# A serverless system architecture for latency-insensitive data processing tasks

Authors: Philippe Louchtch, Ankush Sharma, and Tim Speelman
{P.A.Louchtch, a.sharma-20, T.Speelman-1}@student.tudelft.nl
Course instructors: Dick Epema, D.H.J.Epema@tudelft.nl
Lab assistant: Sobhan Omranian Khorasani, S.OmranianKhorasani@tudelft.nl

## ABSTRACT

Whilst cloud computing has proven to be a valued development, it still has a lot of room for further improvement. One of the emergent paradigms, serverless computing, or Function-as-a-Service (FaaS), aims to make developing cloud-based applications easier, faster and cheaper. We propose and evaluate a serverless, share-nothing architecture built on the Amazon AWS Cloud platform, for easy development of scalable, latency insensitive data processing applications. Using the system, we build a very simple, data processing toy application and analyse the functioning of the features of the system along with its performance. The system functions well and is easy to work with. However, the performance of the application and the supporting functionality of Amazon AWS for serverless systems is lacking.

## INTRODUCTION

Even though cloud computing and *Iaas* (Infrastructure as a Service) by some was cautiously regarded as a passing technological fashion or considered being merely a set of buzzwords, it has become a firmly entrenched concept that has proven its value. While services such as Amazon AWS, Microsoft Azure and Google Cloud have greatly lowered the bar of entry through continuous improvements to the ease of use, management and price models, the utilization of the resources in datacentres remains lower than desired. This is a problem for both the users and the providers of cloud services due to missed income for the providers and over-paying for actual capacity used for the users. Another issue that the users face, is still the need to maintain, administer and monitor the fleet of leased resources. Even though the physical machine is no longer a property, nor a worry of the user, the software side of things still need to be maintained. With the proliferation of Software as a Service (*SaaS*) offerings of contemporary and continued desire of abstracting away the physical and even the operating system, comes the idea of deploying the code directly to the cloud. The code can be executed as response to events or explicit invocations through the API of the service provider. Whole applications developed in such manner have even become its own class of architecture, called "serverless". The most enticing reason for using this technology is to only pay for the actual computation time used, ease of deployment and seemingly boundless scalability on-demand.

To investigate the maturity of the service offering, the performance potential and practical scaling potential of such approach to building applications, we design and evaluate a system/framework for latency-insensitive, data processing tasks. The system is completely serverless, based on the Amazon AWS Lambda *Function-as-a-Service* (*FaaS*) service. Furthermore, it leverages the Amazon AWS SaaS services, S3 for persistence and SQS queueing service for communication between functions.

Other products exist for big-data processing workloads, some optimize for low latency (Murray 2013) (Wu 2015), fault tolerance [REF: Epema Spark paper], efficient system utilization (Shao 2018), cost (Yan 2016), programming model (Murray 2013) or combinations thereof. These products work in predefined cluster environment and do not dynamically auto-scale to accommodate for the load. Our contribution is the investigation of the feasibility of the presented system design for data processing that makes use of the serverless paradigm to allow for auto-scaling to meet dynamic demands of user submitted job requests.

This report presents the background of the application and its requirements in Section 2. Then explains the System design in detail Section 3. Lastly, it provides the Experiment setup and analysis in Section 4 before ending with a discussion and conclusion in Section 5.

## BACKGROUND ON APPLICATION

Before designing the system, we must consider the types of applications it must support. In order to ease the design of the system, we considered the class of applications of high degree of data parallelism. One such application is the still popular case of social media analysis, particularly Twitter. Each message consists of a short amount of text, can contain geolocation information and has a lot of potential, practical uses ranging from sentiment analysis to warning systems (earthquakes, fires etc).

Consider the case of gathering information on where possible fires occurred in Great Britain. Given a time range, the application would filter tweets originating from Great Britain containing some, or combination of, specific keywords and/or hashtags such as "fire" and ~~"fire truck"~~ "fire engine" and counted and grouped by the city. Such application can be composed from a linear sequence of independent operations. Furthermore, implementing a single application does not make much sense. Data scientists might want to be able to prototype various ideas fast or create one-shot type of applications. Therefore, from such data processing ideas we distil an a system architecture that allows for fast prototyping of such applications and can handle changing demands from user-submitted job requests.

We formulate the requirements on this system as an easy to read bullet point list:

- Serverless architecture
- Share-nothing design, for concurrent users
- Persistence/caching of results, prevents double-work
- Queue-based communication, to decouple the components of the system. Making scaling easy to implement
- Auto-scaling to process large jobs fast, keeping costs low for smaller jobs
- System automation (no user intervention in efficient execution of the jobs or management of the system)

## SYSTEM DESIGN

Throughout this report we will refer to both "*invocations of*" and "*lambda function*" itself, simply as a "*lambda*". The meaning should be obvious from the context. Furthermore, from the perspective of the system discussed in this report, we shall use the verb *"spawn"* to refer to the act of *invoking* a lambda function.

## General overview

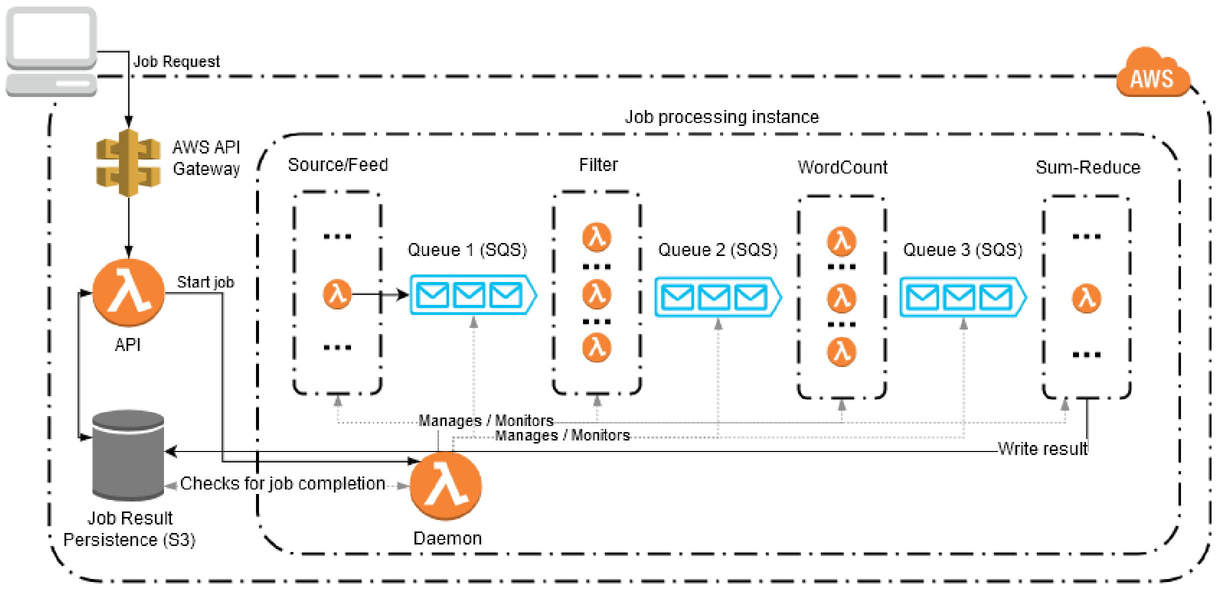

**Figure 1: simple application in system**

As indicated in the background section / introduction, one of the system requirements is to be generic to allow for fast, easy composition or implementation of new processing flows. In other words, to be framework for simple big data processing flows. Once deployed, the system accepts *job requests* containing the job type and parameters which it maps to job results. An example: a `TwitterCountFireTweetsJobRequest` with parameters `(country: GB, from: 1 nov 2018, to: 3 nov 2018)` will start a job that counts all tweets originated in Great Britain, between 1st of nov and 3rd of nov of 2018, that mention fire. Each job type must map to a defined job flow in the system. A job definition is an ordered set of specific processing flow components with possibly some job specific system settings such as scheduling constraints.

The system consists of three main parts, all of which are lambdas: the API, the daemon and the processing components. A processing flow can be best imagined as a directed, acyclic graph where the components are the vertices and the queues between them are the edges.

- The *API lambda* receives job requests and either spawns a daemon for that job or, if the job is already running or has been completed before, returns the status or result of that job. The status/result is retrieved from storage (S3) and is identified by the job-request itself.
- A *daemon lambda* handles the provisioning and orchestration of the processing flow of that job (**automation**). In other words, it spawns all the lambdas for that job, creates the SQS queues, and monitors and scales (**elasticity, performance**) the workers[1].
- The *processing flow components* are the lambdas that do the actual work of processing the job. Each data source/feed, word-counter, summing-reduce are all implementations of such components. There are three main types: those that only send (data feed or source), those that only receive (terminal operations) and those that both receive and send. An instance/invocation of any of these components is also referred to as a *worker*.

Each of the instances of process flow components, and the daemon itself, are exclusive to that job (shares nothing) and therefore are is not impacted by any of the other jobs (**load-balancing**).

The persistence of the job result is handled by the terminal operation in the job processing flow. This persistence in our case is both **caching** and **durability** of results. Caching of intermediary results can

[1] A lambda instance of any processing flow step

be achieved, but is not implemented, for some types of jobs through partitioning it into smaller jobs and generating the final result from the combination of the smaller jobs.

## Resource Management Architecture

In the system architecture, the daemon is the orchestrator of the job. By orchestrator we mean that it is the sole responsible component for the completion of the job. It initializes the processing flow by creating all the queues, sets the job status to processing and uses the chosen or job-defined scheduler policy to spawn all the other components. Furthermore, the system has a set of **monitoring** facilities in the form of "*Xyz*`Metrics`" classes which can be used by any component in the system. Due to the additional complexity of tracking status of lambda invocations, spawn counts are maintained by the scheduler itself. For monitoring SQS queue sizes, there is a polling implementation that can provide statistical and snapshot information on the approximate[2] number of messages in the given queue. A scheduler can make use of these for its scheduling decisions.

Concerning reliability, the system design is tolerant to non-byzantine faults of all lambdas except for the daemon. The **reliability** is achieved by leveraging the reliability of the S3 and SQS services by the system architecture. When receiving messages from SQS, the message is only removed from the queue through an explicit delete operation. However, a message that has been delivered will temporarily become invisible and not receivable until a timeout occurs. In our system, the receive operation will automatically delete the massage if and only if it has been processed successfully. If the machine in AWS that the lambda is running on fails or loses connectivity, the message processing-time will expire and the message becomes available to other lambdas. If the message fails processing due to code bugs, the message will end up in a dead-letter-queue for human analysis and bug-fixing. The rest of the system can still complete the job with the remainder of the messages. The validity of the result then depends on the required accuracy of the result. The API is resilient only to machine failures and is managed by the AWS API Gateway service. Making the daemon reliable is possible through various techniques, but we considered this to be outside the scope of the first system design and implementation.

## System Policies

The system has facilities for implementing various types of scaling policies. We have implemented two basic ones: AlwaysOne and StaticProportional. The AlwaysOne policy only ensures that a single lambda is running for each component. The StaticProportional is specific to the job types that have a known *a priori* size of input data, it calculates the number of lambdas to spawn proportional to the size of the job.

A policy has unilateral control over the scaling. It can make use of any source of metrics to base its decisions on. If more advanced types of metrics are collected or calculated (statistical models, for example), the more advanced the policy can become. In fact, the only limit placed on policy implementations are the facilities it has access to with respect to managing the lambdas (AWS API only allows spawning, with some effort, termination can be implemented) and the interval at which the policy is evaluated at. The latter is a function of the daemon's control loop interval, but this too can be made controllable by the policy implementation through some minor software-engineering work on the daemon.

It is important to note that even a clairvoyant policy is limited by the limitations in scalability of the components in the rest of the system.

## Additional System Features

**Multi-tenancy:** though the share-nothing architecture, the system has inherent fairness for concurrent system users. If two users submit the same job, both get the results at the same time.

[2] AWS SQS is a distributed, simple queue service. It can only provide approximate numbers on the amount of messages in a queue due to the fact that the queue can be/is distributed over multiple machines for performance, reliability and scaling reasons

Concurrent, unique jobs do not influence each other up to a certain limit[3]. Because only the actual computation time of lambda is billed, it does not matter[4] if two jobs run concurrently or sequentially.

**Durability:** the results are persisted in the AWS S3 service. From our experience this a very durable and, for our purposes, reasonably fast storage service.

**Scheduling:** the system has support for simple or complex, static and dynamic scheduling/scaling policies. However, we did not implement any complex/advanced policies.

[3] Account limits on maximum allowable concurrent lambda invocations. Can be raised by contacting AWS support.

[4] Cost-wise

## EXPERIMENTAL RESULTS

We implemented our system design on Amazon Web Services (AWS), using four service groups: AWS Lambda, AWS SQS Queues, AWS S3 Storage and AWS Gateway API. Our Lambdas run Node JS 8.10 and use the AWS JavaScript SDK for communicating with the other AWS components in our design (Lambdas, Queues and Storage). Only the lambda function code need to be deployed. This deployment is automated through Bash scripts that use the AWS Command Line Interface tool to update/deploy the function code. Our TypeScript code-base is open source and can be found at https://github.com/TimSpeelman/in4392-lab.

Our experiments are orchestrated through a custom-built Web Interface. This allows us to send job requests with varying parameters to our system at specific times, enabling us to simulate workloads and subsequently, gather all data needed for analysis. We gather metrics and logs from AWS CloudWatch, and process these data using custom built JavaScript data processing script. The visualization of the resulting data is done using MATLAB.

## Experiment 1, Elasticity

The system scales in two dimensions: in terms of job size (or number of input messages), and in terms of concurrent requests (multiple jobs running at the same time). We will evaluate these in two experiments. The workload consists of different user requests which trigger a stream of n messages sent as quickly as allowed by the AWS infrastructure into our system. The system computes the wordcount of these messages and sums these into a total, which is returned at the end of the job. We measure the metrics Job Completion Time and Billed Time for the total time of Lambda usage billed by AWS. The system parameters for both experiments are described in Table \ref{tbl:sysparams}.

### Experiment 1.1 Elasticity towards Job Size

For larger jobs, our system can spawn multiple concurrent workers (Lambdas) to process the stream. This scheduling depends on the policy used. For experimental purposes we have implemented two policies: AlwaysOne always schedules one Lambda per stage, regardless of the workload. StaticProportional(m) schedules ceil(n/m) lambdas for each stage except Feed, for n input messages.

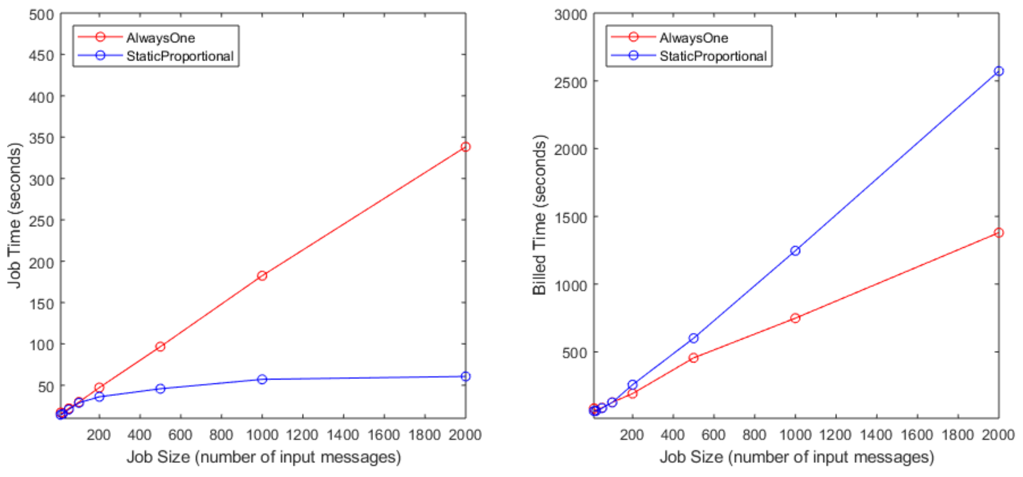

**Figure 2: Job Processing Time over Job Size for different Scaling Policies**

The results show that the job computation time scales linearly for the AlwaysOne policy, which is to be expected. The StaticProportional(150) policy shows significant improvement in terms of Response Time, but the billed time per message is higher due to the diminishing returns of scaling up the components used in our application that have trivial computations and overhead of receiving and sending messages. The used queue service is not optimized for throughput.

### Experiment 1.2 Elasticity towards Concurrency

Because of our system's design, each user request spawns an independent set of queues and lambdas to process the jobs. Hence, we expect that multiple concurrent requests do not influence job completion time at all.

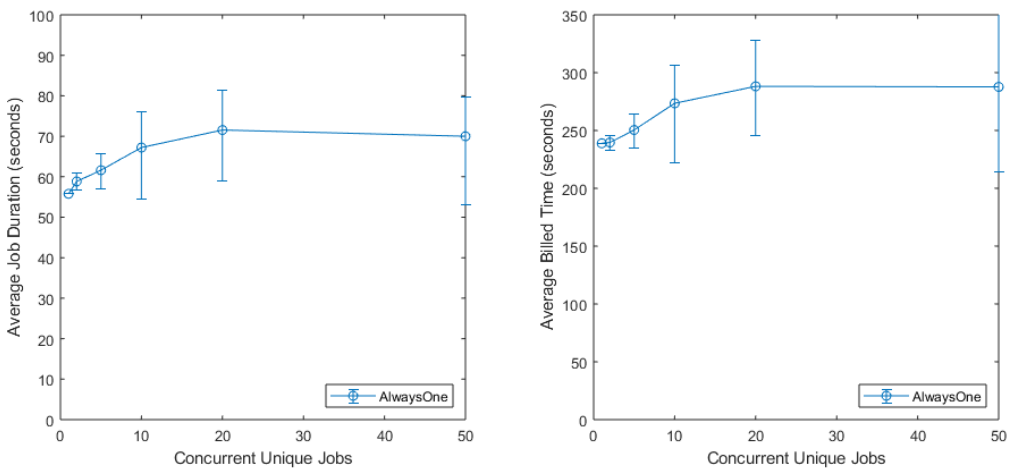

**Figure 3: Job Processing Time over Number of Concurrent Requests**

Contrary to our expectations, the results show a slight increase in the Job Completion Time as the number of concurrent requests grows. It might be the case that we're hitting some bottleneck or being throttled on service level operations such as SQS queue creation or network IO, as evidenced by the large variance in the results. Other than automatic queueing of requests to the API lambda done by AWS Lambda gateway, our system does not share any components. We have not investigated this further

## Experiment 2, Caching

Our system design incorporates caching to re-use earlier requested jobs or already running jobs. An incoming job request which has been computed before will receive the stored result. To measure that this is actually working, we execute the same request multiple times and measure the response times and billed time.

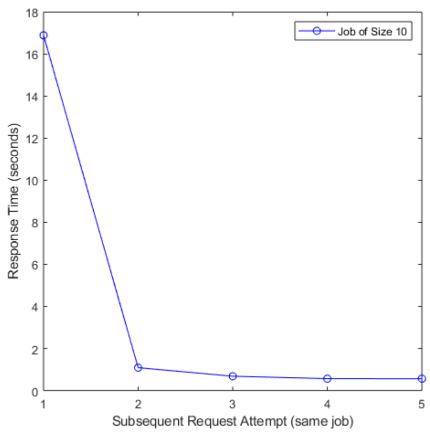

**Figure 4: Response Time for consecutive requests of the same job**

The results clearly show that a job is processed only once. Subsequent requests receive persisted (or cached) result.

## Experiment 3, Automation

Jobs can run on our system without human intervention. Using the Web Interface a user can request a Job and the application will return to the user once the job is done. Job orchestration, auto-scaling and monitoring is handled by the daemon lambda and the scheduling policy configured. To show this orchestration, we perform a single job request and plot the number of queues and workers (instances of processing component lambdas) over time.

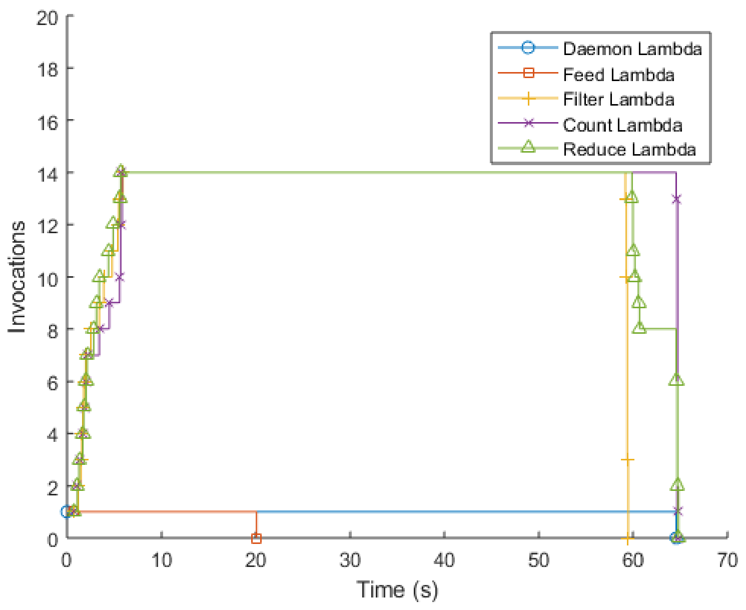

**Figure 5: Number of Lambda instances with a single job with size 2000**

This figure shows the Lambda invocations after a request of size 2000 has entered the system. The strategy being used is StaticProportional(150). The request is triggered at t=0: it shows the Daemon Lambda being invoked by the API call. The Daemon creates the Feed Lambda once and all others 14 times according to the policy.

## Experiment 4, Monitoring

When our system is used for unpredictable workloads and containing processing stages of varying complexity, the Daemon should have the means to identify the system's bottlenecks dynamically. A reasonable indicator of performance and bottlenecks are metrics on the queues. If a queue contains disproportionally large amount of messages, a scheduling policy could decide to spawn more processing lambdas for the step that reads from that queue. The metrics available in AWS CloudWatch for the SQS service are detailed but are available only at 5-minute (free) or 1-minute (paid) intervals. This is not granular enough for our application. Hence, we have implemented a polling mechanism to collect the approximate queue size information directly from the SQS service. Using this mechanism, queue metrics can be exposed to the scheduling policy implementations.

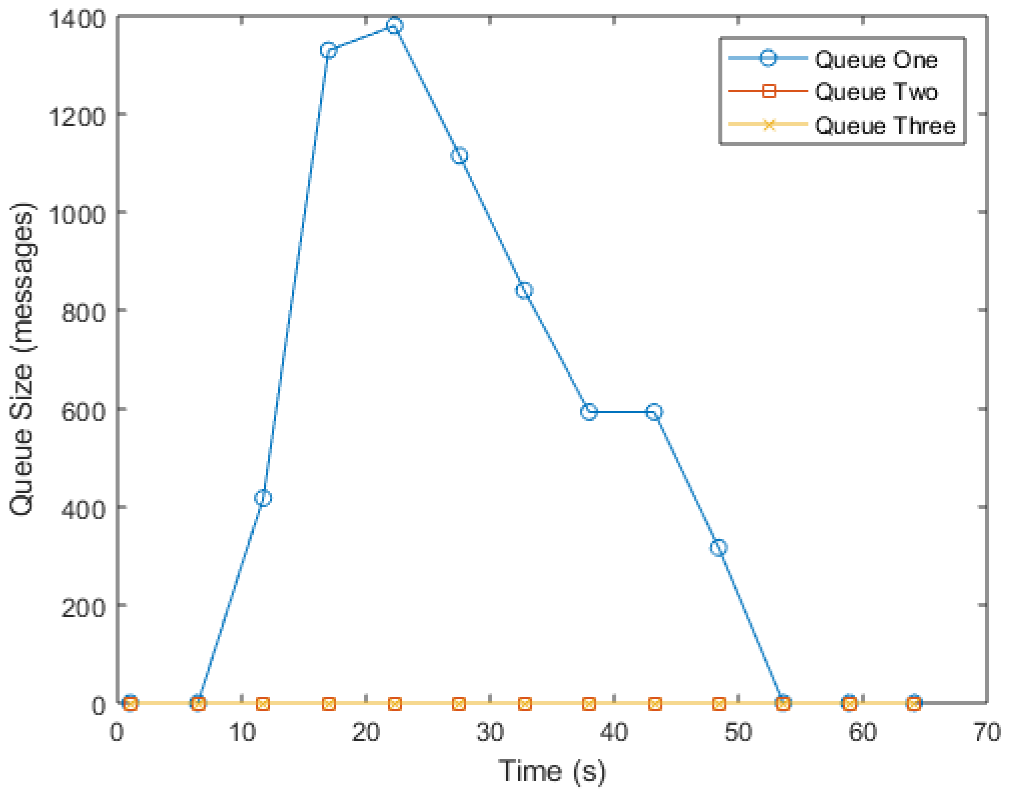

**Figure 6: Queue sizes of a single job with size 2000**

The above picture shows the metrics retrieved by the Daemon at 5-second interval. It clearly shows the size of the first queue to grow quickly as the Feed Lambda injects the messages. The second and third queue are nearly always empty. This shows that the processing in the Reduce and Count Lambdas is at least as quick as done by the Filter Lambda. Based on this information, a more dynamic policy could be created that increases the amount of Filter Lambdas and reduces the Count and Reduce Lambdas.

## CONSLUSION

We have succeeded in implementing a serverless system/framework for creating data processing applications. While the system is functional, and satisfies the requirements, we are a bit underwhelmed with the final performance of the system. It might be that the implementation of the steps of our simple application on top of the system is too light and the main source of overhead coming from the SQS service. Further work would encompass using a more performant queueing mechanism. Another point is the lack of proper, fine-grained metrics in Amazon AWS (Cui 2017). Obtaining better metrics requires extra effort from the developers.

We also acknowledge that our architecture might not have been the best fit for Amazon AWS Lambda service. As suggested by various Amazon AWS developer guides, tt could have been more efficient to implement the system with the lambas being triggered by the services (queues) themselves, thus resulting in a serverless, fully event-based architecture. At the expense of loss of all, but basic, control over the concurrency of the functions. However, for the purpose of this assignment that was not desired.

Overall, we feel we have accomplished a significant software engineering effort and sufficiently explored the strengths and weaknesses of the emerging alternative to *IaaS*, the *FaaS*.

## APPENDIX A: TIMESHEET

This Table documents all hours the team spent on this assignment.

| Activity | Hours | Explanation |
|---|---|---|
| total-time | 199 | |
| think-time | 15 | |
| dev-time | 106 | |
| xp-time | 32 | |
| analysis-time | 15 | |
| write-time | 20 | |
| wasted-time | 11 | |
| (exp 1) total-time | 24 | |
| (exp 1) dev-time | 20 | Most of these hours are spent building a control panel and multiple scripts to fetch and process data. |
| (exp 1) setup-time | 4 | |
| (exp 2) total-time | 5 | |
| (exp 2) dev-time | | |
| (exp 2) setup-time | | |
| (exp 3) total-time | 5 | |
| (exp 3) dev-time | | |
| (exp 3) setup-time | | |
| (exp 4) total-time | 5 | |
| (exp 4) dev-time | | |
| (exp 4) setup-time | | |

| | | |
|---|---|---|
| (exp 5) total-time | 5 | |
| (exp 5) dev-time | | |
| (exp 5) setup-time | | |