Cloud Computing Coursework

COMM034

Alex Daniel Popa
6440987
ap00921@surrey.ac.uk

In the last few years cloud computing is expanding constantly, supporting an extensive environment where new solutions and services can be developed. The Cloud commit to facilitate cheap and flexible services to end-users and support small organizations and individuals to host and offer high-scale services themselves. In this paper, I introduce a cloud application that that determines average value at risk (VAR) for a trading strategy based on a simple moving average. With the examination and processing of Big data, it can help customers by creating a trading strategy. Giving them an analysis using Monte Carlo simulations on the trading signals of buy and sell, users can use this strategy to expand their trading portfolio. I describe the architecture of this system, the implementation steps, the results and an analysis of the costs of running the produced snapshot of this application.

Keywords— cloud computing, dynamic resource sharing, Amazon Web Services, Google App Engine

I. Introduction

The NIST definition accounts the following fundamental features of the cloud computing model: on-demand self-service, resource pooling, broad network access, measured service and rapid elasticity [1]. The system that is described in this paper provides the fundamental features presented in the NIST publication.

All the features of this system are supported and provided by Google App Engine and Amazon Web services as they provide computing capabilities without requiring a user to interact with them. The capabilities of the application are available over the network and enable access through all client platforms. The resources such as storage, processing, etc. are dynamically assigned and reassigned by the service providers in correspondence to the level of usage. Capabilities are elastically provisioned and release to scale with the demand. All the services provided by the providers are measured by and can be accessed by the system administrators for different purposes.

A. Developer

The system is making use of Platform as a Service (PaaS) as the software is deployed without any complexities of managing infrastructure services. The application is deployed on a Public cloud that is provisioned for open use by the general public. This enables the developer to create the application without the need of managing complex services and just by accessing the Public cloud.

B. User

The application by making use of PaaS is enabling the user to operate it with a single click eliminating the need of managing infrastructure services and/or dependencies. The application being deployed on Public cloud it is open for use by the general public and the user can access it freely

II. Final Architecture

A. Major System Components

The system is designed by making use of Google App Engine, subsequent mentions of Google App Engine will be as GAE, as client-side in which the user can be able to run an analysis with a single click. The client-side is designed using Flask as a web framework which makes use of templates that can be plugged to the application, replace if needed and can be possibly reused. This provides expansion capabilities without rewriting the whole client-side.

The logic of the application is handled by the server-side which is making use Amazon Web Services (AWS) to meet all the NIST definitions of essential characteristics [1]. In order to enable stateless communication between the client-side and the services in the AWS back-end, as shown in

, API Gateway services are used to create a REST API that can handle the requests coming from the client and to redirect the responses of those requests coming from the other AWS services.

As shown in

, API Gateway One is triggering emr_lambda Lambda function that is automating the process of creating an Elastic Map Reducer (EMR) cluster that performs a map reduce job in order to produce the Buy/Sell signals needed for the Value at Risk (VAR) analysis and stores the result in an S3 bucket. EMR was chosen for this task because EMR clusters can be dynamic orchestrated on-demand in comparison to a Hadoop AMI instance in EC2 that has to run continuously, thus increasing the cost.

Another advantage of the EMR in comparison to the Hadoop AMI on EC2 is that of the automated dynamic setup which is achieved by a Lambda function emr_lambda and that enables the termination of the cluster after the jobs are completed. The EMR cluster can access data on the S3 bucket trough Hive tables lowering the complexity of setting up the cluster in comparison to EC2 Hadoop AMI.

The second API Gateway is the trigger for the parallelVar Lambda function that splits and and distribute the number of Monte Carlo simulations to the the valueAtRisk Lambda functions to be calculated as shown in

. This facilitate parallel computation that can process large amount of data and reduce the overall time and cost of the process.

For storage S3 buckets are used to store the data of the system. This storage type is used because of the format that Hadoop on EMR produces the results.

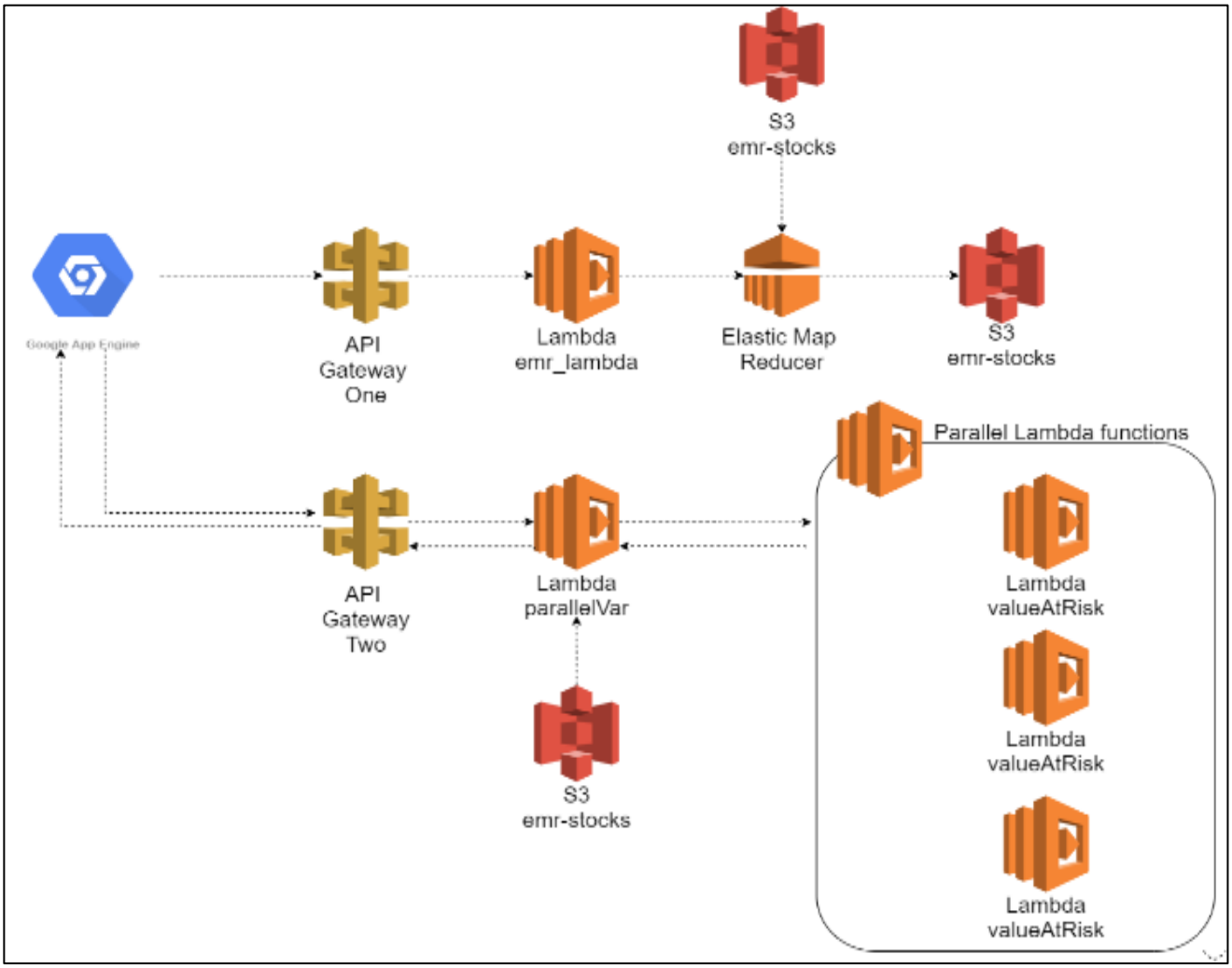


FIGURE 1: SYSTEM ARCHITECTURE

B. *System Component Interactions*

First, to determine the Buy/Sell Signals, data has to be communicated between the major components. As shown in

, the client GAE sends a HTTP request to the API Gateway One with the stock name and the moving average period to be computed. In turn the API Gateway One triggers and passes this information to the emr_lambda Lambda function that creates an EMR cluster. The cluster configuration parameters, the job configuration for Hadoop map reduce job that contain the bucket name and the location of the mapper, reducer, input file and output path in that bucket and the average period to be computed are transmitted to the EMR cluster. At the end of this job the output of the EMR is stored in the emr-stocks S3 bucket.

Second, in order to determine the VAR at each Buy/Sell signal data has to be communicated between the client-side and back-end. This communication is realised trough API Gateway Two, as presented in

, in which the API receives a HTTP request from GAE with the parameters stock name, number of stock units, Moving Average period, VAR period, number of Monte Carlo simulations and the number of Resources. API Gateway Two triggers the parallelVar Lambda function to which is redirected all the HTTP parameters received. In addition, the function retrieves the Signal List data produced previously by the EMR from the emr-stock S3 bucket. After retrieving the Stock List data, the parallelVar Lambda function invokes in parallel a Resource number of valueAtRisk Lambda functions and pass them the number of Monte Carlo simulations according to each function, the VAR period and the Buy/Sell signals data in order to return the VAR determined at each Buy/Sell signal to back to the parallelVar function.

After obtaining the results from all the valueAtRisk lambda functions, parallelVar function do the remaining steps and sends back to the client through the API Gateway Two the Moving Average data, the list of VAR values and the list of profit/loss for each signal.

III. IMPLEMENTATION

A. *Implementation*

The whole system was implemented using Python as a programming language including back-end and front-end. This programming language was chosen because in comparison to Java, Python has a faster cold start time, thus improving the overall costs of running the application.

In order to reduce costs for running an EMR cluster, I decided to make use of a Lambda function that automatically

creates an EMR cluster instance with a map reduce job by making use of boto3 client. As shown in Figure 2: emr_lambda Function, the job requires the paths from S3 bucket for all the required files. Furthermore, because TerminationProtection is set to False by default the cluster will terminate after the job flow is complete reducing the costs.

```
1. Steps = [{
2.     'Name': 'streaming step lambda',
3.     'ActionOnFailure': 'TERMINATE_CLUSTER',
4.     'HadoopJarStep': {
5.         'Jar': 'command-runner.jar',
6.         'Args': [
7.             'hadoop-streaming',
8.             '-files',
9.             '{},{},{}'.format(MAPPER_PATH,
                       REDUCER_PATH,  INPUT_PATH),

10.            '-mapper', MAPPER_FILE,
11.            '-reducer', REDUCER_FILE_ARGS,
12.            '-input', INPUT_PATH,
13.            '-output', OUTPUT_PATH
14.          ]
15.     }
16. }],
```

FIGURE 2: EMR_LAMBDA FUNCTION SNIPPET

In order to parallelise the Monte Carlo simulation, parallelVar Lambda function is using threads and queues.. Each thread is calling the valueAtRisk Lambda function, as shown in line 11 in FIGURE X, and waits for an answer and then merges all the returns. To invoke the other Lambda function boto3 client is used that invokes the Lambda function and then waits for an answer as json. All the results from the valueAtRisk function are averaged in order to be merged and send to the client-side as a json. All the data that is processed in these two functions is saved into an array that is processed and the results sent to the client-side.

```
1. class ThreadUrl(threading.Thread):
2.   def __init__(self, queue, task_id):
3.   threading.Thread.__init__(self)
4.   self.queue = queue
5.   self.task_id = task_id
6.   self.data = None
7.
8.   def run(self):
9.     try:
10.        queue_values = self.queue.get()
11.        var_list = invoke_lambda(queue_values)
12.        self.data = var_list
13.        self.queue.task_done()
14.    except Exception as e: print(e)
```

FIGURE 3: PARALLELVAR LAMBDA FUNCTION SNIPPET

One major challenge in this implementation was to create a Lambda function that creates an EMR cluster with a job in which are specified different parameters. Another challenge was to obtain the data from the S3 bucket, clean it and save it in a format that can pe processed by the Lambda functions. This was possible using boto3 client that retrieves the data from the S3 bucket.

The application is deployed at:
http://coursework-278313.ew.r.appspot.com

B. *Testing*

Unit testing for the Lambda functions was accomplished pytest library that helps the programmer to create tests. All the tests were conducted locally on the machine and not on the cloud to save on cost of running the testing functions.

IV. RESULTS

The following results are produced using BTC-GBP name, 100 Stock units, 101 as VAR period, Moving Average period is represented in the MA column in Table I, the number Monte Carlo simulations are represented in VAR column in Table I.

TABLE I. RESULTS FOR *BTC-GBP*

#	MA	VAR	Profit/Loss	95%	99%
1	10	10,000	498624.51	156048.40	156108.17
2	10	100,000	498624.51	156047.33	156110.36
3	10	1,000,000	498624.51	156048.51	15609.74
4	30	10,000	529972.05	181486.67	181550.16
5	30	100,000	529972.05	181507.93	181576.62
6	30	1,000,000	529972.05	181497.09	181553.41
7	50	10,000	568963.77	215908.68	215923.88
8	50	100,000	568963.77	215919.60	215967.52
9	50	1,000,000	568963.77	215951.12	216015.12

As the results in Table I shows, the only significant difference is in profit/loss is when the Moving Average period is different. This is because a higher Moving Average period, a higher number of closing prices for the stocks that gives a higher accuracy of prices for the buy/sell signals.

As presented in the Figure A: User Interface with user Input in the Appendix the results are obtained filling in the first form with the Stock Symbol and Moving Average period for the EMR to produce the Moving Average for the selected Stock Name. It is required a period of 10 minutes for the EMR to calculate the Moving Average for the selected stock name. After that period the VAR can be calculated using the second form in which all the required parameters need to be filled.

In Figure B: Results of User Input in the Appendix it is presented how the results are presented in a chart that shows the time series, the moving average, the position for each signal and a legend of it. On the right hand-side of the table are the controls for the interactive chart. In the table below are presented the date of the signal, the price at that time, profit/loss and VAR at 95% and 99% regarding the number of units at that date.

V. SATISFACTION OF REQUIREMENTS

Identify requirements that were MET, PARTIALY MET, or NOT MET (in column C, either M/P/N for each entry); briefly justify those MET or PARTIALLY MET, and describe what you could have done in order to meet those NOT MET.

TABLE II. REQUIREMENTS

#	*C*	***Description***
i.	M	Google App Engine, AWS Lambda and Elastic MapReduce (EMR) are used for this system.
ii.	M	The system offer a persistent front-end trough which data can be selected for the analysis.

#	*C*	***Description***
iii.	M	The front-end display all the required elements.
iv.	M	The front-end display all the required elements.
v.	M	All the VAR code is running on the AWS.
vi.	M	EMR is switched off after the completion of the jobs trough the code of emr_lambda function.
vii.	M	A form is provided for the user in the front-end to easily specify the values of the required parameters.
viii.	P	Two forms are provided for the analysis because the EMR takes over 6 minutes to determine the Buy/Sell signals and a request timeout will appear if GAE is waiting too long for a response.
ix.	M	The data is stored on a S3 bucket.
x.	M	The data is stored on a S3 bucket.

The overall application could be improved using AWS Step functions that can run workflows that join services for a better performance. Especially this can improve Requirement viii. and enable the user to run the analysis through a single click.

VI. COSTS

In order to produce the costs of running the scalable services for the snapshot of results, the calculations of cost for Lambda functions are done using the average from of 200 runs of the application. The overall price is calculated per 200 requests.

The Lambda functions have the following settings:

1. Emr_lambda has allocated 128mb of memory
2. parallelVar and valueAtRisk have allocated 256mb of memory

The prices for the US-East-1 for Lambda functions are as follows:

- 256mb : $0.0000004166 per 100ms [2]
- 128mb : $0.0000002083 per 100ms [2]
- Requests : $0.20 per 1M requests [2]
- Duration: $0.0000166667 for every GB-second [2]

TABLE III. COSTS

#	*Emr_lambda*	*parrallelVar*	*valueAtRisk*		*Overall*
	ms	*ms*	*No.*	*ms*	*$*
1	100	11200	10	1300	0.03
2	100	19500	20	4300	0.059
3	100	25300	100	4200	0.384
4	100	10700	10	1000	0.027
5	100	18900	20	4100	0.55
6	100	24600	100	3900	0.358
7	100	10900	10	1200	0.029
8	100	19200	20	4300	0.06
9	100	27400	100	4200	0.41

For the EMR Cluster the pricing is as follows:

- 1 Master node: m5.xlarge, $0.048 per Hour [3]
- 1 Core node: m5.xlarge, $0.048 per Hour [3]
- Average of 10 minutes of running x $0.048 per Hour

Total Price for each run of EMR cluster is $0.00768.

REFERENCES

[1] NIST, "The NIST Definition of Cloud Computing NIST Special Publication 800-145," [Online]. Available: http://csrc.nist.gov/publications/PubsSPs.html#800-145. [Accessed 18 05 2020].

[2] A. W. Services, "AWS Lambda Pricing," Amazon, [Online]. Available: https://aws.amazon.com/lambda/pricing/. [Accessed 16 05 2020].

[3] A. W. Services, "Amazon EMR pricing," [Online]. Available: https://aws.amazon.com/emr/pricing/. [Accessed 16 05 2020].

APPENDIX

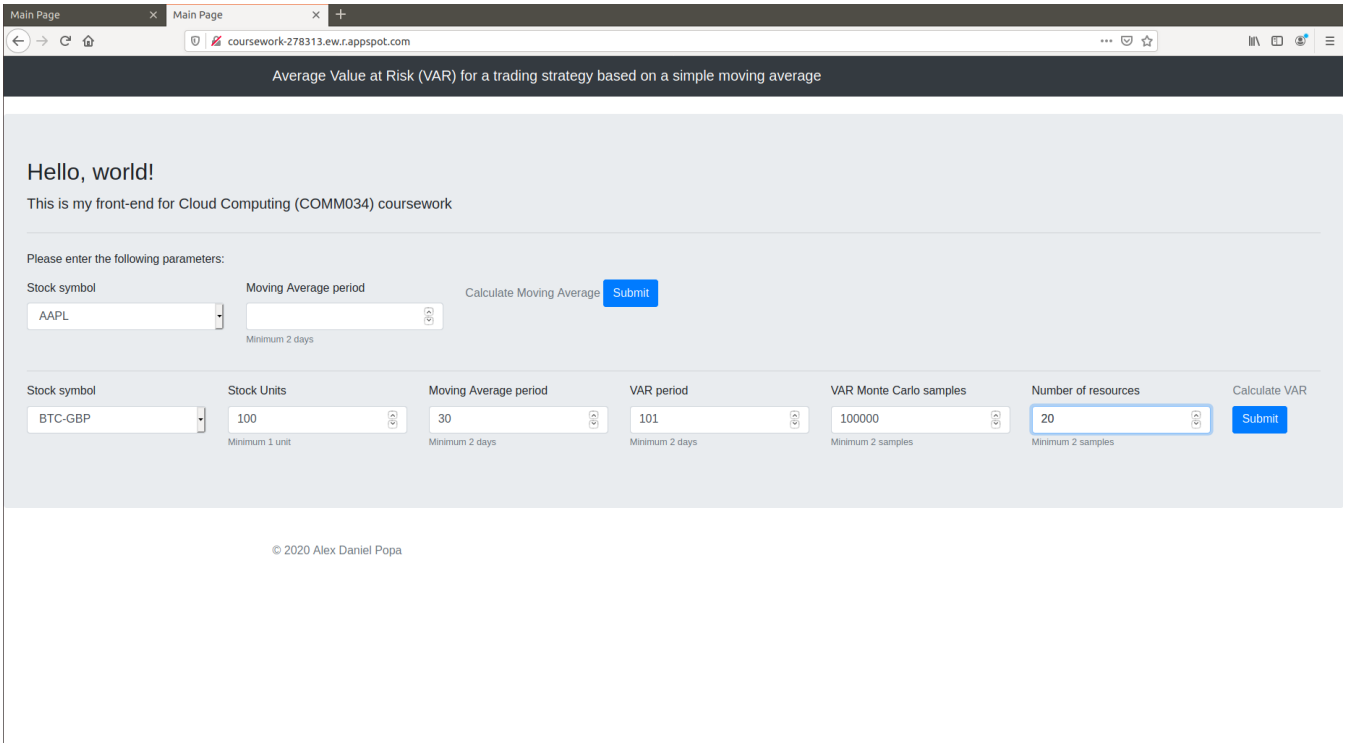
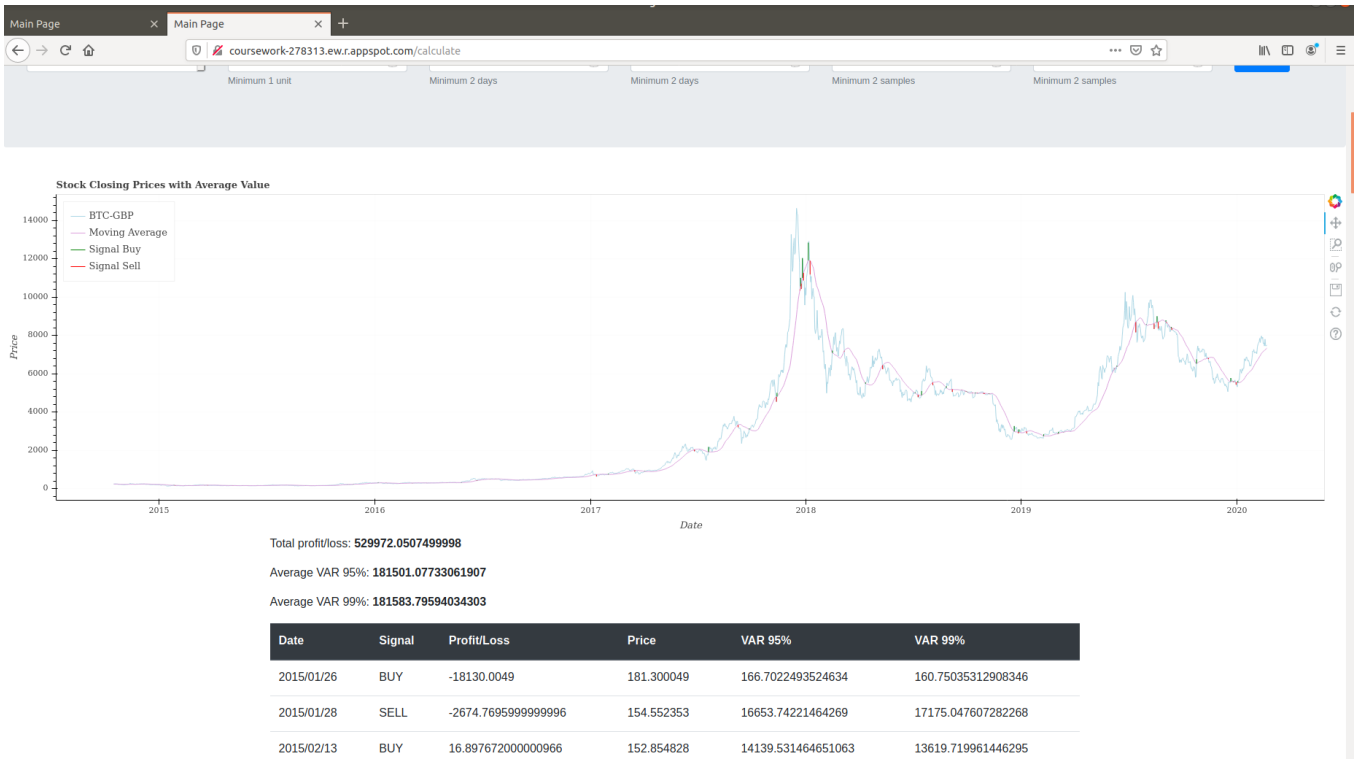


FIGURE A: USER INTERFACE WITH USER INPUT



Date	Signal	Profit/Loss	Price	VAR 95%	VAR 99%
2015/01/26	BUY	-18130.0049	181.300049	166.7022493524634	160.75035312908346
2015/01/28	SELL	-2674.7695999999996	154.552353	16653.74221464269	17175.047607282268
2015/02/13	BUY	16.897672000000966	152.854828	14139.531464651063	13619.719961446295

FIGURE B: RESULTS OF USER INPUT