Implications of Alternative Serverless Application Control Flow Methods

Sterling Quinn, Robert Cordingly, Wes Lloyd
School of Engineering and Technology
University of Washington
Tacoma, WA USA
sterl1789@gmail.com, rcording@uw.edu, wlloyd@uw.edu

ABSTRACT

Function-as-a-Service or FaaS is a popular delivery model of serverless computing where developers upload code to be executed in the cloud as short running stateless functions. Using smaller functions to decompose processing of larger tasks or workflows introduces the question of how to instrument application control flow to orchestrate an overall task or workflow. In this paper, we examine implications of using different methods to orchestrate the control flow of a serverless data processing pipeline composed as a set of independent FaaS functions. We performed experiments on the AWS Lambda FaaS platform and compared how four different patterns of control flow impact the cost and performance of the pipeline. We investigate control flow using client orchestration, microservice controllers, event-based triggers, and state-machines. Overall, we found that asynchronous methods led to lower orchestration costs, and that event-based orchestration incurred a performance penalty.

CCS CONCEPTS

• **Computer systems organization** → Cloud computing;

KEYWORDS

Serverless Computing, Frameworks, Function-as-a-Service, Performance Evaluation, Programming Languages

ACM Reference format:

Sterling Quinn, Robert Cordingly, Wes Lloyd. 2021. Implications of Alternative Serverless Application Control Flow Methods. In Proceedings of 7th International Workshop on Serverless Computing (WoSC7) 2021. ACM, Virtual Event, Canada, 6 pages.

1 Introduction

Function-as-a-Service (FaaS) is a "serverless" cloud computing delivery model where developers provide code that is run in isolated sandboxes provisioned and managed on demand by the cloud provider. These sandboxes known as function instances, provide scalable infrastructure for the code to run that is always available, and resistant to failure [1][2]. This paper leverages AWS Lambda as the FaaS platform combined with additional services to facilitate investigations extending on work from [4].

Function-as-a-Service platforms have recently become a popular option for hosting microservices. These platforms excel at hosting and scaling independent microservices. When adopting these platforms to host larger applications that aggregate microservices together to constitute a task or workflow, it is necessary to orchestrate the service-based application's control flow given the distributed nature of where functions execute in the cloud. We refer to methods that instrument function call chains in serverless applications as "serverless application control flow". Different options are possible for orchestrating application control flow, but it is unclear initially what tradeoffs exist with each option. When choosing between different methods of control flow cost, performance, capability, and ease of development are factors to consider. Methods for remotely triggering serverless calls can have associated charges, while calling those functions from a desktop or laptop client are generally free. A cloud-based virtual machine can provide a low-latency server-side client to instrument control flow but will also incur always-on costs. If speed is the primary consideration, how latency varies between the different control flow options is not intuitive. The capabilities of each method must also be considered. If data must be exchanged between functions, some control flow approaches facilitate the exchange more easily than others. Given that developers time is extremely valuable, the most economical option that requires the least effort to implement may be preferred.

In this paper, we examine implications of four alternate methods of serverless application control flow to orchestrate an Extract-Transform-Load (ETL) style data processing pipeline.

Client Orchestration: This method involves calling the individual pipeline steps *synchronously* from the deve loper's computer or from a centralized VM provisioned in the cloud.

Microservice Controller: This method involves provisioning an additional serverless function to *synchronously* orchestrate execution of the serverless functions in the pipeline.

State-Machine: This *asynchronous* method offered by the cloud provider enables developers to define a state-machine to describe function transitions and data flow. The cloud provider

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
WoSC '21, December 6, 2021, Virtual Event, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9172-6/21/12…$15.00

instruments a client based on the state-machine definition to invoke functions for each step of the pipeline or workflow.

Event-Based Triggers: Cloud providers offer methods to define 'rules' that trigger serverless functions when events occur. These rules can be used to *asynchronously* orchestrate a pipeline.

1.1 Research Questions

This paper investigates the following research questions:

RQ-1 (**Performance**): What are the performance implications for alternate methods of serverless application control flow? Specifically, how does pipeline runtime, latency, and data processing throughput vary?

RQ-2 (**Cost**): What are the cost implications for alternate methods of serverless application control flow?

RQ-3 (**Cold Start**): What are the implications for cold start for alternate methods of serverless application control flow? Specifically, how does pipeline runtime, latency, and data processing throughput vary when pipelines are run from a cold state vs. warm?

RQ-4 (**Memory**): What are the implications for memory reservation size for alternate methods of serverless application control flow? Specifically, how does pipeline runtime, latency, and data processing throughput vary when functions are deployed with different memory sizes?

RQ-5 (**Microservice Controller**): What are the implications for performance and cost for alternate memory reservation sizes and controller programming languages for the microservice controller application control flow pattern?

2 Background and Related Work

This paper leverages an ETL-style data processing pipeline from [4] extending prior work that investigated performance and cost implications of programming language choice for serverless data processing pipelines. Here we examine alternate control flow designs to orchestrate steps of a data processing pipeline. Our efforts specifically aim to quantify cost and performance implications of alternative control flow approaches.

Previously, López et al. examined different platform specific FaaS orchestration systems [6] for application control flow. López examined AWS Step Functions, Azure Durable Functions, and IBM Composer. Their work focused on cloud platform specific orchestration tools, and did not compare other control flow methods. López quantified overhead by subtracting the sum of the runtime of individual workflow functions from the runtime of an aggregate function which combined all of a workflows functions into one. López found that overhead grew linearly with the number of functions in the sequence, and that overhead grew exponentially with the number of parallel functions. They concluded that AWS Step Functions was the most mature and performant orchestration service. The offerings on other platforms were still in experimental phases when the paper was published limiting results of the cost comparison. A key limitation was that the experiments used a function that slept for one second that did not perform computational work. In our study, we leverage a variant of the ETL data processing pipeline with various dataset sizes to gain insight into how alternate control flow methods impact the performance of tasks with different runtimes.

3 Serverless Application Control flow

Methods to orchestrate serverless application control flow can be classified as synchronous or asynchronous. Synchronous methods involve the caller maintaining an active connection to the FaaS platform until a result is received. This provides the advantage of having an immediate awareness of failure, but also requires that the client dedicate an idle thread to wait for a response. Asynchronous methods make a call to initiate function execution without maintaining an open connection. The client either polls for the final result or subscribes to a notification service which pushes the result to the client on function completion. Asynchronous invocations free computing resources by releasing client threads that must wait for a result. Developers must determine an appropriate polling interval that considers the round-trip latency between the client and FaaS platform to strike a balance between retrieving results and not wasting CPU cycles.

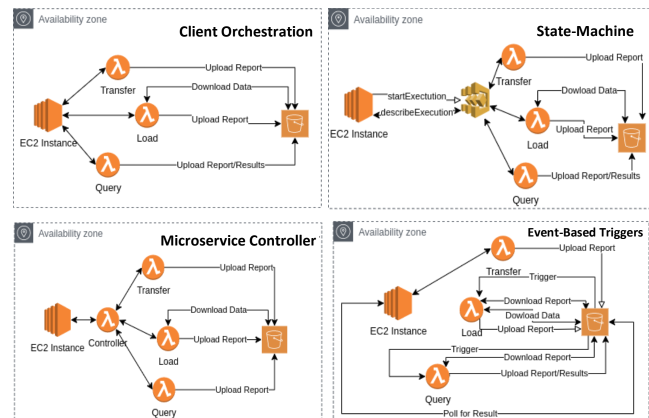

Figure 1: Serverless Application Control Flow Architectures: (*top left*) Client Orchestration, (*top right*) State-Machine, (*lower left*) Microservice Controller, (*lower right*) Event-Based Triggers

In this paper, we investigate four alternate approaches (Figure 1) for serverless application control flow: Client Orchestration, State-Machine (e.g. AWS Step Functions), Microservice Controller, and Event-Based Triggers (e.g. S3 triggers).

For *Client Orchestration*, we performed synchronous control flow from a cloud based VM using scripts derived from the Python-based FaaS Runner tool. In practice, client-side orchestration could involve the use of any client including personal desktops or laptop computers. To minimize latency and improve reproducibility of results in our experiments, we employed c5.xlarge instances in the same region and availability zone as our FaaS functions. We used larger c5 instances to obtain more vCPUs as needed for concurrent testing. In addition, all experiments were performed from the same EC2 instance. Function orchestration performance here will be limited by the

VM's available resources (e.g. vCPUs and memory). For Client Orchestration, we include the cost of the VM when calculating the total cost of the data processing pipeline. VM costs for instrumenting control flow will scale with FaaS pipeline runtime.

For *State-Machines*, we leveraged AWS Step Functions, a service provided by Amazon to instrument server-side asynchronous control flow. When invoked, the client must poll Step Functions to learn when result(s) are available. Step Functions enables developers to define workflows using an easy-to-use GUI, supporting advanced constructs like if statements, while handling data passing between functions enabling complex workflows. Step Function bills per function transition on top of AWS Lambda's charges. Transitions include a pipeline's start and end, meaning a three-step pipeline requires four transitions.

For the *Microservice Controller* control flow approach, we implemented a separate FaaS function to provide server-side synchronous control flow to orchestrate pipelines. We developed controllers in both Java and Python to compare cost and performance differences. The Microservice Controller approach suffers from the double billing problem [5], as cost is incurred for both the function accomplishing the work and the controller. This additional cost can be minimized as the controller function can be provisioned with minimal memory. Specifically, we sought to investigate the performance vs. cost implications of function orchestration using low memory (i.e. cheap) controllers.

Most cloud providers offer event-based triggers to invoke FaaS functions. For the *Event-Based Triggers* approach, we leveraged Simple Storage Service (S3) events to trigger Lambda functions. This way we orchestrate control flow by trapping S3 events occurring on buckets and objects. The event handling approach is asynchronous, requiring FaaS function response JSON objects to be pushed to, and then pulled from S3 for each transition adding cost by increasing function runtime compared to synchronous approaches. Event-Based Triggering additionally incurs costs for S3 operations which are billed per operation. For our data processing pipeline the size of the JSON response objects published to S3 were constant resulting in a fixed cost for control flow using the Event-Based Triggers approach regardless of the size of the dataset processed. This control flow approach was the only one requiring refactoring to instrument the pipeline due to differences in how data was exchanged between functions. This coupled the control flow approach to the function implementation reducing reusability without first refactoring.

4 Methodology

4.1 Evaluation Metrics

To analyze control flow approaches we leveraged a variety of metrics which are described in table I.

4.2 ETL Serverless Data Processing Pipeline

In this paper, we investigated alternate control flow approaches using the three-step ETL-style data processing pipeline from [4]. Our ETL-style pipeline presents a workflow consisting of sequential steps without conditional branching. This pipeline consists of three functions: a *Transform* function to format and convert data, a *Load* function to load data into the serverless Aurora MySQL database, and a *Query* function to perform a series of SQL queries against the backend Aurora MySQL database. Our pipeline processed sales data including product order details, transaction pricing, and customer metadata from CSV files stored in Amazon S3 ranging from 100 to 500,000 rows. Please refer to [4] for additional information regarding the Transform-Load-Query (TLQ) pipeline.

TABLE I. SERVERLESS APPLICATION CONTROL FLOW METRICS

Metric	Description
Pipeline runtime	End-time of last function minus start-time of first function. Equivalent to pipeline elapsed time. Includes function execution time and function transition time
Client runtime	Client-to-server round-trip time to execute the full pipeline. For asynchronous control flow, the round-trip time reflects when the client learns the pipeline has completed through polling or push notifications.
Function runtime	Function execution time excluding transition time.
Latency	This is latency of the control flow approach calculated as pipeline runtime minus the combined function runtimes. This reflects only the transition time between functions.
Billed amount	The total application hosting costs including FaaS functions and any additional control flow charges.
Data throughput	Measured in data rows processed per second. Represents the data processing velocity of the pipeline.

4.3 Tools and Platforms

To investigate serverless application control flow we leveraged the AWS Lambda FaaS platform and related cloud services [7]. To instrument experiments and benchmark performance we leveraged the Serverless Application Analytics Framework (SAAF) and the FaaS Runner tool [8]. SAAF supports profiling performance, resource utilization, and infrastructure metrics for FaaS workloads deployed to AWS Lambda written in Java, Go, Node.js, and Python. Programmers include the SAAF library and a few lines of code to enable SAAF profiling. SAAF collects metrics from the Linux **/proc** filesystem (Linux procfs) appending them to the JSON payload returned by the function instance. FaaS Runner is a client-side application for defining and executing experiments that works in conjunction with SAAF.

4.4 Experiments

To investigate control flow approaches we conducted the following experiments. Dataset sizes refer to rows of CSV data.

EX-1. Overall Performance Comparison

We performed 11 runs of the TLQ pipeline for each control flow method, with 100, 1,000, 5,000, 10,000, 50000, 100,000, and 500,000 row datasets.

EX-2. Cold Performance Comparison

We performed 10 runs of the TLQ pipeline for each control flow method using a 100,000-row dataset, with 45 minutes of sleep time between runs. We set the Aurora Serverless database to stay active for the duration of the experiment to isolate cold start latency to only the Lambda functions.

EX-3. Lambda Functions Memory Size Comparison

We performed 11 runs of the TLQ pipeline for each control flow method using a 100,000-row dataset, for each of the following memory settings: 512, 768, 1024, 1536, and 2048 MB.

EX-4. Microservice Controller Memory & Language Comparison

We performed 11 runs of the TLQ pipeline each using the Java and Python microservice controllers. We performed the experiment using the 100,000-row data set with 128, 192, 384, 512 MB memory. 128 MB was insufficient to run the Java controller, so 192 MB was used as the minimum.

For the Microservice Controller for EX-1, EX-2, and EX-3 we used the Python controller configured with 128 MB of memory. For every experiment except EX-3, we configured the TLQ worker functions with 2048 MB of memory. For every experiment except EX-2, the initial run was discarded to ensure "warm" infrastructure. SAAFs newcontainer attribute was used to verify that all function instances were warm for these experiments. In addition, when reporting metrics, we discarded outliers (runtimes more than two standard deviations from the mean).

Table II provides a price comparison of control flow methods.

TABLE II. PRICE COMPARISON - CONTROL FLOW METHODS

Control flow Method:	Client Orchestration (synchronous)	Microservice Controller (synchronous)	State-machine (asynchronous)	Event-Based Triggers (asynchronous)
FaaS Function cost:	$.0000166667 / GB-second	$.0000166667 / GB-second	$.0000166667 / GB-second	$.0000166667 / GB-second (includes runtime for S3 download)
Flow Control Cost - 1 pipeline execution	Cost of VM (e.g. c5.large EC2 Instance)	Cost of server-side controller - *memory size may differ from pipeline functions*	$.00010 4 transitions @ $.000025 / transition	$0.000012 2 get requests + doesObjectExist call

5 Experimental Results

5.1 Performance Comparison (EX-1)

Figure 2 depicts measured pipeline runtime with alternate control flow methods for EX-1. The event-based triggers control flow approach suffered in performance, consistently producing the most latency as well as function runtime. The higher function runtime was due to the additional time spent downloading the previous function's JSON output file to obtain data passed to the ensuing function. Given the time spent downloading JSON output files and latency remained constant as input data size increased, the overhead was amortized, and the effect minimized on large data payloads. For the 100-row dataset, event-based control flow resulted in 384% slower performance compared to the other asynchronous control flow approach, the state-machine. However, for the largest dataset, the event-based approach had only a 2% performance penalty vs. the state-machine. Normalizing results for the 100,000 row experiments, the Microservice Controller, State-Machine, and VM-client (Client Orchestration) were 28.1%, 27.7%, and 25.6% faster than Event-Based Triggers. This shows the performance difference is relatively small between these three other methods compared to Event-Based Triggers.

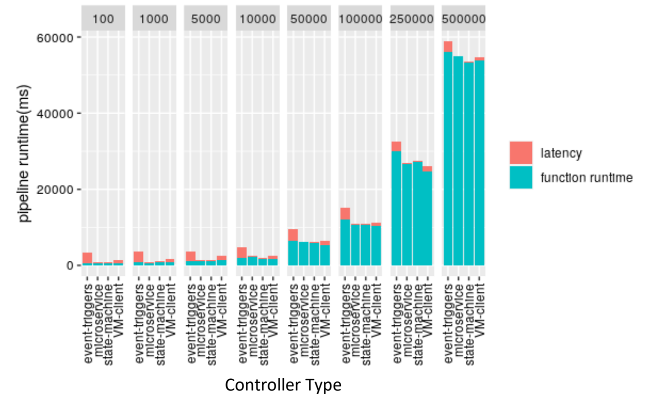

Figure 2: Pipeline runtime comparison of alternate control flow methods

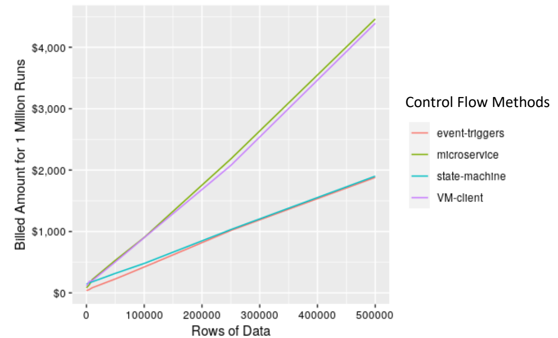

Figure 3: Cost vs. dataset size comparison for alternate control flow methods

Data processing costs for one million pipeline executions is shown in Figure 3. Event-Based triggers were consistently the cheapest option for running the pipeline, though the difference versus the State-Machine becomes relatively small as the dataset size increases. When processing large datasets, the runtime of the Lambda functions overpowers the cost of control flow. The cost of our synchronous Microservice Controller and Client Orchestration control flow methods increases quickly and dramatically when processing larger datasets. This is to be expected due to increased costs for additional cloud infrastructure (e.g., a VM, or FaaS function) when data processing time is long. For the largest dataset, the Microservice Controller was more than twice the cost Event-Based Triggers, meaning a million pipeline invocations using the largest dataset would cost ~$2,580 more using the microservice controller! The elements that constitute data processing costs for our data processing pipeline (compute + control flow) are described in Table II.

5.2 Cold Performance Comparison (EX-2)

Table III depicts average cold vs. warm latency values observer for the four control flow methods with rankings. The highest increase in warm vs. cold latency for the TLQ pipeline was seen when using the Microservice Controller for control flow. The Microservice Controller, however, had the lowest latency of any approach being slightly less than the State-Machine approach implemented using AWS Step Functions. Warm latency was derived from EX-1. Event based triggers had high cold and warm latency. For use cases where real time processing is needed, Event-Based triggers may be ineffective.

TABLE III. COLD-START LATENCY - CONTROL FLOW METHODS

Control flow Method:	Cold Latency ms (rank)	Warm Latency ms (rank)	Cold-to-Warm Latency Increase ms (rank)	Cold-to-Warm latency ratio (rank)
Client Orchestration	2944 (3)	933 (3)	2011 (3)	3.16x (2)
Microservice Controller	1850 (1)	192 (1)	1658 (1)	9.64x (4)
State-machine	1977 (2)	292 (2)	1685 (2)	6.77x (3)
Event-Based Triggers	4910 (4)	2850 (4)	2060 (4)	1.72x (1)

5.3 Function Memory Size Comparison (EX-3)

We examined how the pipeline runtime varied for each control flow approach while increasing the allocated memory for the TLQ functions. Functions were run with 512, 768, 1024, 1536, and 2048 MB using the 100,000 row dataset. Reducing function memory from 2048 to 512 MB using event-based triggers control flow doubled pipeline runtime. Notably, doubling function memory did not halve the pipeline runtime for any control flow method. AWS Lambda often claims doubling function memory should double performance. This result is likely due to pipeline dependencies on external cloud services (S3 and Aurora). Event-Based Triggers experienced the greatest decrease in pipeline runtime between 1024 MB and 1536 MB, 23.1%. Other methods all experienced the biggest decrease from 512 MB to 1024 MB at 20.0%, 20.0%, and 17.7% for the Microservice Controller, State-Machine, and VM-client (Client Orchestration) respectively. Event-based triggers only improved by 12.7% for this memory step. The plot line for event-based triggers in Figure 4 appears nearly linear between 512 and 1536 MB meaning additional memory helped improve performance consistently through this range. The other methods show more concavity between 512 and 1536 MB. Beyond 1536 MB little performance improvement was seen, likely because our functions did not utilize multiple threads. Above 1536 MB, Lambda functions gain access to an additional vCPU [9]. The Event-Based Trigger control flow approach better performance improvements relative to memory size may be attributed to I/O improvements as these functions download the prior function's JSON response object from S3, a step unique to this control flow method.

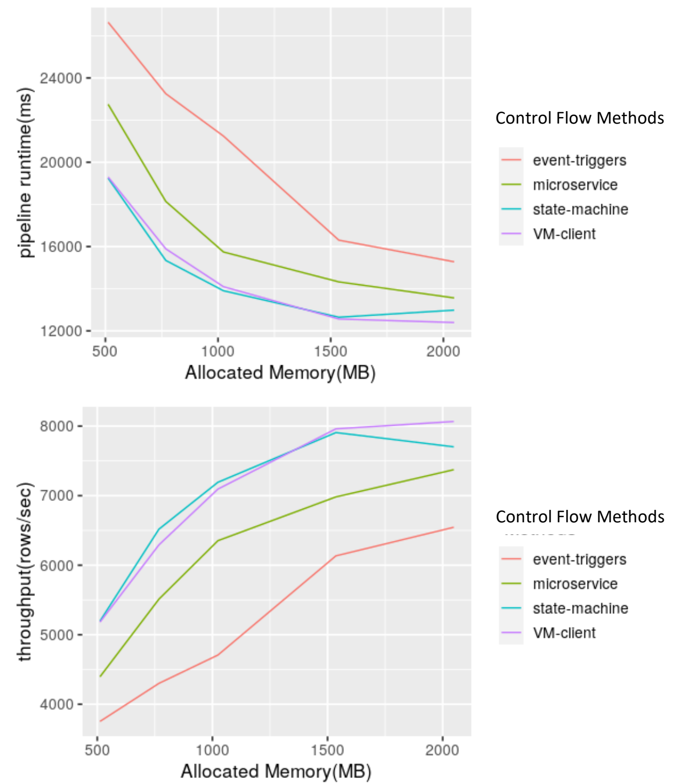

Figure 4: Pipeline runtime vs. memory reservation size (*top*), Pipeline throughput vs. memory reservation size (*bottom*)

5.4 Microservice Controller Comparison (EX-4)

In EX-4 we investigated memory and programming language implications when using a server-side function to orchestrate control flow. Figure 5 depicts pipeline runtime across the different controller memory settings. The figure shows consistent performance improvements for the Java controller. The Python controller, however, did not exhibit a relationship between memory allocation and performance, as the lowest memory setting provided the lowest pipeline runtime! It is important to note that the relative performance difference was very small with the difference between the fastest memory setting and the slowest being 3.2% for the Python controller, and 2.4% for the Java controller. Comparing performance across languages at the 192 MB setting shows a 3.2% improvement for the Java controller vs. Python. Allocating additional memory to the controller only resulted in a small performance improvement. For the 100,000 row dataset, the controller was shown to spend the majority of its time idling because data processing time was much larger than the pipeline latency. Figure 5 (*right*), shows controller price vs. memory allocation. A linear relationship between price and memory allocation is seen because the cost of allocating additional memory to the controller was not offset by a significant

runtime improvement as EX-4 processed a large number of rows (100,000). Doubling memory led to about a doubling in price.

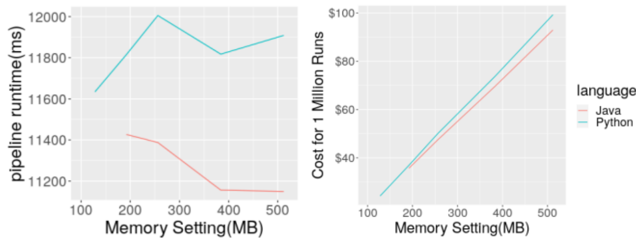

Figure 5: Comparison of pipeline runtime (*left*), and transition costs (*right*) for Java vs. Python microservice controllers

6 Conclusions

In this case study we compared different methods for orchestrating a multi-function data processing pipeline implemented on AWS Lambda. We implemented four alternate control flow approaches and performed experiments to investigate performance and cost implications for hosting a serverless data processing pipeline. We investigated control flow implications of different function memory configurations and infrastructure state (i.e. cold vs. warm). The summary of our findings is as follows:

RQ-1 (Performance): Executing the TLQ pipeline using each control flow method for various data sizes allowed us to examine how latency, pipeline runtime, and cost varied depending on the control flow method. We found that Event-Based Triggers had a noticeable performance penalty vs. the other methods. The difference in pipeline runtime using Event-Based Triggers compared to the State-Machine was 384%, 28%, and 2% for the 100, 100,000 and 500,000 row datasets. Performance of the VM-client (Client Orchestration), Microservice Controller, and State-Machine control flow was relatively similar, with the difference in pipeline runtime for these methods being less than 3% for the 100,000 row dataset.

RQ-2 (Cost): We found that the synchronous control flow methods, Microservice Controller and the VM-client (Client Orchestration), exhibited significantly higher costs compared to the asynchronous methods for all but the smallest datasets. A cost increase of over 200% was seen with the largest dataset. This equates to a premium of ~$2,580 (137%) when using the Microservice Controller compared to Event-Based Triggers for one million pipeline executions with the 500,000 row dataset.

RQ-3 (Cold Start): Our cold start experiment revealed that the Microservice Controller had the lowest latency, even less that the State-Machine supported by AWS Step Functions. Event - Based Triggers had the highest warm and cold latency and may not be suitable where very fast response times are needed.

RQ-4 (Memory): Event-Based triggers experienced the largest improvement in pipeline runtime (43%) at 2048MB compared to 512MB. Scaling TLQ function memory from 512 MB to 1024 MB resulted in the largest pipeline runtime improvements for the other control flow approaches, while pipeline runtime only improved by nearly half as much for Event-Based Triggers. Event based triggers exhibited the largest performance improvement when scaling from 1024 MB to 1536 MB. All control flow methods improved the least when scaling from 1536 MB to 2048 MB.

RQ-5 (Microservice Controller): Performance and cost of the Microservice Controller control flow approach had only minimal improvements when varying the controller memory allocation or programming language (Java vs. Python). Performance improved by 2.4% with additional memory for the Java implementation, while Python showed no discernible relationship between memory and performance with the lowest memory setting performing over 2% better than the highest. Given that the controllers spent most of their active time idling, the additional memory increased costs while not providing a significant performance improvement. The Java controller at 512 MB was the fastest, while the Python controller at 128 MB was the cheapest.

In conclusion, synchronous control flow methods such as the Microservice Controller exhibited higher costs, but low latency, while Event-Based Triggers required FaaS function refactoring, while having low costs, though high latency. The State-Machine approach implemented with AWS Step Functions provided a good balance of high performance and intuitive developer experience.

ACKNOWLEDGMENTS

This research is supported by the NSF Advanced Cyberinfrastructure Research Program (OAC-1849970), NIH grant R01GM126019, and AWS Cloud Credits for Research.

REFERENCES

[1] Wang, L., Li, M., Zhang, Y., Ristenpart, T. and Swift, M., 2018. Peeking behind the curtains of serverless platforms. In Proc. of the 2018 USENIX Annual Technical Conference (ATC '18), pp. 133-146.
[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] Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uță, A. and Iosup, A., Serverless is more: From PaaS to Present Cloud Computing. IEEE Internet Computing, Sept. 2018, 22(5), pp.8-17.
[4] Cordingly, R., Yu, H., Hoang, V., Perez, D., Foster, D., Sadeghi, Z., Hatchett, R. and Lloyd, W.J., Implications of Programming Language Selection for Serverless Data Processing Pipelines. In Proc. of the 6th IEEE Intl Conf on Cloud and Big Data Computing (CBDCom 2020), August 2020, pp. 704-711.
[5] Baldini, I., Cheng, P., Fink, S.J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P. and Tardieu, O., The serverless trilemma: Function composition for serverless computing. In Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017), Oct 2017, pp. 89-103.
[6] López, P.G., Sánchez-Artigas, M., París, G., Pons, D.B., Ollobarren, Á.R. and Pinto, D.A., Comparison of FaaS Orchestration Systems. In 11th IEEE/ACM International Conference on Utility and Cloud Computing Workshops: 4th Workshop on Serverless Computing (WoSC '18), December 2018, pp. 148-153.
[7] "AWS Lambda," AWS, [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 13-Dec-2020].
[8] Cordingly, R., Yu, H., Hoang, V., Sadeghi, Z., Foster, D., Perez, D., Hatchett, R., Lloyd, W., The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation for Serverless Software, 2020 21st ACM/IFIP Int. Middleware Conference: 6th Int. Workshop on Serverless Computing (WoSC '20), Dec 7-11, 2020.
[9] Cordingly, R., Heydari, N., Yu, H., Hoang, V., Sadeghi, Z., Lloyd, W., Enhancing Observability of Serverless Computing with the Serverless Application Analytics Framework, Tutorial Paper. 2021 12th ACM/SPEC Int. Conference on Performance Engineering (ICPE '21), Apr 19-23, 2021.