Application to determine risks of using signals as a trading strategy using Monte-Carlo method

Venkatesh Nagasubramanian : 6721597, vn00197@surrey.ac.uk

https://comm034-coursework-386115.nw.r.appspot.com

***Abstract*—A web application is built using cloud services like Google Cloud Platform (GCP) and Amazon Web Services (AWS) to determine risks of using Three White Soldiers and Three Black Crows method as a trading strategy. The stock analysed here is BP p.l.c. (LSE: BP. L). This paper provides a detailed overview of the architecture, the requirements, and the cost of the all the services used in building the application. The results have also been discussed.**

***Keywords*—GCP, AWS, lambda, EC2, Three white soldiers, three black crows, S3 bucket**

I. Introduction

The developed website uses the stock data of BP.L to check the risks of using three white crows and three black soldiers as a trading strategy. This website is developed by closely following the standards defined by the National Institute of Standards and Technology for Cloud Computing [1]. The essential characteristics of the application are as follows:

On-demand self-service: The user and the developer can provision the cloud resources to scale up or down as required. This is done automatically, and no human intervention is required. However, the credentials for the AWS CLI need to be changed every time for running the application without errors – this is a limitation of the AWS Academy.

Broad network access: The application is always accessible for anyone with the link, and works on every browser (which supports HTML5) irrespective of the location the user is in. (works faster if accessed near London, as it is deployed in europe-west2 server in the Google Cloud Platform)

Resource pooling: The application can be used by multiple tenants and the resources are dynamically assigned according to the users' demand. AWS Lambda is scaled automatically according to the demand and EC2 resources are specified by the user themselves according to the requirements. The region of the resources, however, cannot be specified by the user, it is set in us-east-1 (N. Virginia).

Rapid elasticity: Google App Engine provides automatic scaling for applications that are hosted using this service. For a higher number of requests in the application, resources are allocated automatically.

Measured service: The usage of resources like Lambda and EC2 are metered, and costs are calculated according to the documentation from AWS. The runtime of the resources are optimised to run parallelly using multiple threads, so that the time is preserved. These costs are displayed to the user to evaluate their expenses.

Given below is a detailed explanation of what the developers and the users of this website will experience.

A. Developer

The application has made use of two different providers of cloud infrastructure – Google Cloud Platform (GCP) and Amazon Web services. These entities are bound together by creating a Flask application, utilizing Python 3 as the main programming language. Hence, this is a *hybrid cloud* deployment model.

GCP provides the service App Engine which is used to deploy the model on the cloud server for public use. The backend of the application is made through Python 3 and Flask. The frontend of the application is built using HTML and CSS, with a touch of JavaScript for programming only the back button.

The interface provides a way for the user to provide inputs to calculate the risk values. AWS services like Lambda and EC2 are used here to calculate the risk values. The EC2 resource used here is Ubuntu Linux. The costs of using these resources are calculated and sent back to the user. The costs of using these resources have been minimised by using threads to create a parallel run for the number of resources that the user mentioned to be cost-effective to the user.

The summary of the parameters and the results are stored in an S3 bucket. It is stored and retrieved for the user to estimate the costs for different parameters.

B. User

The application is a *Software as a Service (SaaS)* model. The user can use the application to determine the risks of using Three White Soldiers and three black crows as a trading strategy. This is accessible to anyone with the link on a browser. The user does not manage the resource configurations, but only the number of resources that they would want to make these calculations.

The user will have a provision to select the service and the number of resources to initialize or warm up instances of EC2 (invocations for Lambda). After the user warms up the resources, the user can input the parameters to calculate the risk values for each signal. Users can enter the minimum history parameter, which takes the number of days to simulate the risk values, the number of shots to perform the Monte-Carlo, select to check the Buy or Sell signal, and the number of days after which to check profit or loss. Clicking 'calculate' will send the parameters to the data to calculate the risk values and render the results page.

The results page has three parts: an 'Audit' table with all the parameters that the user had entered for the analysis with the net profit (the loss is represented as a negative number) and the cost undertaken for the analysis; a results table with the 95% and 99% risk values for each signal and the date of each signal; and a line graph depicting the risk values for each signal and average of these values as a dotted line.

A separate audit page is provided as a hyperlink on the top which redirects to a page with a table containing all the parameters, averaged risk values, cost of the analysis, and the net profit (or loss if negative).

The user can reset the analysis, by clicking on a reset button, to provide a way to run the analysis once again, but without having to initialise more resources. There is also a terminate button, to terminate all resources and make sure there aren't any further costs.

II. Final Architecture

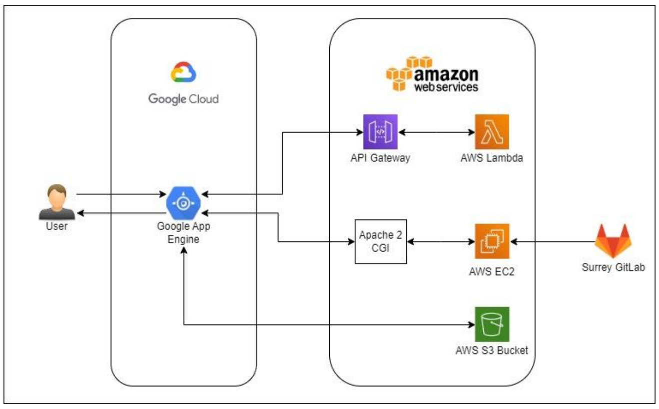

Fig. 1. System components and architecture

I. *Major System Components*

There are two main parts of this application – GCP and AWS. They are bound to work together by Flask framework in Python 3

Google Cloud Platform: The application is deployed into the cloud to be used for public access using the *Google App Engine*. It provides automatic scaling, which will provide computational resources according to the number of requests sent to the website. The server this website has been hosted is *europe-west2*, located in London.

Amazon web services: Another major provider of cloud infrastructure, three services are used for the development of the application – AWS Lambda, Elastic Compute Cloud (EC2) and Simple Storage Service (S3). The region of all these services is set to *us-east-1* (N. Virginia).

AWS Lambda – Lambda is a serverless platform that can run code when invoked. Per the number of requests and the time taken to run, it scales up or down. This is used to calculate the 95% and 99% risk values for the parameters specified by the user.

AWS EC2 – EC2 provides elastic compute capacity, directly in the cloud to run applications. The machine used in this application is a t2.micro system, running in Ubuntu Linux, with 1GB of memory with EBS storage only. Similar to the above, the EC2 also is used for calculating the 95% and the 99% risk values.

AWS S3 – S3, like the name suggests, is a simple storage service which is used to store values. The bucket contains one empty json file, which is a dictionary of lists. These lists are appended and overwritten with every execution, so they can be retrieved and rendered for the audit page.

A brief representation of the architecture is given in Fig.1.

B. *System Component Interactions*

When a user 'visits' the website, they are prompted to enter the values of the scalable service and the number of resources for the selected service. After the user initialises the services, the dataset is created by the Google App Engine.

The columns 'Date', 'Close', 'Buy' and 'Sell' are converted into lists and added to a dictionary (string) with the values of 'H', 'D', 'T' and 'P' to send to the selected service for analysis.

AWS Lambda: The lambda function takes this 'string dictionary' as event through the API Gateway. Lambda has the code to undertake the calculations and returns the risk values, which are averaged and sent back to the GAE to render the website. To send and receive values from Lambda, a POST request is sent to the url of the function.

AWS EC2: This has the same functionality as the lambda, but the method is different. The EC2 is initiated to install packages like python and apache2. The configuration file for apache2 is overwritten to create the LAP server (Linux, Apache2 and Python) in the /var/www/html directory. The python file to complete the calculation is also kept in this directory. Both these files are kept in a GitLab repository so that they can be fetched faster. Similar to the above, a POST request is sent to the url of each resource to send and receive the values in the GAE.

AWS S3: Before every calculation is rendered on the results page, the values of S, R, H, D, T, P are sent to a json file in the S3 bucket. The values that previously existed in this file will be fetched, and the new values will be appended to it, and the file will be updated in the bucket. When the user clicks on the 'Audit page' this data will be fetched from the bucket to the GAE to display on the website.

Google App Engine: This acts as the middleman for the user and the AWS. GAE takes input from the user and sends and receives data accordingly to and from the services mentioned above. However, GAE does some very important calculations. It averages all the risk values which are received from the services for each resource. It also calculates the Profit/Loss for each signal.

All these services work together effectively to provide the best experience for the user and help judge the performance and the risks involved while using Three White Soldiers/ Three Black Crows as a method to trade and predict the stock prices.

III. Satisfaction of Requirements

Table I provides the summary of all the requirements. Requirements that are fully met are denoted by M, partially met are denoted by P, and the requirements not met are denoted by N. Some code was referenced and modified from other sources, and the appropriate reference to the source, and the code added to it are given in the table below.

TABLE I. SATISFACTION OF REQUIREMENTS AND CODE USE/CREATION

#	*C*	*Description*	*Code used from elsewhere, and how used*	*Code you needed to add to what you used from elsewhere*
i.	M	Use GAE, AWS Lambda and another scalable service in AWS. In this application, that scalable service used is EC2.	The code for index.py, app.yaml files were taken from Lab 1 of the course. Steps to use AWS Lambda was taken from Lab3, and the steps to initialize EC2 instances were referenced from Lab 4 and Lab 5.	The index.py file was modified to have the code to fetch and initialize the BP.L dataset. Additional code for passing values in lambda and EC2 were added accordingly. The app.yaml file was modified to have a higher timeout.
ii.	M	System will offer a page to warm up and terminate all resources. However, this wouldn't work properly due to a limitation in AWS Academy as stated in Lab 4.	The code for creating a form was taken from Lab1. The code to initialise EC2 resources was taken form Lab 4. The code that will terminate the ec2 instances is taken from boto3 documentation. Additionally, the css file from Lab1 was used to develop the user interface.	The form was modified to take the S and R values to initialise R number of resources of the service S. In addition to initialising EC2 instances, a hyperlink at the top is included to terminate the resources. The code was modified from the documentation and customized for the application. The css file was altered for a better design language.
iii.	M	Scalable services from AWS will calculate the risk values, and the GAE will average them to show results. It was not required to send both the buy and sell signals but could have sent just one of them to the services depending on the T value. This would have decreased time consumption.	The code to send the values to AWS Lambda was referenced from Lab 3. To initialise and send the .py file to EC2, code from Lab4 was used.	Code was modified from Lab 3 to include the values that the user had passed to the application. For EC2, code was created to input and calculate the risk values, and the code was modified to send the .py file to the instance via gitlab instead, to make it faster.
iv.a	P	The application will enable the user to specify S (scalable service, either Lambda or EC2), and R (the number of resources). Submitting or clicking the initialise button will 'warm-up' the resources. Also, the code will capture the time required for warmup. The time, however, will be added to the execution time so it could be calculated for the cost.	Lab 1 code for the form has been referenced to make the form which contains this feature. The code from Lab 3 was taken to create the connection for AWS Lambda. Code from Lab 4 is used to initialise the resources for EC2. The code to create the apache2 server was taken from Lab 5.	The form has been modified so it can ask the user to select from either using Lambda or EC2. The link to connect to Lambda was changed accordingly. For EC2, the bash script was directly given in the 'UserData' parameter instead of fetching the file from GAE to make it faster. The .py files were hosted in Surrey gitlab instead of GAE to make it faster. Time calculations were also added.
iv.b	M	The application will give the user a provision to specify H (price history), D (data points), T (Buy or Sell signal) and P (the number of days after which to check profit) and calculates the risk values across R. The values are then averaged and rendered on the results page.	Lab 1 code provided the code for the form to get all the values. Also, Lab 1 contained the code for rendering the output on the results page. The code from Lab 3 was used to implement multithreading in Lambda and EC2. The method to retrieve the data from the LAMP server in EC2 is used from Lab 5.	The codes were modified to have separate logic to execute for Lambda and EC2 services. The method to send the inputs to the EC2 instances is another variation[2] to what was done to give inputs to lambda in Lab 3. A LAP server was created, because MySQL was not required. The data was retrieved in a json format from the EC2, rather than a html output.
iv.c	M	The results page will show three figures – the inputs of the user with the time and the cost of the execution, a chart of the risk values and averages, and a table with the risk values and profit/loss values for each signal. Clicking the Audit hyperlink, a page opens with the history of all analysis. The page will show all analysis values by all users till date, and not just for the user performing the analysis.	The code for the charts is referenced from the image-charts documentation. The code to show the values on the results page and the audit page was taken from Lab 1. The code to store and retrieve values form S3 bucket was referenced from the documentation of boto3.	The image-charts api documentation was referred for syntax but customised to create the charts for the risk and the average values. For using the S3 bucket, the code was modified to read, append values and then overwrite the file so it can be fetched and rendered to the audit page.
iv.d	M	The system provides a way to reset the analysis and get back to giving different values, without requiring to 'warm up' new resources.	Code to create a reset button was taken from w3schoools website. Also, the code to go back on the browser was taken from the w3schools[3] website.	Buttons for the reset and go back were designed and placed accordingly.

#	*C*	*Description*	*Code used from elsewhere, and how used*	*Code you needed to add to what you used from elsewhere*
		There is no one single button for this, but a workaround where the user must click 'Go Back' for the results page, and then click reset on the form to reset the values on the form.		
iv.e	M	The system has one hyperlink which upon clicking will terminate all instances of EC2.	The code that will terminate the EC2 instances is taken from boto3 documentation.	The code was modified from the documentation and customized for the application to enable the termination. The application will redirect to the main page, and then display the number of resources that are terminated.

IV. Results

Table II shows the summary of results from the system. Three tests are conducted for each service, each having one or three resources, with 10000 or 20000 shots. The profit/loss values are not provided in the table because of the lack of space. All the values are rounded to 5 decimal places. Time is rounded to the nearest 0.01 seconds. But please note that the time taken to warm up the resource is added to the time of the calculations which gives a very large time for the user.

TABLE II. Results

S	R	H	D	T	Average 95%	Average 99%	Cost	Time
EC2	1	100	10000	buy	-0.03546	-0.05071	$0.0 2654	118. 85
EC2	1	100	20000	sell	-0.03558	-0.05081	$0.0 2852	127. 72
EC2	3	100	10000	sell	-0.03548	-0.05070	$0.0 7694	114. 83
lambda	1	100	10000	buy	-0.03549	-0.05091	$0.0 0018	9.02
lambda	1	100	20000	sell	-0.03563	-0.05075	$0.0 0083	13.2 8
lambda	3	100	10000	sell	-0.03550	-0.05077	$0.0 0056	9.10

From the table, it is understood that for higher resources and higher number of shots, the time and the costs have been higher. Figure 2 shows chart for one run of the system.

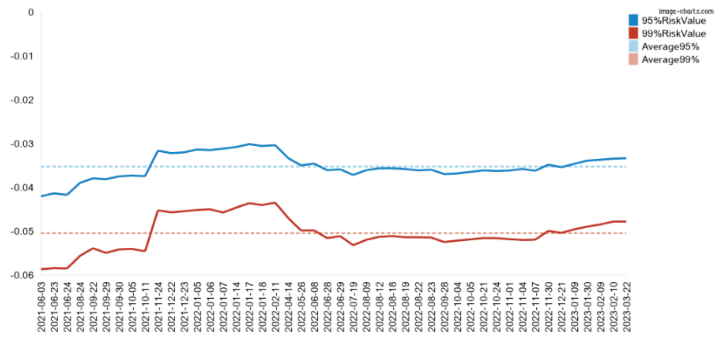

Fig. 2 Risk values and the average risk values against each signal

V. Costs

There are two kinds of costs for this website. One cost is for the user and the services they use to calculate risks, and the other, is the operational costs. These costs are based on the official documentation of the service providers. All the costs are explained below.

A. Service Costs

Elastic Compute Cloud (EC2).

The cost of one instance of t2.micro, Ubuntu Linux, for a EBS storage only, and 1 GB of memory, the cost per hour is $0.0134. The total cost of using R resources per T seconds is:

$$\mathbf{\$0.0134 * R * T / 3600}$$

AWS Lambda

The cost of one request of Lambda for the first 6 billion GB-seconds of the month, is $0.0000166667 for every GB-second. The total cost of R requests for 128MB of memory for T seconds is:

$$\mathbf{R * 128 / 1024 * T * \$0.0000166667}$$

B. Operational Costs

These costs are hidden from the user.

AWS Simple Storage Service (S3)

The file in the S3 bucket takes up 5 KB of storage. For the first 50 TB per month the storage charges is $0.023 per GB. The charge of one get and one put request is around $0.0000054. The total cost for 5KB of storage per month and with one get and post request is:

$0.000009536 * 5 + $0.0000054 = $0.00005308 per month

As we can see, this cost is very negligible.

Google App Engine

For GAE, the billing is dynamic, and billed based on the instance hours (**$0.06 per hour**) and the outgoing bandwidth (**$0.12 per GB**), according to the billing section in the App Engine Dashboard. More research is required on the actual costs of App Engine.

References

[1] Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD. [online] Available at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf [Accessed 23 May 2023].

[2] "Python Requests post Method." W3Schools. Refsnes Data, 2023. https://www.w3schools.com/python/ref_requests_post.asp. Accessed 23 May 2023.

[3] "History.back() Method." W3Schools. W3Schools, 2023. https://www.w3schools.com/jsref/met_his_back.asp. Accessed 23 May 2023.