AWS Lambda — Composition and Control Flow

AB Testing of Service Compositions and Control Flow Options for a TLQ Pipeline

Jered Wiegel
School of Engineering and Technology
UW Tacoma
Tacoma, WA, USA
jwiege@uw.edu

Hyunggil Woo
School of Engineering and Technology
UW Tacoma
Tacoma, WA, USA
hyunggilswoo@gmail.com

Hui Wagner
School of Engineering and Technology
UW Tacoma
Tacoma, WA, USA
huiwagne@uw.edu

Joshua Barbee
School of Engineering and Technology
UW Tacoma
Tacoma, WA, USA
joshbar@uw.edu

ABSTRACT

This report compares thirteen implementations of a serverless TLQ pipeline on AWS Lambda, with an implementation for each meaningful service composition-control flow pair for four composition options for the three constituent services and four control flow options. The round-trip time from the initial invocation of the pipeline to the final retrieval of results is compared for each of these implementations to distinguish the overhead associated with each pair, and the performance variation (CV) is compared to determine which implementations offer the most stable performance. We found composing the services into a single function provides the lowest average round-trip time and the lowest performance variation. With more functions (less composition), having the functions call each other in a chain within Lambda provides the best performance with two functions, and an extra Lambda controller function provides the best performance with a three-function pipeline.

CCS CONCEPTS

• Computer systems organization~Architectures~Distributed architectures~Cloud computing • Software and its engineering~Software creation and management~Designing software~Software design tradeoffs

KEYWORDS

Cloud Computing, Service Composition, Control Flow, AB-Testing, TLQ, ETL, AWS, AWS Lambda, SAAF

1 Introduction

This report details a comparative analysis of a TLQ pipeline regarding service compositions and control flow options. The TLQ pipeline involves the following three services: Transform/T (load sales data from a CSV file on S3 and modify the local copy), Load/L (generate a SQLite database from the modified CSV data), and Query/Q (load the SQLite database and execute some query). The original input CSV file is retrieved from a single S3 bucket for all implementations, and some implementations additionally write data to the same S3 bucket for subsequent functions to read. All code for the Lambda functions is written in Python and uses the Serverless Application Analytics Framework (SAAF) to distinguish warm and cold function instances [1], and the calls that start pipeline invocations are made by custom Bash scripts. The exact details and reasoning for each implementation are given in Section 2.

1.1 Research Questions

We investigate the following research question:

RQ-1: How do different combinations of service compositions and control flow options for a microservices application impact round-trip time and performance variation when hosted using FaaS platforms? We investigate the alternate compositions of our Transform, Load, and Query services, as well as the control flow options of a client controller, Lambda-function controller, Step Function, and having each function synchronously call the next.

2 Case Study

This serverless application consists of a data pipeline of the three services T, L, and Q, using the Python *pandas* package for CSV loading and manipulation in T and L, the *sqlite3* Python package for creating and managing short-term databases on ephemeral storage, and AWS' *boto3* Python package for communication between two Lambda functions and between a Lambda function and S3. The three services are hosted on AWS Lambda as one, two, or three separate functions, and additional resources such as a fourth Lambda function or an AWS Step Function are also associated with some implementations for different control flows. Thirteen distinct implementations of the pipeline are considered in this report.

Hosting such a data processing pipeline on a FaaS service provides three notable advantages: (1) simplified pipeline setup by simply specifying the application code and its required resources — without configuring a VM and manually installing dependencies, (2) on-demand scaling based on the number of requests to the pipeline, and (3) no always-on costs, which is beneficial in periods when the pipeline is infrequently called [2].

2.1 Design Tradeoffs

Thirteen combinations of service compositions and control flow options are evaluated. The four considered compositions include T_L_Q (which denotes the T, L, and Q services all being in separate Lambda functions), TL_Q, T_LQ, and TLQ (which denotes all services composed into a single Lambda function). The four control flow options are LC (Laptop Client; a remote client directly calls each function in sequence and manages inter-function communication outside of the cloud), WL (Within Lambda; each function automatically invokes the next function in the pipeline), CF (Controller Function; an additional Lambda function calls each function of the pipeline in sequence), and SF (Step Function; an AWS Step Function calls the pipeline's functions in sequence). Each pair of a service composition and a control flow option is considered, excluding WL, CF, and SF for TLQ because only one function exists for this composition. Each implementation is denoted by its control flow option, followed by a period, followed by its composition (e.g., LC.T_L_Q). These combinations are shown in Table 1.

Control Flow → Composition ↓	LC	WL	CF	SF
T_L_Q	LC.T_L_Q	WL.T_L_Q	CF.T_L_Q	SF.T_L_Q
TL_Q	LC.TL_Q	WL.TL_Q	CF.TL_Q	SF.TL_Q
T_LQ	LC.T_LQ	WL.T_LQ	CF.T_LQ	SF.T_LQ
TLQ	LC.TLQ			

Table 1: Implemented Control Flow-Composition Pairs

LC.TLQ and other implementations with fewer separate functions are expected to have lower round-trip time and thus higher throughput, as less communication overhead is required between the services. The LC implementations are expected to have higher round-trip times and thus lower throughput than implementations with the same compositions but different control flow options, as additional latency will be added by additional requests and responses passed between the client and the cloud.

2.2 Serverless Application Implementation

The 1-3 Lambda functions for the services accept JSON data to specify the S3 bucket and the S3 key(s) to read/write data to. The *pandas* package is used to read, modify, and save local copies of CSV data for the T and L services and converts CSV data to a SQL table for L. All 1-3 of the service functions return SAAF data to validate runs as warm or cold, except for LC.TLQ, due to all attempts to return SAAF data from the function consistently returning errors and thus preventing any testing with the intended behavior. Subsequently, the warm and cold tests for LC.TLQ cannot be definitively validated as warm or cold.

The Transform service in all implementations initially reads the specified CSV file from the specified bucket in S3., and the following example transformations are applied: add an Order Processing Time column (the difference between Shipping Date and Order Date), map the Order Priority column letter codes to full words, add a Gross Margin column (Total Profit divided by Total Revenue), and remove rows with duplicate Order ID values. The modified data is then saved to S3 with a consistent naming convention (for the T_LQ and T_L_Q compositions) or kept in memory (for the TL_Q and TLQ compositions) to be accessed by the Load service.

The Load service reads CSV data from a request-specified S3 location (for T_LQ and T_L_Q) or from memory (for TL_Q and TLQ), uses *pandas* and *sqlite3* to create a SQLite database table in ephemeral storage with the same data, and either puts the database file to S3 (for TL_Q and T_L_Q) or keeps it only in ephemeral storage (for T_LQ and TLQ) to be accessed by the Query service.

The Query service reads a SQLite database file from a request-specified S3 location (for TL_Q and T_L_Q) or uses the existing file in ephemeral storage and the existing database connection (for T_LQ and TLQ), executes a SQL query based on the aggregations and filters and columns to group by specified as JSON in the request body, and returns the query results as JSON in the request body. An exception to returning the results in the request body is in the compositions T_L_Q and TL_Q, which instead store the resulting JSON to S3; this behavior was accidentally retained when the tests were run, but all of the several on-cloud runtime checks with SAAF between this incorrect handler and a corrected handler (after the recorded tests were run) — to determine the severity of this mistake — showed differences of less than 1 second. The tests were not redone due to time constraints and the likely insignificant differences.

The SF implementations only return the JSON output for the final function in the pipeline (Q in T_L_Q and in TL_Q, or LQ in T_LQ).

The implementations are intended to support only synchronous pipeline calls. Also, while the services' handlers allow input and output filenames to be specified, allowing different pipeline invocations to read from and write to different files to support multiple users and different input data, the tests described below did not use this feature.

All implementations are currently only configured to support calling all of the functions in sequence, with no support for reusing the result of Transform for multiple Load calls or Load for multiple Query calls.

2.3 Experimental Approach

All Lambda functions (including the controller functions for CF) were configured with 1300 MB of RAM (which is sufficient for the input dataset's size of 178 MB but less than the 1769 MB RAM threshold for receiving a full vCPU per call) [3], a timeout of 3 minutes, 512 MB of ephemeral storage to store the database file in all implementations, the AmazonS3FullAccess role for reading and possibly writing to S3, and the AWSSDKPandas-Python311 layer for data processing with *pandas*. The services' functions for WL and the controller for CF also used the AWSLambdaRole and AWSLambdaExecute roles.

All Lambda functions and S3 buckets were within the us-east-2 region; a single region was used to represent a realistic scenario for deploying a multi-resource application with minimal network communication overhead. No VPC or availability zone information were configured.

Local clients outside of the cloud made requests to the Lambda functions or Step Functions with the AWS CLI.

While on-cloud runtime could not be recorded for all tests, the round-trip time from the client's call to the pipeline to the client receiving the final results from the final service was always recorded. The round-trip time is relevant for comparing the implementations because the mechanisms for communication between the services and thus potentially the total overhead for communication differ between each composition-control flow pair; round-trip time captures these differences to compare the performance of the various control flow and composition options.

A sales dataset with 1.5 million records [4] was uploaded to an S3 bucket to use as the input to Transform for all implementations. Custom Bash scripts were created to run each pipeline with the appropriate JSON input, and another Bash script was created to run a specified pipeline's script 100 times concurrently, recording the JSON output and round-trip time for each call [5]. For the LC tests, the functions T, L, Q, TL, LQ, and TLQ were tested independently and had their round-trip times added in the appropriate combinations to acquire data for each composition; the other control flow options ran the whole pipeline for each test.

Each pipeline was first tested with a single call to ensure proper output, followed by 100 concurrent warm tests for the pipeline, followed by 100 concurrent cold tests for the pipeline after 40 minutes of inactivity for the functions to cool. After acquiring the data for all 100 runs, the "newcontainer" attribute of the JSON output was checked to confirm that the warm tests used existing containers and that the cold tests used new containers, and any tests intended to be warm/cold that were not correctly warm/cold were filtered out of the results to ensure the validity of warm and cold results. This check could not be performed for LC.TLQ, so these warm and cold results are presented as-is. Also, Q in the CF implementations was almost always warm while the other functions were cold. This data is kept.

The same S3 bucket, input file, CSV filename for T's output, and SQLite database filename were used for all concurrent tests. Because S3 provides strong read-after-write consistency [6], and all input data is identical, and all data is manipulated in ephemeral storage before being pushed to S3 all at once, data is not corrupted by using the same database filename for concurrent tests. Concurrent tests were used to allow large amounts of data to be generated soon enough.

3 Experimental Results

RQ-1

LC.TLQ is used as a baseline because it has the least communication overhead in all cases.

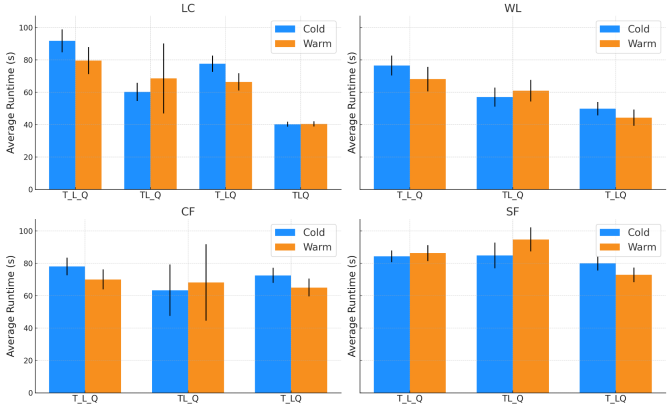

Figure 1: Warm and Cold Average Round-Trip Time (with Standard Deviation) for Each Implementation.

	LC	WL	CF	SF
T_L_Q	97/128	89/90	73/94	114/110
TL_Q	70/50	41/42	69/58	135/111
T_LQ	64/93	23/24	61/81	80/99

Table 2: Percent Difference (100[B-A] ÷ A) in Average Round-Trip Time. LC.TLQ vs. All Others (Warm/Cold)

	LC	WL	CF	SF
T_L_Q	173/106	191/113	131/86	47/15
TL_Q	720/149	185/175	799/567	103/147
T_LQ	113/71	195/127	121/68	62/49

Table 3: Percent Difference (100[B-A] ÷ A) in CV of Round-Trip Time. LC.TLQ vs. All Others (Warm/Cold)

Figure 1 shows that all implementations are slower and have higher standard deviations than LC.TLQ. Table 2 further shows that all implementations are at a minimum 23% slower on average than LC.TLQ for both cold and warm round-trip times, and the compositions with two functions in each column (i.e., for each control flow option) except SF show faster round trips than the compositions with three functions. This trend is consistent with the reduced communication overhead between the services when there are fewer separate functions. The higher times for SF across all categories mean that SF's overhead for the state machine greatly exceeds the overhead for any other control flow option for a given composition.

Additionally, Table 2 shows that WL is much faster than the other control flow options for the higher compositions, meaning that — with more composition — the overhead for LC's network latency and CF's and SF's additional separate control resources exceeds WL's overhead for each function directly invoking the next function; except, when no composition is used (T_L_Q), CF becomes the optimal choice, meaning that WL's overhead for requiring each function to load the *lambda* resource with *boto3* and then invoke the next function exceeds CF's overhead for the additional controller function (which only has to load the *lambda* resource once and calls all functions from the same controlling instance) when the number of functions in the composition increases to three.

Finally, considering the performance variation of the implementations, the large CV percent differences in Table 3 show that the LC.TLQ had by far the smallest performance variation across its tests. Because there was sufficient memory for the processed data to be stored in memory many times over, this means that the network latency for a single round trip between the off-cloud clients used to launch the pipeline (in the Seattle metropolitan area) and the cloud resources in the us-east-2 region vary less significantly than the total overhead for any other option because (1) for the other control flow options, more overhead variables related on-demand cloud resources are added to the network latency and (2) for all options other than LC.TLQ, the higher overall overhead becomes a larger fraction of the round-trip time and evidently varies more than the processing time of the services (which perform the same operations on the same input data in all the tests). Additionally, the SF CV values being lower than the values for the other control flow options indicates that the overhead for AWS Step Functions — although larger — is more consistent than the overhead for any other control flow option for a given composition (and across compositions).

4 Conclusions

Thirteen implementations for the combinations of the control flow options LC, WL, CF, and SF and the service compositions T_L_Q, TL_Q, T_LQ, and TLQ were tested and compared for a TLQ pipeline. For each implementation, round-trip time for the entire pipeline and the coefficient of variation for this time were captured to compare these implementations by overhead and performance variation, respectively, to answer RQ-1.

The experiments show that LC.TLQ (composing all services into a single function) provided by far the lowest round-trip times and the most consistent performance among all compositions for the pipeline. When other compositions are used, the control flow option with the lowest average round-trip time is WL (the service functions calling each other in a chain), until the number of functions increases to three, at which point CF (an additional Lambda function calling the service functions in sequence) becomes the best choice.

ACKNOWLEDGMENTS

This project was completed with instructions, guidance, and the sales dataset from Professor Wes Lloyd.

REFERENCES

[1] Robert Cordingly. et al. 2020. The Serverless Application Analytics Framework. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. doi:10.1145/3429880.3430103.

[2] Amazon Web Services. 2023. AWS Lambda Pricing. *Serverless Computing — AWS Lambda*. Available at: https://aws.amazon.com/lambda/pricing/.

[3] Amazon Web Services. 2023. Configuring Lambda function options. *AWS Lambda Developer Guide*. Available at: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-memory-console.

[4] Wes Lloyd. 2018. 1500000 Sales Records.zip. Available at: https://faculty.washington.edu/wlloyd/courses/tcss562/project/tlq/sales_data/1500000%20Sales%20Records.zip.

[5] Jered Wiegel. 2023. Cloud-Computing-Project. Available at: https://github.com/Sinoffate/Cloud-Computing-Project.

[6] Amazon Web Services. 2023. Amazon Simple Storage Service. *Amazon S3 User Guide*. Available at: Amazon Web Services. 2023. Configuring Lambda function options. AWS Lambda Developer Guide. Available at: https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-memory-console.