ETL Pipeline Project Report

Tanner Brown, Khanh Nguyen, Ali Nemati
University of Washington
Tacoma, Washington USA
tbrown13@uw.edu, rowuyen6@uw.edu, anemati@uw.edu

***Abstract* - We present four different implementations of an ETL application with microservices utilizing resources from Amazon Web Service such as AWS Lambda, API Gateway, Step Function, S3, Simple Queue Service. This application is serverless and written in different languages for its services.**

***Index Terms* - AWS, Cloud Computing, Go, Java, Lambda**

I. INTRODUCTION

In this project, the Lambda Amazon Web Service is used to implement an ETL Pipeline serverless application. It supports the case study of "Application Flow Control" with four different designs of the ETL pipeline: Client Flow Control, Microservice As Controller, AWS Step Function, and Asynchronous. The goal is to compare the overall cost and performance from different methods of ETL pipeline's implementations. All models have shown in fig. 1.

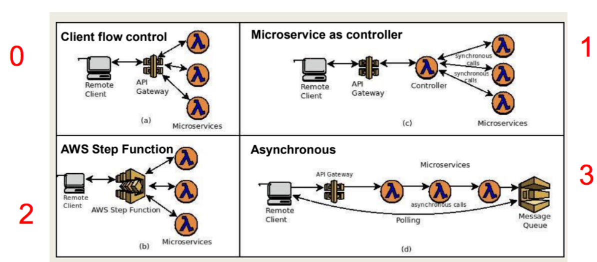

Fig. 1. Application flow control

II. ETL IMPLEMENTATIONS

A. Services:

1. Transform: We implement this service in the Go programming language because we wanted to experiment with how fast Go runs in a cloud environment in comparison to Java. This service accepts a filename and an S3 bucket from the user and then processes the file by modifying and appending some columns. It then uploads a new output file in another directory within the same bucket to be processed by the Load service. This service uses the standard Go library, with the addition of the Go AWS SDK and a third party library for String to Time conversion using custom time formats.

2. Load: This service is implemented in Java and uses SQLite with file-based database. When a request comes in with the location of a csv file in S3, load service will first check if the csv file exists locally or not. If the csv is not already under local /tmp directory, it will get the csv object from S3 and save locally. Next, this service will attempt to query from the database corresponding to this csv file (for example, 1000SalesRecords.csv will have sale_1000.db, 50000SalesRecords.csv will have sale_50000.db etc.), if the database doesn't exist, then a new database with a new table will be created and inserted data from the csv file. The local SQLite database is uploaded to S3 bucket for other services to use.

3. Extract: This service is implemented in Java and uses SQLite local database. It receives a request with the database name, table name, and location of the S3 bucket to download the database file from. After having the database on local disk under "/tmp" directory, it will query from this database and compile the results into two uniquely named files: <transaction-id>-filtering.csv and <transaction-id>-aggregation.csv. The result of filtering and aggregation will be uploaded to S3.

B. Different design methods

1. Client flow control: Client flow control is applied to limit the flow of data among a client and servers, or a server and another server in order to communicate and send messages by using Amazon API Gateway (AAG). AAG is a fully managed service that makes it so simple for developers to develop, publish, manage, monitor, and reliable APIs.

2. Microservice as a controller: Due to the performance that we noticed with the Transform service, we decided to use Go for the Microservice controller as well. The transform service showed a low memory requirement, and a fast "cold" startup time, both of which we determined to be crucial for the controller. Because this controller would run for the entirety of the three-service pipeline, it was important that it be efficient to minimize cost. This controller calls all three ETL services individually, and waits for a response. Upon successful completion, all three service responses are included in the controller's response.

3. AWS step function: Using AWS step function, three services "transform", "load", "extraction" are added into a pipeline as shown in figure 2.

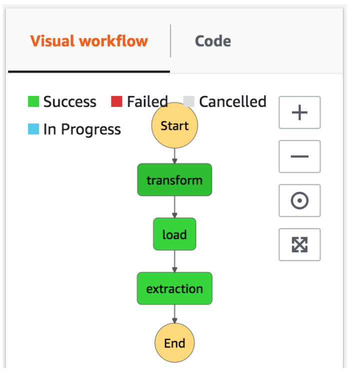

Fig. 2. AWS step function model

4. Asynchronous: For Asynchronous design method, a gateway is used to send a POST request and invoke the first lambda, which is transform service. When transform service is done processing the CSV file, it invokes the load lambda. Similarly, after load service has completed its work, it invokes the extraction lambda. In the extraction service, besides sending a regular Response, it will also send a copy of response data to SQS to be retrieved by the clients later.

III. EXPERIMENTAL RESULTS

All of the collected data are from warm starts. Memory Size and Max Memory Used are in MB, Billed Duration and Duration are in milliseconds.

A. Test 1: Individual service testing.

1. Transform: We tested each of the individual services on different data sizes to find optimum memory allocation is for 1,000 10,000 and 50,000 row files. These optimum values would then be applied to these services when testing the pipeline to keep our test parameters consistent. This service was the first to be tested, so it was tested before we discovered our troubles with the Load and Extraction services. Because of this, we were able to test Transform on additional file sizes. Testing showed that until the file sizes began to reach 500,000 rows, very small amounts of memory were sufficient to run this service. As a result, we initially set the Transform service to 128mb of memory, for the 1,000 10,000 and 50,000 row file sizes. After the first few tests, we increased the memory on the 50,000 row file to allow for better overall runtimes since the cost difference between the two was minimal.

2. Load: To find the best memory size and pricing for each of the data file size, load lambda is ran with different memory size. After selecting the best memory size, we run our combination tests using the chosen memory setting for each file size.
For 1000 rows csv file: Memory size of 512MB is used.

For 10,000 row csv file: The experiments show that the cost between 640MB and 1024 very close, however, cold start when using 1024MB is better than 640MB, so the 1024MB is chosen for 10,000 rows csv file instead of 640MB.

For 50,000 rows csv file: this is a good example of smaller memory size (512MB or 640MB) isn't always the cheapest, smaller memory size (512MB, 640MB) takes longer to finish the task and leads to cost more. The final memory size of 1024 MB is chosen for 50,000 rows csv.

3. Extraction: Our test result has shown on figure 3. It indicates that maximum memory size has selected is 1024 mb and maximum memory used is 119 mb for 50,000 rows. In addition, 29400 ms is for a maximum of build duration by having maximum duration 29321.21 ms from 1000 rows csv file.

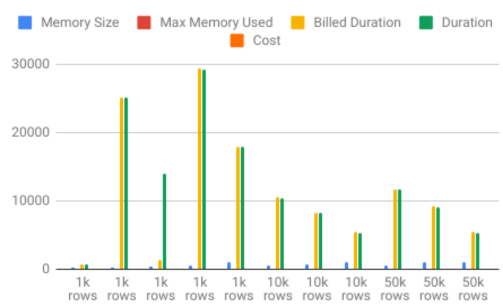

Fig. 3. The result of Extraction lambda function

These are final configurations for lambdas to use during testing:

- 1000 rows: transform (128MB), load (512MB), extraction (512)
- 10,000 rows: transform (128MB), load (1024MB), extraction (640MB)
- 50,000 rows: transform (256MB), load (1024MB), extraction (640MB)

B. Test 2: Client flow control

Table 4 has shown that the averages of the tests for the client flow control with the entire application: having better memory use in 1000 records with 1152 MB of maximum memory size. In addition, less cost and less billed duration occurred during our test for the same data files. Comparing 50,000 records to 1000 records illustrates that we would incur higher cost of memory usage for bigger csv files.

C. Test 3: Microservice as controller

Result data is shown in table 3. The "duration total" column displays the start to finish runtime of the service, and the cumulative column combines runtimes of all 3 services and the controller. The controller was able to easily operate at the minimum memory setting which contributed to times and overall prices that were lower than expected.

D. Test 4: AWS step function

Result data is shown in Table 1. Each column: memory used total, memory total, billed duration total, and cost is a combination of three services. The duration total is collected from the Step function final Elapsed time in milliseconds.

E. Test 5: Asynchronous

Result data is displayed in Table 2. Each column: memory used total, memory total, duration total, billed duration total, and cost is a combination of three services.

IV. CONCLUSIONS

We implemented 4 methods for the ETL pipeline and collect memory, timing, cost data for each combination as well as individual services in each combination. Step function seems to have the best runtime performance, but is more expensive. This method would be useful for large-scale projects with a budget that can afford it. The Asynchronous is the worst of all with runtime as well as cost as the csv file getting larger. However, perfecting this implementation could result in speeds and cost that could make this method viable in the right circumstances. The Client flow control method is the cheapest option since it doesn't have any additional cost besides the three lambda functions, however it doesn't have the best runtime compared to other methods. This indicates that it could be the best method for small-scale, low-budget projects. The Microservice controller is also quite cost effective, although more expensive than the gateway dues to the cost of an extra lambda for the controller, it's still cheaper than Step function or Asynchronous. The controller could potentially be much more effective if implemented asynchronously.

Overall understandings of this project are listed below:

1. Minimizing memory use is not always a most efficient use.
2. Even with maximum memory and timeout, lambda function might not be able to handle more than 500000 record files in our implementation.
3. Lambda Step Function is a great option for optimal runtime, but costly in the long run.
4. Many different methods for implementing a serverless pipeline exist, and the best option can vary depending on the needs of the user.

ACKNOWLEDGMENT

AWS Educate cloud credits are used for learning and completing this project.

REFERENCES

[1] aws.amazon.com, “AWS SDK for Java,” https://aws.amazon.com/sdk-for-java, 2016, [Online; accessed 12-December-2018].

[2] amazon.com, “AWS SDK for Go,” https://aws.amazon.com/sdk-for-go/, 2016, [Online; accessed 12-December-2018].

[3] aws.amazon.com, “How API Gateway Works,” https://aws.amazon.com/api-gateway/, 2016, [Online; accessed 12-December-2018].

[4] aws.amazon.com, “HowAPIGatewayWorks,” https://aws.amazon.com/api-gateway/, 2016, [Online; accessed 12-December-2018].

[5] T. Brown, “Transform Microservice Controller,” https://bit.ly/2GhX0gU,2018,[Online;accessed12−December−2018].

[6] K. Nguyen, “Load,” https://bit.ly/2rL5i73 , 2018, [Online; accessed 12-December-2018].

[7] A. Nemati, “Extraction,” https://bit.ly/2QVumGp ,2018, [Online; accessed 12-December-2018]

TOTALS	Memory Size Total	Max Memory Used Total	Billed Duration Total	Duration total	Cost
1k	1152	248	1500	1403.800	$0.000110006
1k		250	1300	1175.630	$0.000007712
1k		248	1000	879.040	$0.000005836
avg - 1k		248.667	1266.667	1152.823	$0.000041185
10k	1792	314	2800	3068.000	$0.00011521
10k		315	3100	3360.000	$0.00001885
10k		315	2600	2733.000	$0.00001500
avg - 10k		314.667	2833.333	3053.667	$0.00004968
50k	1920	558	8000	7998.000	$0.00015211
50k		559	7500	7788.000	$0.00005408
50k		559	6000	6145.000	$0.00005250
avg - 50k		558.667	7166.667	7310.333	$0.00008623

Table 1. Data collected from Step Function in total for each file size.

TOTALS	Memory Size Total	Max Memory Used Total	Billed Duration Total	Duration total	Cost
1k	1152	245	2700	2567.89	$0.00001313
1k		245	2200	2082.73	$0.00000771
1k		250	2100	1917.97	$0.00001000
avg - 1k		246.67	2333.33	2189.53	$0.00001028
10k	1792	307	5300	5145.72	$0.00003291
10k		310	4300	4098.73	$0.00002624
10k		310	4400	4264.96	$0.00002603
avg - 10k		309	4666.67	4503.14	$0.00002840
50k	1920	565	12800	12698.39	$0.00009088
50k		572	12800	12663.26	$0.00010586
50k		576	10500	10387.55	$0.00009084
avg - 50k		571	12033.33	11916.40	$0.00009586

Table 2. Data collected from Asynchronous in total for each file size

TOTALS	Memory Size	Max Memory	Billed Duration	Dur (cumulative)	Duration total	Cost
1k	1196	281	3270	3719.42	1642.02	$0.00009006
1k		277	3300	3027.69	1617	$0.00001438
1k		277	2300	2030.78	1116.91	$0.00000917
avg - 1k		278.33	2956.67	2925.96	1458.64	$0.00003787
10k	1920	334	8500	8415.05	4282.4	$0.00003123
10k		337	7300	7095.99	3643.39	$0.00002811
10k		336	6300	6055.77	3125.69	$0.00002165
avg - 10k		335.67	7366.67	7188.94	3683.83	$0.00002700
50k	2048	617	16200	15970.32	8073.65	$0.00006334
50k		618	15400	13576.66	7665.88	$0.00006063
50k		618	15500	15372.04	7778.51	$0.00006125
avg - 50k		617.67	15700	15475.19	7839.35	$0.00006174

Table 3. Data Collected from Microservice Controller in total for each file size. Note this table includes a column titled *Dur (cumulative)*, which includes the microservice in the overall runtime.

TOTALS	Memory Size	Max Memory	Billed Duration	Duration total	Cost
1k	1152	270.00	2400.00	2433.79	$0.00001230
1k		257.00	2200.00	2071.07	$0.00000917
1k		263.00	4000.00	1888.58	$0.00002063
avg - 1k		263.33	2866.67	2131.15	$0.00001403
10k	1792	500.00	3300.00	3202.62	$0.00001187
10k		503.00	2700.00	2536.40	$0.00000895
10k		365.00	9300.00	9124.97	$0.00002435
avg - 10k		456.00	5100.00	4954.66	$0.00001506
50k	1920	571.00	10900.00	12498.05	$0.00006255
50k		566.00	7400.00	12963.36	$0.00005254
50k		586.00	11600.00	12501.52	$0.00005421
avg - 50k		574.33	9966.67	12654.31	$0.00005643

Table 4. Data collected from Api Gateway in total for each file size.

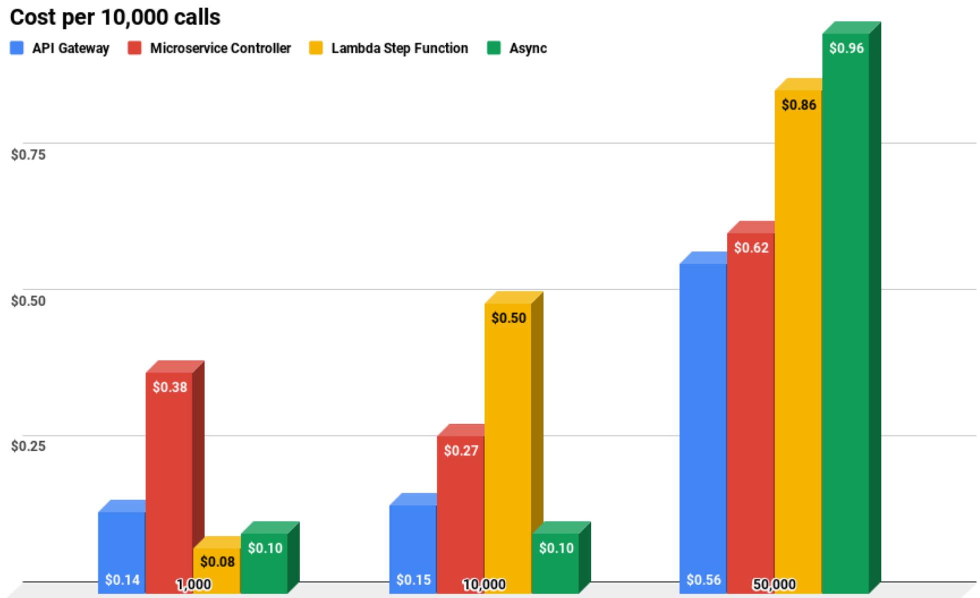

Table 5. Comparison of cost to run each design implementation 10,000 times on each file size.

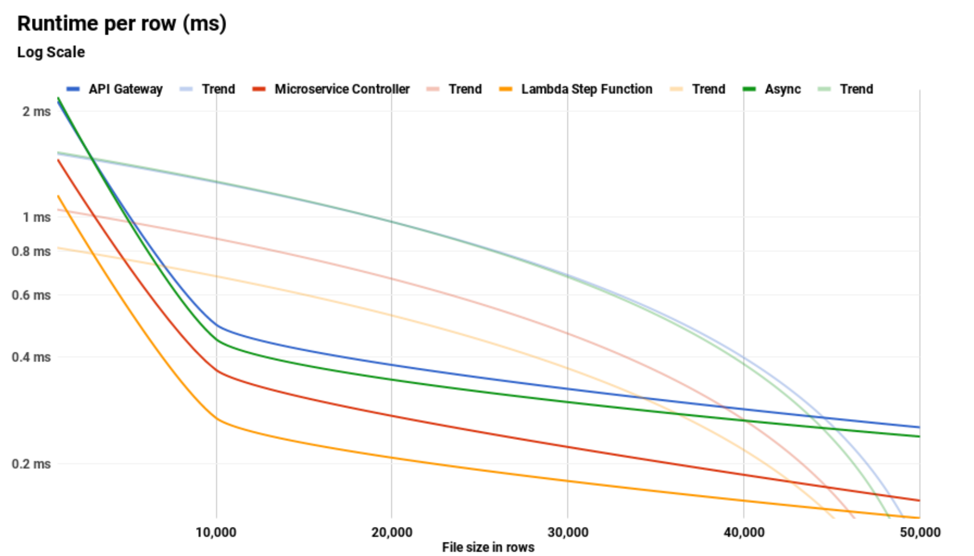

Table 6. Comparison with trendlines of time to execute individual rows by service and file size.

		API GATEWAY		MICROSERVICE AS CONTROLLER		LAMBDA STEP FUNCTION		ASYNCHRONOUS	
		Time	+/-	Time	+/-	Time	+/-	Time	+/-
1,000	RUNTIME (start to finish)	2131.15	+978.33	1458.64	+305.82	1152.82	-305.82	2189.53	+1036.71
	RUNTIME (combined)	N/A		2925.96		N/A		N/A	
	runtime per row (ms)	2.131	+0.978	2.926	+1.773	1.153	-0.978	2.190	+1.037
	BILLABLE TIME	2866.67	+1,633.33	2956.67	+1,690	1266.67	-1066.66	2333.33	+1066.66
	COST TOTAL per call	$0.00001403		$0.00003787		$0.00000785		$0.00001028	
	Cost 10,000 calls (est)	$0.14	+$0.06	$0.38	+$0.30	$0.08	-$0.02	$0.10	+$0.02
	COST - SERVICES	N/A		$0.000003189		N/A		N/A	
10,000	RUNTIME (start to finish)	4954.66	+2,263.48	3683.83	+992.65	2691.18	-992.65	4503.14	+1811.96
	RUNTIME (combined)	N/A		7188.937		N/A		N/A	
	runtime per row (ms)	0.495	+0.226	0.368	+0.099	0.269	-0.226	0.450	+0.181
	BILLABLE TIME	5100.00	+2,266.67	7366.67	+4,533.34	2833.33	-1,833.34	4666.67	+1833.34
	COST	$0.00001506		$0.00002700		$0.00004968		$0.00002840	
	Cost 10,000 calls (est)	$0.15	+$1.59	$0.27	-$0.01	$0.50	+$0.23	$0.28	+$0.01
	COST - SERVICES	N/A		$0.000016363		N/A		N/A	
50,000	RUNTIME (start to finish)	12654.31	+5,663.68	7839.35	+848.72	6990.63	-848.72	11916.40	+4925.77
	RUNTIME (combined)	N/A		15475.188		N/A		N/A	
	runtime per row (ms)	0.253	+0.113	0.157	+0.017	0.140	-0.113	0.238	+0.098
	BILLABLE TIME	9966.67	+2800.33	15700	+8533.33	7166.67	-4866.66	12033.33	+4866.66
	COST TOTAL	$0.00005643		$0.00006174		$0.00008623		$0.00009586	
	Cost 10,000 calls (est)	$0.56	-$0.06	$0.62	+$0.06	$0.86	+$0.3	$0.96	+$0.4
	COST - SERVICES	N/A		$0.000016363		N/A		N/A	

Table 7. Comparison of average values across all 4 control types and file sizes.
Note: *Combined runtime* includes the runtime of the Microservice Controller.