Cloud based Web Application to determine risk of trading using Monto Carlo Method by analysing Trading Signals

Kowshik Kesavarapu
6678793 , kk01002@surrey.ac.uk
https://coursework-6678793.ew.r.appspot.com

***Abstract*— Main objective of this paper is to understand and develop a Cloud based web application to determine the risk associated with trading. The risk is calculated using Monto Carlo method by detecting the signals such as "Hammer", "Inverted Hammer", "Hanging Man", "Shooting Star". The web application is used to obtain the selection from the user and run the analysis with chosen parameters and resources. The result is shown as a graph and a table. All the analysis done are captured and stored for future audit as well.**

Keywords—Cloud, Web Application, Monto Carlo Analysis, Lambda , EC2 ,S3 , AWS , GCP

I. Introduction

This Application is built using several different Technologies , Frameworks and Services. At high level the main objective is to provide option between using two different services in AWS cloud (Lambda and EC2) to analyse the risk associated with trading. The main information that related to this analysis is provided by the user and the necessary calculation are done with the help of selected resource and provide an easy and informative output in the form of an interactive graph, table and few vital information.

A. *Developer*

From the perspective of developer, the main goal is to optimise the experience so that the wait time for the user is as short as possible. Here developer should consider all the possible ways the system may not work and figure out a way to handle them. From developer's perspective there are two main components Front End and Back End.

Front End is where the options are presented to the user to choose from which is mainly done using HTML.

Back End is where all the magic happens here the developer should capture values and with respect to those values Resources should be called in case of Lambda or Instances are created In case of EC2 and after performing all the required calculations the results are provided to the user.

B. *User*

Here User is the person who is accessing the web application through the public URL and requests the analysis. In ideal situation a user will visit the web application and performs all the analysis takes the data they need and perform the termination of instances if they used EC2. Or they can just simply use the audit function by clicking on History where all the past analysis is summarised.

II. Final Architecture

A. *Major System Components*

The main Components of this Application are AWS Lambda, AWS EC2, AWS S3 Bucket, Google Cloud Platform and AWS API Gate way.

1) *Aws Lambda*

AWS lambda is a service provided by AWS to run small snippets of code without worrying about underlying compute resource handling and scaling. It is also a classic example of serverless compute. Here the scaling, Managing and updating of the underlying resources are handled by the AWS itself. Its very Optimal to run short bursts of code execution. Aws Lambda is charged by how many times its triggered, how long it's running and how much data it used and transmitted out of AWS system. If you have a lambda function sitting in AWS until and unless it is triggered it won't be billed. In respect to this application the Lambda is used to calculate the risk values and report them back.

2) *AWS EC2*

Amazon Elastic Compute Cloud (ECC ~ EC2) is a service which lets the user to rent a server from amazon and use it to host web pages, videos or whatever the developer likes. This is the classic example of Infrastructure as Service. This lets the Developer to choose the exact specification required to run their application without the need of going out and buying the physical hardware. It offers a wide range of collection of different resource to select from and recently added ARM based compute which is becoming more and more sought after. Here billing is done with the respect to the time the instance (Virtual Server) is up and running and the amount of data is transferred. In respect to this application EC2 is essentially used as a server and all the processing is done within the EC2 instance and only the output is reported back.

3) *AWS S3*

Amazon Simple Storage Service (SSS ~ S3) is basically a storage as service. It's used to store all sorts of information like logs, Results or in case of Netflix the whole video catalog is stored on S3 buckets. In respect to this project the S3 Bucket is used to store the history of all the runs and store them for the purpose of audit

4) *Google Cloud platform*

Google Cloud Platform (GCP) is essentially a competitor of AWS but it's more focus is on Web Based applications and Machine Learning kind of tasks. In respect to this application GCP is used to host the Webpage and to do some basic calculations and string manipulation.

5) AWS API gateway

AWS API gateway provides an easy and quick way to interact with several components in AWS like Lambda. In this application API gateway is used to send requests to lambda function and to store and retrieve data from the S3 Bucket.

B. System Component Interactions

The System Interaction starts with the user Requesting the analysis through web page located on GCP then the information is captured and with respect to the selection the flow of the interaction is determined. If lambda is selected there is a quite a lot that happens in GCP. Initially the data is downloaded using YFinance and the mean and standard deviation is calculated, and the Risk values are then calculated in the Lambda using API gateway triggers . Once all the calculation is done the data is then displayed as a chart and table. If EC2 is selected, then the requested number of resources are created, and the request is sent to run the code present in the EC2 instance, and the results are reported back. Then the data is displayed as a chart and table. Once the data is reported back from the either Lambda or EC2 the same is stored along with some more information to s3 bucket. This is done by sending data to a lambda function using API gateway trigger and the lambda then stores the data into the S3 bucket. For retrieval of the data from S3 bucket another API gateway trigger is used to invoke a lambda function which then retrieves the data from S3 bucket and sends back to be presented to user.

Several different Technologies , Frameworks and Languages are used in order to achieve this interaction . The majority of which is done using Python and flask framework. Other major things used are Jinja2 which is a web template engine for python-based web applications , HTML and CSS for designing web pages, Java script to implement graph using Google Charts API and finally json to send requests and capture data.

Official Documentation[1] of respective frame works, and packages are consulted extensively to build this web application. I started by using lab 1 code where it was demonstrated how to capture values from html forms and do operations on them as a base line and started adding more and more functionality. First two more pages were added in templates one for capturing data like Shots , Minimum History and Signal(Buy/sell) to be used and other template to show historical logs for audit. In the initial pages the user was given choice to select what resource to select (Lambda /EC2) and how many parallel resources of them to be used. Then they are redirected to page that asks for the remaining information . if the lambda is chosen the analysis function in python triggers a function which gets data using YFinance package then calculates mean and std with respect to the min history and signal selected then these values along with shots is sent to a lambda function using API gateway to get the values for 95 and 99% risk values. In lambda the parallelisation is also implemented used python inbuilt thread pool executor to send each request as thread and then once all the values are received, they are splitted into two lists using string manipulation for each thread and then an average is calculated for each day by adding all the responses and dividing with number of parallel resources used. Now these two lists are used to calculate the overall average and then the data is stored in S3 bucket by calling another lambda function which consists of Boto3 call to add all the information as a json object inside a S3 bucket .

Now for EC2 , Ubuntu based instance is used to create an Apache server and modified its configuration file to use CGI (Common Gateway Interface) by following the same steps demonstrated in lab 5.Then a Python code is developed which uses the “QUERY_STRING” Key from the OS environmental parameters to capture three values that are passed in it (Signal ,Shots and Min history)to calculate the 95 and 99 Risk values. All the required packages are installed, and an AMI is created which has all the packages installed and contains the code. Now when user requests for 4 resources Boto3 is used to create 4 instances using this AMI. This does add some cost as storing an AMI cost’s around 0.002$ per month but this allows the service to run each time without any error and decreases the chances of encountering a problem .Then the parameters are passed using a delimiter as “&” and appended to the public DNS followed by my python code (ec2_calc.py). To capture the data HTTP connection is established and “GET” request is used to receive response which is then cleaned by handling some junk values and also all the responses generated by different instances are averaged just like it was done in Lambda part.

Once Responses are received from the either service and all the averaging among multiple resources and average within the list is done the data is then packed into two tuples one for displaying chart and other for constructing the table. For the Chart Google Image API[2] is used and the data table for the chart is constructed by using Jinja and Jinja is also used to create a dynamic table which adapts to the length of the tuple and create a table with dates , 95 and 99 Percentages of the date.

For the storing an S3 bucket is created where all the values are stored as a json file object . Here I am capturing 8 values, Service selected, Number of resources of said service used, Shots, Signal, Minimum days, Average of 95%, Average of 99% and time it took to process the request. These are then passed to a lambda function which in turn stores them into a S3 bucket as a json object.

For the History page another Lambda Function is created which reads all the objects from the bucket and send them back. After which the output is cleaned using some string manipulation, and a tuple is created which is passed to the html page in which jinja is used to construct the table.

In total there are 7 Web pages,1 CSS file , 3 Lambda function , 3 API gate way triggers. 1 index.py file where all the python code is done and finally ec2_run.py which is present in the ec2 instances are the main components of this application.

I did face an issue where when the application is deployed to GCP the requests are getting terminated without waiting for the response this took a lot of time to research and find the way to change this behaviour. The solution is found in Unicorn Documentation[3] which is adding a timeout tag so that the request doesn’t timeout. This did create some entries into audit table with 0.0 in 95% and 99% risk values . These entries can be ignored.

The reason EC2 is chosen over other scalable services like EMR and ECS is EC2 is less complex to implement compared to others. EMR also works better when combined working on large data sets and tasks such as data modelling. It does

Have benefits overs EC2 as EMR can be scaled much efficiently compared to EC2. ECS is at a high level is EC2 with support to docker and here also the main advantage is the scalability. For these reasons I have Chosen EC2 over EMR and ECS

III. Satisfaction of Requirements

#	*C*	*Description*	*Code used from elsewhere, and how used*	*Code you needed to add to what you used from elsewhere*
i.	M	Used GCP for the front End and Lambda and EC2 are used to calculate the risk	Lab codes from Week 1 to Week 5 are used as a base line and achieved this	
ii.	M	User can go through the front end can create instances and terminate instances. History is also implemented.		
iii.	M	All the Risk calculations are done using scalable services Google App engine is only used to calculate aggregate and average values		
iv.a	P	In initialization user can specify the selection between Lambda and EC2 but the warmup is not achieved. The Warmup is done at the later stage. The time values are also captured. i , ii , iv are met and iii is not met	Lab codes from lab1 is used as a baseline template and achieved this.	
iv.b	M	For the risk analysis user can select the values for Min history, Shots and signal and the results are displayed and stored for audit.	Lab codes from Lab 3, 4, 5 used for this purpose and added upon them to add functionality. Boto Documentation is referred extensively to create instances and also to store data to s3 bucket	
iv.c	M	Output with chart , and table is produced. Audit page is also produced	Used Google charts API to get a sample code for chart and then used jinja to create data table. For table Used jinja to create a dynamic table.	
iv.d	N	I was not able to reset the system to provide a way to make more analysis		
iv.e	M	Used Boto 3 to terminate all the running instances	Used Boto 3 Documentation[4] to understand how to terminate the instances	

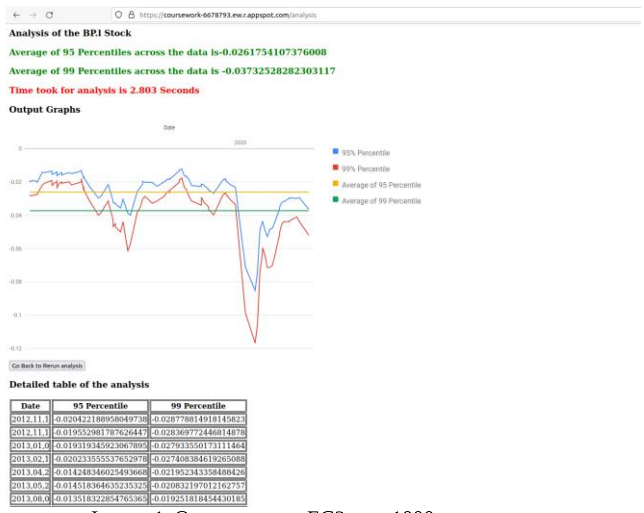
TABLE I. Satisfaction of Requirements and Code use/Creation

IV. Results

Here we can see the EC2 takes significantly less time to produce output for same parameters than lambda. This is because lambda is not designed to perform these kinds of tasks. It's most common use cases are file processing and website authentication .Whereas EC2 is essentially a server designed to do these kinds of tasks.

TABLE II. Results

Parameters Selected	*Avg 95%*	*Avg 99%*	*Time Taken*
S-Lambda R -2 H -100 D – 100000 T – buy	-0.026187549697484745	-0.03698912128459559	143.317
S-Lambda R -2 H -100 D – 10000 T – buy	-0.026232439717570416	-0.037021779542412675	29.04
S-EC2 R -2 H -100 D – 10000 T – buy	-0.026147773283133485	-0.036993814465382045	4.637
S – EC2 R -2 H -100 D – 100000 T - buy	-0.026190473073132048	-0.037002016664783 29	29.187
S – EC2 R -5 H -100 D – 10000 T – buy	-0.02614941289781443	-0.03695398067570761	12.074
S – EC2 R -5 H -100 D – 1000 T - Sell	-0.026145816928711253	-0.03735748663563436	7.026


https://coursework-6678793.ew.r.appspot.com/analysis
Date
2020
0
-0.02
-0.04
-0.06
-0.08
-0.1
-0.12
🟦 95% Percentile
🟥 99% Percentile
🟨 Average of 95 Percentile
🟩 Average of 99 Percentile
Go Back to Rerun analysis

Analysis of the BP.l Stock

Average of 95 Percentiles across the data is-0.0261754107376008

Average of 99 Percentiles across the data is -0.03732528282303117

Time took for analysis is 2.803 Seconds

Output Graphs

Detailed table of the analysis

Date	95 Percentile	99 Percentile
2012,11,1	-0.020422188958049738	-0.028778814918145823
2012,11,1	-0.019552981787626447	-0.028369772446814878
2013,01,0	-0.019319345923067895	-0.027933550173111464
2013,02,1	-0.020233555537652978	-0.027408384619265088
2013,04,2	-0.014248346025493668	-0.021952343358488426
2013,05,2	-0.014518364635235325	-0.020832197012162757
2013,08,0	-0.013518322854765365	-0.019251818454430185

IMAGE 1. OUTPUT USING EC2 AND 1000 SHOTS

V. COSTS

For calculating the cost amazon cost calculator[5] is used .

For Lambda it would cost around ~0.01 per month while factoring in one request an hour with 2 parallel resources which would be around 550 requests per hour after factoring in some extra requests for audit and storage it comes up to 750 requests per hour which costs 0.01 (511,000 requests x 0.0000002 USD = 0.10 USD)

For EC2 we need to add wait time as well in calculation t2.micro is charged at a rate of 0.0072 USD per hour .Considering 2 hours of use a day for 4 instances it will cost around 2.9 USD (4 instances x 0.0116 USD x 61 hours in a month = 2.83 USD)

For S3 it won't cost anything until and unless the amount of history stored reaches to a crazy amount like a million objects because each entry only takes 100bytes of storage.

REFERENCES

[1] https://flask.palletsprojects.com/en/2.1.x/tutorial/layout/

[2] https://developers.google.com/chart/interactive/docs/datesandtime

[3] https://docs.gunicorn.org/en/stable/settings.html#timeout.

[4] https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html#EC2.Client.terminate_instances

[5] https://calculator.aws/#/