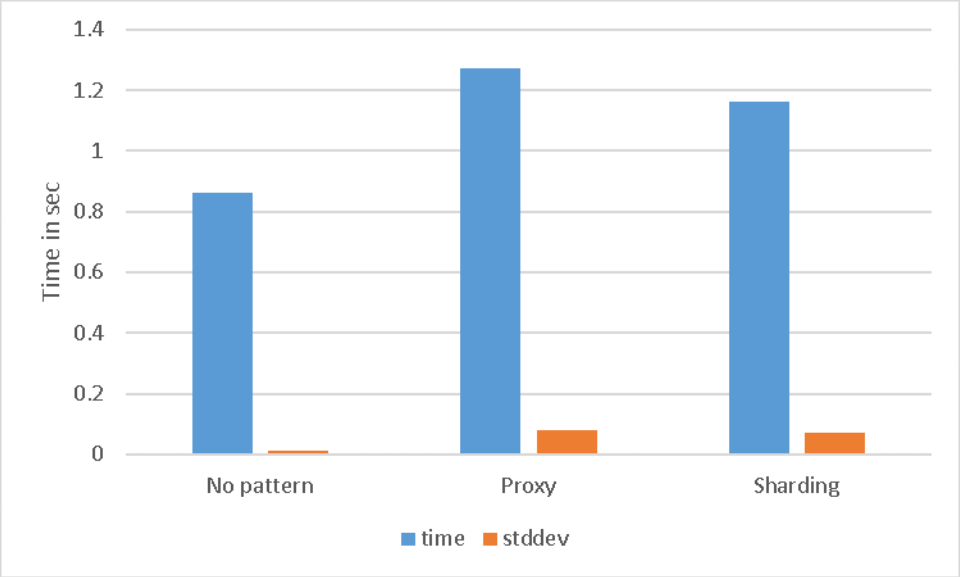
Figure 3: Execution time and standard deviation of lambda_handler1

5.4 Comparison

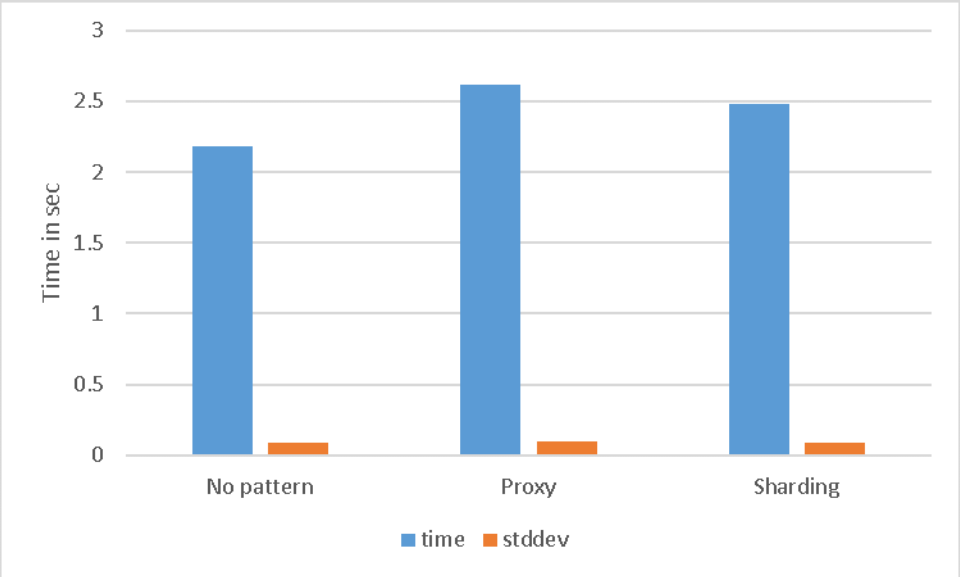
We compared each result to understand the impact of the pattern on the performance of the system. Figure 3 shows the comparison of the execution time and standard deviation for the first lambda_handler1 function .

The figure shows that the results are almost the same with or without pattern. This is obvious as the pattern was implemented only on the lambda_handler2 function and the lambda_handler1 function is the same for every experiment. Now let's look at the execution time of the lambda_handler2 function that implements the pattern, as shown in figure 4a

The figure shows that the implementation with no pattern has the lowest execution time (0.86 sec) while the the proxy pattern has the highest execution time (1.27 sec) and the sharding pattern is between the two with execution time of 1.16 sec. This means that the sharding pattern has a better performance.



(a) Execution time and standard deviation of lambda_handler2



(b) Execution time and standard deviation of prediction and execution

Figure 4: Execution time and standard deviation of lambda_handler2 and the prediction and saving process

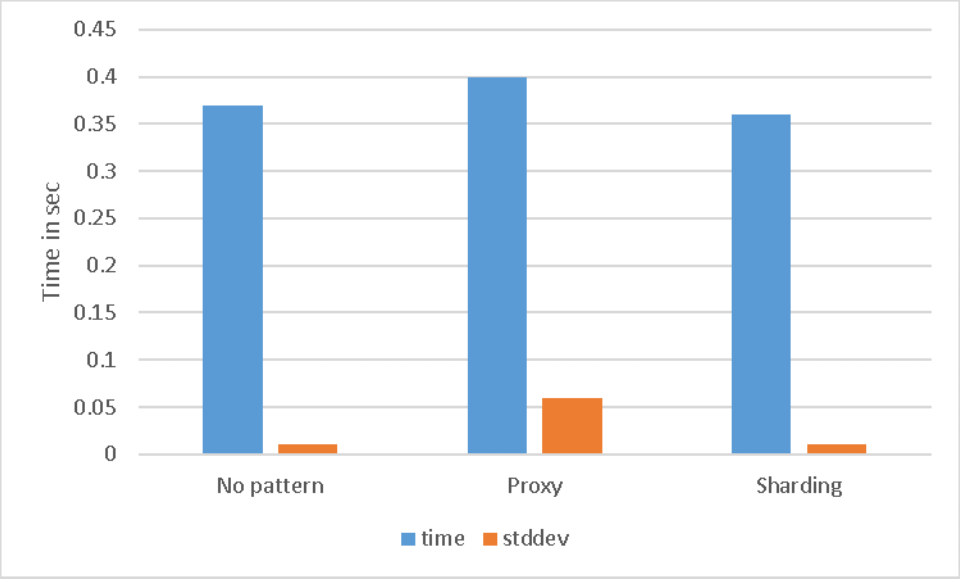


Figure 5: Execution time and standard deviation of data retrieval process

Logically, the execution time for predicting and saving (figure 4b) follows the same schema as figure 4a, as it combines lambda_handler1 and lambda_handler2.

For the data retrieval process, there is only a slightly difference between the execution time, as shown in figure 5. The proxy pattern still has the lowest performance (the highest execution time). This can be explained by the random distribution of the data in the instances, and the database needs to go through the whole instances to gather the data, while on the sharding pattern, the k first data are in the first instance and the others are in the second. This makes it easier to retrieve because the database selects the range of indexes, instead of checking the indexes one by one. However, the difference between the execution time of the proxy and sharding pattern is low (0.03 sec).

For the overall execution time of all the application, figure 6 shows the comparison between the different experiments.

As expected by analysing each function's execution time, the proxy pattern has the highest execution time (3.02 sec), which means it has the lowest performance, compared to the others. Also, the figure states that it may be better not to implement any pattern if we want a better performance, as the no pattern implementation has the lowest execution time. This can be true for very small databases. As stated by [4], the size of a table has an impact on the performance of queries to that database. While small database don't have a meaningful impact, large databases can have a big impact on the performance.

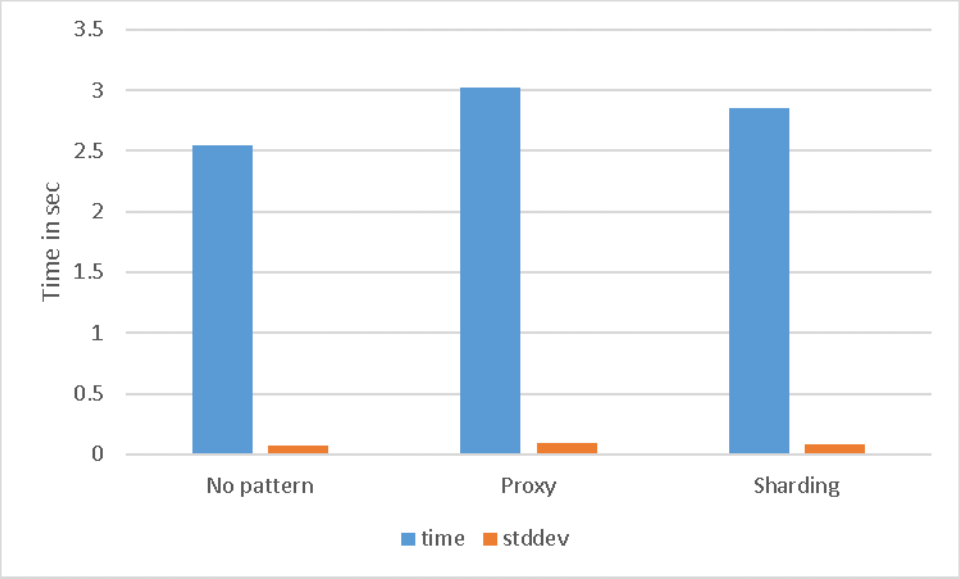


Figure 6: Execution time and standard deviation of the whole process

5.5 Threats to validity

- First, assessing the performance using just the execution time of the whole can't be completely valid because of the network latency and the intermediate activities not related to the application, like the firewall filters from AWS [3]. However, this was mitigated by also analysing the individual time of each functions and compared with the overall time. This was done by getting the sum of each process, then compared with the whole execution time automatically calculated by the application. We found that the latency was very slow, about 7 ms.

- Second, as the size of the database has an impact on its performance [4], the dataset we got from the assignment having only 287 records was not big enough to assess and eliminate the bias it may create on the performance. This bias can be assessed by conducting this experiment with different datasets with different size.

6 How to reproduce this work?

This report is given with five other files:

- **rds.txt**: This file contains the connection information to retrieve the data from the RDS instances

- **lambda_handler1.py**: This file contains the code to be implemented in the lambda_handler1 lambda function

- **lambda_handler2.py**: This file contains the code to be implemented in the lambda_handler2 lambda function
- **flask_ec2.py**: This file contains the code for the flask app to be launched in the ec2 instance.
- **results.json**: This file contains the extracted sentiment from the dataset provided by the assignment.

To reproduce this work, just execute the following instructions:

1. Go to AWS and Create your lambda functions lambda_handler1 and lambda_handler2.
2. Make sure to give the appropriate permission to lambda_handler1 to access the lambda_handler2 function or AWS will block the requests from lambda_handler1 to lambda_handler2. This can be done by giving the appropriate role or profile in the IAM console.
3. Add your bucket name to lambda_handler2.py. The variable name is bucket and is initialized at "aqui-tp3". If you don't change it, you will get an error
4. Connect to your ec2 instance and make sure python3 and pip are installed. Don't forget to open the port 5000, as the flask uses that port for inbound and outbound communication.
5. Upload the flask_ec2.py file to your ec2 instance and run python3 flask_ec2.py
6. Now send a post request to your server ip/sentiment/proxy to use the proxy pattern or your server ip/sentiment/sharding to use the sharding pattern, with the json data to analyse in the body. The HTTP answer from the app will be the analysed data, while the python console will print the execution time and message results as the one used in this paper to analyse each pattern

7 Conclusion

We report the results of our analysis for sentiment analysis and pattern comparison by implementing a sentiment detection algorithm using the VADER library. The steps followed was guided by the set of the instructions provided in assignment and some additional recent works with clear references where necessary. The results show that the proxy pattern has the lowest performance than the sharding pattern.

References

[1] Mawal Ali and Mahmoud O Elish. A comparative literature survey of design patterns impact on software quality. In *2013 international conference on information science and applications (ICISA)*, pages 1–7. IEEE, 2013.

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[3] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 35–46, 2010.

[4] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, page 287–296, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[5] Gab C. Vader sentiment analysis explained.

[6] Shihab Elbagir and Jing Yang. Twitter sentiment analysis using natural language toolkit and vader sentiment. In *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2019, 13-15 March, 2019, Hong Kong, pp12*, volume 16.

[7] Gilbert E.E. Hutto C.J. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth International Conference on Weblogs and Social Media (ICWSM-14)*, 2014.

[8] Muhammad Ehsan Rana, Wan Nurhayati Wan Ab Rahman, Masrah Azrifah Azmi Murad, and Rodziah Binti Atan. The impact of flyweight and proxy design patterns on software efficiency: An empirical evaluation.