Implications of Flow Control Methods for Serverless Data Processing Pipelines

by

Sterling Quinn

Supervised by Dr. Wes J. Lloyd

A thesis submitted in partial fulfillment of the departmental honors requirements for the degree of

Bachelors of Science

Computer Science and Systems

University of Washington Tacoma

December 2020

Presentation of work given on: 12/16/2020

The student has satisfactorily completed the Senior Thesis, presentation and senior elective course requirements for CSS Departmental Honors.

Faculty advisor: Wes J. Lloyd ____________ Date ____________12/16/2020___
CSS Program Chair: Donald Ch____________ Date ____________12/16/2020___

© Copyright 2020

Sterling H. Quinn

Abstract

Implications of Flow Control Methods for Serverless Data Processing Pipelines

Sterling Ho Quinn
Supervised by Dr. Wes J. Lloyd

Serverless computing platforms offer a compelling option for developers to host applications without configuring servers, shifting the burdens of scaling, handling hardware failures, and guaranteeing availability to the cloud provider. 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. FaaS platforms are well suited to host microservices, small modular applications that perform specific tasks encouraging code reuse and enabling greater agility by reducing reliance on hard to change monolithic applications. Using smaller functions to decompose sequential processing steps of larger tasks or workflows introduces the question of how to instrument individual function calls to orchestrate the original task or workflow. In this thesis, we examine the implications of using different methods to orchestrate the flow control of a serverless data processing pipeline composed as a set of independent microservices. We performed experiments on the AWS Lambda FaaS platform and compared how four different patterns of flow control affected the cost and performance of the pipeline. Methods of flow control include a virtual machine client, microservice controller, event-based triggers, and cloud-based state-machine. Overall we found that asynchronous methods led to lower orchestration costs and that event-based orchestration comes with a performance penalty.

TABLE OF CONTENTS

Chapter 1. Introduction **1**
- 1.1 Serverless Computing 1
- 1.2 Flow Control 2
- 1.3 Case Study 3
- 1.4 Research Questions 3

Chapter 2. Background and Related Work **4**
- 2.1 Serverless Function Development Choice Implications 4
- 2.2 Conceptual Issues with Serverless Flow Control 5
- 2.3 Comparison of FaaS Orchestration Systems 5

Chapter 3. Serverless Application Flow Control **6**
- 3.1 Overview 6
- 3.2 Billing 7
- 3.3 Metrics 8
- 3.4 Flow Control Methods 10
  - 3.4.1 Client Flow Control 10
  - 3.4.2 Step Functions 11
  - 3.4.3 Microservice Controller 12
  - 3.4.4 Event Based Triggers 13

Chapter 4. Methodology **14**
- 4.1 AWS Lambda 14
- 4.2 Transform Load Query Data Processing Pipeline 15
- 4.3 SAAF 16
- 4.4 Experiments 17

Chapter 5. Results **19**
- 5.1 EX-1 Overall Performance Comparison 19
- 5.2 EX-2 Cold Performance Comparison 21
- 5.3 EX-3 Lambda Functions Memory Size Comparison 22
- 5.4 EX-4 Microservice Controller Memory Size and Language Comparison 24

Chapter 6. Conclusions **26**
- 6.1 Flow Control Performance Comparison 26
- 6.2 Flow Control Development Perspectives 28
- 6.3 Outcomes 31

Bibliography **32**

List of Figures

Figure 1. Diagram of pipeline orchestration using Amazon EC2 as the client

Figure 2. Diagram of pipeline orchestration using AWS Step Functions

Figure 3. Diagram of pipeline orchestration using a microservice controller

Figure 4. Diagram of pipeline orchestration using Event Based Triggers

Figure 5. Pipeline runtime comparison of alternate flow control methods

Figure 6. Billed amount for across dataset sizes for different flow control methods

Figure 7a. Pipeline runtime vs. memory reservation size

Figure 7b. Pipeline data processing throughput vs. memory reservation size

Figure 8a. Comparison of pipeline runtime for Java and Python microservice controller implementations

Figure 8b. Comparison of flow control costs for Java and Python microservice controller implementations

List of Tables

Table 1. Price comparison for alternate flow control methods

Table 2. Cold start latency comparison

Table 3. Comparison of capabilities of flow control methods

ACKNOWLEDGEMENTS

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

Chapter 1. Introduction

1.1 Serverless Computing

Cloud computing's popularity comes in part from the level of abstraction it offers organizations and developers. Instead of having to purchase and configure physical hardware, organizations can rely on cloud providers for server infrastructure, allowing them to focus more resources on their core business functions. A further level of abstraction is provided by serverless computing platforms where the burden of provisioning and scaling infrastructure is placed on the cloud provider. While serverless technology still leverages servers behind the scenes, many details regarding management of the servers are abstracted from the user and handled by the cloud provider. Users request services on demand, and are billed for only what they use. This can reduce costs for workloads having variable user demand, removing the need to have always-on idle servers ready to handle spikes in traffic. With serverless technology, this responsibility to respond to server failure or sudden spikes in traffic is shifted to the cloud provider.

Function-as-a-Service (FaaS) is a serverless computing platform 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 an isolated environment for the code to run guaranteeing consistent, stateless performance [1].

Amazon Web Services (AWS) is a leading provider of cloud computing services owning almost half of the publicly available cloud infrastructure in 2018 (47.8%), more than three times its closest competitor Microsoft Azure at (15.5%) [2]. This thesis leverages the AWS FaaS platform known as Lambda combined with additional AWS services to facilitate investigations.

1.2 Flow Control

The move toward microservices is sweeping across technology companies [3]. A microservice is an application that handles a specific job and is only loosely coupled to other services. This is in contrast to monolithic applications that encapsulate entire business processes in a single application. Microservices that accomplish common tasks can be reused by other developers and smaller modular pieces of software can be updated or changed more easily. 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, some method to orchestrate independent service calls is required. We will refer to these methods that instrument function call chains as *"serverless application flow control"*. Cloud providers offer different options for connecting services but it is unclear what tradeoffs come with each option. When choosing between different methods of flow control for serverless functions, 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 is generally free. Using a cloud-based virtual machine will also come at a cost. If speed is the primary consideration, it isn't clear how latency varies between the different flow control options. The capabilities of each method must also be considered. If data must be exchanged between functions, some flow control approaches facilitate the exchange more easily than others. Given that software developers time is extremely valuable, developers may consider the most economical option as the approach that requires the least effort promoting "ease of implementation" to be the highest priority requirement for some use cases.

1.3 Case Study

We examine the implications of flow control for serverless applications by investigating four different methods of orchestrating a data processing pipeline:

1. **Client Control**: This method involves calling the individual pipeline steps synchronously from the developer's computer or from a computer provisioned in the cloud.
2. **Microservice Controller**: This method involves provisioning an additional serverless function to orchestrate the execution of the functions.
3. **State-Machine**: This method offered by the cloud provider enables developers to define a state-machine to describe function transitions and data flow. The cloud provider instruments a client based on the state-machine definition to invoke functions for each step of the pipeline or workflow.
4. **Event Based Triggers**: Cloud providers also provide methods for defining 'rules' that will trigger serverless functions when a certain event occurs. These rules can be used to orchestrate serverless pipelines.

1.4 Research Questions

In comparing these alternate methods of flow control, we investigate the following research questions. Empirical questions include:

RQ-1. (Performance) What are the performance implications for alternate methods of serverless application flow control? 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 flow control?

RQ-3. (Cold Start) What are the implications for cold start for alternate methods of serverless application flow control? 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 flow control? 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 flow control pattern?

Qualitative questions include:

RQ-6. (Feature Comparison) From a development perspective, what unique capabilities does each application flow control method provide?

RQ-7. (Developer Effort) How does developer effort vary for alternate application flow control methods? Which provides the best developer experience?

CHAPTER 2. BACKGROUND AND RELATED WORK

2.1 SERVERLESS FUNCTION DEVELOPMENT CHOICE IMPLICATIONS

This thesis complements the findings of a paper by the UW Tacoma Cloud and Distributed System Research group regarding the implications of programming language choice for serverless data processing pipelines [4]. In the paper, the researchers implemented a three step data processing pipeline in multiple languages to analyze the impact of language choice on the

cost and performance of the pipeline. In this thesis, we examine alternate choices of flow control to orchestrate the steps of the same data processing pipeline. These efforts aim to demystify cost and performance implications of alternative serverless application flow control methods available to developers.

2.2 Conceptual Issues with Serverless Flow Control

Other researchers have analyzed outstanding conceptual problems in orchestrating serverless functions. In a 2017 paper, IBM researchers address a problem common to synchronous flow control methods, double billing [5]. This problem occurs any time a serverless function that is billed per unit time is used as a controller to compose a workflow of serverless functions. This function must be active while waiting for each result from synchronous serverless function calls. The user is billed for two functions at the same time. The user is charged for the controller function that is idly waiting for results, and for the other function that accomplishes the work. While this paper introduced this problem, it did not quantify the cost implications with concrete data.

2.3 Comparison of FaaS Orchestration Systems

López et al. examined different platform specific FaaS orchestration systems in a 2018 paper [6]. The systems examined were AWS Step Functions, Azure Durable Functions, and IBM Composer. This report focused on these three cloud provider orchestration tools and did not compare them to other flow control methods. The primary metric focused on by this report was the overhead associated with orchestrating a set of functions, which the researchers obtained by subtracting the sum of the runtime of those functions from the runtime of the sequential composition of those functions. The researchers varied both the number of functions in sequence,

and the number of parallel invocations of sequences of the same size. They found that "overhead grows linearly with the number of functions in the sequence" but in contrast "overhead grows 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 and cost data for those platforms was not clear at that point making a direct cost comparison impossible. The experiments conducted in this report used a function that slept for one second and did not perform any work. By conducting our flow control study using functions performing common data processing tasks with various dataset sizes, we hope to gain insight into how alternate flow control methods can impact the performance of tasks with different runtimes.

CHAPTER 3. SERVERLESS APPLICATION FLOW CONTROL

3.1 OVERVIEW

The flow of execution between serverless functions presents a unique set of challenges. Serverless functions should be treated as stateless since each invocation may be in a fresh environment. When possible, serverless platforms reuse function instances from previous invocations, this means that in some cases any data written to the environment from a previous invocation may be available, however if a long enough period has elapsed since the last invocation of a function, or if all the previously initialized function instances are in use, a fresh function instance will be initialized that persists no previous data. Because of this, any stateful data must be stored using an additional service.

Methods of orchestrating the flow of functions can be classified as synchronous or asynchronous. Synchronous methods involve the caller maintaining a connection to the cloud platform. This provides the advantage of having an immediate awareness of failure, but also

requires that the client dedicate an idle thread to wait for work to be completed by the serverless functions. Asynchronous methods make a call to initiate function execution without maintaining an open connection. The client must either poll for the availability of the final result to then pull it from the server, or subscribe to a notification service which will then push the result to the client upon completion. Asynchronous function invocations free up computing resources by not dedicating resources that must wait while work is accomplished elsewhere. Developers must determine an appropriate polling interval that considers the round trip latency between the client and the cloud. Approaches must strike a balance between retrieving results in a timely manner and not wasting CPU cycles.

3.2 Billing

How serverless workflows are billed introduces additional considerations to account for in choosing the optimal method of flow control. The serverless functions accomplishing work are billed per GB/second, rounded to the nearest millisecond. This means that the cost of running functions will always scale with the amount of work. FaaS platforms allow developers to configure the amount of memory available to the function. The amount of memory allocated to the function by the developer will affect the price per second, but in some cases improve the runtime of the function. Many FaaS platforms such as AWS Lambda and Google Cloud Functions scale CPU resources with the function's memory reservation size. This prompts users to allocate memory, not only for the memory required to run the function, but also to scale the CPU capacity allocation of a function. Choosing the optimal memory setting for a function is important because setting the memory too low can increase runtime to the point that it eclipses any cost savings, leading to lower performance and higher costs. On AWS Lambda, performance improvements also decline when reserving above 1536 MB, the point where the function

sandbox receives a second virtual CPU core. If the function's code is not multithreaded, allocating memory beyond this threshold leads to higher costs with minimal discernable performance gains [4]. Flow control methods that require the FaaS functions themselves to perform additional work will incur additional costs. All methods of flow control add additional costs for orchestrating transitions which we discuss in section 4.4.

3.3 Metrics

To analyze the performance of the various methods of flow control we collected a variety of metrics to characterize our experimental runs. We introduce these metrics before covering the flow control methods studied to help readers better understand how we expect these metrics to vary with the choice of flow control. We profiled the following primary metrics:

pipeline runtime:

This metric is calculated by subtracting the startime of the first function in the pipeline from the endtime of the last function to derive the pipeline's elapsed time. This captures processing time of the worker functions as well as the time taken to transition between functions.

client runtime:

This metric captures the client-to-server round trip time and is calculated by subtracting the client's wall clock time before invoking the pipeline from the wall clock time when the client is notified of pipeline completion. For the asynchronous flow control methods, this metric will include a variable amount of extra elapsed time because notification of pipeline completion is obtained through polling.

function runtime:

This metric records only the time used by the worker functions and neglects the transition time.

latency:

This metric is obtained by subtracting the combined **function runtimes** for the worker functions from the **pipeline runtime** to obtain just the transition time between functions.

billed amount:

This metric is the estimated cost of hosting the data process pipeline. Billed amount is calculated by multiplying the function runtime in ms by the published Lambda billing rate per ms for each function, and then adding any additional transition costs specific to the method of flow control. This includes costs associated with the use of other AWS services such as S3, and Step Functions.

throughput:

Data processing throughput (row/sec) provides a metric for comparing the efficiency of data processing pipelines. Throughput describes the velocity that data is processed offering a key big data metric to describe the flow rate of data processing for a given test. Visualizing a pipeline's throughput when scaling the memory reservation size of the pipeline's functions can help to visualize flow rate of data processing.

3.4 Flow Control Methods

3.4.1 Client Flow Control

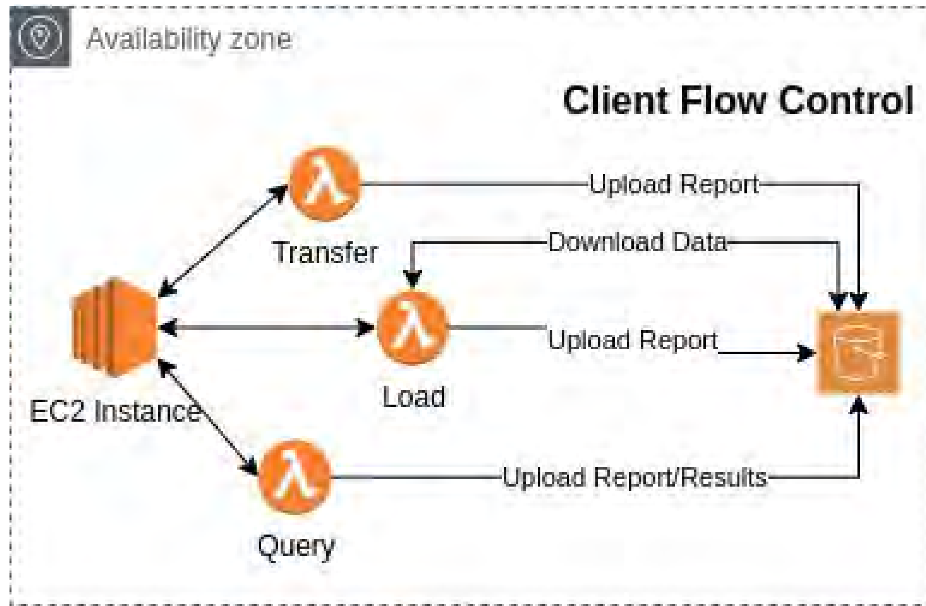

Figure 1. Diagram of pipeline orchestration using Amazon EC2 as the client

Client Flow Control refers to orchestrating the pipeline from a developer desktop or laptop, or a cloud-based virtual machine. All of our experiments involved triggering the pipeline from the same EC2 instance. Performance of this method is limited by the resources of the EC2 instance (e.g. vCPUs, memory). For client flow control, we included the cost of the EC2 instance while calculating the total **billed amount** of the data processing pipeline using this flow control method. Including the cost of the EC2 instance means that the portion of the **billed amount** metric associated with transitions between functions will scale linearly with **pipeline runtime.**

3.4.2 Step Functions

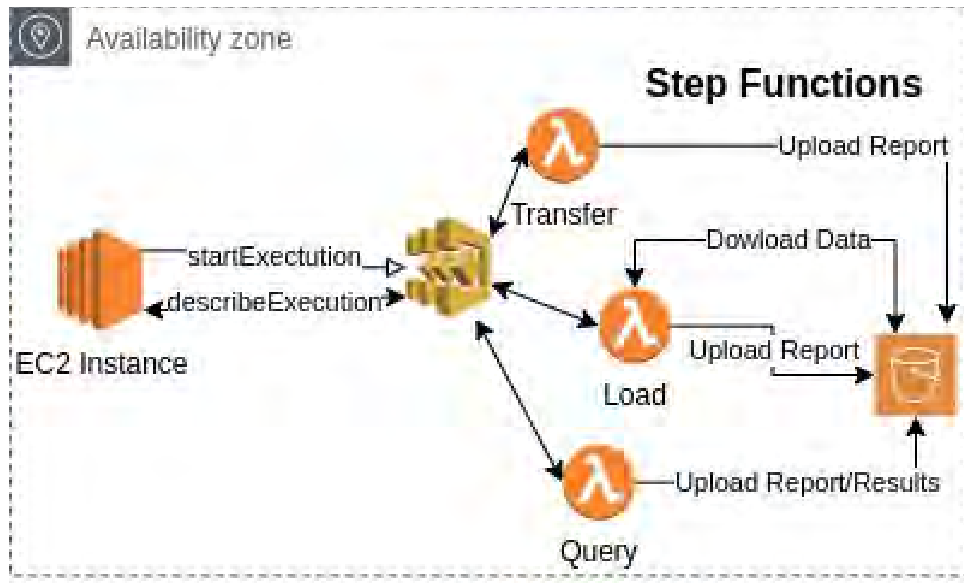

Figure 2. Diagram of pipeline orchestration using AWS Step Functions

Step functions is an AWS provided service for workflow orchestration. This service allows developers to define workflows in an easy to use GUI. For a simple pipeline without branches this was trivial to implement. The service also supports more advanced constructs like if statements enabling more complex workflows. Step functions handle passing data from each step (e.g. Lambda function) to the next. The service is billed per transition, not function runtime, including the pipeline's start and end, meaning a three step pipeline requires five transitions. One advantage with Step Functions is that the **billed amount** is unrelated to pipeline runtime enabling this method to have a constant transition cost based on the number of function transitions.

3.4.3 Microservice Controller

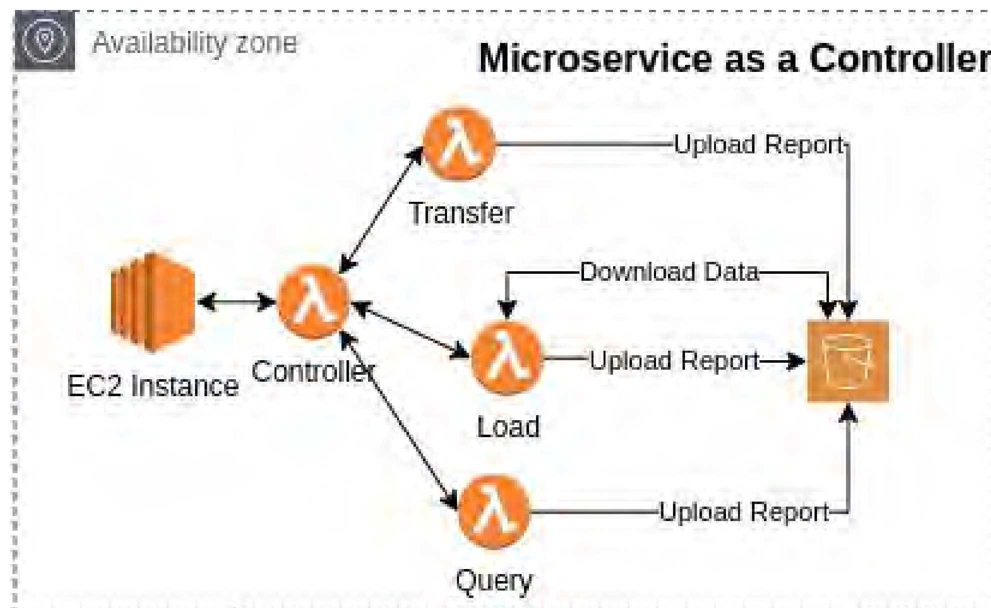

Figure 3. Diagram of pipeline orchestration using a microservice controller

The Microservice Controller flow control approach involves developing a separate Lambda function to orchestrate the pipeline. For this project we developed controllers in both Java and Python to compare cost and performance differences. This flow control method suffers from the double billing problem, because cost is incurred for both the function accomplishing the work and the controller [4]. For the controller microservice, the relatively simple task of orchestrating function calls can be accomplished with minimal memory. Consequently, we investigate the tradeoff between the memory reservation size for the Microservice Controller function vs. the ensuing performance of our data processing pipeline.

3.4.4 Event Based Triggers

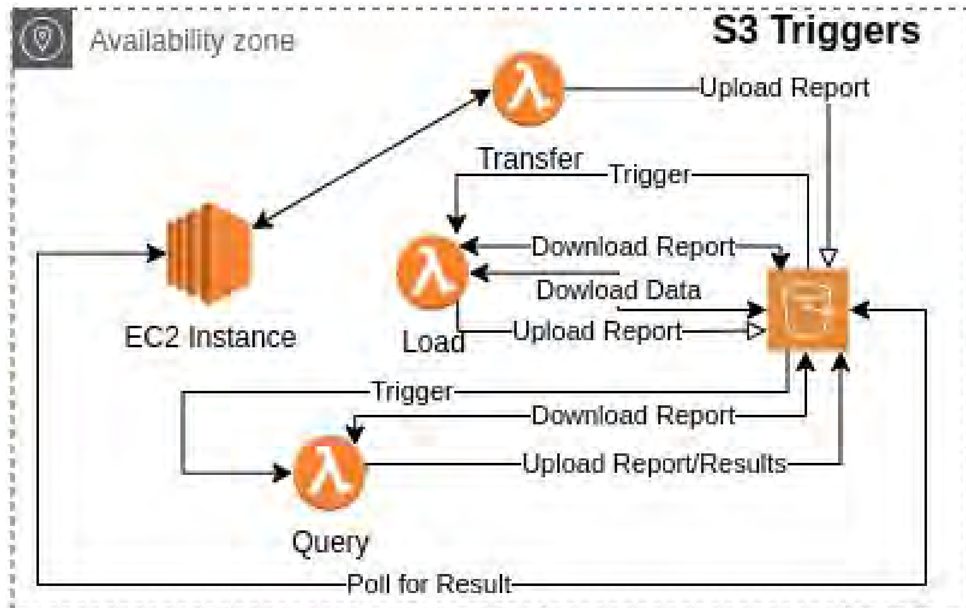

Figure 4. Diagram of pipeline orchestration using Event Based Triggers

AWS provides an option to trigger Lambda functions based on events occurring to buckets and objects in S3. For this workflow, we upload report files to S3. AWS provides a mechanism for defining triggers matching certain events that target other resources. To orchestrate our pipeline, we defined an event handler for S3 PutObject operations with a filename matching the function. The event handler calls the next function in the pipeline. Because the pipeline requires data from the previous function, this method required downloading the report files (e.g. Lambda function response JSON objects) from S3, adding additional costs by increasing the **function runtime.** This additional cost could be avoided through multiple workarounds. Static data could be hardcoded, or provided through Lambda environment variables. Dynamic data provides a more difficult problem because the only data provided to the function triggered by S3 is the file name, which is limited to 1024 characters. However this could be used to transmit a limited amount of dynamic data if necessary. For our experiments we accepted the additional overhead of downloading the report file as a penalty for this method of

flow control. Additional costs for this method are S3 operations which are billed per operation. Because the size of the report files (e.g Lambda function. JSON response objects) do not scale with the size of the dataset processed, and the S3 operations used stay constant with each pipeline execution, the transition cost of this method is constant in relation to data size.

It is worth noting that this method of flow control was the only one requiring code modifications and additional deployments for the worker functions. Because S3 triggers provide an S3Event argument instead of a standard HashMap, the code had to be refactored to accept that input. This is undesirable from a microservice architecture standpoint because it limits the reusability of the services by tying them to a particular method of flow control. Services defined to accept an S3Event as an argument are no longer useful for clients that may want to invoke them through other means.

CHAPTER 4. METHODOLOGY

4.1 AWS LAMBDA

AWS Lambda is a serverless compute service that lets developers run code without provisioning servers, creating scaling logic, or managing runtimes. With Lambda, developers can run code with zero administration. Developers can upload code as a zip or JAR file, or even write code in browser and Lambda will automatically allocate an execution environment on demand and run the code, for any scale of traffic [6].

4.2 TRANSFORM LOAD QUERY DATA PROCESSING PIPELINE

In this case study, we examine the implications of using alternate methods of flow control to instrument a three step data processing pipeline implemented on the AWS platform. This pipeline was originally developed to study implications of programming language choice for serverless data processing [3]. The pipeline processes sample sales data that includes information such as product order details, transaction pricing, and customer metadata. Each dataset is a CSV file stored in Amazon S3 ranging from 100 to 500,000 rows. The pipeline steps are described as follows:

1. Transform Function

   The transform function takes a CSV file stored in Amazon S3 and applies multiple transformations to the data. This function removes duplicate rows, creates additional columns containing order processing time, and calculates the gross margin of each sales transaction. Once the transformations are applied the modified CSV file is saved to Amazon S3.

2. Load Function

   The load function pulls the transformed CSV file from S3 and loads it into an Amazon Aurora serverless MySQL database. The function creates SQL insert queries for each row in the CSV file. These queries are executed in batches of 1,000 to improve performance by reducing the number of distinct database transactions.

3. Query Function

   The final function queries the newly loaded database by performing five separate SQL aggregation queries where results are joined with a UNION . This function then saves the results of the queries to S3 for future access. After the queries are complete a stress test is

performed using a " SELECT * " query to retrieve every row from the database to measure the data transfer throughput (row/sec) between the database and the FaaS function.

4.3 SAAF

To help identify factors responsible for performance variation on FaaS platforms, researchers in the UW Tacoma Cloud and Distributed Systems research group have developed the Serverless Application Analytics Framework (SAAF) [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.

Attributes collected include Linux CPU Time Accounting metrics such as CPU idle, user, kernel, and I/O wait time. Other metrics include wall-clock runtime and several to characterize memory usage. To identify infrastructure state, SAAF stamps function instances with a unique ID and the existence of a stamp identifies if the environment is new (cold) or recycled (warm). A function instance is stamped by writing a UUID file to /tmp. For this case study, we collected data from each pipeline step by uploading results compiled by SAAF to Amazon S3 for later analysis. We augmented the data collected by SAAF with additional metrics collected on the client.

4.4 EXPERIMENTS

To analyze our various methods of flow control with respect to the metrics described in section 3.3 including **function runtime, pipeline runtime, client runtime, latency,** and **billed amount,** we conducted the following experiments. Dataset sizes refer to rows of CSV data.

EX-1. Overall Performance Comparison

In this experiment we performed 11 runs of the TLQ pipeline for each flow control method, for each of the dataset sizes: 100, 1,000, 5,000, 10,000, 50000, 100,000, and 500,000.

EX-2. Cold Performance Comparison

In this experiment, we performed 10 runs of the TLQ pipeline for each flow control method using a 100,000 row dataset, with 45 minutes of sleep time between runs. To isolate the effect of cold starts to the Lambda functions and flow control methods, we configured our database to stay active for the duration of the experiment by disabling the "Pause compute capacity after consecutive minutes of inactivity" setting.

EX-3. Lambda Functions Memory Size Comparison

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

EX-4. Microservice as a Controller Memory Size and Language Comparison

In this experiment, we performed 11 runs of the TLQ pipeline for each of the two different implementations of a microservice controller: one in Java, and one in Python. We repeated the experiment using the 100,000 row data set for the following controller

memory settings: 128, 192, 384, 512 MB. For the Java controller, 128 MB was an insufficient amount of memory to run the controller so this setting was skipped.

For the experiments comparing different flow control patterns (e.g. **EX-1, EX-2,** and **EX-3)**, the microservice controller used was the Python implementation configured with 128 MB of memory. For every experiment except **EX-3,** we configured the worker functions with 2048 MB of memory. For every experiment except **EX-2**, the initial run was discarded to ensure "warm" infrastructure. SAAF's newcontainer attribute was used to verify that all function instances were warm for these experiments. In addition, when reporting metrics, we discarded any run for which **pipeline runtime** was more than two standard deviations from the mean. Resource contention on the public cloud can introduce a level of unpredictability to analyzing cloud computing performance so by removing outliers we hope to perform more meaningful analysis.

The below table provides a price comparison for each flow control method.

Table 1. Price comparison for alternate flow control methods

Flow Control Method	Client Orchestration	Microservice Controller	State Machine	Event Based Triggers
function runtime costs	\$0.0000166667 for every GB-second	\$0.0000166667 for every GB-second	\$0.0000166667 for every GB-second	\$0.0000166667 for every GB-second (has additional **function runtime** for S3 download)
Transition cost	Cost of EC2 Instance	Cost of running controller	0.000125\$ (5 transitions at .000025\$ per transition)	0.000012\$ (2 get requests + doesObjectExist call)

CHAPTER 5. RESULTS

5.1 EX-1 Overall Performance Comparison

The focus of this experiment was determining the overall cost and performance implications of the choice of flow control. For this experiment we configured the T, L, and Q functions with 2048 MB of memory. Throughout this experiment, the event-based model of flow control suffered in performance, consistently producing the most **latency** as well as **function runtime.** The higher **function runtime** for the event-based flow control was due to the additional time spent downloading the previous function's JSON output file to obtain data that was passed to the functions in the other methods. However, because both the time spent downloading JSON output files and the latency stays constant as the data size increases, the relative difference decreases as data size increases. In essence this additional latency is amortized with increasing larger input data. For the 100 row data sample, using the event based system results in 384% slower performance when compared to the other asynchronous method of flow control, the state-machine. However for the largest datasize, choosing the event based system comes at only a 2% performance penalty vs. the state-machine. Normalizing the results from the 100,000 row experiments for the microservice controller, state-machine, and VM-client based on the results from the event-based trigger method results in a 28.1%, 27.7%, and 25.6% faster performance for the microservice controller, state-machine, and VM-client respectively. This demonstrates that the performance difference is relatively small between these three methods compared to the event-based method.

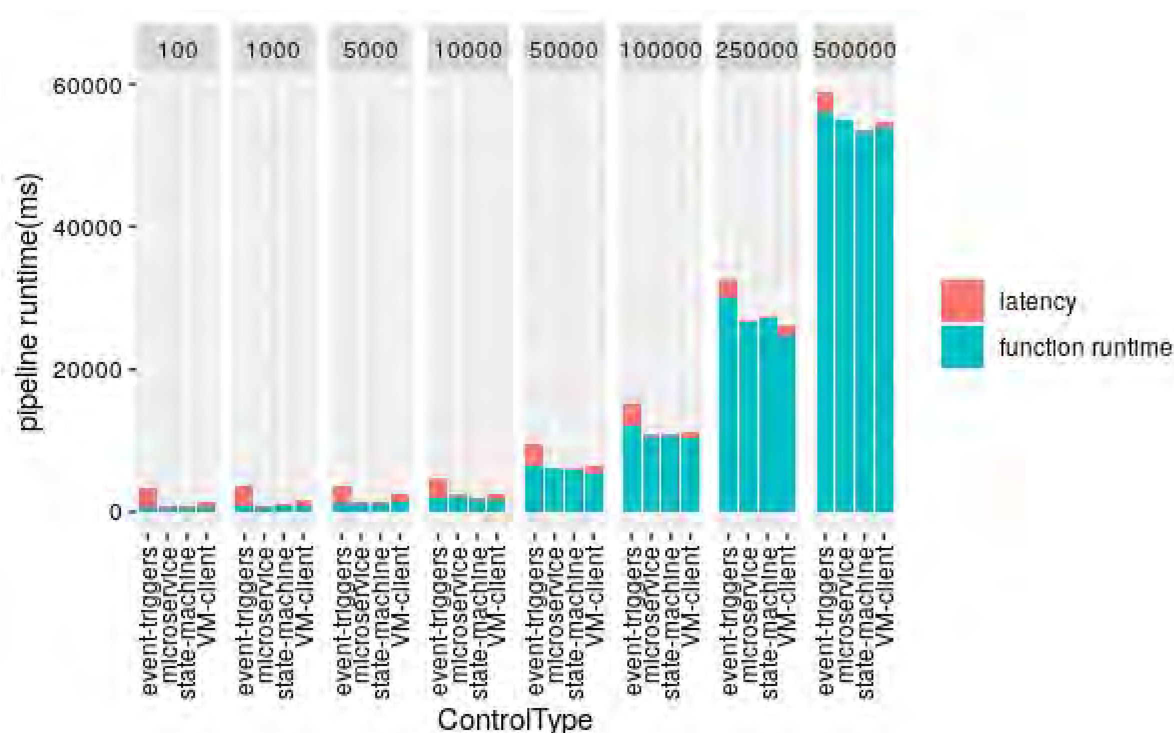

Figure 5. Pipeline runtime comparison of alternate flow control methods

It is clear that using event-based triggers comes at a performance penalty, however when considering the price of each method, the event-based model is a competitive option. Event-based triggers are consistently the cheapest option for running this pipeline, although the difference between this method and the use of the state-machine becomes relatively small as dataset size increases and the runtime of the Lambda functions begins to overpower the price of function transitions. The cost of both of our synchronous methods increases quickly and dramatically as the dataset size increases. This is to be expected as the transition cost will continue to scale as the data size increases, while our asynchronous methods have constant transition costs. For the largest data size, choosing the microservice controller pattern results in more than twice the **billed amount** of the event based system, meaning that a million invocations of the pipeline for the largest data size would cost ~$2,580 more using the microservice controller.

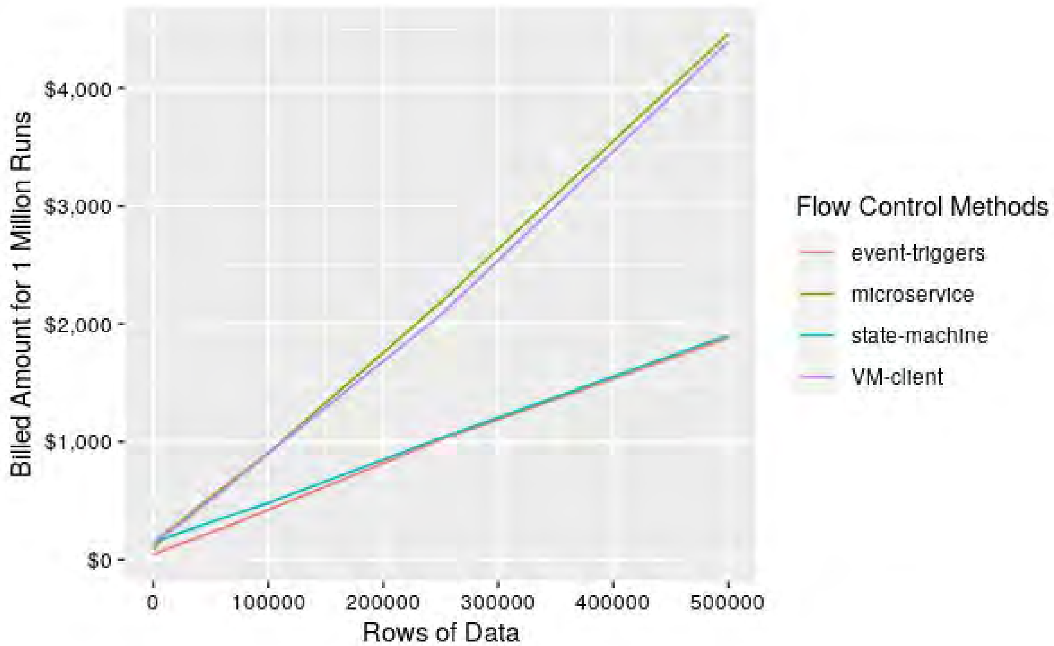

Figure 6. Billed amount for across dataset sizes for different flow control methods

Table 2. Cold start latency comparison

Flow Control	Cold Latency (ms)	Warm Latency (ms)	Cold-to-Warm Latency Increase (ms)	Cold-to-Warm Delta Ratio
VM-client	2944	933	2011	3.16x
state-machine	1977	292	1685	6.77x
event-triggers	4910	2850	2060	1.72x
microservice	1850	192	1658	9.64x

5.2 EX-2 Cold Performance Comparison

This experiment revealed that the ratio of cold to warm start latency for the data processing pipeline was most impacted by the microservice controller flow control pattern, and least impacted for event-based triggers when comparing the data collected in this experiment to the data using the same data size from **EXP-1**. For the microservice controller, cold latency will include the cold start for the controller Lamba. This additional cold start time, combined with the best warm performance gave it a very high delta ratio. Event based triggers were least affected as a ratio as the warm latency is already quite high for that method.

5.3 **EX-3** Lambda Functions Memory Size Comparison

To examine how the different methods of flow control were affected by varying allocated memory for the worker functions, we examined how each flow control method's **pipeline runtime** varied as we increased the allocated memory for the worker functions. The T, L and Q functions were run with 512, 768, 1024, 1536, and 2048 MB for this experiment using the 100,000 row dataset. Comparing figure 7a to figure 5 from **EXP-1** demonstrates that the performance improvement for the flow control method with the greatest **pipeline runtime**, event based triggers, was most noticeable, with choosing the lowest memory setting (512 MB) resulting in over twice the **pipeline runtime** compared to the largest memory setting (2048 MB). We note that doubling the memory allocation did not halve the **pipeline runtime** for any flow control method. The event-based method experienced its biggest decrease in **pipeline runtime** between 1024 MB and 1536 MB, 23.1%. The 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 respectively. Event-based triggers only improved by 12.7% for this memory step. The plot line for event-based triggers is relatively linear between 512 MB and 1536 MB meaning that additional memory continues to translate to better performance throughout this range. The plots for the other methods show a clear concavity over that period, meaning that as the available memory increased, the performance improvement for adding memory decreased. For every flow control method, the graph flattens between 1536 and 2048 MB, with the state-machine even experiencing a drop in performance. This may not be the case if our pipeline consisted of multithreaded functions because above 1536 MB, functions gain access to an additional CPU core, which the T, L, and Q functions do not utilize [3]. It is worth mentioning here that the Lambda functions used for the pipeline with event-based trigger flow control are different from those used by the other three methods, as they download the prior

function's JSON response object from S3. This could be the reason why this flow control method has higher relative improvements at higher memory settings than the other methods. Figure 7b depicts pipeline data processing throughput relative to function memory reservation size.

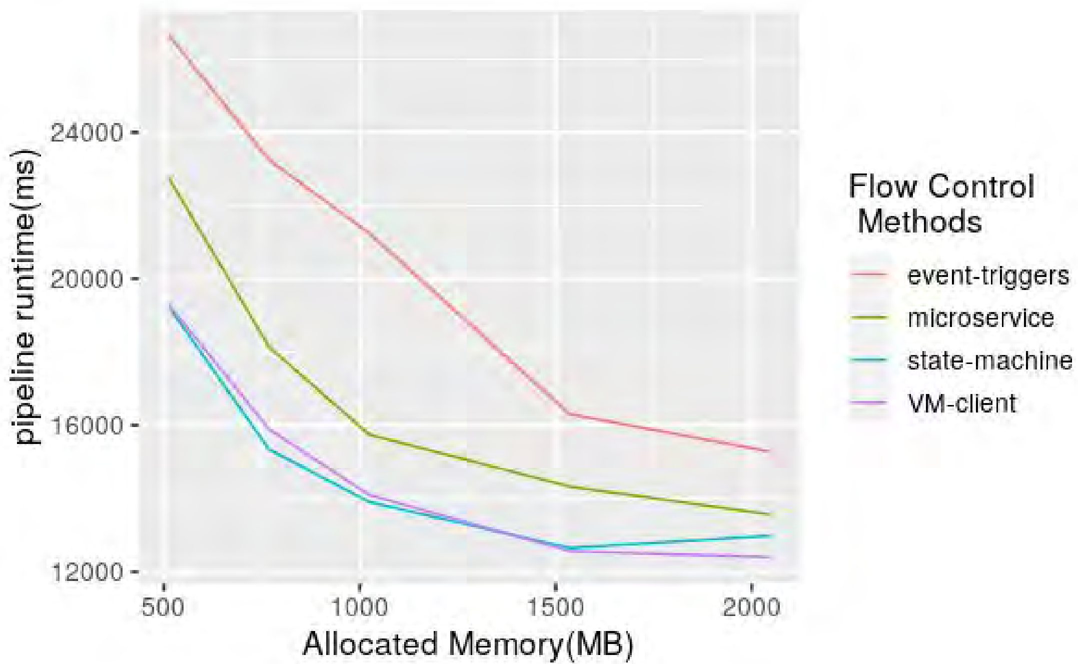

Figure 7a. Pipeline runtime vs. memory reservation size

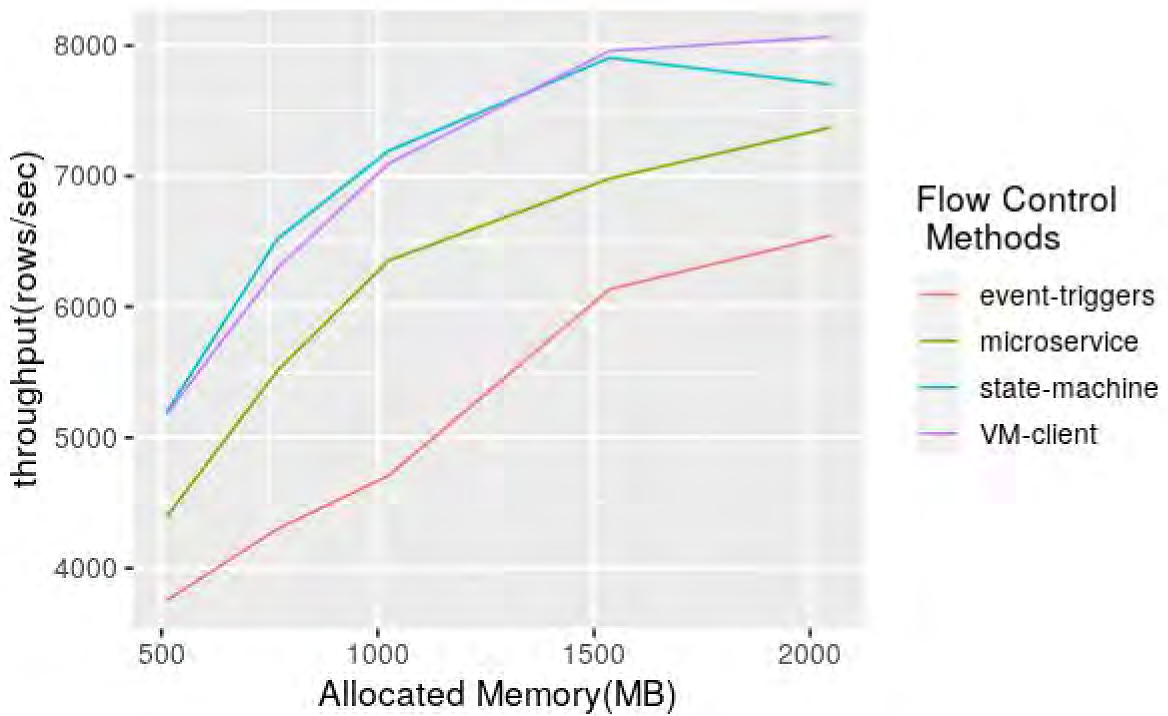

Figure 7b. Pipeline data processing throughput vs. memory reservation size

5.4 EX-4 Microservice Controller Memory Size and Language Comparison

To investigate how language choice and memory setting affected the microservice controller pattern, we experimented with various controller memory configurations using controllers implemented in Java and Python, by processing a 100,000 row dataset. We evaluated the Python controller with 128, 192, 256, 384, and 512 MB of reserved memory. For the Java controller, 128 MB of memory was not a sufficient amount of memory to run the controller so we excluded this setting and evaluated the other 4 memory settings. The performance graph in Figure 8a depicts **pipeline runtime** across the different controller memory settings. The figure shows consistent performance improvements for the Java controller. The Python controller, on the other hand, does not show a consistent relationship between memory allocation and performance, as *the lowest memory setting provided the lowest **pipeline runtime**!* It is important to note that the percentage performance difference is very small for both languages, 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. Referring back to figure 5 from **EXP-1** shows why allocating additional memory to the controller only results in small performance improvements. For the 100,000 row dataset, the controller was shown to spend the majority of its active time idling because **function runtime,** the execution time of the worker functions, clearly dominates **pipeline runtime**, while latency, the metric that is affected by better controller performance is only a small fraction. Figure 8b, which shows controller price vs. memory allocation, shows a linear relationship between price and memory allocation because the cost of allocating additional memory to the controller is not offset by a significant runtime improvement. Doubling memory led to about a doubling in price.

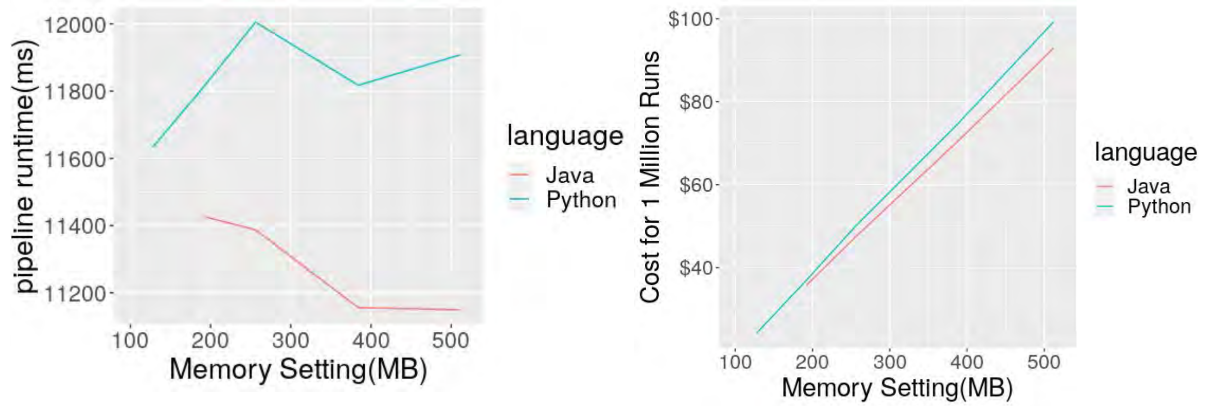

Figure 8. Comparison of pipeline runtime (left, a) and transition costs (right, b) for Java and Python microservice controller implementations

Chapter 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 distinct flow control patterns and performed experiments to investigate how the choice of flow control pattern impacts the cost and performance of a data processing pipeline. Our experiments investigated implications for flow control using scenarios with different function configurations and infrastructure state (i.e. cold vs. warm).

6.1 Flow Control Performance Comparison

We now summarize our research findings for each research question.

RQ-1: Executing the Transform-Load-Query data processing pipeline using each flow control method for various data sizes allowed us to examine how **latency, pipeline runtime,** and **billed amount** varied depending on flow control method. **We found that using event-based triggers came at a noticeable performance penalty vs. the other methods**. This difference in **pipeline runtime** using event-based triggers compared to the state-machine orchestrated pipeline was 384%, 28%, and 2% for the 100, 100,000 and 500,000 row datasets. The performance for the VM-client, microservice controller, and state-machine pipelines was relatively similar, with the difference in **pipeline runtime** for these methods being less than 3% for the 100,000 row dataset.

RQ-2: We found that the synchronous flow control methods we examined, the microservice controller, which suffers from the "double billing" problem, and the VM-client, both exhibited significantly higher costs compared to the asynchronous methods for all but the smallest dataset. This led to a cost increase of over 200% for the largest dataset. **This equates to**

a premium of ~$2,580 (137%) for using the microservice controller over the event-based method for one million pipeline executions using the 500,000 row dataset. This results demonstrates how flow control approaches that require renting a programmable client incur additional charges.

RQ-3: Our cold start experiment revealed that the latency of the microservice controller was most affected by cold infrastructure, exhibiting 9.6x more latency when compared to running the pipeline using the same dataset and control pattern on warm infrastructure. Event based triggers were least affected, experiencing only 1.7x times more latency for cold vs. warm infrastructure. Latency increased 6.8x for the state-machine, and 3.2x for the VM-client respectively.

RQ-4: When varying the memory allocated to the worker functions, the event-based triggers experienced the largest improvement in **pipeline runtime** at 43% when comparing performance for the smallest memory allocation (512 MB) to the largest (2048 MB). Scaling the allocated worker function memory from 512 MB to 1024 MB resulted in the largest **pipeline runtime** improvements for the microservice controller, state-machine, and VM-client, 26.5%, 25.9%, and 23.7% respectively, while **pipeline runtime** only dropped 14.6% for event based triggers between those memory settings. Event based triggers experienced the largest improvement from 1024 MB to 1536 MB, 21.7%. Overall, the event-based triggers pipeline experienced more relative improvement at higher worker function memory settings than the other three pipelines. All flow control methods improved the least from 1536 MB to 2048 MB.

RQ-5: Comparing the effects of implementation language and memory allocation on the performance and cost of the microservice controller pattern showed only minimal improvement in performance when providing the controller with additional memory for the Java implementation, 2.4%. The Python implementation showed no discernible relationship between

memory allocation and performance with the lowest memory setting performing over 2% better than the highest memory setting. Because the controllers spent the majority of their active time idling, allocating additional memory to the controller results in much higher costs, without significant performance improvement. The Java controller configured at the highest memory setting we measured (512 MB) was the fastest, while the Python controller configured at 128 MB was the cheapest.

6.2 Flow Control Development Perspectives

RQ-6: Considering the different capabilities of each flow control method is also important in choosing the right pattern for a given use case. Table 3 summarizes the difference in capabilities. Desired capabilities are highlighted in green.

Flow Control Method	Asynchronous	Built-in payload passing	Requires refactoring worker functions	Requires additional infrastructure	Vendor lock-in
Microservice Controller	No	Yes	No	No	Yes
Event-based Triggers	Yes	No	Yes	No	Yes
State-Machine	No	Yes	No	No	Yes
VM-Client	Yes	Yes	No	Yes	No

Table 3. Comparison of capabilities of flow control methods

Our findings from **EX-1** show that both synchronous methods of flow control experience much higher costs than the asynchronous methods and these costs continue to increase as the runtime of the worker functions increase. With this in consideration, synchronous methods offer multiple benefits. Much of the analysis reported in this thesis focuses on **pipeline runtime**,

which records the time elapsed from the start of the first service to the completion of the last. For synchronous methods this will be very close to **client runtime**, the time elapsed from when execution is triggered by the client, to the time when the client is notified that the pipeline has completed. Only a small amount of additional latency will be added for the trip to the server and back. For our asynchronous methods, the **pipeline runtime** and **client runtime** can be quite different as the results must be retrieved through polling which results in additional latency due the gap between pipeline completion and when the user next polls to retrieve the results.

For workflows that require dynamic data to be passed between services, implementing the event based triggers approach is the only method that requires refactoring the original pipeline's function source code. While this is by no means an insurmountable problem, it requires extra effort to implement this method of flow control. Additionally these source code changes that make specific changes to satisfy the event based trigger flow control approach reduce the immediate reusability of the functions source code which is undesirable from a software engineering perspective.

The VM-client method was the only method that required persistent infrastructure to be deployed. When calculating costs for this method, we calculated the cost of the VM based on **client runtime,** however, this is a simplification that assumes an active EC2 instance is already available. Having this EC2 instance idling clearly does not match the goals of serverless computing. It is possible to script the deployment of the client instance to minimize charges to only when the pipeline is called as AWS allows EC2 instances to be paused and restarted with only minimal charges for storage of the disk image, however this approach incurs additional latency for starting the EC2 instance.

The VM-client is the only method that is not completely tied to AWS. Although even with this method some refactoring would be required to migrate to another cloud provider in our

case because we were directly using the AWS CLI (Command Line Interface). A developer desiring a truly platform agnostic method of flow control could provide worker functions that are invoked through HTTP endpoints and trigger these functions using the cURL HTTP shell client.

RQ-7: Developer effort is a hard metric to quantify, but an important one as labor costs are a major factor for technology companies. We can address this subject from the perspective of initial development as well as maintenance and change tolerance. For a simple sequential workflow with no branches like the one studied here, AWS Step Functions provided the simplest initial development experience, providing an in browser GUI where the developer can define the workflow using familiar JSON. The VM-client and both versions of the microservice controller, Java and Python, required handling the dependencies needed to invoke the functions and process the JSON payloads. The Java controller presented the most difficulty here as it required the Maven build system to supply the necessary dependencies. For the Python controller, AWS Lambda provides an easy to use browser-based Python editor, and the environment that runs the code comes pre-loaded with all the necessary dependencies. The VM-client required installing the AWS CLI, as well as the jq JSON parser. The Python controller has the clear advantage between the synchronous flow control methods, both for initial development and maintenance, since updating the dependencies will be handled by the cloud provider instead of the developer. Using S3 triggers for flow control is clearly the loser from a development perspective because of the requirement of modifying the worker functions. Not only does this require additional upfront development, but it limits the reuse of the functions through other invocation methods, and requires additional coding to modify the flow control method later on.

6.3 OUTCOMES

This thesis provides a comparison of different methods of flow control for serverless applications that can help to demystify the tradeoffs that developers face when considering different serverless designs. Each of the four methods examined presented their own advantages and disadvantages. We found that choosing a synchronous method of flow control resulted in additional costs, while choosing an event-based system required refactoring worker functions and resulted in a performance penalty. For our use-case, the AWS version of the state-machine pattern, Step Functions, provided the best developer experience while also providing the best balance of performance and cost.

BIBLIOGRAPHY

[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] J. Su, "Amazon Owns Nearly Half Of The Public-Cloud Infrastructure Market Worth Over $32 Billion: Report," *Forbes*, 02-Aug-2019. [Online]. Available: https://www.forbes.com/sites/jeanbaptiste /2019/08/02/amazon-owns-nearly-half-of-the-public-cloud-infrastructure-market-worth-over-32-billio n-report/?sh=1c2568c729e0. [Accessed: 13-Dec-2020].

[3] Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uţă, 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] "SAAF: Serverless Application Analytics Framework." *UW Tacoma Cloud and Distributed Systems Research Group,* https://github.com/wlloyduw/SAAF.