An Application that estimates the value of Pi(π) to a specifiable degree using a so-called Monte Carlo Method

Pratyaksh Rao
6612540
pr00358@surrey.ac.uk

***Abstract*— This paper proposes a basic cloud app using two architectures while exploring costs, security and storage adding up to accessibility. It provides the user to be able to estimate the value of pi according to the user's choices. The system's architecture and implementation have been also discussed. Finally, a test run of the app is also showcased in which results with variation in parameters are shown, alongside the running cost of the service with the satisfaction of the requirements.**

Keywords—Cloud computing; cloud storage; AWS; GCP; Lambda; Monte Carlo.

I. Introduction

The cloud application system implements a requestable self-service to estimate the value of pi using Monte Carlo method over the architecture of two independent cloud service providers being AWS and GCP. A web application has also been hosted which acts as a front end to the users to be able to collect information as provided by the user and calculate pi value in the cloud services according to that. With being able to parallelize the computation the system is efficient and faster for the execution providing the feature of elasticity to the user.

The cloud system can be explained in the following two ways:

A. Developer

As a developer the system's anatomy is able to provide on-demand actions, resource management, parallelisation of executions, showing the scalability factor essential to a cloud application. Optimisation and selection of resources is a critical factor in developing efficient system.

The overall system can be divided into two major components which are:

1. The front end system: part of the platform as a service (PaaS). The underlying feature and structure are avoided with the provision of technologies as per the providers services.
2. The back end system: part of the function as a service (FaaS). The servers are not needed to be managed by the developed

The system uses two different cloud service providers making it a public cloud system. This helps both the developer and the user to get the best available services and options from each individual service.
Amazon Web Services provides the back end while Google Cloud Platform provides the front end.

B. User

As a user, although being a complicated system, appears to be a simple and straightforward one. From the user's perspective it is self-service application which provides the result from just a click of a button. Can be easily accessed on the web browser. Additionally, due to parallel computation option the user can also get faster results. The previous results are stored and can also be viewed as part of history.

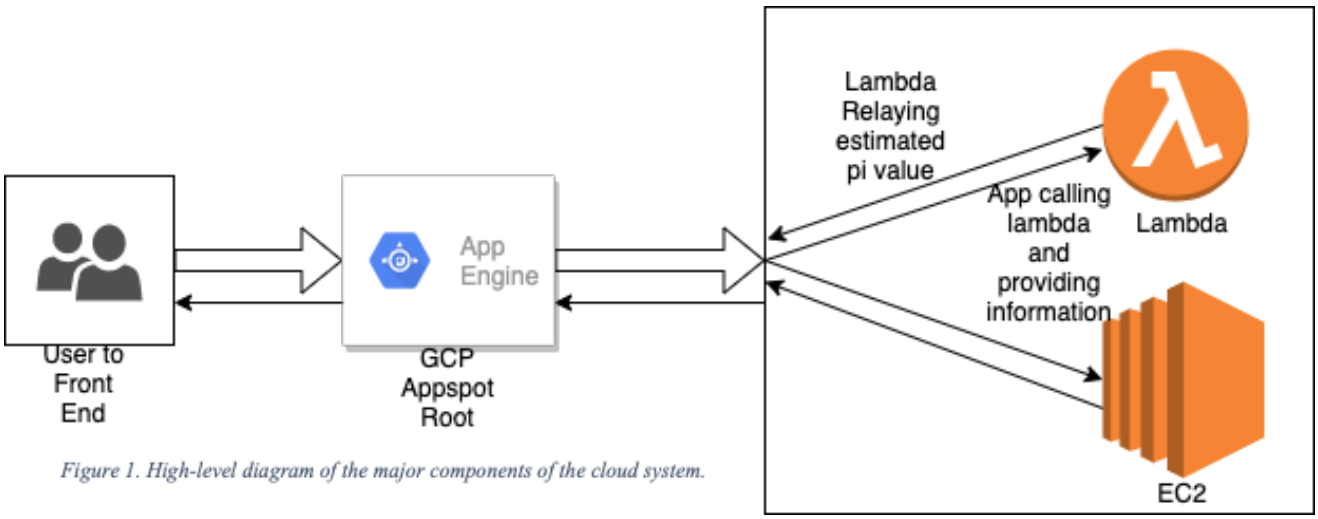


Figure 1. High-level diagram of the major components of the cloud system.

II. Final Architecture

A. Major System Components

There are two major components and four minor components to the application being:

1) Back End Components:

(a) Amazon API Gateway:

REST API is being defined by the system for many purposes such as publishing and maintaining an API because of the Amazon API Gateway. The API gateway has a role to provide by publishing HTTP methods to gain access to further functions.

(b) AWS Lambda:

AWS Lambda is the set piece that has to execute the code which does the core computation (pi estimation in this case). It has no role in handling the infrastructure of the system. Most of the functions are computed through AWS Lambda in REST API.

(c)AWS EC2:

AWS EC2 is the other set piece that can also execute the main computation code. It sets itself apart from lambda by providing the ability for heavy pi estimation calculation by the creation of disposable clusters and parallel jobs.

2) Front End Components

(a) Google App Engine:

The web application (front end portal for users) is developed and hosted through the GAE. The web application can connect with the REST API which ultimately enables the execution of the functions as per the user's request.

The Scalable service that is used primarily is Lambda which is capable of running code in event response and makes it easier as a developer to manage and compute resources, as compared to other viable scalable service options such as EC2 which is excellent in providing resizable capacity to compute or ECS which helps in easily running and management of docker containers dealing with clusters of EC2 instances.

B. System Component Interactions

The lifecycle of the system begins with the first web page being rendered which has the initial form that is filled by the user. The HTML page is designed to create a form that takes in input from the user for total shots, total resources, total reporting rate and the selection of the AWS cloud service to be used. the user can make a choice between Lambda and EC2. As per the requirements the input of the total number of shots will always be in the multiple of 1000. If an input of a number not a multiple of 1000 is provided the webpage will reset until the correct and valid input is given to the form. The user will then hit the submit button to submit the form.
After submission of the form the GAE sends a request through the REST API gateway to AWS Lambda (or EC2) for further computations to occur which includes the plotting of the pi estimation chart and the result table.
As the user data is relayed to Lambda it is then provided to the pi estimation code using monte Carlo simulations which is written in the lambda function. The code requires the values for Shots (S), Resources (R), Reporting rate (Q) in order to calculate the pi value. Since the lambda function was deployed earlier with a HTTP link known as the invoke URL, it is called in the main python program in order to get the results of the computations i.e. pi estimates.
Those pi estimates are then presented in the form of a chart which plots the true value of pi vs the estimated ones and a table is created to show the different estimates in accordance to different values of S, R, and Q.
Additionally, in order to meet the requirements, the tactical use of resources has been carried out in the main python program
Which provides triples and the numbers of each R are determined by using the Maximum of Q and 10% of S/R, all coded in the main flow of the program, being provided to the lambda function before it computes the pi estimate. These modified values of Resources are being shown in the result table on the second webpage.
Finally, all the resource runs are being carried out parallelly only for increased efficiency and better optimization to encompass future scope for scalability and elasticity
The app engine requests the pi estimations available before the page is rendered by calling It by HTTP GET method. The list of pi estimates is then sent from lambda to app engine via the API gateway as like a HTTP response. This list of estimates is saved as a list of lists in the main python program which is then printed in the form of a table.

III. Implementation

APPSPOT- https://eco-shift-268116.appspot.com/

The computation function in lambda and the main function are both written in python and the web pages are written in CSS and HTML.
First the back end system was developed which included making the web form and sending the information to the lambda function. Following that the lambda function was make giving the invoke URL. Until now resources have not been modified (in order to meet requirements). Manual inputs are tested on the lambda function and then the same is tested after being routed to the main python function. The pi estimate values were checked to be received or not.
Now that the pi estimate values were being successfully received and printed in the terminal for the main function, we could move on to modifying in accordance to the tactical use of the resources. This included makes the resources be determined by using the Maximum of Q and 10% of S/R, writing a function for it using `list_combo`. then we had to parallelize the approach of resources for faster computation and more importantly, scalability. We wrote a function for that as well using `threadpool`.
The back end system is being hosted in N.Virginia region for AWS. Due to its serverless application model (SAM) it is used for the benefit of infrastructure as a code for the development of the model. The API Gateway methods and lambda functions are defined in the main python template to be able to deploy the model with minimum commands.
The API gateway is designed using the 'Representational State Transfer'(REST) software which generates the HTTP.

Common dependencies such as pandas, boto3, NumPy along with the native Python libraries were used for the Lambda

Function. Various Documentations such as AWS Services, Google Cloud, Pandas were referred. Some solutions to the problems in code which occurred during the programming were solved with a reference to many Stack Overflow Article. The most challenging part of the coding was the tactical use of resources and resource functionality. All instances are to produce a parallel result which can be merged, deleted or stored as history.
For the front end portion, commonly used code from Labs were used alongside python dependencies like Flask and Unicorn's HTML templates were mostly developed from the lab documentation and w3schools.com.

While none of the tests that were ran against the code were automated, they are divided based on the nature of the code:

1. Back End Tests: initial tests were carried out locally. First in the terminal then in localhost. Lambda function tests were also running locally in the lambda console environment. The link for Lambda and API Gateway and GAE were run to check the functionality. The API were run with HTTP Clients
2. Front End Tests: The web app was opened manually in the local browser and local host. Then it was deployed in the GAE

IV. Results

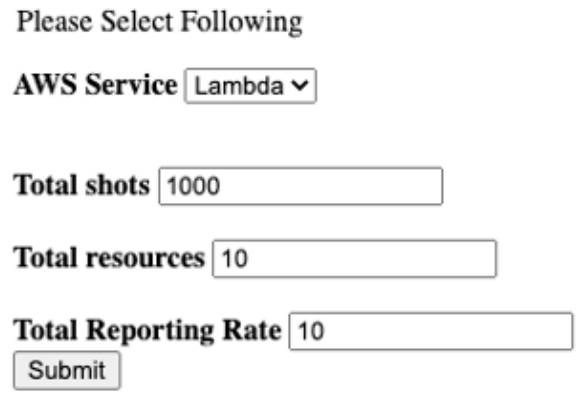
The following test run was executed to observe the variation of the results over the varying values of Shots [1000,10000,100000] and Resources [10,50,100]

Shots	*Resources*	*Process*	*incircle*	*Shots*	*Iterating Shots Value*
1000	10	0	9	110	10
1000	50	0	8	30	10
1000	100	0	8	20	10
10,000	10	0	8	1100	10
10,000	50	0	7	220	10
10,000	100	0	9	110	10
100,000	10	0	8	11000	10
100,000	50	0	9	2200	10
100,000	100	0	8	1100	10

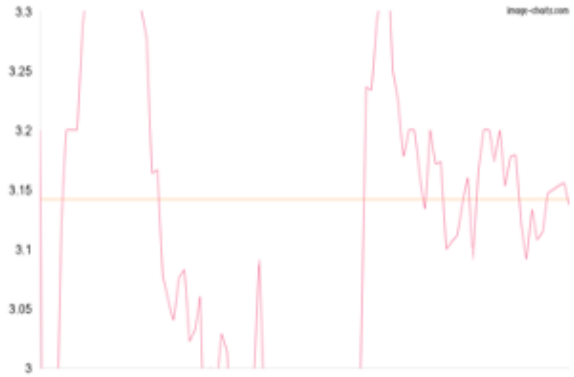
TABLE I. Results

After observation of TABLE 1, we analyze that (i) iterating shots value is kept same for the sake of this table. (ii) Lower value of resources corresponds to Higher value of Shots. (iii) incircle value remains similar throughout the variations.

The highest value of shots was observed to be at 100,000 total shots and 10 total resources used. In conclusion higher total shots and lower total resources works better for our pi estimation



Page 1: Initial Form



Yellow = True pi Value(3.14) , Red = Estimated pi value

incircle

	process	incircle	shots	iterating_shots_val
0	0	8	110	10
1	0	13	110	20
2	0	21	110	30
3	0	29	110	40
4	0	39	110	50
5	0	48	110	60
6	0	56	110	70
7	0	64	110	80
8	0	74	110	90
9	0	83	110	100
10	0	92	110	110

Page 2: Results table and Chart

V. SATISFACTION OF REQUIREMENTS

#	*C*	***Description***
i.	P	GAE, AWS Lambda and EC2 are used but EC2 is not yet routed to the main app
ii.	M	As Described the front end provides users to input information to estimate the value of pi and open webpage
iii.	M	As shown in the screenshot the webpage presents a chart that shows true pi value vs estimated one using Image-charts
iv.	M	The Pi estimation code runs on AWS services such as Lambda and EC2 and is presented to the front end
v.	M	The Resources are launched and used in parallel only for pi estimation as specified and switched off automatically
vi.	M	As mentioned earlier the services are switched off automatically after computation
vii.	M	The Values of S, R, Q and Service selection are specifiable in the Front End
viii.	P	a), b) are met in the main python code for tactical use of resources. “nice” numbers can be further found for optimization
ix.	M	Through the front end the user can run analysis with just one click
x.	P	S, Q, R, pi estimate and Time taken are calculated but can be extended to be stored and presented in the front end

VI. COSTS

The costs are calculated using the run timing information and AWS Pricing for N.Virginia. Note that these values are for the Free tier educate account.
It is calculated from the time of the function and the memory used and the cost per each call of function

shots	*Resources*	*Lambda Time (ms)*	*Cost*
1000	10	94300	~0.3
1000	50	125000	~0.7
1000	100	63200	~0.2
10,000	10	69600	~0.2
10,000	50	142700	~0.6
10,000	100	35200	~0.2
100,000	10	82300	~0.3
100,000	50	132000	~0.6
100,000	100	52900	~0.2

The cost can be reduced by using parallel execution making it much faster and optimized.
As number of shots increases significantly (~100,0000) the computational speed decreases significantly as well. Hence parallel computing is the most optimal solution to this problem. Additionally, adding security layers can also add to cost but result in safer architectures.

REFERENCES

[1] L. Gillam, “Lecture Week 4/5”, Cloud Computing, 2020, University of Surrey.

[2] Amazon Web Services, Inc.,”What is Amazon API Gateway?”, AWS Documentation, 2020

[3] Google Ltd, “Google App Engine Documentation”, 2020

[4] Pandas Development Team, “pandas Documentation”, 2020

[5] Amazon Web Services, Inc.,”Boto3 Documentation”, 2020

[6] L. Gillam, “Lab Week 1”, Google Cloud,Cloud Computing, 2020, University of Surrey.

[7] L. Gillam, “Lab Week 2”, Lambda, Cloud Computing, 2020, University of Surrey.

[8] L. Gillam, “Lab Week 4”, EC2, Cloud Computing, 2020, University of Surrey.

[9] Pallets, “Jinja”, Jinja project, 2007

[10] Refsnes Data, “HTML Tutorial”, w3schools.com, 2020

[11] Refsnes Data, “CSS Tutorial”, w3schools, 2020

[12] Amazon Web Services, Inc.,”AWS Lambda pricing”, AWS, 2020

[13] “Estimating Pi using Monte Carlo Method”,101computing.net , 2020

[14] The SciPy Community,”NumPy Manual”, Documentation, 2019

[15] Amazon Web Services, Inc.,”AWS SAM”, AWS serverless Apllication Model, AWS, 20