Serverless Web Application for Cost estimation of Pi

Bhavyasree

***Abstract*— This document presents a cloud system which provides the user the capability of determining the cost for estimating Pi up to the specified digits of accuracy. The system architecture and implementation are discussed in detail. An experiment is explained with varying parameters and their effect on accuracy of Pi. Finally, a cost analysis of the scalable services involved are presented along with the requirements satisfied.**

Keywords*— Cloud Computing, AWS, GCP, Serverless, lambda, EC2, S3, Monte Carlo simulation.

I. Introduction

The cloud system presented in this document implements an on-demand self-service application that determines the cost of estimating the value of Pi accurate to specified number of digits and further identify the value of pi at fixed intervals, generating and storing the cost of the analysis and its relation with user inputs. The web application permits broad network access to any customer. The system offers rapid elasticity to the user by providing the capability of estimating Pi using an on-demand cluster of EC2 instances and lambda functions to parallelise and speed up the overall computation.

The cloud system can be described from two points of view.

A. Developer

From a developer's perspective, the system offers on-demand functionalities by a set of resources and can be scaled if demanded. The system monitors, captures and reports the resource usage to the provider and user.

The overall system can be divided into two service models.

1. The front- end system: The front-end system is developed under Platform as a Service (PaaS) model. The underlying cloud infrastructure is not managed by the developer; however, the application and its hosting environment are controlled by the developer [1].
2. The back-end system: The back-end system is developed under function as a service (FaaS) model. This lets the developer write and update code on the fly and completely abstracts them from managing server or containers for each function [2].

The whole system was deployed under the public cloud model using two service provides. Amazon Web Services (AWS) was used as the back-end system and Google Cloud Platform (GCP) was used as the front-end and for deployment.

B. User

From a user's point of view, the system provides an on-demand self-service application available over the internet, that can be accessed via the browser. Due to parallel computation the system provides fast results to the user.

The user can also review the cost for Pi estimation for all the previous runs stored in the system.

II. Final Architecture

A. Major System Components

The system comprises of 5 different components, grouped into back-end and front-end components for simplicity:

1. Back-end Components

a) Amazon API Gateway

Amazon API Gateway enables the system to create, publish, monitor and secure the REST API [3]. It provides the functionality to access history from the previous runs and to estimate Pi accurate to specified number of digits. The API provides four HTTP methods to access these functionalities.

b) AWS Lambda

AWS lambda provides compute services without having to manage the infrastructure [4]. Two lambda functions are implemented in the system. One is used to generate Monte Carlo simulations and the other is concerned with storing and retrieving objects from S3.

c) Amazon Elastic Compute Cloud (EC2)

Amazon EC2 provides resizable computing capacity [5]. A group of EC2 instances is launched on-demand in parallel for heavy estimation of Pi. Amazon EC2 provides quick deployment process, including the python code executed in the instances and therefore was preferred over Amazon ECS or EMR.

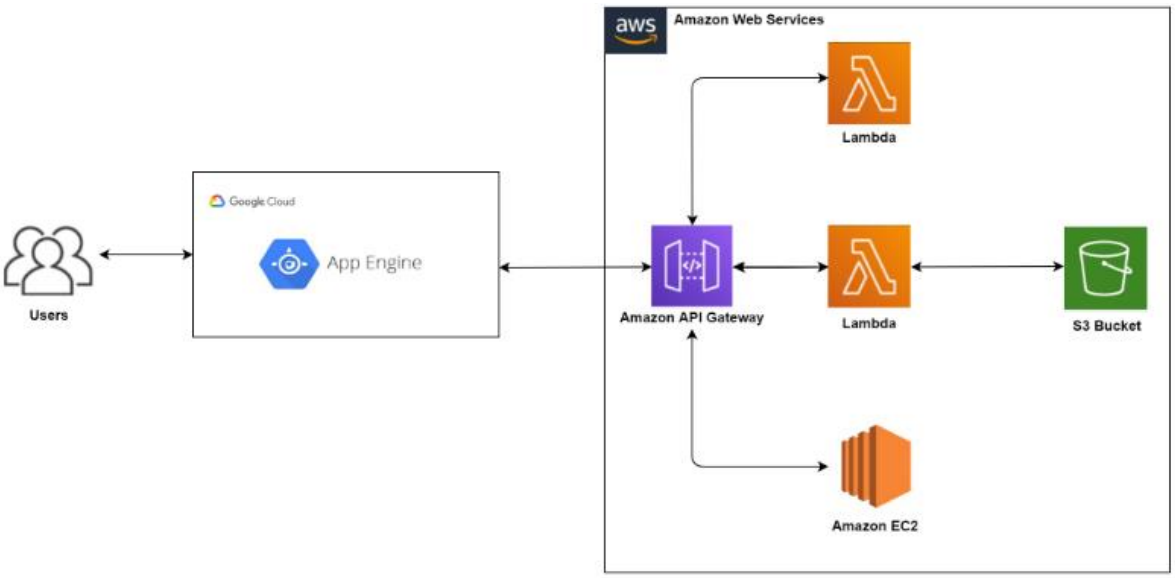
d) Amazon Simple Storage Service (S3)

Amazon S3 is a web service used to store and retrieve any amount of data from the web [6]. The user inputs along with the final estimate of Pi and its cost are stored in S3. Amazon S3 was preferred over S3 Glacier because of the need for frequent access to data. The ease for accessing and storing data, cheaper cost and lack of need for advanced data storage functionalities led to choose S3 among other options like Amazon DynamoDB and Amazon Aurora.

2. Front-end Components

a) Google App Engine (GAE)

GAE is a cloud computing platform for developing and hosting web applications. It provides a platform for the web application to interact with the rest API, allowing the user to perform the functionalities mentioned earlier.



B. System Component Interactions

Various data objects are required for the functioning of the system. The user specifies the scalable service to be used (lambda or EC2), total shots, number of resources, number of digits and reporting rate. The user inputs and EC2 software package is required before the start of the system. The interactions are grouped by functionality, and the two lambda functions in the system are listed:

- Pi Estimator Function (A)
- Read/Write to S3 Function (B)

The system starts when the app engine renders the first webpage which contains the form to be filled by the user.

1) Create a Pi estimate in lambda.

When the user selects lambda and submits the form, the app engine sends them to API Gateway using the proper HTTP POST method of the REST API (1-App Engine to API Gateway). The parameters are then passed to lambda function A and is executed (2-API Gateway to Lambda). This lambda function performs the Monte Carlo simulations. The code requires the value of shots (S), number of resources (R) and reporting rate (Q) for estimating pi. Parallel execution is achieved by making use of threads. Lambda returns all the pi estimate per Q to API Gateway and then to App Engine (3- Lambda to API Gateway, 4- API Gateway to App Engine). The final value of pi is estimated by the values returned by all the resources. This value is then compared with the math. Pi value to the specified number of digits (D). If D is not met further runs are carried out the same way. This continues till D is met or threshold has been reached. The final estimate is obtained after the entire runs. A chart depicting how the estimate proceeds with given rate of shots is then obtained using the estimates at Q. This is then displayed to the user along with a table containing the incircle, shot values and resource IDs per Q. The time taken for the runs are monitored and used to estimate the cost. The final estimate of Pi, cost, choice of scalable service, reporting rate, number of resources and digits are then passed to API Gateway using the proper HTTP POST method (5- App Engine to API Gateway). The lambda function B is then invoked with the parameters (6- API Gateway to Lambda). The function stores the details of the run to S3 bucket as objects (7- Lambda to S3). This can be later retrieved by the user for analysis.

2) Create a Pi estimate in EC2

The interactions in this functionality are similar to the one described previously.

However, before the app engine sends the request, it creates EC2 instances based on the number of resources. The EC2 instances the installs the software package from an existing AMI Image and warms up. Once the instances are in the running state, the app engine sends the user input to the API Gateway using the proper HTTP methods. (1- App Engine to API Gateway). The EC2 then executes the Monte Carlo simulations using the parameters provided (2- API Gateway to EC2). EC2 instance then returns the pi estimate per Q to the App Engine (3- EC2 to API Gateway and 4- API Gateway to App Engine). The values returned by all the instances collectively contribute to estimating the final value of Pi. If D number of digits is not met, then further runs are carried out. A chart and table are displayed based on the results and the result of the analysis is then stored similarly to the previous functionality. The results are passed using HTTP POST method in API Gateway (5- App Engine to API Gateway). The lambda function B is executed with the parameters (6- API Gateway to Lambda). The function then stores the values to s3 bucket as objects (7- Lambda to S3)

3) Return the history of estimates.

A method in App engine is triggered when the user requests the history of all the previous runs. The App Engine then sends a proper HTTP POST method to API Gateway (1- App engine to API Gateway). This in turn triggers the lambda function B (2- API Gateway to Lambda). This function retrieves all the stored estimates from S3 (3- Lambda to S3). The estimates are read from S3 as objects and passed as json to lambda (4- S3 to Lambda). Lambda then returns this history to API Gateway and then to App Engine (5- Lambda to API Gateway and 6- API Gateway to App Engine). The App Engine then displays the history to the user.

III. Implementation

The public web address of the application is:

<https://your-applicatioon-address.appspot.com/>

A. *Implementation Process*

The cloud system was mainly implemented with Python.

The backend system was developed with a functionality for generating Monte Carlo simulations. Both Lambda and EC2 runs the same code for Monte Carlo simulations. While using Lambda, ThreadPoolExecutor was used to run the simulations in parallel. Code from lab 3 was used for implementing ThreadPoolExecutor [7]. The results of these threads were then combined to form the final estimate of Pi.

When using EC2, an Ubuntu server was launched as a master instance, flask and Apache were then installed on this instance [8]. Apache server was configured to start as soon as the instance starts. The python code for Monte Carlo simulations were then uploaded to this server. An Amazon Machine Image (AMI) was created for this instance. AMI provides the information required to launch an instance. Multiple instances with the same configuration can be launched from an AMI [9]. When EC2 is selected by the user, multiple instances are launched from this image, each having the code for Monte Carlo simulations running in the server. Once the instances start, the Public IP Addresses of these instances are obtained. Using the ThreadPoolExecutor the user inputs were passed to all the servers simultaneously. A final estimate of Pi was obtained by combining all the results from each of the servers. The EC2 instances are stopped automatically when the estimate completes. The user can terminate these resources from the front end by selecting the terminate option.

The time taken for the Monte Carlo simulations are captured and are used to calculate the cost of compute resources. The final estimate along with the cost and other input parameters are then stored to S3 bucket. This is done by the lambda function B. This function stores and retrieves data to and from the S3 bucket.

The backend system is hosted in us-east-1(N. Virginia) region. The Representational State Transfer (REST) architecture was used to design the API gateway methods. The 3 HTTP methods used are:

- /estimate POST
- /history POST
- /terminate POST

For developing the system, dependencies like boto3, pandas, numpy and other native Python libraries were used [10,11,12]. The system was developed with the help of documentations from AWS, python, boto3 as well as other blogs and post from Stack Overflow. The most challenging part to implement was the parallel Monte Carlo simulations in EC2. Here, EC2 instances had to be created based on the number of resources the user enters. The Apache server starts when the instances are created, and the calculations are done only after the instances have reached their running state. This provides warm up for the resources. Additionally, the final estimate is obtained by combining the results from all the resources and this must be stored for future reference.

The front-end system was developed by reusing the code from Lab 1 and Lab 3 [13,7]. Flask and Gunicorn along with native python libraries were used extensively in developing the front end. The HTML template and CSS sheets were developed by extensively referring w3schools.com.

The web app renders an error template when a non-existent path is called.

B. *Tests against the code*

The code was tested manually. Performance testing was not conducted. The functionality testing conducted can be divided into two:

1) Front-end tests

Front end tests were carried out manually by browsing the webpage both locally and after deploying in GAE.

2) Back-end tests

Manual testing was done locally in a mock environment, and some tests were also conducted in the AWS console, to ensure the functionality of lambda, API gateway, EC2 and S3 bucket. JSON events were given as arguments for testing both lambda and API Gateway. EC2 was tested using ssh to connect to the instance, and later using the public Ip of the instance.

IV. RESULTS

The following experiment was executed to relate the number of shots required to achieve a specific decimal place of accuracy for Pi. The scalable services used were Lambda and EC2. The values considered for shots were: [1000000,1200000, 1800000,2400000]. Values for resources were: [5,6,8,60]. The digits given were: [5,7].

On comparing the results in Table 1, the number of resources does not seem to affect the specified accuracy of Pi. When the number of shots remain the same (1200000) and number of resources is increased steadily (6,8,60), the Pi value has achieved only four digits accuracy.

On increasing the number of shots, the Pi value gets closer to the specified number of digits. In order to get more accuracy, more shots can be fired.

It can be concluded that the number of shots is directly proportional to the accuracy obtained.

TABLE I. RESULTS FOR LAMBDA

Shots	Resources	Digits	Pi-Value
10,00,000	5	5	3.142200
12,00,000	6	7	3.141477
12,00,000	6	7	3.141819
12,00,000	8	7	3.141128
12,00,000	60	7	3.142379
18,00,000	6	7	3.141259
18,00,000	8	7	3.071291
18,00,000	60	7	3.141016
24,00,000	6	7	3.141866

Two screenshots are presented: Fig. 1 shows the user interface and Fig. 2 shows the results for the inputs. The graph is obtained by plotting pi values at Q for two thresholds among the ten threshold runs conducted.

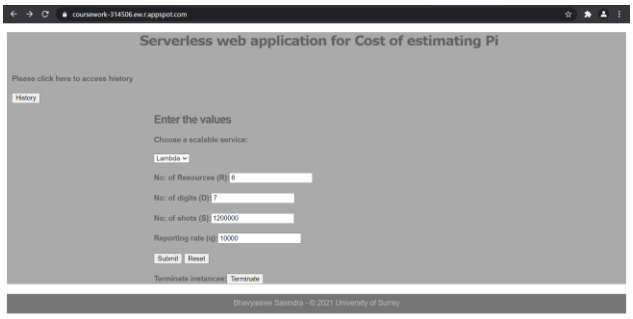


Figure 1. User Interface

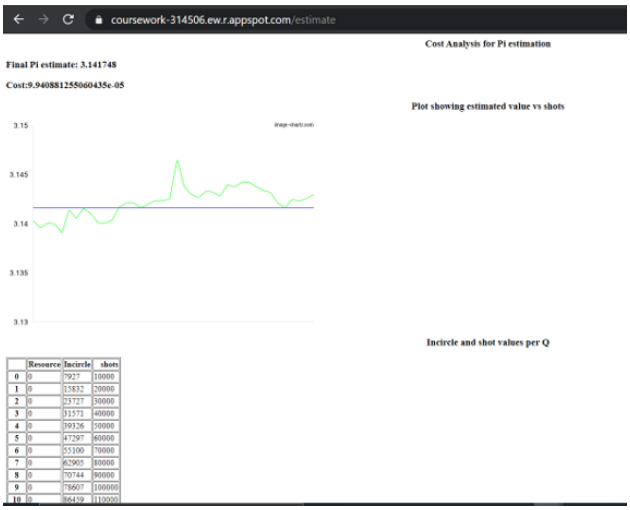


Figure 2. Results of the experiment

V. Satisfaction of Requirements

TABLE II. Requirements

#	*C*	*Description*
i.	M	GAE, AWS Lambda and EC2 were used to build the system
ii.	M	As discussed earlier, user can provision and terminate resources from the front-end, and receives information about the runs to meet the value of Pi
iii.	M	Scalable services run the pi estimation code
iv. a	P	Data is passed to EC2 only after the instances have been warmed up i.e, created and are running. However, the time for this provisioning is not captured separately to report to the user but is included in the total cost.
iv. b	M	As described previously, the user specifies shots, resources, reporting rate and digits. Pi estimation is done in parallel by the resources and each reports its own incircle and shot values
iv. c	M	A chart showing the estimate proceedings is obtained by image charts. A table showing resourceId, incircle and shots per Q is obtained. The final estimate of Pi and cost is presented to the user. History page showing details of all runs is made available to the user.
iv. d	P	The system automatically stops the resources thereby leading to 'zero' analysis. However, new instance needs warm up for the subsequent runs. To avoid warmup, the system must check the existing stopped resources and start them and create new only if required.
iv. e	M	A terminate button is provided in the front-end for the user to terminate the EC2 instances.

VI. Costs

The costs shown in Table III are calculated based on the run time and AWS pricing for east-us-1 region. Free tier discounts are excluded from this calculation.

The lambda cost is determined by the number of requests, the execution time of the function and the amount of memory allocated to the function [14].

The EC2 cost is determined by the running time of the instance and the data transfer in and out of the instance [15]. Data transfers within the same availability zone are free. The data transfer for our application is very small and hence neglected.

TABLE III Costs Results

Shots	Resources	Time	Cost	Service
10,00,000	5	44.85392	0.000104	lambda
12,00,000	6	43.53743	0.000104	lambda
12,00,000	6	160.8375	0.000518	EC2
12,00,000	8	65.7661	0.000155	lambda
12,00,000	60	95.50601	0.000331	lambda
18,00,000	6	59.4061	0.000140	lambda
18,00,000	8	275.9902	0.000889	EC2
18,00,000	60	106.9881	0.000355	lambda
24,00,000	6	77.73488	0.000175	lambda

In conclusion, it can be observed from the table that executing lambda in parallel is cheaper than launching multiple EC2 instances in parallel.

References

[1] P.Mell, T.Grance, "The NIST definition of Cloud Computing," NIST Special Publication 800-145 (SP 800-145), 2011.

[2] L. Gillam, "FaaS (PaaS) overview", Lecture Week 4/5, COMM034-Cloud Computing, University of Surrey, 2021.

[3] Amazon Web Services, Inc., "What is Amazon API Gateway?", Amazon API Gateway, Developer Guide, AWS Documentation, 2021.

[4] Amazon Web Services, Inc., "What is AWS Lambda?", AWS Lambda, Developer Guide, AWS Documentation, 2021.

[5] Amazon Web Services, Inc., "What is Amazon EC2?", Amazon EC2, Developer Guide, AWS Documentation, 2021.

[6] Amazon Web Services, Inc., "Amazon Simple Storage Service, User Guide", AWS Documentation, 2021.

[7] L. Gillam, "AWS Lambda: 'Function as a Service'", Lab Session 3, COMM034-Cloud Computing, University of Surrey, 2020-2021.

[8] Amazon Web Services Inc., "Amazon Machine Images", Amazon Elastic Compute Cloud User Guide, AWS Documentation, 2021.

[9] Thiagarajan,V. "Setting up Flask and Apache on AWS EC2 Instance". Available at: Setting up Flask and Apache on AWS EC2 Instance (vishnut.me)

[10] Amazon Web Services Inc., "Boto 3 Documentation", Boto3 docs 1.17.79, 2021

[11] Pandas Development Team, "Pandas: powerful Python data analysis toolkit", Release 1.2.4, 2021.

[12] Python Software Foundation, "Python 3.9.5 documentation", 2021.

[13] L. Gillam, "Google App Engine (using Python)'", Lab Session 1, COMM034-Cloud Computing, University of Surrey, 2020-2021.

[14] Amazon Web Services Inc., "AWS Lambda Pricing", Pricing, AWS, 2021

[15] Amaazon Web Services, Inc., "Amazon EC2 Pricing", Pricing, AWS, 2021.