Risk Analysis for Netflix Stock Using Cloud Technologies

Name: Sharayu Dileep Salunkhe : URN: 6725863, Email: ss03755@surrey.ac.uk

URL: https://cloud-new-sharayu.nw.r.appspot.com

***Abstract*— A Netflix (NFLX) stock data analysis will be carried out to assess the potential risks associated with implementing the Three White Soldiers and Three Black Crows trading strategies through the creation of a web application utilizing Google Cloud Platform (GCP) and Amazon Web Services (AWS). The abstract detail the application's framework, prerequisites, and comprehensive cost breakdown of every service involved in its development. As part of the analysis, the results and implications are detailed and explained thoroughly.**

Keywords— Cloud Computing, Google Cloud Platform, AWS Lambda, EC2

I. Introduction

Multi-cloud web application for assessing risk using Netflix's financial data aligns with cloud applications' five key attributes. Adaptable computing resources and seamless, on-demand services are offered by the system. According to the core principles of a cloud-based application, this section describes how the system is designed. The APIs built for the purpose of analysing the risks and profitability of the stock data comply the NIST standards as discussed below.

Essential Characteristics:

On-demand self-service: The user can make use of the cloud resources without any need of an administrator intervening into the architecture. Scaling of the resources can be done automatically. When it comes to the limitation, AWS CLI credentials require frequent manual updates to run the application smoothly, due to limitations within the AWS Academy framework. [1]

Broad network access: A key element of this strategy is the provision of cloud resources and services over a network, commonly the internet, and the ability to access them through a wide range of devices and platforms. The APIs which are deployed in the Google Cloud Platform can be accessed from any location and system.

Resource pooling: In cloud computing, resource pooling refers to aggregating and sharing computing resources, such as processing power, storage, and memory, among multiple users. By pooling resources, it becomes possible to allocate them efficiently and distribute them dynamically according to demand. Multi-tenant models allow for flexibility, scalability, and cost-effectiveness through efficient resource sharing. Resources can be specified by parameters like geographic region, but consumers typically have limited control over the exact location of resources.

Rapid elasticity: The availability of resources can be automatically increased and decreased as demand changes. The elasticity of this system allows it to be adjusted quickly to accommodate higher or lower usage levels. In order to reduce computation time, the user can specify how many resources are to be run in parallel. With this storage or memory requirements can also be scaled up as per the needs.

Measured service: This involves automated resource management through metering, tailored to service types like storage, processing, and users. Continuous monitoring, control, and reporting of usage provide transparency for both providers and users. With the use of parallel resources, the overall time of computation is reduced significantly which also reduces the cost. This cost information is provided to the user to evaluate the overall expenses incurred for running the service.

Service Models:

Software as a Service (SaaS): Python flask is used which acts as a Software as a Service. Flask enables the user to create the applications which can be hosted on the cloud services. Image-charts are used to generate the charts for risk values. The service can be accessed over the internet without users needing to install or maintain any software locally. Providing a cloud-based solution for creating and embedding charts into various applications and websites, it falls into the SaaS category.

Platform as a Service (PaaS): Cloud-hosted applications minimize the complexity of managing underlying resources for hosting by utilizing a cloud infrastructure (GAE). In addition, AWS Lambda, a Platform as a Service (PaaS) that simplifies the management of multi-cloud web applications, is employed to calculate risk. Resources can be managed more efficiently and effectively in this way.

Infrastructure as a Service (IaaS): Amazon EC2 and S3 are both the IaaS service used in the building the services. EC2 provides instances which users have the provision to configure as per their needs. S3 is used to store the risk values and historic data. A complete management of the operating system and storage resources, all without the need to possess the underlying infrastructure is provided.

Deployment models:

Public, Private, Hybrid, and Private Cloud are the four deployment models specified in NSIT 800-145. The API services are deployed on a Hybrid cloud since we have made use of GCP for deploying the APIs constructed with Flask framework and AWS lambda and EC2 service for the rest interaction.

A. Developer

In NIST SP800-145	Developer uses and/or experiences
On demand self service	The application is deployed via Google cloud platform so that it is accessible to anyone at any point of time.
Broad Network access	The service is created considering that it must be accessible from any device over the internet. AWS lambda, EC2 and Google Cloud platform provides this which needs appropriate inputs on location. Location service used for the development of this service is N-virginia which provides low cost service.
Resource Pooling	Virtual machines (instances) are created in EC2 for the purpose of parallel computation. Memory can be increased in both Lambda and EC2 as per the requirement.
Rapid Elasticity	The developer configures the necessary computing resources for the web application's optimal performance, including storage and memory, which can be readily adjusted for both increased and decreased needs.
Measured service	A developer provides a provision to the user to see what is the cost incurred for each request.

B. User

In NIST SP800-145	User uses and/or experiences
On demand self service	The user can warmup the resources according to his needs, which means he can scale them up and down. With AWS Lambda, users can leverage serverless functions, and cloud services (GCP) to provision and configure virtual machines (EC2 instances), without the need for administrative intervention.
Broad network access	A number of platforms are supported by the application, including desktops, laptops, tablets, and smartphones, regardless of where the user is physically located. The endpoints can be hit by the user to see the data from anywhere
Resource Pooling	Since user can scale the resources, without leveraging any physical infrstructure, cost effective and efficient resource management is fostered
Rapid Elasticity	The user can seamlessly increase the processing power, storage and resources based on the payload
Measured Service	Users can understand and manage their usage of computing resources based on insights into their consumption by the cost function included in the service

II. Final Architecture

A. Major System Components

There are several core components of the service, including Google App Engine, Amazon Lambda, Amazon Elastic Compute Cloud, Amazon S3, and Python Flask. These are discussed as below.

Google App Engine: Google App Engine hosts the application in the cloud, allowing public access. The platform offers automatic scaling, adjusting computing resources based on incoming website requests. Google App Engine supports various programming languages so there is no issue while hosting the service.

AWS Lambda: AWS Lambda provides event-driven serverless computing that dynamically scales as demand rises. When users opt for AWS Lambda to determine risk values, the service computes these values for analysis. Using user-defined parameters, these values are employed to compute the 95% and 99% risk values. After collecting risk values, they are forwarded back to GAE for further analysis.

AWS EC2: Ubuntu instances will be launched using an Amazon Machine Image (AMI) when EC2 is selected for calculating risk values. AMIs are preconfigured templates that consist of an operating system and software. In this case t2.micro system is being used which has 1GB of memory. Once the risk values are calculated, these instances will respond to GAE with the results.

AWS S3: AWS S3 is a cloud-centric storage solution providing object storage capabilities. In this context, user-conducted analysis results are securely stored using the S3 service. Data is divided into objects, each with a key. This data is then retrieved and displayed on the audit API call.

Python Flask Framework: All the APIs are written in the Flask framework. Flask is a micro web framework used for developing APIs and web services. Through flask various routes are created which are called to fetch the response data. These APIs interact with Lambda and EC2 as per the selection by the user.

B. System Component Interaction

As a first API interaction, the user warms up the resources by sending a POST API request where they specify which services (s) they wish to use between Lambda and EC2 and the number of parallel resources (r). Once this API is executed, the value of s and r both are stored in a global variable which can be accessed later on. With another GET API it is checked if the resources are warmed or not. So, for EC2 instance all the instances are checked if they are in running state or not. Another API is executed to calculate the warmup cost. To implement the get_endpoints, GET API is written which further communicates with the Lambda function to fetch the DNS of the running EC2 instances and these are stores in a global variable.

After this implementation we move on to analyse API which is a POST API. There are five elements in the price history: length (h), shots (d), buy/sell (t), and the number of days after which to check profit/loss (p). These variables are stored in variables. Another function is called from this endpoint which aims to fetch the NFLX data. This function is getting the data parallelly as per the resources specified by the user. AWS lambda plays a role here to get the desired output and the final result for 95% and 99% risk values which are stored in another global variable. If the scalable service is EC2 then the API communicates with another function where the DNS addresses from the global variables are taken and data is parsed to get the results for 95% and 99% risk which are stored in global variable again.

To proceed with the another, GET API for getting the signal var9599 values which takes the data stored from the analyse API endpoint. It does the manipulation on the

data to display only first 20 values by date. Similarly other GET APIs are requested to get the average, profit loss, and total profit loss. For chart url, Image-charts are being used. To carry out the audit logs, S3 bucket is used to store the data and then results are fetched and displayed.

To finally terminate the EC2 instances GET API makes a call with the lambda function, where boto3 is used to terminate all the running instances. Finally, a GET API to check if the instances are terminated or not gives a Boolean response for the same.

III. Satisfaction of Requirements

The following table discusses the implementation part and from where the code was referred and where the necessary changes were needed.

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 of Google App Engine to deploy the APIs. AWS Lambda and EC2 are the scalable services used.	The python code present in the index.py containing routes is taken from Lab1. app.yaml is also taken from the same lab 1. Lab 3 was used to set up AWS Lambda, while Lab 4 and 5 covered initializing EC2 instances. [6]	There are multiple APIs written in the index.py so several aap.routes are added which further makes connection to the AWS.
ii.	M	The APIs being deployed can be accessed through curl requests and offers persistent API for user interaction.	The way to make curl requests is taken from Lab 3. All the AWS lambda functions can be accessed through the calls directly. [6]	The code is modified as per the requirement and logic functionality needed for the communication of python index.py file with the lambda.
iii.	M	The scalable service Lambda and EC2 are used to generate simulated risk values.	A lambda function was used to calculate the risk values var95 and var99 based on the values passed to it.The code used is from the coursework and the from Lab 3. [6]	A Lambda function has been implemented to process the payload data and compute risk values.
iv	M	Utilizing Lambda functions to facilitate interaction with EC2 while mitigating credential issues.	The code taken from lab 4 was utilized to establish EC2 instances through Lambda, by using the boto3 library. [2] [6]	The code for this the communication via Lambda is written within the AWS lambda function to create the instances. Further it is modified to terminate the instances and modifications are done as per the documentation.
v.a	M	The following API requirement is met and fulfilled through Lambda and EC2 For initialization- i. warmup ii. resources_ready iii. get_warmup_cost iv. get_endpoints For risk analysis- i. analyse ii. get_sig_vars9599 iii. get_avg_vars9599 iv. get_sig_profit_loss v. get_tot_profit_loss vi. get_time_cost vii. get_audit For reset, terminate, resources_terminated i. reset ii. terminate iii. resources_terminated	The *warmup* of the resources depending on ‘s’ and ‘r’ is carried out with the help of code from lab 3 where creation of EC2 is carried out from lambda function using the lab 4 code. [6] To implement the *resources_ready* and *get_endpoints*, boto3 documentation is used. [2] For *get_warmup_cost* the costing is referred from AWS costing documentation For the *analyse* API the code provided in the coursework is used and for parallel functioning thread pool is used mentioned in lab 3. For *get_sig_vars_9599, get_avg_var9599, get_sig_profit_loss, get_tot_profit_loss, get_time_cost* the results from the above analyse	The code was modified to get the results in the specified format for *warmup*. To get output for *resources_ready* the code was modified and the running state of the instances was tried to be figured out. For *get_endpoints* modifications done to get the results as per specification. For finding the *get_warmup_cost* the code calculates the time taken to run and then calculate the billable cost, which is added and required response is being sent. For *analyse* the code is modified to get the desired results which includes list and dictionary manipulation. For other API endpoints *get_sig_vars_9599, get_avg_var9599,*

#	*C*	*Description*	*Code used from elsewhere, and how used*	*Code you needed to add to what you used from elsewhere*
			API are used and manipulation is done. For *get_audit* endpoint the code for storing and fetching values from an S3 bucket was derived from the boto 3 documentation. For *reset* all the values are set to zero or null. For *terminate* the code is referred from the boto3 documentation. [2] For *resources_terminated* the code is referred from boto3 documentation. [2]	*get_sig_profit_loss, get_tot_profit_loss, get_time_cost* the code is changed for the computation required. For *get_audit* endpoint the code is modified to get the results in the desired format. Dictionary manipulation is done when the data is parsed from the index.py file to the Lambda function. For *reset* endpoint no modification or additional code is required. Code was modified for *terminate* endpoint to get the output. To get output for *resources_terminated* the code was modified and the terminated state of the instances was tried to be figured out.
v. b	N	The API endpoint to generate the graph showing 95% and 99% risk values along with the average is not available. Get_chart_url	This API could be fulfilled with the use of Google Chart service or Image charts. This is not implementated	

IV. RESULTS

Summary of the results is given in the following table.

TABLE II. RESULTS

Parameters selected	*Total profit loss*	*Avg95* *Avg99*	*Time taken*	*Cost*
S = Lambda, R = 2, H = 100, D = 5000, T = Buy, P = 5	149.14	-247.17 -337.38	9.93 sec	0.000005
S = Lambda, R = 4, H = 120, D = 10000, T = Sell, P = 7	157.75	-267.96 -362.68	17.82 sec	0.000018
S = Lambda, R = 6, H = 110, D = 12000, T = Buy, P = 10	239.87	-261.56 -354.71	21.336	0.000033
S = EC2, R = 3, H = 100, D = 10000, T = Sell, P = 6	204.99	-246.69 -335.82	6.72	0.000064
S = EC2, R = 2, H = 105, D = 15000, T = Buy, P = 8	152.51	-247.24 -335.93	6.64	0.000042
S = EC2, R = 5, H = 115, D = 20000, T = Sell, P = 10	239.87	-264.13 -360.01	16.95	0.00027

V. COSTS

Following is a breakdown of how much it costs to host the application on a real-world basis, outside of the free tier.

AWS Lambda: In the first 6 billion GB-seconds of the month, Lambda requests cost $0.0000166667 per GB-second. [3] Assuming usage falls within the first tier of up to 6 billion GB-seconds / month, the price per 1ms for 128mb is $0.0000000021. So, for R requests and T seconds the following pricing formula will be applicable:

Lambda Cost = R * 128/1024 * 0.0000000021 * T * 1000

AWS EC2: In the case of an application that uses EC2 instances with one virtual CPU and 1GB of memory, the cost of using scalable service would be as follows [4]:

EC2 Cost = R * 0.0116 * T/3600

AWS S3: S3 bucket costing depends on two factors, i.e. the size of the file and a monthly charge of $0.023 per gigabyte for the initial 50 TB. [5]

Google App Engine: The GAE costing for the region it is being deployed is $0.06 per hour as seen on the dashboard.

REFERENCES

[1] Mell, P.M. and Grance, T. (2011) *The NIST definition of cloud computing* [Preprint]. doi:10.6028/nist.sp.800-145.

[2] https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/ec2.html

[3] https://aws.amazon.com/lambda/pricing/

[4] https://aws.amazon.com/ec2/pricing/

[5] https://aws.amazon.com/s3/pricing/

[6] https://surreylearn.surrey.ac.uk/d2l/le/lessons/239801/units/2606979