Risk Evaluation of Trading Signals Through Monte Carlo Simulation

https://lateral-journey-377016.nw.r.appspot.com

***Abstract*— BP.L data trading signals are simulated using the Monte Carlo method using a command line interface (CLI) application that employs Cloud Computing characteristics to assess the risk. Using cloud services like Google App Engine, AWS Lambda, EC2, and S3, the application provides on-demand service, scalability, and flexibility. This article explores how different cloud components are employed in achieving that and includes architecture specifics. The cost associated with obtaining the appropriate results for this application is also covered in the article.**

Keywords—Cloud Computing, AWS Lambda, EC2, Google App Engine, On Demand Service

I. INTRODUCTION

The five main cloud features stated in NIST SP 800-145 are followed by the CLI-based application. The application employs cloud concepts for scalability and flexibility and provides on-demand service using CLI, also the cloud infrastructure has a physical layer and an abstraction layer. All services are evaluated based on time-related consumption

Essential Characteristics:

On-demand self-service: The application leverages the server and network storage as necessary for risk assessments without sacrificing functionality; there is no human interaction with other service providers. The application also offers lambda and EC2 instance parallel processing, with instances being created based on the necessary quantity of resources.[1]

Broad network access: The application is available throughout any geographical location, any time zone, and any device, achieved through standard mechanisms[1]

Resource pooling: The application pools Lambda and EC2 employ a multi-tenant strategy, with physical and virtual resources being dynamically assigned and reassigned in response to customer demand. Lambda and EC2 handle resources like memory, computation, storage, and network bandwidth at the location where the load is lower and the cost per use is lower (us-east).[1]

Rapid elasticity: In order to scale quickly outward and inward in accordance with demand, Lambda, and EC2's memory and storage are configured. This makes it easy to use for any quantity at any time.[1]

Measured service: For storage, compute, bandwidth, automatic control, and resource optimisation, Lambda and EC2 use metering capabilities with abstraction. Resource usage reporting and monitoring promotes transparency for both users and service providers.[1]

Service Models:

Software as a Service (SaaS)

The consumer uses the application hosted on a cloud infrastructure. Through a Command Line Interface the application is available to the user. The user is given a very minimal configuration choice and does not manage or control the network, servers, operating systems, storage, or even specific application capabilities. The chart of the analysis is created using image charts which is also a SaaS[1]

Platform as a Service(PaaS)

The application created using the Python Flask framework is deployed on GAE to significantly reduce the efforts for using the resources in deploying the application. Changing the programming language services libraries doesn't actually affect the deployment of the application. The user can configure the deployed applications and doesn't have to worry about how to use the underlying resources. Similarly, AWS lambda(also a PaaS) also provides the ability to configure according to requirements and specify the language, service, and libraries used for risk calculations, storage, and accessing other scalable services.[1]

Infrastructure as a Service(IaaS)

The application uses EC2 to calculate Risks which provides the capability to set up Software, Operating Systems, and Applications. Ubuntu OS is configured in an EC2 instance and Apache2 and Python applications are installed to calculate the risks. Some network profiles can be configured but don't allow full capability. The control is over the OS and installed applications, and storage but again doesn't manage the underlying infrastructure.

Deployment Models:

The cloud infrastructure is made up using, GAE and AWS each of which is a separate legal entity, the application is deployed over GAE which is public, and uses the services by AWS which is private so the Deployment model is hybrid.

A. Developer

In NIST SP800-145	Developer uses and/or experiences
On-demand self-service	The application can be easily scaled with respect to load and requirement of the storage along with this it also offers concurrency.
Broad network access	The application interface is created using the Python flask framework and deployed on GAE irrespective of location we can deploy the application to any of the regions. Similarly, AWS and EC2 are available whenever and whenever time we use them with minimal configuration.
Resource pooling	The application handles the pooling of resources so according to the demand it can allocate the available resources from the pool. If EC2 requires more storage we can easily attach a volume to it. If Lambda requires more memory we can easily increase it through config settings.
Rapid elasticity	The memory, storage, and load of AWS Lambda and EC2 can be easily configured from config settings to meet the user's demands.
Measured service	The cost for each resource that is consumed is calculated using the cost management system inside each respective service which gives an honest consumption rate

B. User

In NIST SP800-145	User uses and/or experiences
On-demand self-service	User can directly consume the endpoints to see the risk analysis and doesn't have to worry about how many concurrent users are executing and doesn't have to worry about abstract services.
Broad network access	Users can consume the endpoints from any location irrespective of location and time.
Resource pooling	The resources are allocated to the user who is consuming the endpoints the more the requests are made by the user the more is resources allocated to them.
Rapid elasticity	Users can use this application without being worried about the storage and memory being consumed during his/her operation.
Measured service	The user can use the application according to his choice for which there is some cost associated. The service usage and cost for it are handled by the provider.

II. Final Architecture

The architecture comprises system components and system components interactions:

System Components:

a) Google App Engine: It is a Platform as a Service and hosting the application which can be accessed through the command line. The application is using Flask framework to achieve this.

b) AWS Lambda: It is event-driven, serverless which automatically controls the resources needed to run code in response to events. It offers risk calculations if the user selects lambda to perform analysis and responds back to GAE when called from it.

c) API Gateway: AWS API gateway is used to deploy REST API the API are GET and POST with respect to the requirements of the GAE application. It routes the request to the respective lambda function assigned to it during creation.

d) AWS EC2: It offers the configuration of a system according to the requirement. These virtually created environments are called instances which run the analysis inside them with the use of applications and OS stated at the time of creation. The instances are created from images.

e) AWS S3: It offers storage of data inside a bucket which can be used throughout the AWS independently. The results of the analysis along with some data are used to pass between different lambda functions.

EC2 is used with Lambda to proceed with risk analysis.

System Interactions

The Python-based GAE-hosted CLI application is interacting with API Gateway through REST API and performs the risk analysis in one of the scalable services. The *warmup* API which is the POST method is taking the scalable service type (*s*) and the number of resources(*r*) to be warmed up. It is going to the respective scalable service and warming the resources as requested and throwing the status of whether the application has finished the operation. The total time of completion of this task is also calculated. The request data of this is stored in the S3 bucket. The *resource_ready* API which is GET will check whether the resources are ready or not by checking the status of the warming-up resource. The analyze API will take the history(*h*), data points(*d*), buy/sell(*t*), and profit days(*p*) to calculate risks by sending it to chosen scalable services fetch the response, and save it in the S3 bucket to make the results available for other analysis. The risk analysis will generate 95% and 99% confidence interval signals to simulate the risks involved. The number of responses generated depends on the number of resources provided during warmup. The *get_warmup_cost* API will calculate the costs with the time consumed by the warmup API. The *get_sig_vars9599* will consume the already generated results from S3 and display the 20 results sorted by date. The *get_avg_vars9599* will also use the results from the bucket and average vars 95 and var 99 values and display the pair with respect to the number of resources. The *get_sig_profit_loss* is using the close value of the stored results and calculates the profit by adding the profit days to it and comparing the close values to get profit or loss values. The *get_tot_profit_loss* is using the data from signal profit loss and then does the total to finally check whether it is profit or loss. The *get_chart_url* will generate a chart using Image-Charts parameters after giving it var 95, var 99,avg var 95, and avg var 99 values there will be *r* no of charts url. The *get_time_cost* will calculate the total time involved in the analysis so far and will calculate the total cost based on consumption on each of the analysis endpoints. The *get_audit* will retrieve all the information from the analysis and will push the results into the S3 bucket and fetch all the previous response and display it to the user. The *reset* will clear up all buckets except "audit" and clear all the global variables and results stored elsewhere used for earlier analysis using general manipulations. The *terminate* will fetch all the resources having the state running and then terminate everything by using the client. The *resources_terminated* will check whether all the instances with the state terminated are there and throw the response according to it.

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 elsewhere*
i.	P	Using of Google App Engine, Lambda and EC2 services	The initial code required is taken from lab 1 where configuring app. yaml file with respect to requirements The code needed to warm up the lambda and EC2 is used from lab 3 and lab 4 respectively	The code was modified to create an instance and access lambda using the HTTP request code. The code was also modified to give a cold call to the Lambda function.
ii.	M	Confirmation of warmed-up resources.	The code was taken from AWS boto3 documentation.[5]	The code taken was modified to check whether the state of the function is "running" if it's not running it will false.
iii.	M	**Warmed-up Billable Time**	The total time taken was calculated using global variables and the cost was calculated using AWS and EC2 price documentation [2]	The code was modified to apply the formula that calculates the cost of the selected scalable services
iv	P	Uses of Lambda to initiate another scalable service	The code was taken from lab 4 to create EC2 instances from lambda using boto3[5]	The code was modified to get instance-id which is used later in the application for termination and other purposes .
v	***M***	*Warmup,* *resources_ready,* *get_warmup_cost* *get_audit* *terminate* *reset* *resources_terminated*	In *warmup* the resources are created using the code from lab 4. In *resources_ready* the code is taken from boto3 official documentation describe instance. The *warmup_cost* calculates the cost using the prices from Amazon pricing documents. *Audit* The audit function uses the code from Boto3 documentation to put the response into the bucket and fetch from it. *Terminate* the code to terminate was taken from amazon describe instances. *Reset* it nulls the results and analysis using general manipulation. *Resources_terminated* the code was used from boto3 documentation.	The code was modified to check weather it has fired a command to warm up resources. The code was modified to get the state of the instance. The code returned the results of the formula. The response data was modified to get the desired results. The code was modified to sent terminate command by using id of the instances. The code didn't require any more modification. The code was modified to get the status 'terminated' of all the running instances.
v	***P***	*Get_sig_vars9599,* *get_avg_vars9599,* *get_sig_profit_loss,* *get_tot_profit_loss,* *get_time_cost* *get_end_points*	*Get_sig_vars9599* The code to generate risks is taken from coursework documents and parallel running of function is achieved using thread pool. *get_avg_vars9599* the code uses the above generated signals from s3 bucket and takes the avg of all the signals. *get_sig_profit_loss* The profit and loss code is data manipulation and comparing the values. *get_tot_profit_loss* The code just took the general data manipulation *get_time_cost* The code figures were used from AWS and *get_end_points* the end points list of lambda function is given by hard-coding links taken from lambda function. All the endpoints requirements are met by lambda but are not met by EC2	The code was modified to get var95 var99 signals. The code was modified by general data manipulation to obtain the average. No modifications done No modifications done No modifications done
v	***N***	*Get_chart_url*	Used google Image charts api to generate the chart	Used the documentation to modify the chart type as per requirement.

IV. RESULTS

TABLE II.

The results in the table show the results obtained by the application using concurrency. We observe that when the parallel resources are used for the analysis it takes approximately the same time as running the resources linearly.

The application is also performing effectively on increasing the number of shots.

s	*r*	*h*	*d*	*t*	*p*	*P and l*	*Time*	*Cost*
L	2	105	2000	sell	5	147.70	12.36	0.0004
L	4	105	1500	sell	7	-57.74	9.48	0.00063
L	5	150	10000	sell	9	51.79	11.64	0.000971
L	6	120	10000	buy	15	255.14	13.26	0.0013
L	10	125	20000	buy	11	164.54	8.21	0.00137

V. COSTS

This section discusses the costs incurred in the real world to deploy the application.

a) GAE: The price of the Google app engine for the deployed region is based on hourly usage which is **$0.06/hour** from the billing dashboard although it is limited to particular bandwidth if we want more bandwidth then further investigation into pricing is required.

b) AWS Lambda: The price of the lambda is based on the no. of requests made to lambda along with execution time and memory allocated to each lambda function.The Lambda is invoking other Lambda function along with keeping the files in bucket so so the memory taken is 1024Mb for each lambda function that is used for analysis. Since the application is designed in such a way that it uses Lambda functions thoroughly so the cost is expected to be high. The prices of Lambda functions differ from region to region with some regions operating at low costs the price for N. Virginia (US-East) region is $0.00001666667 / GB second until the request is 1M then charged at &0.20/M request after that. The concurrent executions are counted as separate. So the total execution cost of lambda is:[2]

Usage cost: T*memory configured / 1024 *0.0000166667 * no. of resources

The consumption charge=(r * 0.020)/1000000.

Adding the two will give the Cost of Lambda resources consumed.[2]

c)AWS EC2: The cost of EC2 is based on hour. The per-hour price of EC2 is **$0.0134** which provides 1 GB of memory and 1 vCPU.[3] To Compute the EC2 cost :

$0.0134 * resources * Time(in s)/3600

d) AWS S3: The price of storing data in S3 is $ **0.023/GB** for the first 50TB per month. Similar to other services the pricing of some of the regions is different than others.[4]

REFERENCES

[1] The NIST Definition of Cloud Computing, “Peter Mell and Timothy Grance” in Special Publication 800-145

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

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

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

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