Performance Analysis and Cost Implications for Serverless Data Processing Pipelines

(Evaluation between AWS EC2 and AWS Step functions using TLQ pipeline)

Sreenavya Nrusimhadevara, Deeksha Rao Gorige, Simerpreet Kaur, Sumitha Ravindran
School of Engineering and Technology
University of Washington
Tacoma, Washington USA
Email IDs: sreevpk@uw.edu, dgorige@uw.edu, simkaur@uw.edu, sumitr@uw.edu

***Abstract*— This paper presents the performance evaluation of application flow control system in which we compare the performance and cost implications between Amazon Web Services (AWS) - Elastic Compute Cloud (EC2) and Step Functions against a TLQ pipeline comprising of three services. We considered execution runtime and throughput as performance metrics of lambda micro services when invoked from an EC2 instance, in comparison with a state machine. The results of this performance evaluation could help the cloud users in designing the orchestration of micro services in their serverless application.**

Keywords*— *TLQ Pipeline, Serverless, AWS EC2, AWS Step functions, throughput, runtime.

I. Introduction

"Transform-Load-Query (TLQ)" pipeline is similar to an "Extract-Transform-Load (ETL)" pipeline, except the transform phase additionally includes extract. In TLQ pipeline, service 1 performs both extract and transform, service 2 performs load, and service 3 performs query. In TLQ pipeline, extract has been simplified because input data is provided in a single easy-to-use format. We used an input sales dataset in CSV format stored in AWS Simple Storage Service (S3). This data is transformed using a set of predefined rules and the resultant data is loaded to AWS Relational Database Service (RDS) Aurora database using SQL queries so that this data is available for querying using SQL filters and aggregators. This TLQ pipeline is deployed in cloud in the form of AWS lambda microservices. The orchestration of TLQ pipeline from EC2 instance is performed by invoking these micro services in a sequence by transferring the state from one output of a lambda function to the next function as input using bash scripts. While in step functions service, we designed a state machine which does the orchestration for us once we initiated an execution of the workflow from a client. The performance evaluation is based on the runtime, throughput, memory and cost metrics for both workflows.

Lambda services are designed to be stateless and short lived i.e., the maximum execution time is 15 minutes. Amazon has launched a workflow execution service on top of lambda called step functions with built-in retry operations and error handling and also facilitates designing multi-step applications and tracing the stages of the workflow. However, the method of invocation is an important consideration here. The previous research in this area , investigating the performance from a local client (non-cloud) accessing lambda services and evaluated on the basis of latency proving that the lambda services perform better when invoked from a cloud native application (Further discussed in Design tradeoff section). The current approach, evaluates the performance of a serverless data processing application comprising of lambda micro services when executed from cloud native application (from EC2 host) vs AWS step functions workflow.

A. *Research questions*

In the previous research, it was evident that the latency varied from a local client to cloud native client. The current research direction of this paper is towards evaluation of two cloud native applications against a data flow application. The scope of this paper is to consider the runtime and throughput performance metrics of our TLQ pipeline when invoked from a EC2 host and step functions separately. While doing this, the memory and time out configured for each lambda service is noted. While the current research proved that the AWS lambda functions has low latency, we here, attempt to establish a benchmark between EC2 and step functions and investigate the following research questions:

RQ-1: What would be the freeze-thaw metric of EC2 and Step functions against TLQ pipeline?

RQ-2: Which application workflow implementation is efficient between AWS EC2 and AWS Step functions?

RQ-3: What would be the cost implications of these different workflow implementations?

II. COMPARISON STUDY

FaaS is slowly emerging a booming industry standard because of its cost effectiveness and security. The serverless application implemented in this project was TLQ pipeline. The reason behind implementing the pipeline as a FaaS application case study as FaaS provides faster development and requires less maintenance. Also FaaS is highly secure and can be scaled as per user needs. The other reason is FaaS is affordable when compared to other service models.

B. Design Tradeoffs

The case study on Application Flow control can be done on methods other than AWS EC2 and AWS Step Functions. With a laptop as the client, the laptop transfers the data amongst the services and call all the services synchronously. However , because the location of the laptop may vary , this could impact the analysis due to network variability and network latency. So, for actual benchmarking results, we used AWS EC2 instead.

With Controller Functions, a FaaS function controls the flow of a multi-function sequence by running synchronously and issues several FaaS function calls. Running the controller synchronously doubles the cost as it is idle most of the time and waits for other functions to respond.

Within Lambda , the client calls Lambda Service A asynchronously , then Service A calls Service B (via simple notification service (SNS) or using simple queueing service (SQS)

C. Application Implementation

In the TLQ implementation, the transform service extracts the input dataset from S3 bucket and performed transformations such as - generating columns “Order Processing Time” calculated from (i), and “Gross Margin” calculated from (ii), Replacing the values of “Order Priority” column (L=Low; H=High; M=Medium; C=Critical) and Removing duplicate records in the column “Order ID”.

In the Load service, we connect to the Aurora database, check if the table exists, if not, create a table with the column headers in the transformed CSV file in S3.The column Order ID is created as a Primary key.The data in the transformed file is loaded into the database using a batch of insert statements.

In the query service, we connect to the RDS cluster Aurora database and query the database.On the column “Units_Sold”, the minimum,maximum,average and sum of the values are obtained by using SQL aggregators SUM,AVG,MIN and MAX.The sum of “Total_Revenue”, “Total_Profit” columns is obtained by SUM. The number of total orders placed is obtained by using COUNT. The “Order_ID”s of orders with unit price between 207.70 and 437.20 are listed using the filters GROUP BY and WHERE clauses and “Region” is listed with the”Item_Type” is Cereal. The output for each query is given as a separate JSON object in a JSON array.

Languages that are used for the implementation were JAVA and MySQL. The IDEs Apache Netbeans IDE 11.1 and IntelliJ 2.3 and the Cloud services AWS lambda, AWS EC2, AWS S3, Amazon RDS Aurora 5.6 database and AWS Step functions are used.

In our implementation, the input to each service is passed in the form of a JSON object.For the transform service, input is, the name of S3 bucket and key, pointing to the dataset on which the queries has to be performed. The input for load service is the names of bucket and a key pointing towards the transformed dataset.The Query service is provided with the RDS cluster name and table name created by the load function. It then connects to the database to query on the loaded data.So, the states of the TLQ application are the transformed file in S3 bucket and the loaded database in RDS. From EC2, the state is traced at the client side between the service calls. In the case of step functions, once the input bucket name and key name are passed while invoking the state machine, the transform service passes its output of transformed key and bucket name internally.Similarly for the query service, the load generates the JSON output with database details. Hence, the state of the application is traced internal to the state machine. In both the workflow implementation models, the lambda functions synchronously.

The implementation of the services can be found in the link https://github.com/SreenavyaN/TCSS562_Group8

D. Experimental Approach

The case study on Application flow control will basically compare methods to implement a sequence of service calls and their subsequent data exchange. This is done with both AWS EC2 instance and AWS step functions. In the EC2 instance case, we invoked the individual lambda functions explicitly via AWS CLI. We have done this by writing a script which invokes all the three lambda functions. In case of the Step function method analysis, the client needs to call the state machine to execute all the three lambda functions. In order to invoke the state machine we have written a BASH script. This is the client-side infrastructure that we followed.

On the Server side, we configured memory and timeouts according to the requirements. Datasets with large number of rows were taking more memory when compared to the datasets with less number of rows. For each service we have taken different memory sizes. For example for analysis with 10000 row data set, we raised the bar of memory to 640 MB for the transform function, 512 MB for the load function and 512 MB for the Query function. In this way we have changed the memory range differing from one data set to another. In a similar approach that we followed for the memory configurations, timeouts were changed according to the size of the data sets. For example, for the dataset with 10000 rows, we adjusted the timeout to 5m15sec for the transform function, 2m15sec for the load function and 2m15sec for the query function.. The Network configuration was set up to “No VPC” for the transform function.Since we have chosen RDS to load the data for querying, and RDS cluster runs in a VPC set up,Load and Query functions are configured with the same VPC settings. We have used US East (N. Virginia) region for our evaluation. We also run the tests with constant memory (max) and timeout (max) in order to compare the results on common grounds.

D. *Equations*

In the Transform function, we use formulas to generate new columns from the existing columns.For calculating “Order processing time” we have calculated the time difference between order date and ship date.

Order processing time = [Ship date] - [Order date]. -- (i)

The formula for calculating “Gross margin” is,

Gross margin = [Total Profit] / [Total Revenue] -- (ii)

II. Experimental Results

The freeze-thaw cycle for lambda micro services is important in determining the metrics[2]. We invoke the pipeline via bash client script for both EC2 and step functions. The Table.1 below depicts the cold-warm start performance with two consecutive runs. The first run usually has the cold start problem which can be observed from the tabulated values.This cold-warm start exhibits similar latency for almost all the cloud services. For the current TLQ application, the EC2 and step functions takes different time with the same lambda micro services and with the same volume of data i.e., 5000 row dataset. EC2 has a better cold start compared to step functions.

To determine the efficiency of selected models AWS EC2 and AWS Step Functions, testing was performed using different volumes of data in the range 600 KB and 12 MB and for 100 runs of the application. The runtime of each run is obtained using time command in bash ,and the average runtime of 100 runs is computed for each volume of data.The fig.1 shows the data size Vs runtime plot .

AWS EC2 (in sec)	Step Functions (in sec)
70	115
16	21

Table 1: Freeze-Thaw metric for AWS EC2 and AWS Step Functions

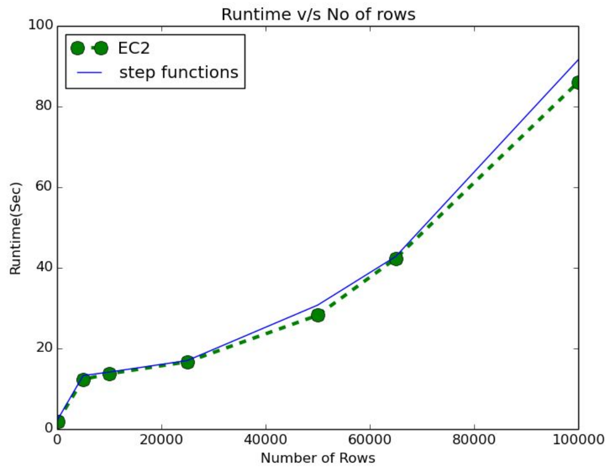

Fig 1: Runtime vs No. of Rows . The graph presents the comparison of AWS EC2 and AWS Step Functions for runtime v/s no of rows.

From Fig 1, we can clearly deduce that AWS EC2’s performance is better as compared to AWS Step Functions when dealing with huge data set.

Fig. 2 shows the throughput metric for each dataset volume.It could be observed that EC2 offers a better throughput when compared to the step functions.The performance of individual services namely transform(T), load(L), query(Q) is also evaluated which can be seen from Table 2.

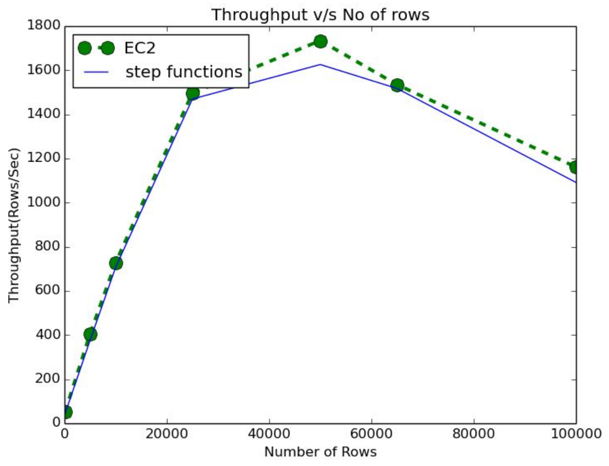

Fig 2: Throughput vs No. of Rows The graph presents the comparison of AWS EC2 and AWS Step Functions for throughput vs no. of rows.

Service	Stand alone Runtime(in sec)
Transform	305.259
Load	104.33
Query	81

Table 2: The table presents the comparison of all the Services when running Standalone forth both EC2 and Step Functions.

In table 2, the stand alone performance of each function is mentioned for a 5000 dataset volume. To our observations, transform was taking more time when compared to load and query.

In table 3, the cost incurred for our serverless application is shown. It can be clearly seen that most of the cost incurred for our application is by Relational Database. Thereby, the cost incurred for the lambda function was Zero. For the total application the cost incurred was $ 6.94816. We have observed that Amazon incurs different charges for GET and PUT requests on AWS S3.For our comparison between workflow mechanisms EC2 and step functions, we have observed that using step functions is cost effective.

Service	Cost($)	Usage
EC2	1.5892	274 hours
Step Functions	0.21465	>4000 state transitions
Lambda	0	11762 requests
S3	0.02398	13000 requests
Relational Database	5.45793	>10000 requests
Cloudwatch	1.2516	>5GB
Total	6.94816	-

Table 3: The table presents the cost of all the Services used for the application.

III. Conclusions

In our experiments, we obtained the solutions for our research questions. For the first question, we evaluated the freeze-thaw metric and came to the conclusion that AWS EC2 was performing better when compared to AWS Step functions. For the second question we could answer that AWS EC2 instance was performing better when compared to AWS step functions. We have run our experiments for data sets 100, 5000, 10000, 25000, 50000, 65000 and 100000. and as per the results, AWS EC2 shown better performance compared to AWS Step functions both in terms of runtime and throughput. This might be due to the orchestration over head of the workflows in step functions when compared to our own EC2 workflow implementation to invoke various lambda functions sequentially. For our workflow needs, it seems EC2 is a better choice for performing the application flow control when compared to step functions.However, in cost perspective, we have observed that defining the workflow using step functions is a lot cheaper compared to the costs incurred by running the workflow from an EC2 instance which answers the third research question.This is because we are paying for running the EC2 host even when we are not actually using it for running our client application..

IV. Acknowledgment

This research was facilitated by AWS cloud Educate credits and under the guidance of Wes Llyod. The term assignments were the rudimentary routes for exploring serverless FaaS platforms and grew our interest in the field of serverless computing.

V. References

[1] Zach Hill and Marty Humphrey “A Quantitative Analysis of High Performance Computing with Amazon’s EC2 Infrastructure: The Death of the Local Cluster?” in 10th IEEE/ ACM International Conference on Grid Computing (Grid 2009). Oct 13-15 2009. Banff, Alberta, Canada

[2] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S., Serverless Computing: An Investigation of Factors Influencing Microservice Performance, IEEE Int. Conf. on Cloud Engineering (IC2E 2018), Apr 17-20, 2018.

[3] Understanding AWS Lambda Coldstarts, https://www.iopipe.com/2016/09/understanding-aws-lambda-coldstarts/

[4] Amazon Elastic Compute Cloud (Amazon EC2)

[5] https://blog.boltops.com/2018/04/17/aws-step-functions-benefits-and-disadvantages

[6] https://www.hcltech.com/blogs/why-faas-aka-serverless-has-bold-future

[7] https://aws.amazon.com/step-functions/features/

[8] https://aws.amazon.com/aws-cost-management/

[9] https://en.wikipedia.org/wiki/Extract,_transform,_load