Implications of Programming Language Selection for Serverless Data Processing Pipelines

Robert Cordingly, Hanfei Yu, Varik Hoang, David Perez, David Foster, Zohreh Sadeghi, Rashad Hatchett, Wes J. Lloyd
School of Engineering and Technology
University of Washington
Tacoma WA USA
rcording, hanfeiyu, varikmp, daperez, davidf94, zsadeghi, rhatch26, wlloyd@uw.edu

***Abstract*— Serverless computing platforms have emerged offering software engineers an option for application hosting without the need to configure servers or manage scaling while guaranteeing high availability and fault tolerance. In the ideal scenario, a developer should be able to create a microservice, deploy it to a serverless platform, and never have to manage or configure anything; a truly serverless platform. The current implementation of serverless computing platforms is known as Function-as-a-Service or FaaS. Adoption of FaaS platforms, however, requires developers to address a major question- what programming language should functions be written in? To investigate this question, we implemented identical multi-function data processing pipelines in Java, Python, Go, and Node.js. Using these pipelines as a case study, we ran experiments tailored to investigate FaaS data processing performance. Specifically, we investigate data processing performance implications: for data payloads of varying size, with cold and warm serverless infrastructure, over scaling workloads, and when varying the available function memory. We found that Node.js had up to 94% slower runtime vs. Java for the same workload. In other scenarios, Java had 20% slower runtime than Go resulting from differences in how the cloud provider orchestrates infrastructure for each language with respect to the serverless freeze/thaw lifecycle. We found that no single language provided the best performance for every stage of a data processing pipeline and the fastest pipeline could be derived by combining a hybrid mix of languages to optimize performance.**

Keywords— Serverless Computing, Function-as-a-Service, AWS Lambda, FaaS, Programming Languages

I. Introduction

Serverless computing recently has emerged as a compelling approach for hosting applications in the cloud [1][2][3]. Serverless computing platforms promise autonomous fine-grained scaling of computational resources, high availability (24/7), fault tolerance, and billing only for actual compute time while requiring minimal setup and configuration. To realize these capabilities, serverless platforms leverage ephemeral infrastructure such as MicroVMs or application containers. This serverless architectural paradigm shift ultimately promises better datacenter utilization as cloud providers can merge user workloads at the service-level to increase server utilization and save energy. Re-architecting applications for the serverless model promises reduced hosting costs as fine-grained resources can be provisioned on demand and charges reflect only actual compute time.

Function-as-a-Service (FaaS) platforms leverage temporary infrastructure to deploy, host, and scale resources on demand for individual functions known as "microservices" [4] [5] [6]. These microservices make use of function instances that contain user code plus dependent libraries and are created and destroyed on demand to offer granular infrastructure for each service [7]. Granular code deployments enable cloud providers to minimize idle servers better than with VM placements [8] [9]. Users are not billed based on the number of function instances, but instead on the total number of service invocations, runtime, and memory utilization to the nearest tenth of a second. Serverless platforms have arisen to support highly scalable, event-driven applications comprising of short-running, stateless functions triggered by events generated from middleware, sensors, microservices, or users [10]. Common use cases include: multimedia processing, data processing pipelines, IoT data collection, chatbots, short batch jobs/scheduled tasks, REST APIs, mobile backends, and continuous integration pipelines [5].

When developing a serverless application, developers make design decisions that directly impact the cost of hosting their application in the cloud. FaaS platforms allow functions to be developed and deployed in a variety of different programming languages and the set of supported languages varies across platforms. This paper investigates the implications of programming language selection on the overall performance and cost of a serverless application.

Unlike IaaS clouds, where cost accounting is as simple as tracking the number of VM instances and their uptime, serverless billing models are directly connected to the runtime of the application. Application deployments consist of many microservices that must be individually tracked [11]. As runtime is the primary factor in FaaS billing, it is important to design FaaS functions to be as fast as possible. FaaS platforms support only a limited number of programming languages, making the problem of selecting the best programming language for performance critical to minimize both runtime and cost. FaaS platforms encourage applications to be decomposed into many functions that are hosted and scaled separately with independent infrastructure. Decomposition of serverless applications into independent microservices allows applications to combine functions written in multiple languages. Aggregating functions written in different programming languages has the potential to offer a unique way to improve the performance of serverless applications, and in particular, data processing pipelines.

To save server capacity, cloud providers deprecate FaaS infrastructure after periods of inactivity, causing significant initialization latency to produce "cold" service requests [12]. Infrastructure recycling on serverless platforms causes a freeze/thaw cycle [13][14], that contributes to significant

performance variation. As programming languages feature different runtime environments on FaaS platforms, choice of programming language can substantially impact function instance initialization time. Without fully understanding the nature of FaaS platforms, developers are left to make ad hoc choices for programming language selection to avoid pitfalls such as the freeze/thaw lifecycle. These decisions can lead to slower applications with significantly higher hosting costs.

The primary goal of this paper is to investigate performance implications of programming language selection on serverless FaaS platforms. For serverless applications, the programming language that is expected to offer the fastest performance, in reality may not. *Many factors can obfuscate a language's expected performance.* For example, C# and Java exhibit greater cold start latency than interpreted languages such as Python resulting from overhead from initializing the Java Virtual Machine (JVM) or deploying required libraries (.NET) [15]. For short running functions, an interpreted language may be faster and cheaper than a compiled one. Conversely, long running functions may execute faster in a compiled language. Another factor is that across FaaS platforms, some language runtimes may have been the target of more optimizations by the cloud provider. Moreover, it cannot be assumed that functions in every language effectively scale the same. Many FaaS platforms scale the CPU timeshare relative to a function's reserved memory. We cannot assume that function performance scales identically for every language, as memory reservations are increased or decreased. The amount of memory that offers the best price-to-performance ratio for each language may vary.

To investigate these factors, we implemented an identical serverless Transform-Load-Query data processing pipeline in Java, Go, Python, and Node.js. We used static code analysis to compare similarity of each language's specific implementation. We then deployed our pipelines on AWS Lambda leveraging the simple storage service (S3) for object storage, and Amazon Aurora as a serverless relational database. Using this Transform-Load-Query pipeline as a case study, we investigate the implications of programming language selection on overall application performance, scalability, freeze/thaw initialization, and FaaS memory configuration performance scaling.

A. Research Questions

This paper investigates the following research questions:

RQ-1: (Performance) How does the choice of programming language (Java, Go, Python, Node.js) impact the overall performance and throughput of a serverless data processing pipeline?

RQ-2: (Scalability) How does the programming language choice impact the scalability of a serverless data processing pipeline when processing many concurrent data payloads?

RQ-3: (Infrastructure State) How does the choice of programming language impact cold FaaS performance compared to warm FaaS performance for a data processing pipeline?

RQ-4: (Memory/Cost) How does performance vary for a serverless data processing pipeline across alternate memory settings for implementations in different programming languages?

B. Research Contributions

This paper provides the following research contributions:

1. We investigate implications of programming language selection for serverless FaaS applications using a case study consisting of the multifunction Transform-Load-Query data processing pipeline written in Java, Go, Python, and Node.js. Using static code analysis, we verify that each language implementation is equivalent.
2. We profile each pipeline using metrics collected by the Serverless Application Analytics Framework (SAAF) [16], observing language specific performance, scalability performance, cold start latency, and how performance scales relative to memory size. We identify that hybrid pipelines that mix functions written in different languages can offer performance improvements over those in a single language.

II. Background and Related Work

The challenge of understanding pricing and performance of serverless platforms, including the need to address performance variation resulting from cold-start latency was identified in [17]. The authors identify how pay-as-you-go pricing models, and the complexity of serverless application deployments, leads to the key pitfall: *"Serverless computing can have unpredictable costs"*. When deploying serverless applications, developers make important choices that potentially impact the overall performance and cost of their applications. In contrast to hosting applications with VMs, serverless platforms complicate budgeting as organizations must understand service demand to estimate hosting costs. Features of FaaS platforms, such as the freeze/thaw lifecycle have been shown to favor some programming languages over others identified in [15]. In this section, we review related work on language performance comparison, performance evaluation of FaaS platforms, and language comparison within serverless applications.

A. General Programming Language Performance Comparison

In [18], L. Prechelt compared seven languages (C, C++, Java, Python, Perl, Rexx, TCL) to investigate language runtime, variability, and memory performance. Prechelt recruited volunteers of different skill levels and diversity to write programs in a variety of programming languages. Prechelt then obtained the runtime for these programs separated by language and group (e.g. C/C++ vs. Java, Java vs. scripting) and made comparisons using statistical tests.

These tests consisted of two stages, an initialization phase where files are loaded into memory, and a search phase where the file data is processed. For the initialization phase, programs in scripting languages ran at least 3.2x as long as those in Java. For the search phase, no significant differences were observed among any of the groups, but tests written in scripting languages exhibited 2.1x less performance variation than Java. Another performance metric compared was memory consumption where at least 20 MBs more memory (98 percent) were consumed by Java relative to tests written in scripting languages. In summary, Prechelt found that programs written in scripting languages ran at least 5.7% longer than Java implementations.

The performance of six different programming languages (Python, SML, C++, Java, Perl, C#, C) was compared in [19]. For comparison, the authors implemented programs that computed large numeric factorials using native language datatypes without dynamic memory. Their comparison focused

on identifying correctness of the factorial computation computing factorials from 1 to 999. Results show that Python was able to compute the longest correct factorial, while Perl was second longest.

However, [19] has several limitations. This paper did not compare source code differences using static metrics (e.g. LOC) and the authors did not compare runtime. Additionally, the comparison was limited to a mathematical use case. Their comparison did not consider application use cases common to serverless computing such as those with data-intensive operations and interactions with services.

B. Performance and Cost Evaluation of Serverless Platforms

Several efforts have investigated the performance implications for hosting scientific computing workflows on serverless platforms [20][21][22][23]. Other efforts have evaluated FaaS performance for machine learning inferencing [24][25], and even neural network training [26]. To support cost comparison of serverless computing vs. IaaS cloud, Boza et al. developed CloudCal, a tool to estimate hosting costs for service-oriented workloads on IaaS (reserved), IaaS (On Demand), and FaaS platforms [27]. CloudCal determines the minimum number of VMs to maintain a specified average request latency to compare hosting costs to FaaS deployments. FaaS resources, however, were assumed to provide identical performance as IaaS VMs when functions were allocated 128 MB RAM. Wang et al. identified AWS Lambda performance at 128 MB as only ~1/10th of 1-core VM performance in [9] suggesting potential inaccuracies with CloudCal. Other efforts have conducted case studies to compare costs for hosting specific application workloads on IaaS vs. FaaS [14][28], and FaaS vs. PaaS [29]. We extend previous efforts by characterizing performance variation of workloads across FaaS runtime implementations.

C. Language Comparison Within Serverless Applications

Jackson and Clynch compared latency of FaaS functions across different programming languages on AWS Lambda for Node.js, Python, Go, Java, and C#, and or Azure Functions with Node.js and C# in [15]. Their comparison, however, was limited to measuring latency using empty functions that performed no operations to quantify FaaS platform overhead.

In [30], Shrestha compared the performance of computing Fibonacci series as a compute-bound test case. Shrestha implemented the same task with all supported languages and compared the runtime while also benchmarking cold start performance. Node.js, Python, Go, Java, Ruby completed cold starts within 800ms, whereas C# was a distinct underdog with cold starts spanning between 0.8 and 5 seconds. Compiled languages (e.g. Java, Go, C#) demonstrated slower cold starts due to the large number of dependencies compared to interpreted languages (e.g. Node.js, Python, Ruby). Shrestha also noted that Java packages are large because of the requirements for the Java Virtual Machine (JVM), which then increased the overall size of the functions. Cold start overhead only impacted the cold start initialization, as the performance was excellent after the initialization phase. C# exhibited a much longer cold start time, which is also mentioned in [15]. This cold start overhead may result from the use of the open-source .NET CLR (Common Language Runtime) library on AWS Lambda, a Linux-based FaaS platform. Here, AWS Lambda is unable to leverage native .NET C# under the Windows operating system.

Another limitation of [30] is the simple compute-bound test case whereas many common applications written for FaaS platforms are data intensive. This is one of our motivations for choosing a data processing pipeline for our case study.

III. Methodology

In this section, we detail tools and techniques used to investigate our research questions (**RQ-1**, **RQ-2, RQ-3, RQ-4**). Section III.A describes the programming languages we investigated, their differences, and why we chose them. Section III.B describes the serverless Transform-Load-Query pipeline we developed to use as our programming language case study. Section III.C provides a discussion on the functional equivalence of our data processing pipelines in each language. Section III.D details our experimental workloads, and section III.E describes the tools and platform used to collect metrics, run experiments, and host our data processing pipelines.

A. Programming Languages: Java, Python, Go, Node.js

Each FaaS platform offers a different set of programming language runtimes that functions can be deployed with. For example, AWS Lambda supports functions written in Java, Python, Go, Node.js, Ruby, and C# [31]. For this case study, we focused on Java 8, Python 3.7, Go 1, and Node.js 12. These four languages feature major differences in both language design and FaaS platform implementation. Go, Java 8, and Python 3.7 use Amazon Linux 1 whereas Node.js uses version 2.

We focused on Java and Python to compare two fundamentally different programming languages. Python is a high-level interpreted programming language while Java is compiled to platform independent byte code that is run on each platform using a custom interpreter known as the Java Virtual Machine (JVM) [18]. Being a compiled language, Java offers better performance compared to interpreted languages such as Python [18]. While Java may be faster, on FaaS platforms the JVM has been shown to cause significant cold start latency [15]. The two interpreted and compiled language classes also apply to Go (compiled) and Node.js (interpreted). The goal for selecting these languages is not simply to make a direct performance comparison, but to also identify characteristics of use cases (i.e. compute-bound, I/O-bound, small vs. large data) where one programming language excels above the others.

B. Transform-Load-Query Pipeline Use Case

To investigate performance impacts of programming language selection, we developed a Transform-Load-Query multi-function serverless application in all four languages [32]. We built the pipeline to process 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. To process this data, the pipeline consists of three microservices:

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.

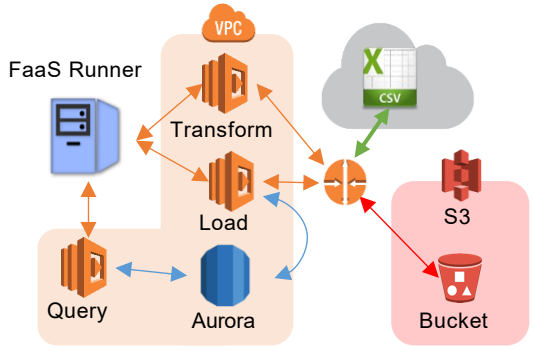


Fig. 1. Transform-Load-Query Data Processing Pipeline Services and Tools

C. Static Code Analysis and Function Equivalence

To ensure function equivalence between the different language implementations, we needed to ensure that the services for each language were as similar as possible in design and structure. To achieve this, we wrote the Java version first and used it as the reference implementation to write the Python, Go, and Node.js implementations.

TABLE I. STATIC CODE ANALYSIS METRICS BY SERVICE AND LANGUAGE. INCLUDES NUMBER OF FUNCTIONS, VARIABLES, SOURCE LINES OF CODE, LOOPS, AND CLOUD SERVICE USAGE

Service	Lang	Funcs	Vars	SLOC	Loops	Cloud Service Usage
S1	Java	3	40	86	2	S3 Get/Put
S1	Python	3	28	64	3	S3 Get/Put
S1	Go	3	30	77	1	S3 Get/Put
S1	Node.js	3	24	96	1	S3 Get/Put
S2	Java	3	25	77	2	S3 Get, DB Conn x1
S2	Python	3	21	57	3	S3 Get, DB Conn x1
S2	Go	3	15	65	1	S3 Get, DB Conn x1
S2	Node.js	4	18	83	1	S3 Get, DB Conn x1
S3	Java	4	36	111	7	S3 Put, DB Conn x2
S3	Python	5	44	96	9	S3 Put, DB Conn x2
S3	Go	4	34	104	8	S3 Put, DB Conn x2
S3	Node.js	5	17	74	1	S3 Put, DB Conn x2

The focus was to translate the Java implementation as closely as possible into the target language. We did not employ language specific optimizations and retained the same logic across all implementations. Due to fundamental language differences, such as required use of callback functions and frequent asynchronous code, static code analysis metrics for Node.js exhibited the largest differences while being functionally equivalent. We also minimized the use of 3rd party libraries, excluding those required to interact with AWS (e.g. boto3), to eliminate as much under-the-hood code as possible. Our goal was to replicate the behavior and implementation of each language-specific function implementation. This was especially challenging with Aurora database interactions. Each language has its own MySQL driver and library with different features, configurations, and limitations to consider. Table I provides static code analysis metrics to compare our pipelines implemented in Java, Go, Python, and Node.js.

D. Experiments

To investigate the implications of programming language choice for our serverless case study, we conducted four experiments to evaluate different aspects of the FaaS language runtimes.

1. Overall Performance Comparison

In this experiment, we executed each pipeline 11 times with 8 different workload sizes gradually increasing the size of the data payload processed by the pipeline. The first run of each workload was thrown out to prewarm FaaS infrastructure to ensure execution was in the warm state. The number of rows in the data payload was gradually increased as follows: 100, 1000, 5000, 10000, 50000, 100000, 250000, and 500000 rows. The goal of this experiment is to measure the overall warm performance of each language and measure how performance scales as the workload size changes (**RQ-1**). All runs were configured with the maximum memory setting (3008 MBs), and executed sequentially to minimize tenancy and resource contention between function instances. Alongside observing runtime, we also use these runs to profile Linux CPU metrics to evaluate function resource utilization using SAAF (described in section D) to compare how the processing requirements differ for each language.

2. Scalability Performance Testing

The goal of this experiment is to measure how the performance of each language changes as the number of concurrent function invocations increases (**RQ-2**). All functions execute the same 100,000 row workload, and we gradually increased the number of concurrent invocations. Starting with one function invocation to warm up infrastructure, we increased function calls in steps of 5 up to 50 concurrent invocations. We performed this experiment with the maximum memory setting (3008 MBs) to force the cloud provider to assign our application the maximum amount of infrastructure. To eliminate database resource contention, each pipeline was allocated a dedicated Amazon Aurora serverless database instance. This limited our scalability testing as the maximum number of databases serverless Aurora supported in one user account was 50.

3. Cold-Start Performance Testing

In all previous experiments, function infrastructure was prewarmed to profile warm function performance. The goal of experiment 3 was to measure cold start performance for each language (**RQ-3**). Similar to experiment 1, each function was called sequentially with a fixed memory setting (3008 MBs). To

minimize network latency, an EC2 instance was used as the client to invoke these functions. We created an c5n.large EC2 instance in the same availability zone and subnet of our functions (us-east-1a). We note that our functions were deployed using a virtual private cloud (VPC) to fix their placement to the us-east-1a availability zone. After each pipeline iteration, the client executing the experiment slept for 1 hour to guarantee cold infrastructure. All function invocations used the smallest 100 row workload since the metric this experiment measured was cold start latency. Sleeping 60 minutes ensured the FaaS platform deprecated FaaS infrastructure between calls returning each function to a cold state. A previous study showed that on average AWS Lambda required ~45-minutes to deprecate all warm FaaS Infrastructure for a given function [14]. We used the SAAF framework to identify infrastructure state, and verified that indeed all FaaS infrastructure used to execute functions in this experiment was cold.

4. Memory Configuration Comparison

The fourth experiment focused on identifying performance differences for different programming languages with respect to a FaaS function's memory reservation size. For this experiment, each function was called sequentially and with a fixed workload. After each test, we reduced the memory setting. Starting with the 3008 MBs memory setting, we repeated the experiment with 2560, 2048, 1536, 1024, 768, and 512 MBs memory settings. The goal was to measure how performance changes in each language as we changed the memory configuration (**RQ-4**).

E. Tools and Platforms

To help identify factors responsible for performance variation on FaaS platforms, while quantifying their extent, we have developed the Serverless Application Analytics Framework [16]. SAAF supports collection of 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 appending them onto the JSON payload returned by the function instance. Attributes collected include Linux Time Accounting metrics such as CPU idle, user, kernel, and I/O wait time, wall-clock runtime, and 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`**. To generate concurrent FaaS workloads, retrieve metrics, and aggregate results from SAAF we developed FaaS Runner. Implemented in Python, FaaS Runner provides a client-side application used in conjunction with SAAF, to automate profiling experiments on FaaS platforms. FaaS Runner combines performance, resource utilization, and configuration metrics from SAAF enabling observations not possible when profiling individual FaaS functions calls

For this FaaS programming language comparison, we deployed each of the data processing pipelines to AWS Lambda, stored sample datasets in S3, and leveraged the serverless Aurora relational database. We focused our study using AWS as it is the only platform that presently offers a horizontally scalable serverless relational database. Combing AWS Lambda, S3, and Aurora allowed our pipeline to be fully serverless.

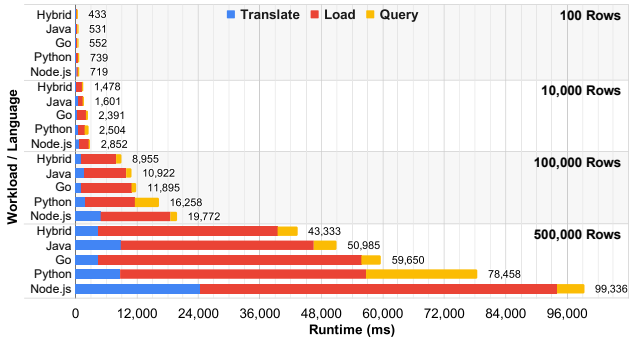
IV. Experimental Results

To investigate our research questions, we deployed each of our data processing pipelines to AWS Lambda using the default VPC in the Virginia region. We pinned all Lambda functions to execute within the same availability zone (us-east-1a) to minimize network latency and increase the likelihood that experiments ran under identical conditions.

A. FaaS Language Performance Comparison

In the vast majority of the trials, Java had the lowest runtime. The only exception is where Go outperformed Java with the 250,000-row workload. **On average across all workloads, Go took 13.1% longer compared to Java, Python took 45.9%, and Node.js took 64.6%.** Processing large datasets exacerbated the difference between Java and Node.js performance. Node.js required 94.8% more runtime to process the 500,000 row dataset than Java. Our results show that language choice of a serverless function can have a significant impact on the overall runtime and price of a serverless application. Figure 2 shows the average runtime of each function and workloads.

Fig. 2. Function Runtime with workload sizes of 100, 10000, 100000, and 500000. Some workloads are excluded for simplicity. Hybrid pipeline consists of the Go Translate/Query and Java Load functions.



For comparison purposes for a serverless application, it is useful to extrapolate the cost to run each pipeline 1,000,000 times. For the worst-case scenario to process the 500,000-row dataset the price for 1,000,000 pipeline invocations would be on average $2,549, $2,982, $3,922, and $4,967 for Java, Go, Python, and Node.js respectively. **By implementing a data processing pipeline in Node.js instead of Java, a serverless application may cost ~95% more for 1,000,000 pipeline invocations.** This price comparison shows how FaaS runtime directly determines the cost of a serverless application.

Across our four single language pipelines, the Load phase was by far the slowest. This phase took on average 69%, 60%, 56%, and 59% of the entire pipeline's runtime in Go, Java, Python, and Node.js respectively. An interesting observation is that no single language was consistently the fastest across all functions. On average, Go delivered the fastest runtime for the Transform function, Java for the Load function, and Go for the Query function. Go's long runtime of the Load function causes it to have a longer overall runtime than Java: 14010ms vs. 12927ms respectively.

From our available implementations, the fastest pipeline would be to use Go for the Transform and Query functions, and Java for the Load function. This demonstrates a potential benefit for building decoupled multi-function pipelines on serverless platforms. The freedom of being able to mix multiple languages

and tools allows developers to create optimal ***hybrid*** pipelines. Compared to our single language pipelines, our Go/Java/Go hybrid pipeline processed the 500,000 row dataset 37%, 17%, 81%, and 129% faster than the Go, Java, Python, and Node.js pipelines respectively.

Using SAAF, we are able to profile Linux CPU Time Accounting metrics for our functions. This provided insight into what functions are doing while running. We can observe how much time is spent executing user code, idling, waiting for network I/O, or executing privileged kernel code in the operating system. Figure 3 shows the profile of each function at 3008 MBs running the 500,000-row workload. This graph depicts differences in resource utilization for each language's function implementations.

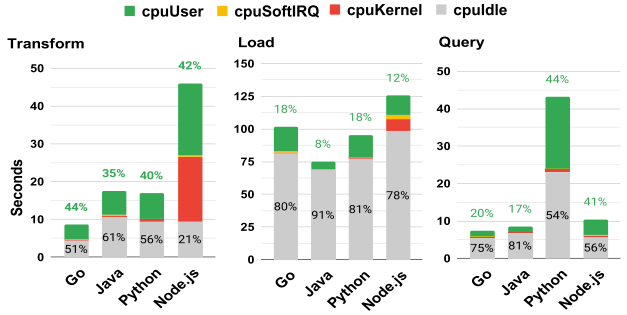


Fig. 3. Linux CPU Time Accounting profile for each function in the pipeline at 3008 MBs with 500,000 row workload.

Across all implementations, Node.js required more kernel time than other languages. For the first function (Transform), Python and Java had similar resource utilization, while Go required less CPU User and Idle time. Node.js required a significant amount of Kernel time, consisting of 37% of the total runtime. For function 2 (Load), all functions exhibited a significant amount of CPU Idle time of ~70-100 seconds. Go, Python and Node.js required a similar amount of CPU User time while Java used about half. Like function 1, Node.js required much more Kernel time than the others and exhibited more soft interrupt request time potentially as a result of function callbacks. This function is more network bound due to loading data into the database so it was expected that the vast majority of the time the CPU would be idle. For function 3 (Query), Go, Java, and Node.js performed similarly, while Python required significantly more user and idle time to execute the same task as the other two languages. Beyond basic runtime, analyzing Linux CPU Time Accounting metrics affords a deeper understanding of what the aspects of a workload some languages excel at vs. others. For example, our Python MySQL driver loaded data very quickly (`INSERT`), similar to the other two languages, while query performance (`SELECT`), presumably for data retrieval, was significantly slower.

B. FaaS Platform Scalability Comparison

One of the most important features of a FaaS platform is how quickly they respond to demand and scale up the amount of infrastructure allocated to an application. Once many function instances are deployed, performance can be impacted by multi-tenancy and resource contention. In this test, we gradually increased the number of concurrent function invocations to monitor if there was any change in performance as the number of concurrent requests increased. Each pipeline was provided an independent AWS Aurora Serverless (MySQL) relational database instance.

Figure 4 shows the average runtime of the pipelines as the number of parallel requests in the trial were scaled up. The Python, Java, and Go pipelines show minor linear increases in runtime as the number of concurrent invocations increase. Conversely, between 20 and 35 concurrent invocations of Node.js, there was a large ~5000ms increase in total pipeline runtime. **At 50 concurrent invocations, there was 8%, 9%, 19%, and 35% increase in runtime for Python, Java, Go and Node.js respectively compared to 1-20 invocations.**

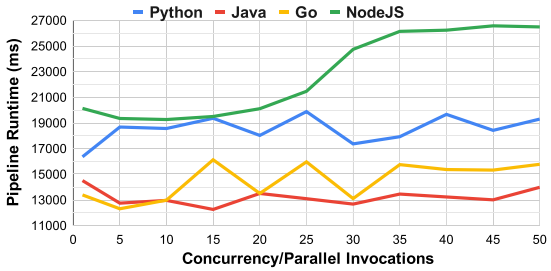


Fig. 4. Overall Pipeline Runtime vs Concurrency.

C. Runtime Cold-Start Performance

FaaS platforms dynamically scale the amount of infrastructure used to host function instances. Given that allocation of function instances is not instantaneous, initial function invocations exhibit "cold-start" latency.

In experiment 3, we invoked our pipelines twice to observe both cold and warm latency and then waited one hour to allow AWS Lambda to deprecate FaaS Infrastructure. We calculated latency by first recording the client's time immediately before a function invocation, and after receiving the response; also known as the round-trip time. We then subtracted the runtime reported by SAAF from the round-trip time giving us the overall latency caused by networking and infrastructure provisioning.

TABLE II. FUNCTION COLD AND WARM-START LATENCY BY SERVICE, COLD VS WARM DELTA, AND FUNCTION PACKAGE SIZE. ALL FUNCTIONS RAN USING THE DEFAULT VPC IN THE US-EAST-1A AVAILABILITY ZONE

Lang	Cold (ms) (Service T, L, Q)	Warm (ms) (Service T, L, Q)	Delta (ms) (Service T, L, Q)	Package Size (Service T, L, Q)
Go	871, 886, 886	402, 402, 406	469, 484, 436	15, 15, 7.8 MBs
Java	988, 1119, 1139	401, 395, 398	587, 724, 741	6.3, 8.7, 8.7 MBs
Python	962, 1034, 1033	408, 406, 410	554, 628, 622	6, 173, 174 KBs
Node.js	889, 1140, 1076	390, 389, 388	499, 751, 688	704, 705, 707 KB

Table II shows how the cold start latency, and overall cold vs warm speedup varies between each language. On average, cold invocations had latency of 867, 1082, 1010, and 1035 milliseconds for Go, Java, Python, and Node.js respectively with an overall Coefficient of Variation (CV) of 10.8%. Once functions were warmed, each language had nearly identical latency averaging 404, 398, 408, 389 milliseconds for Go, Java, Python, and Node.js with a CV of just 1.9%. The difference in latency between cold start and warm functions averaged 463, 684, 602, 645 milliseconds for Go, Java, Python, and Node.js respectively.

Our results provide two interesting observations. On average, Java had the most cold-start overhead; this was expected. Alongside that, Node.js and Python were only 39ms and 82ms better than Java. We assumed that these interpreted

languages would be much better than Java. It should be noted that cold-start overhead is higher when functions are deploying using a VPC. Another observation is that package size does not appear to impact cold-start overhead across the languages. The Python Transform function, with its tiny 6 KB package, still had more cold start latency than Go's 14.9 MB package. When comparing package sizes for the same language, the smaller 6.4 MB package for Java Transform did result in less cold start latency versus the 8.7 MB packages for the Load and Query functions (587ms with 6.3 MB package and ~730ms with 8.7 MB package).

Cold start latency is one of the greatest examples of FaaS language bias that a developer making a serverless application cannot do anything about. Languages may offer better performance on some FaaS platforms vs. others as a result of each platform's specific language runtime implementation. For example, by developing an application in Java rather than Go on AWS Lambda, an application will always have more cold start latency (here ~221ms) because of how Lambda initializes function instances. For frequently invoked long running applications this runtime may be insignificant, but for every function in an application it compounds the latency. **Choosing Java, Python, or Node.js instead of Go resulted in an average of 20.2% more cold latency.**

D. Memory Configuration Performance Scaling

We profiled the resource utilization for each function using Linux Time Accounting metrics [33] to determine how much time each function spent executing user code, idling, waiting for network I/O, or executing code in the operating system's kernel. AWS Lambda throttles performances of functions with lower reserved memory not by reducing clock speed, but by adjusting the CPU time share to effectively increase CPU idle time when a function is running. Figure 5a depicts how as the memory setting increases or decreases, the CPU user time remains fairly constant. Conversely, Figure 5b shows that when the memory setting is decreased, the CPU idle time increases.

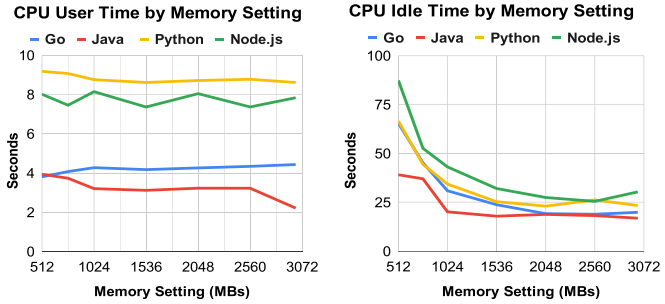


Fig. 5. (a) Left CPU User Time (b) Right CPU Idle Time

Altering the CPU time share has a massive impact on performance. Ideally, function performance on AWS Lambda should scale linearly with the memory setting. If a function invocation has double the memory as another, the runtime should be twice as fast. Perfect scaling of the CPU time share would produce perfectly balanced billing. Regardless of the memory setting, functions would always have the same cost, with the only difference being that with higher memory settings the same work is performed much faster. In this perfect scenario, the highest memory setting would always be the best choice. With our case study, this is not the case for multiple reasons.

Figure 6 shows how performance of each pipeline scales based upon the memory setting. In this graph, pipeline runtime is normalized to a percentage of each memory setting's runtime compared to the maximum memory setting. This allows us to compare differences between how performance of each language scales relative to the memory setting. With perfect scaling, 3008MBs would be 100%, 1536MBs would be 50%, 768MBs would be 25% and so on.

For our pipelines, we did not see significant performance improvements above 1536 MBs. In some trials, higher memory settings produced worse performance than lower memory settings. For example, the Go pipeline took on average 1895ms with 2560 MBs of RAM while it took 1992ms with 3008 MBs. The primary reason we did not see performance improvements after 1536 MBs was because our functions are all single threaded. Above 1536MB, functions gain access to a second CPU core on AWS Lambda which our functions do not utilize. This provides an example of how FaaS platforms obscure pricing. Without testing, a developer may assume that 3008 MBs would always be the fastest memory setting when they actually get nearly equivalent performance to 1536 MBs while then paying double.

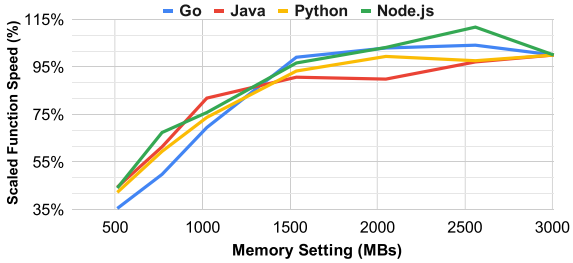


Fig. 6. Relative Performance Decrease of Low Memory Settings

With low memory settings, we did not observe perfect scaling as memory size was decreased. Going from 1024 MBs to 512 MBs relative function speed dropped from 69%→35% (x0.51), 82%→44% (x0.54), 74%→42% (x0.57), and 76%→44% (x0.58) for Go, Java, Python, and Node.js respectively. At 1024 MBs and below, Go consistently saw lower scaled performance compared to the other three languages with memory settings lower than 1536 MBs. **For 1,000,000 invocations, choosing the 3008 MB setting instead of 1536 MB would result in paying $310, $220, $374, and $581 more for Go, Java, Python, and Node.js respectively.** This equates to paying up to 95% more for a mere 4.8% performance improvement.

V. Conclusions

In this paper, we developed a multi-function Transform-Load-Query data processing pipeline in Java, Python, Go, and Node.js. Using these pipelines, we investigated the overall performance variation, cold start latency, scalability, and implications of memory size. We investigated how programming language selection impacts the efficiency of serverless data processing pipelines with a series of experiments.

Our research findings include: **RQ-1:** Executing each pipeline with varying workload sizes showed which language was able to run each function of the pipeline the fastest. Go had the lowest runtime for the Transform and Query functions, while Java performed the Load function the fastest. Due to poor Load function performance in Go, Java achieved the best performance

on average for the entire pipeline. For these workloads, choosing Node.js instead of Java resulted in 94% higher costs. We also identified the potential for performance improvements by building hybrid pipelines that combine functions written in multiple languages. The fastest configuration was a pipeline using Go for Translate/Query functions, and Java for the Load function. Our hybrid pipeline provided performance improvements ranging from 17% to 129% compared to single language pipelines. **RQ-2:** Running up to 50 concurrent instances of our data processing pipeline introduced minor linear increases in runtime for Python, Java, and Go (8%-19% increase in runtime). Node.js was affected much more negatively (35% runtime increase). **RQ-3:** Our cold state latency testing for each language agreed with previous research in [15] where Java had the highest cold start latency, while Go was found to outperform every other language. Reducing package size reduced cold latency when comparing functions in the same language, but not when comparing functions from different languages (e.g. Python and Go). Due to cold start latency, using Java, Python, or Node.js to process small datasets was 20% more expensive than Go. **RQ-4:** Performance scaled approximately linearly for memory sizes up to 1536MBs for each language. For memory settings higher than 1536MBs, there was no major performance improvement, but significant increases in hosting costs (95% cost increase for 4.8% performance improvement). Notably, runtime of our Go pipeline was impacted more at memory settings below 1024 MBs, resulting in lower relative performance compared to the other languages.

This paper provides a comparison of a multi-language serverless data processing pipeline that developers can refer to when considering serverless designs. Overall, there is likely no definite best language for all serverless applications. Ultimately developers should consider resource requirements, and profile performance to optimize their designs before deployment on serverless platforms to mitigate any potential pitfalls.

ACKNOWLEDGMENTS

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

REFERENCES

[1] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, 2016, p. 5.

[2] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[3] I. Baldini *et al.*, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*, 2017.

[4] A. Sill, "The Design and Architecture of Microservices," *IEEE Cloud Comput.*, 2016, doi: 10.1109/MCC.2016.111.

[5] "Openwhisk common use cases." https://console.bluemix. net/docs/openwhisk/.

[6] "Fn Project – The Container Native Serverless Framework." https://fnproject.io/.

[7] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean Lambdas with Large Libraries," in *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, 2017, doi: 10.1109/ICDCSW.2017.32.

[8] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 2018, doi: 10.1109/IC2E.2018. 00039.

[9] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," *2018 USENIX Annu. Tech. Conf. (USENIX ATC 18)*, 2018.

[10] I. Baldini *et al.*, "The serverless trilemma: Function composition for serverless computing," in *Onward! 2017 - Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, co-located with SPLASH 2017*, 2017, doi: 10.1145/3133850.3133855.

[11] A. Eivy, "Be Wary of the Economics of 'Serverless' Cloud Computing," *IEEE Cloud Comput.*, 2017, doi: 10.1109/MCC.2017.32.

[12] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," 2017, doi: 10.1145/ 3106237.3117767.

[13] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for container-based architectures," *Futur. Gener. Comput. Syst.*, 2018, doi: 10.1016/j.future.2018.01.022.

[14] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley, "Improving application migration to serverless computing platforms: Latency mitigation with keep-Alive workloads," in *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, 2019, doi: 10.1109/UCC-Companion.2018.00056.

[15] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," in *Proceedings - 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion, UCC Companion 2018*, 2019, doi: 10.1109/UCC-Companion.2018.00050.

[16] "SAAF: Serverless Application Analytics Framework." https://github.com/wlloyduw/SAAF.

[17] E. Jonas *et al.*, "Cloud programming simplified: a berkeley view on serverless computing," *arXiv Prepr. arXiv1902.03383*, 2019.

[18] L. Prechelt, "Empirical comparison of seven programming languages," *Computer (Long. Beach. Calif).*, 2000, doi: 10.1109/2.876288.

[19] P. Singh, S. Shukla, S. Chandra, and V. Dixit, "Performance evaluation of programming languages," in *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, 2017, pp. 1–4.

[20] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: the prospect of serverless scientific computing and HPC," in *Communications in Computer and Information Science*, 2018, doi: 10.1007/978-3-319-73353-1_11.

[21] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," *Future Generation Computer Systems*, 2017.

[22] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "Serverless execution of scientific workflows," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, doi: 10.1007/978-3-319-69035-3_51.

[23] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018, doi: 10.1007/978-3-319-75178-8_34.

[24] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, 2018, doi: 10.1109/IC2E.2018.00052.

[25] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, Jun. 2019, pp. 23–33, doi: 10.1109/IC2E.2019.00-10.

[26] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring Serverless Computing for Neural Network Training," in *IEEE International Conference on Cloud Computing, CLOUD*, 2018, doi: 10.1109/CLOUD.2018.00049.

[27] E. F. Boza, C. L. Abad, M. Villavicencio, S. Quimba, and J. A. Plaza, "Reserved, on demand or serverless: Model-based simulations for cloud budget planning," in *2017 IEEE 2nd Ecuador Technical Chapters Meeting, ETCM 2017*, 2018, doi: 10.1109/ETCM.2017.8247460.

[28] M. Villamizar *et al.*, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*, 2016, doi: 10.1109/CCGrid.2016.37.

[29] L. F. A. Jr, F. S. Ferraz, R. F. A. P. Oliveira, and S. M. L. Galdino, "Function-as-a-Service X Platform-as-a-Service : Towards a Comparative Study on FaaS and PaaS," *Twelfth Int. Conf. Softw. Eng. Adv. Funct.*, 2017.

[30] S. Shrestha, "Comparing Programming Languages used in AWS Lambda for Serverless Architecture," Master's Thesis, Metropolia University of Applied Sciences, 2019.

[31] Amazon, "AWS Lambda - Serverless Compute," *Amazon Web Services, Inc*, 2014. .

[32] "Multi-function Translate Load Query Data Processing Pipeline." https://github.com/wlloyduw/FaaSProgLangComp.

[33] W. J. Lloyd *et al.*, "Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives," *IEEE Trans. Cloud Comput.*, 2015, doi: 10.1109/tcc.2015.2430339.