Financial Stock Trading Analysis Using Monte Carlo Simulations Across Different Cloud Services

Anthony Awobasivwe : 6442385, aa02350@surrey.ac.uk

https://comm034-coursework-381720.nw.r.appspot.com

In the current climate of financial stock trading, making data-driven decisions is of high importance. With the continuing rise of cloud computing, financial traders now have access to powerful computational resources which can be used to run various kinds of analysis on stock data using methods such as monte carlo simulations. This project proposes a cloud-based application built to run Monte Carlo simulations on stock data gotten from Zoom Video Communications Inc, utilizing trading signals based on the three white soldiers and three black crow's approach. These signals are then used to run a high number of simulations through scalable services provided by amazon web services**.** The application is deployed and hosted on google cloud.

Lambda , EC2 , S3 , GAE , PaaS , SaaS , IaaS , API

I. Introduction

The system is made up of a total of 4 services and is designed to follow the standards mentioned in NIST Special Publication 800-145. Firstly, we consider the viewpoints of both the developer and user with respect to the system and the aforementioned standards. Considering both perspectives ensure system compatibility and user satisfaction alongside adherence to standards.

A. *Developer*

GAE and Lambda are both characterised under PaaS. In the case of GAE, it allows the developer to deploy and manage the application without worrying about the underlying infrastructure. Lambda, is a serverless computing environment that allows the developer to manage the code via the AWS lambda console where the developer can make changes such as the environment run type to be worked in and memory storage allocated as well as change, execute and test function code. EC2 is IaaS , in the context of this project this means the developer is responsible for choosing what operating system to use and installing all relevant libraries/software needed which can then be preconfigured on an AMI . S3 can be considered as IaaS however in the context of this application it is more closely associated with PaaS given it is being used for simple storage without the developer needing to manage any run time environments, operating system, etc. The developer has access to IAM roles meaning communications between GAE and EC2 / S3 can be made via lambda functions hence negating the need for credentials .

B. *User*

The user has certain limitations when interacting with the system which is consistent with a SaaS experience. When working with lambda the user doesn't have control over specifications such as the memory allocated to each function, or service time-out settings meaning the user doesn't experience an on-demand self-service. Restrictions are also present with EC2 with the user not allowed to start up any other instance types barring t2.micro or have control over the service location which also applies to GAE. The system is broadly network accessible as users can run the simulations using curl requests or through platforms such as Postman on various devices such as their phones, pcs, and laptops provided they have the URL for the Google app engine address and subsequent API endpoints. This also means the user is not provisioned with a front-end user interface. The user can provision a maximum of 10 resources due to restrictions present in AWS education . For provisioned resources unlike lambda which automatically scales no load balancing is implemented for ec2 meaning the ec2 system doesn't scale up or down and hence isn't rapidly elastic. The user can ascertain the costs incurred from running simulations and warming up of resources across both services which showcases the system's partial capability of being a measured service. Partial meaning the current system doesn't show users the cost incurred for GAE or storage via S3. The current system does not support resource pooling as it is not capable of serving two users at the same time.

II. Final Architecture

The system is built using Flask and uses gunicorn as its web server gateway interface . The system comprises a total of 15 endpoints which cater to different functionalities and interactions within the system. Of these endpoints , two are designated for POST methods. The post methods are responsible for sending json payloads for warming up or running analysis via google app engine to the corresponding service URL's . Upon system start-up/warmup the system sends a payload via GAE containing the signal data to an s3 bucket for later use via a lambda helper function. There are a total of 6 helper lambda functions that act as intermediaries for communications between GAE and AWS EC2 by performing functionalities such as checking if any ec2 instances are running, terminating instances all of which is done via get request calls to the required lambda function with the subsequent response then sent back to GAE. Out of the 15 endpoints, 5 of them interact directly with these Lambda functions. The remaining 10 endpoints function exclusively through GAE, allowing the system to establish a flexible and efficient network of communication channels. The system makes use of session cookies in order to store data persistently . By leveraging the power of Flask, combined with Google App Engine and AWS the system exhibits a cohesive and responsive interaction model, able to cater to a variety of tasks and processes. However, the current system is quite fragile given some endpoints make the assumptions other endpoints have been run before running them. From figure 1 we see a high-level overview of the system with different interactions taking place across the 4 services in use .

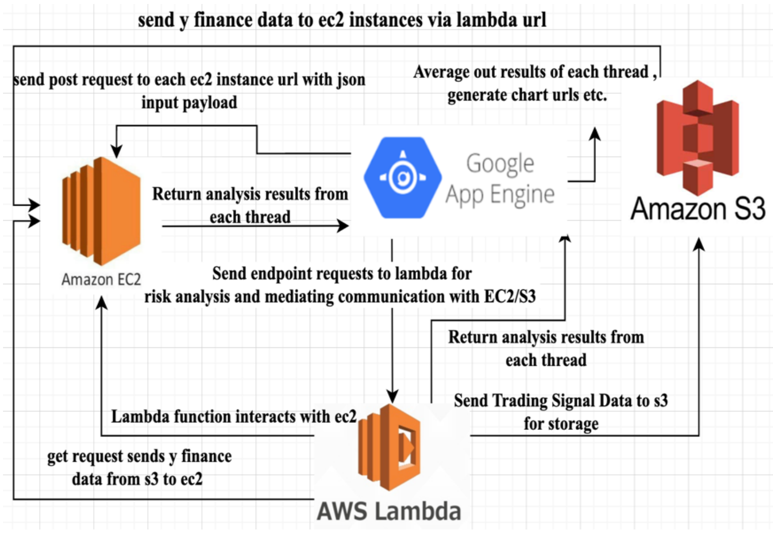

send y finance data to ec2 instances via lambda url
send post request to each ec2 instance url with json input payload
Average out results of each thread , generate chart urls etc.
Google App Engine
Amazon S3
Return analysis results from each thread
Amazon EC2
Send endpoint requests to lambda for risk analysis and mediating communication with EC2/S3
Return analysis results from each thread
Send Trading Signal Data to s3 for storage
Lambda function interacts with ec2
get request sends y finance data from s3 to ec2
AWS Lambda

Figure 1. System Interactions for multi cloud application

III. Satisfaction of Requirements

TABLE I. SATISFACTION OF REQUIREMENTS AND CODE USE/CREATION

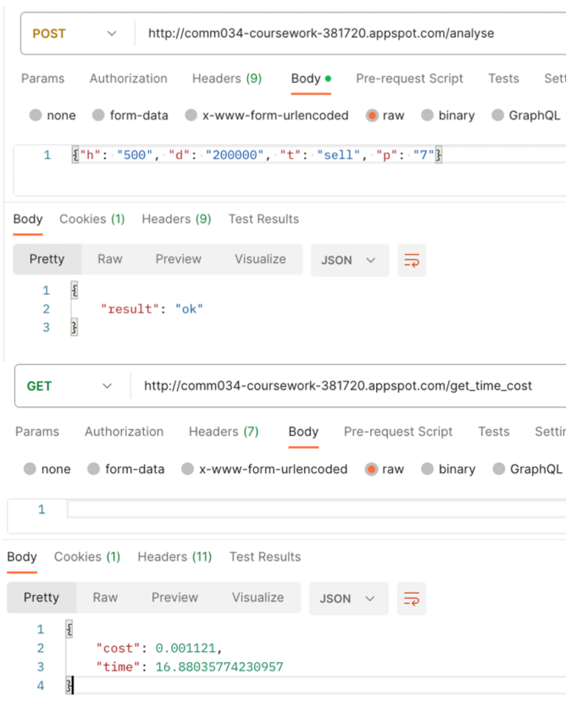
#	*C*	*Description*	*Code used from elsewhere, and how used*	*Code you needed to add to what you used from elsewhere*
i.	M	The calculations are done in lambda and ec2 while hosting of the application is provisioned be GAE. I sent payload requests from GAE to lambda and ec2 and received the response	I made reference to the code from labs 1, 2, and 3 to learn how to deploy to multi-cloud service.	As the application no longer supports a front-end, I added code to parse incoming JSON payload data via the request object provided by flask
ii.	M	The system is provisioned to make calls via curl or application such as Postman . There are a total of 15 endpoints	Reference was made to the coursework description and lab 3 for making curl requests	Some research was done in order to determine how to make curl requests using Postman
iii.	M	GAE sends payloads gotten from user input to the endpoint URL of the AWS service where calculations are done. GAE sends requests for the AWS service to be run in parallel and averages results of each thread	COMM034 Lab 3 Page 15 – Referenced this code for parallel runs to understand how to send requests to lambda from GAE and run the function in parallel	Building on the logic used in lab 3 I used a similar approach to integrate GAE with ec2 with simple logic by sending the request to the ec2 DNS host URL instead of the lambda function URL.
iv	M	To mediate communications between AWS and GAE , I made use of the LabRole . I created several lambda functions which aided in communications e.g. a lambda function called ec2handler which handled ec2 instance creations.	COMM034 Lab 3 Page 13 – I referenced codes from this lab for how to make post communications to lambda	I added code to handle get requests from a lambda function as some lambda functions created for example one which checked if any instances were available solely used a get method
v	M	**<u>Numbered List of Endpoints Met</u>** 1. /resources_ready 2. /get_endpoints 3. /analyse 4. /get_sig_vars9599 5. /get_avg_vars9599 6. /get_sig_profit_loss 7. /get_tot_profit_loss 8. /get_chart_url 9. /get_time_cost 10. /get_audit 11. /reset 12. /terminate 13. /resources_terminated	**A. Endpoints 1 and 2** **I**. I used code from reference [3] in order to learn how to use requests to get url's **II**. I used code from reference [4] in order to find how to get information from an ec2 instance such it's launch time , instance Ip and ids. **B. Endpoint 3(analyse)** **I.** COMM034 Lab 3 Page 15 – Referenced code from this lab for using the thread pool executor for running AWS services in parallel. **II**. COMM034 CW handout pages 12/13 – Used this code in order to get the signal data **III.** Made use of this code from reference [8] in order to create the	**A. Endpoints 1 and 2** Added checks to make sure number of ec2 instances requested by user is equal to the number retrieved . Some instances may take longer than others to start up so without this check the user may get a false response from the system when checking if resources are ready. **B. Endpoint 3(analyse)** **I.** For ec2 I made use of the .submit call when using the thread pool executor instead of .map as I had trouble getting that to work. **II.** I added functionalities for converting the data to a list of dictionaries then sending it to s3 via a lambda function. I then implemented the required functionalities for calculating the next 6 endpoints.

#	C	Description	Code used from elsewhere, and how used	Code you needed to add to what you used from elsewhere
v	P		bucket via lambda which is needed for storing signal data. **IV.** COMM034 Lab 5 Page 13 – I used this code as a starting point for my ec2 back-end script in order to understand how to interact said script. **C. Endpoints 4 , 5 ,6, 7** Made use of code from reference [1] in order to know how to transpose the returned list of results in order to get averages across threads. **D. Endpoint 8(get_chart_url)** COMM034 Lab 1 Page 23 – I made use of the logic from Image charts used in this lab paired with logic from the image charts official website [7] in order to build my line chart. **E. Endpoints 9(get_time_cost)** COMM034 Lab 3 Page 13 – Referenced this code for setting timer to record calculation times. **F. Endpoints 10 , 11** Made use of code from reference [5] in order to learn how to use sessions for persistent storage for the audit and clearing the session for the reset functionality. **G. Endpoints 12, 13** I made use of code from reference [6] in order to terminate instances and code from reference [4] to check if there are still any instances left after termination.	**III.** Built on this by implementing functionality to send the data as a payload to the s3 bucket and also, I created a separate lambda function which retrieves the stored data. **IV.** I built on this code by converting the content type from html to json . I then added code to dump the json calculation results back to flask. **C. Endpoints 4 , 5 ,6, 7** I added simply python code to get the averages of requested endpoint after the list was transposed **D. Endpoint 8(get_chart_url)** I added some customizations to the generated chart such of the risk value lines and averaging out lines **E. Endpoints 9(get_time_cost)** Added calculations to get cost from calculation or elapsed time based on pricing provided by the AWS service. **F. Endpoints 10, 11** Added check to make sure user can't get any results once session has been cleared. **G. Endpoints 12, 13** Added a check so if resources terminated returned as true the user won't be able to run any calculations or get results using ec2.
v	P	**Warmup Endpoint** The warmup functionality is fully functional for ec2 . For lambda however currently to warm up all I am doing is sending some dummy data across as a payload which doesn't necessarily prevent a cold start . In order to meet this requirement, I would investigate warming up lambda using CloudWatch **get_warmup_cost** This endpoint is fully met for lambda . For ec2 the warmup time was not implemented as I had timeout issues when calling wait until running .Hence I assumed a warmup time of 30 secs and calculated the corresponding cost based on that. To meet this requirement, I would need to research ways to prevent timeout errors for ec2 start-up via lambda.	**Warmup Endpoint** ***A.*** COMM034 Lab 3 Page 11 – For Sending Post Requests to Lambda. ***B.*** COMM034 Lab 4 Page 15 – For Launching EC2 Instances **get_warmup_cost** COMM034 Lab 3 Page 13 – Referenced this code for setting timer to record warmup times.	**Warmup Endpoint** For lambda I added to code to record warmup times during warmup . For EC2 I implemented code for the ec2 instances to be created via a lambda function. I also added a check for ec2 via another lambda function which checks if ec2 instances are already running , if true the user can't start new ec2 resources . **get_warmup_cost** Added calculations to get cost from warmup or elapsed time based on pricing provided by the AWS service.
v	N	N/A	N/A	N/A

IV. Results

The system was run via postman as seen in figure 2, considering two scenarios, the first scenario involved running the system with an incrementing number of shots multiplied by 2 starting at 100k and with a constant number of resources (3) across both services. From Table II we see as the number of shots increases the time taken for calculations is significantly higher for lambda compared to ec2. at 100k shots, the calculations take roughly 4 seconds for both services with this value doubling for lambda as the number of shots are doubled. Interestingly on the first ec2 start-up, the calculation with a lower number of shots took more time than the second calculation which had double the shots. This suggests the current system has no load balancing is currently implemented. Scenario two involved incrementing the number of resources by multiples of 2 starting from 2 resources across a constant number of shots (200k) for both services. From Table III we again see the calculation times for ec2 across all resources is significantly less than the time taken for lambda. For both services, we see as the number of resources increases as the calculation times increase with lambda times increasing at a higher rate. Although the calculation time for lambda using 2 and 4 resources is almost identical which suggests the lambda function isn't properly warmed up and hence a cold start.

Figure 2. System run showing analyse and get time cost endpoint calls for lambda via postman.



POST http://comm034-coursework-381720.appspot.com/analyse

Params · Authorization · Headers (9) · Body · Pre-request Script · Tests

none · form-data · x-www-form-urlencoded · raw (selected) · binary · GraphQL

```
{"h": "500", "d": "200000", "t": "sell", "p": "7"}
```

Body · Cookies (1) · Headers (9) · Test Results

Pretty · Raw · Preview · Visualize · JSON

```
{
    "result": "ok"
}
```

GET http://comm034-coursework-381720.appspot.com/get_time_cost

Params · Authorization · Headers (7) · Body · Pre-request Script · Tests

none · form-data · x-www-form-urlencoded · raw (selected) · binary · GraphQL

Body · Cookies (1) · Headers (11) · Test Results

Pretty · Raw · Preview · Visualize · JSON

```
{
    "cost": 0.001121,
    "time": 16.88035774230957
}
```

TABLE II.

Service	*Resource*	*Shots*	*Time(Sec)*	*Cost($)*
Lambda	3	100000	4.506	0.000112
Lambda	3	200000	8.639	0.000215
Lambda	3	400000	17.516	0.000436
EC2	3	100000	4.396	0.0019
EC2	3	200000	2.815	0.0024
EC2	3	400000	4.876	0.0029

TABLE III.

Service	*Resource*	*Shots*	*Time(Sec)*	*Cost($)*
Lambda	2	200000	8.990	0.000149
Lambda	4	200000	9.389	0.000312
Lambda	8	200000	16.880	0.001121
EC2	2	200000	3.284	0.0006
EC2	4	200000	5.296	0.0014
EC2	8	200000	7.263	0.0021

V. Costs

Lambda costs work on a pay-as-you-go basis which is priced per millisecond of memory usage. The current system is using 512 MB storage which means costing per calculation is very small unless the user is calculating millions of shots. This contrasts with ec2 which is billed at an hourly rate of $0.0116. This would mean users are billed even if they aren't running any calculations as seen from Tables II and III with ec2 costs not being directly related to simulations run meaning running the same calculations using both lambda and ec2 could cost the user significantly more for ec2 if care is not taking to terminate all running instances once simulations are over. Currently, the system is restricted to a single user only so ideally the service with less cost would be a lambda, however in a case where multiple users were using the system sending a high number of requests costs will substantially increase especially for lambda almost to a point which negates the reasoning for financial trading so in this scenario ec2 would be a better option, also considering apart from the lambda function handling simulations, the 6 other lambda functions integrated into the system to mediate communications with GAE. In a real-world scenario with multiple users on the system, efficiency may be of high importance meaning for lambda memory storage may have to be increased and ec2 which is currently being run on t2.micro, would now potentially need to run on larger instance types such as t2.large which is nine times the cost of t2.micro. This would also apply to GAE which is priced based on a Cost per hour per instance with higher instance classes costing substantially more. The current system uses S3 to store the original signal data so although increased, the cost for S3 would be minimal even at high storage with a cost of $0.023 per Gigabytes.

References

[1] datagy.io. "How to Transpose a List of Lists in Python with Examples.". Available online: https://datagy.io/python-transpose-list-of-lists/
[2] Program Creek. "Example of flask.session.clear.". Available online: https://www.programcreek.com/python/example/79006/flask.session.clear
[3] W3Schools. "Python Requests get() Method.". Available online: https://www.w3schools.com/python/ref_requests_get.asp
[4] Middleware Inventory. "AWS CLI for Amazon EC2.". Available online: https://www.middlewareinventory.com/blog/aws-cli-ec2/
[5] TestDriven.io. "Understanding Flask Sessions.". Available online: https://testdriven.io/blog/flask-sessions/
[6] Choudhary, Dheeraj. "Launch, List, and Terminate AWS EC2 Instances Using Python Boto3 Script.". Available online: https://dheerajchoudhary.hashnode.dev/launch-list-and-terminate-aws-ec2-instances-using-python-boto3-script
[7] Image Charts. "Chart Image Generation Service.". Available online: https://www.image-charts.com/
[8] Saturn Cloud. "How to Create an S3 Bucket Using AWS Lambda & Python.". Available online: https://saturncloud.io/blog/how-to-create-an-s3-bucket-using-aws-lambda-python/
[9] Dr Lee Gillam. COMM034 Lab 1,3,4,5 and Coursework Handout
[10] NIST. "The NIST Definition of Cloud Computing." Available online: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf