AMESoS: A Scalable and Elastic Framework for Latency Sensitive Streaming Pipelines

Michail Tsenos
tsemike@aueb.gr
Athens University of Economics and Business
Athens, Greece

Aristotelis Peri
ariperi@aueb.gr
Athens University of Economics and Business
Athens, Greece

Vana Kalogeraki
vana@aueb.gr
Athens University of Economics and Business
Athens, Greece

ABSTRACT

Serverless computing, and in particular Function as a Service (FaaS), has become an increasingly popular cloud programming model in recent years. The serverless computing model offers an intuitive, event-based interface for developing cloud-based applications, that makes the writing and deployment of scalable microservices easier and cost-effective. Existing orchestrators in serverless systems are mainly designed for short-lived functions. Nevertheless, an increasing number of applications are deployed as pipelines that comprise a sequence of functions that execute in a specific order or pattern and must meet a wide range of throughput and latency targets to be practical. In this paper, we present AMESoS, our scalable and elastic framework for latency-sensitive streaming pipelines. AMESoS (i) enables developers to build predictable pipelines that meet their latency demands, (ii) employs prediction to *proactively* estimate the most appropriate number of active replicas needed for each function in the pipeline, and (iii) dynamically scales the number of replicas for each pipeline's functions in the presence of overloads. Our experimental results demonstrate the efficiency and benefits of our approach over state of the art systems.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **Real-time system architecture**.

KEYWORDS

Serverless Computing, Elasticity, Latency Sensitive, Streaming, Pipelines

ACM Reference Format:
Michail Tsenos, Aristotelis Peri, and Vana Kalogeraki. 2022. AMESoS: A Scalable and Elastic Framework for Latency Sensitive Streaming Pipelines. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27–30, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524860.3539642

DEBS '22, June 27–30, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539642

1 INTRODUCTION

Serverless computing, and in particular Function as a Service (FaaS), has become an increasingly popular cloud programming model in recent years, fueled by the recent demand to host services on provisioned cluster infrastructures and the paradigm shift towards interconnected IoT applications, devices and platforms. The serverless computing model offers an intuitive, event-based interface for developing cloud-based applications, that abstracts all infrastructure handling and makes the writing and deployment of scalable microservices easier and cost-effective. All major commercial cloud service providers are now offering serverless computing platforms, including AWS Lambda[1], Google Cloud Functions[16] and Azure Functions[28], where they take care of code maintenance and execution so that developers can deploy new code faster and easier.

The recent advantages of edge computing technologies have enabled a wide range of latency sensitive applications attributed to the reduced latency and bandwidth savings that edge computing provides. As a result, a number of applications (i.e., alerting in smart buildings, emergency response, video analytics) that require real-time data analysis and decision making in real-time, have emerged. However, a typical edge cluster has considerably fewer resources compared to cloud servers and may be capable of handling a small amount of requests, yet it becomes inadequate when encountering bursty requests. Addressing these issues is not a straightforward process. For instance, approaches such as [31] and [35] are limited as they either propose application-specific solutions or do not take the real-time streaming characteristics of the applications into consideration.

We propose AMESoS a scalable and elastic framework for latency-sensitive streaming pipelines. AMESoS dynamically computes the queueing delays at the serverless functions' queues to determine whether the pipeline will meet its latency target. By computing the queueing delays at the individual queues, AMESoS can dynamically explore the auto-scaling ability for each function and scale up (or down) to adapt to bursty workload. AMESoS dynamically determines an efficient pipeline configuration to meet the requested timeliness target, without wasting excessive resources. To do that, AMESoS (a) uses a profiling mechanism, (b) estimates the impact of the queueing delays on the latency of the end-to-end pipeline, and (c) utilizes a time-series forecasting mechanism that can estimate the incoming requests in the near future and scale accordingly in advance.

Our approach makes the following contributions:

(i) We propose a methodology and have developed system components to support the execution of latency-sensitive pipelines in serverless environments, at the edge.

(ii) We use a prediction model to estimate the incoming invocation rates for each pipeline. Using this model we can proactively determine the most appropriate number of active replicas needed for each function in the pipeline to maximize the probability that the latency constraint of the pipeline is met.
(iii) We present a scalable and elastic framework that dynamically scales the number of replicas for each pipeline functions. In the presence of overloads, our algorithm reacts dynamically to changing demands to ensure that adequate replicas are allocated to the pipelines to meet their latency constraints. When the queuing delay leads to a violation of the deadline constraint, our system will identify which function in the pipeline is the bottleneck and dynamically increase the number of the function replicas and allocate them to nodes in the cluster.
(iv) We provide detailed experimental results to illustrate the benefit of AMESoS and that it achieves as much as 70% improvement in deadline misses compared to its competitors.

2 BACKGROUND

2.1 Serverless Computing

In recent years the trend of Serverless Computing or Function as a Service (FaaS) has emerged as a popular computing model in the cloud computing industry. Unlike traditional systems, serverless computing offers an intuitive, event-based interface for developing cloud-based applications, in which, developers can deploy and deliver services, where they are only responsible for the application logic. While in traditional host solutions, it is typical to rent some dedicated virtual or physical machines for a specified desired amount of time, the Serverless model offers the ability to utilize ephemeral containers as needed based on the workload. The Serverless computing model has additional advantages including its lower operational and deployment costs due to its unique pricing policy (based on a pay-as-you-use model) where users do not explicitly provision or configure virtual machines (VMs) or containers but they only get charged based only on the number of resources consumed by an application during its execution.

In addition to its appealing pricing model that leads to significant cost savings, there are three more characteristics that make the Serverless model beneficial for our setting. First, the Serverless model allows for easier management compared to server-based approaches. There is no need to have dedicated system administrators or custom-built automated tools to manage the existing infrastructure. Second, the Serverless computing model is superior in terms of elasticity and scalability, by providing a seamless method for autoscaling without the need to know the internals of the underlying technologies, using ephemeral containers with a life span equal to the time required by the workload. The number of active instances can be selected either by the user or can be adapted dynamically according to the request rate. For example, during periods of high load, the number of active instances can be adapted automatically to compensate for the increased traffic. On the other side, during extended periods of inactivity, the number of active instances can decrease to zero (scale to zero) to keep the total execution cost low. As a result, Serverless systems have smaller administration overhead compared to typical infrastructures. Finally, the developers can choose from a wide variety of available programming languages to develop their services. The lack of strict programming rules and grammar allows us to deploy our generated intermediate representation functions easily without having to modify the general architecture of the system.

Serverless systems can be deployed practically everywhere, from cloud to edge clusters and from high-end computers to low-end devices such as Raspberry Pi. Typically, functions are deployed within Containers that run in a distributed fashion over Kubernetes [23] or some other Container orchestrator such as Docker Swarm[12], Nomad[19] or Apache Mesos[4]. There are open-source serverless projects such as OpenFaas[29], Apache OpenWhisk[5], KNative[22] and Fission[13], and commercial ones such as Google Cloud Functions[16] and Cloud Run, Amazon's AWS Lambda[1], Microsoft's Azure Functions[28].

2.2 Pipelines

The ability to build and deploy *Serverless Data Pipelines* (or *pipelines*) that comprise a sequence of functions that execute in a specific order or pattern, has recently emerged. Consider, for example, an AMBER Alert application that uses traffic cameras across a city to search for specific individuals or moving vehicles [31]. Such pipelines must often meet throughput, latency, deadline or cost targets to be useful in practical settings. For example, in the AMBER Alert application, the pipeline must be configured to meet strict deadline constraints, while a pipeline that detects vehicles that are banned from entering specific streets in a SmartCity setting could be configured to be more cost-efficient. Although serverless environments provide horizontal scalability for individual functions, orchestrating the system to support such pipelines is a complex process.

In the serverless data pipeline the data is being processed through a directed acyclic graph (DAG) where each node represents an operation (*i.e.,* a serverless function) that will be applied to the data and each edge shows the corresponding data flow. Nodes in the pipeline are called Pipeline Nodes (or *PNodes* for short). Pnodes are instantiated in VMs or containers and are possibly replicated to meet workload demands.

The nodes represent a range of operations from simple filtering to complex processing like ML-based classification algorithms. The in-degree of each node denotes the number of edges that end up in the node, while the out-degree denotes the number of edges that originate from the node. A node with an in-degree greater than one represents a *Merge Node*. During splits of the data flow, each message is sent to all the out-degree edges of the node, while in a Merge Node a function executes only when it receives all messages from all its in-edges. A node with an out-degree zero is called a sink and represents the end of the processing flow in that pipeline. Typically a sink node either stores the results in some external storage or these are returned to the user.

2.3 Problem Statement

Consider an edge cluster comprising K nodes that host several serverless functions. A typical edge cluster has fewer resources compared to cloud servers, but is capable of running reasonably complex

operations (e.g., linear regressions, object detection). Pipelines are triggered by external events (*i.e.,* user requests or messages generated by streaming platforms). Each pipeline is represented by a DAG, where the pipeline nodes *PNodes* denote the serverless functions (possibly replicated) and the edges represent the corresponding data flows among the functions.

The execution time of a pipeline depends on (1) the PNodes invoked, (2) the dependencies among them, as well as (3) the other pipelines concurrently executing in the system. Each pipeline is characterized by its Deadline. The pipeline's latency is defined as the elapsed time from the issuing of the pipeline invocation to the time that the sink operator externalizes the results.

Our goal is to exhibit low latency for both short-running and long-running workflows. Under normal conditions, we expect that the pipeline invocations will meet their deadlines, but under unpredictable conditions of the streaming system data sources, this is not always guaranteed as the system might experience sudden bursts in the number of requests. In the case of unpredictable bursts, the system may not be able to process them by their deadline. On the other hand, when we experience a decrease in the volume of messages although no deadlines will be missed we may end up under-utilizing and wasting resources while paying extra costs.

Our goal is to design a system that dynamically adjusts (scales up or down) the number of function replicas according to the current request rate factoring in queuing delays. During overloads the system should scale up the functions of each *PNode* accordingly, to ensure a high percentile of requests meeting their deadline while in under-load situations our system scales down the replicas in order not to waste resources.

3 AMESOS ARCHITECTURE AND SYSTEM MODEL

3.1 The AMESoS Architecture

AMESoS (shown in Figure 1) comprises three main components, the Connector, the Pipeline Manager and the Scaler.

Connector: The Connector component is the linkage between the data sources and the Pipeline Manager. We use one Connector per Pipeline Manager. Its purpose is to subscribe to an event stream provided by a third-party system (such as Apache Kafka or S3/Minio) and trigger the flow invocation for each received event. The event payload is used as the initial message of the flow invocation. For example, the event payload can be an ID of an object residing in an S3 bucket. Moreover, the developer can override the default Kafka connector and implement a custom connector that subscribes to any event stream or webhook.

Pipeline Manager: The Pipeline Manager is the main component of our system. It is in charge of managing the execution of the pipeline and maintaining the corresponding metadata. It executes the pipeline by continuously receiving the flow invocations from the Connector and executing the set of invocations for each PNode of the pipeline. For each flow invocation, it instantiates a JSON file called DataFlow that maps the returned result from each function with the corresponding function. The initial invocation payload that is received from the Connector is mapped to a special reserved *key* called *input*. The Pipeline Manager instantiates a set of PNodes as described in the pipeline configuration file. In

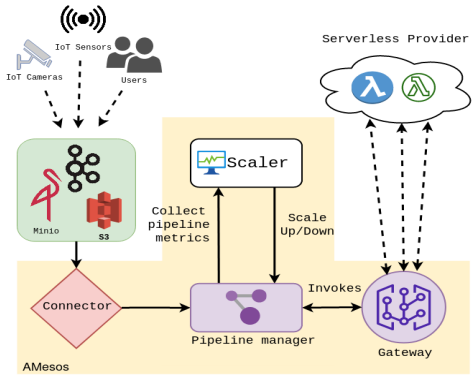


Figure 1: The AMESoS architecture.

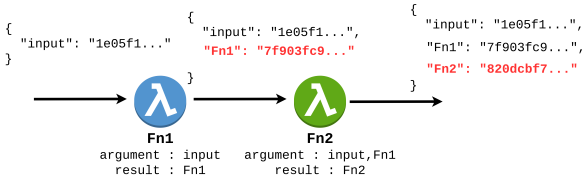


Figure 2: Pipeline flow and Dataflow.

the pipeline configuration file, the user describes the arguments that each function expects and the Pipeline Manager matches these arguments with the corresponding keys from the DataFlow file (Fig. 2). Furthermore, the Pipeline Manager is responsible for computing and storing performance and resource statistics that are used for monitoring the pipeline flow. We present those in section 3.2.

Although it is possible for the Pipeline Manager to support the execution of many pipelines in parallel we recommend different Pipeline Managers to manage different types of pipelines.

Gateway: The Gateway is responsible for interacting with the Serverless environment. This is done through the API that is provided by the Serverless platform. The Gateway uses the provided API to deploy new functions, receive information about the deployed ones, stop the execution of idle functions or even modify their resource parameter CPU/Memory allocation. In addition to the above, the Gateway also acts as a proxy to the functions; if multiple function replicas are deployed it uses a Round-Robin scheduling policy to load balance the function invocations among all actively deployed functions replicas.

Scaler: The Scaler is in charge of dynamically deciding and triggering in real-time the number of function replicas for each PNode. This is done by exploiting the resource and performance statistics provided by the Pipeline Manager. In section 4 we describe in detail our scaling algorithm that calculates in real-time the appropriate number of instances that need to be deployed for each function of the Pipeline.

3.2 System Model

AMESoS supports the execution of pipelines in a serverless environment where computations are decomposed into small steps that fit

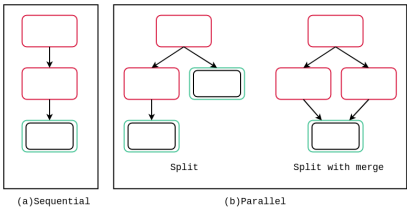


Figure 3: Processing Graph Types

within serverless functions; these are arranged in sequence, branch or parallel and can re-execute whenever triggered. The pipelines are triggered via messages, in response to specific events (*i.e.,* events being generated from a Kafka[3] event source).

Pipeline: A *Pipeline* denotes a sequence of serverless function invocations, typically triggered by a Message. The Pipeline is represented as a Directed Acyclic Graph (DAG), (shown in Figure 3), where nodes correspond to serverless functions, named *Pipeline Nodes* (or *PNodes* for short), and the edges in the DAG denote the corresponding invocations. Each DAG has its own unique ID.

Each PNode can serve one function only and has its unique ID. Pnodes can be instantiated in VMs or containers and can be replicated to deal with workload demands. The degree of replication depends on configuration settings as well as latency and throughput criteria. Each PNode maintains a pool of HTTP Connections to the serverless function replicas. The connections are established through the Gateway provided by the Serverless Provider, as the function containers might not be accessible from outside the provider network. The number of HTTP Connections is equal to the number of function replicas. Furthermore, each PNode i is characterized by $progress_i$ which indicates the processing progress in the Pipeline. This value is used to estimate the remaining processing time. It can be provided by the user during the initialization of the system, but it can also be calculated periodically during the run-time by the Pipeline Manager after a small monitoring period.

Our system supports high-volume workloads like data analytics pipelines with parallel tasks. We consider two types of DAG pipelines, as shown in Figure 3, sequential and parallel. In a parallel pipeline, the processing logic can branch into two or more separate sequential flows, which can execute in parallel and upon their completion, the results either merge into a single flow or stay split. In the sequential pipeline, each vertex has at most one input and one output. The pipeline configuration is provided by the user in JSON format. The execution of the pipeline is managed by the Pipeline Manager.

Pipelines are triggered via events received through the Connector component. AMESoS Pipeline Manager executes the pipeline by subsequently triggering the execution of each PNode in the pipeline. AMESoS handles the dependencies between PNodes. The Pipeline Manager considers which PNodes need to execute next and passes the data as well as the corresponding attributes to the corresponding PNodes.

Pipeline Metrics: Each pipeline p is associated with the following parameters:

- $pipeline_{id_p}$: The id of the pipeline.
- $deadline_p$: The time interval, starting at the receipt of the invocation request, within which the entire pipeline should complete.
- $totalEstimate_p$: The estimated execution time of the pipeline. The initial $totalEstimate_p$ can be given by the user or computed based on previous executions and updated periodically by the Pipeline Manager.

PNode Metrics: Each PNode i is characterized by the following parameters:

- $pnode_{id_i}$: The id of the pnode.
- λ_i: The invocation request rate at PNode i.
- $laxity_{i,90}$: The 90^{th} percentile laxity value of the messages that are invoking PNode i. It is measured periodically within a time window.
- $avgProcessTime_i$: The average process time of PNode i for each function invocation. This is updated periodically.
- c_i: This metric denotes the number of active function replicas represented by the PNode. By default our system maintains at least one replica of each function. The number of replicas is determined dynamically by the Scaler.
- μ_i: The rate with which requests are processed at PNode i.
- $queueSize_i$: The queue size of PNode i.

Message Metrics: A Message encapsulates the event payload for each function invocation. Each Message m is associated with the following invocation parameters:

- $timeToDeadline_m$: The remaining amount of time that the pipeline must complete, before its deadline is violated. This is initially set as deadline and it is updated during execution. As the pipeline executes, if the $timeToDeadline_p$ becomes negative this means that the invocation has missed its deadline and we considered that it expired.
- $queuingDelay_{m,i}$: The time that each message remained in PNode's i message queue.
- $processTime_{m,i}$: The time it takes for PNode's i function to process message m.
- $execTime_{m,i}$: The time taken by the message to execute at pnode i, including queueing, calculated as:

$$totalExecTime_{m,i} = queuingDelay_{m,i} + processTime_{m,i} \quad (1)$$

- $estimate_m$: The estimated remaining execution time of each message. This value is updated after each PNode execution. As we describe in Section 3.1 each PNode is associated with a $progress_i$ value that indicates the percentage of the processing in the pipeline so far. The initial $estimate_m$ is equal to $totalEstimate_p$ and after each PNode excecution is updated as follows :

$$estimate_m = (1.0 - progress_i) \cdot estimate_m \quad (2)$$

- $laxity_m$: The difference between the $timeToDeadline_m$ and the $estimate_m$ after each PNode execution. Negative values indicate that the message will miss its deadline. We keep track and update the laxity value of the pipeline after each function execution.

4 OUR APPROACH

In this section, we describe our proposed solution for satisfying the constraints of latency-sensitive pipelines. Our objective is to provide a system that meets the following requirements: (i) the urgency demands of each pipeline, expressed in terms of a latency value, are met and (ii) we use a lightweight event-driven scaling algorithm to adapt dynamically to meet the workload demands. To meet the above requirements our approach makes the following contributions:

- We propose a methodology and have developed system components to support the execution of latency sensitive pipelines in serverless edge clusters.
- We develop a dynamic algorithm to schedule the execution of the pipeline invocations in the system. The schedule is driven by the urgencies of the pipelines (represented via the laxity value) factoring in queueing delays experienced by the pipeline during execution.
- We use a prediction model to predict the incoming invocation rates for each pipeline. Using this model we can proactively determine the most appropriate number of active replicas needed for each PNode to maximize the probability that the latency constraint of the pipeline is satisfied.
- We present a scalable and elastic algorithm that dynamically scales the number of replicas for each PNode. In the presence of overloads, our algorithm reacts dynamically to changing demand to ensure that adequate replicas are allocated to the pipelines to meet their latency constraints. When the queuing delays lead to a violation in the deadline constraint, our system will identify which function in the pipeline is the bottleneck and dynamically increase the number of the function replicas and allocate them to nodes in the cluster.

4.1 Queuing Theoretic Model

Let λ_i denote the rate of arrival requests for PNode i. Each PNode i has c_i number of function replicas and each replica is deployed and runs in its own container. We assume that all containers are homogeneous and have the same resource allocations. Under the assumption that the containers are homogeneous, the function replicas can all service requests at the same rate. Thus, we assume that all replicas of PNode i have the same $avgProcessTime_i$. Consequently, all function replicas will have the same service rate μ_i. Thus, we can compute the average service rate of each PNode μ_i as:

$$\mu_i = \frac{c_i}{avgExecTime_i} \tag{3}$$

In order to estimate the number of containers in real-time, we assume that time is split into the same duration periods that we call *epochs*. In each epoch we recalculate the λ_i and $avgProcessTime_i$ of all the invoked PNodes.

Assuming a Poisson distribution with λ_i the rate for the incoming requests and that services times are exponential, we can model this system using an M/M/c queuing system with c_i function replicas for each PNode i to process incoming requests in parallel. In a streaming environment, similar to [35], we can assume that for each epoch the arrival rate of the requests can be approximated via a Poisson distribution, although this may not hold for the entire system duration.

Queuing analysis of an M/M/c system is well known in the literature where the most important metric that must be satisfied is the stability of the utilization factor. With utilization factor ρ_i we denote the percentage of time that the system is occupied with jobs and for each PNode we calculated ρ_i as:

$$\rho_i = \frac{\lambda_i}{\mu_i} \tag{4}$$

where λ_i denotes the arrival rate at the PNode and μ_i is the average service rate of each PNode. In all queuing systems, the higher the average utilization factor, the longer the wait time for each job in the queue. Moreover, when the average utilization factor is higher than 1 the queue size will tend to infinity and as a result, the average waiting time will tend to infinity. As a rule o thumb, each ρ_i should not exceed 80%. We should note that these performance metrics depend on multiple aspects such as (i) the scheduling decisions, (ii) the arrival pattern of the incoming data, as well as (iii) the distribution of the input values.

4.2 Estimating the laxity value

Given a user-defined deadline constraint specified for a message, the Pipeline Manager dynamically computes whether the message will be able to meet its deadline target. While the majority of the existing systems assume linear pipelines with predictable invocations and latencies, we examine the applicability of a commonly used scheduling metric, laxity, that depicts how close the estimated execution time is to the user-defined deadline

Essentially, the laxity of a message is computed as the difference between the deadline and the estimated remaining execution time of the message in the pipeline. More formally, $laxity_m = timeToDeadline_m - estimate_m$ where $estimate_m$ is the estimated remaining execution time for the pipeline. The smaller the laxity value is, the higher is the probability of a deadline miss. A negative laxity value indicates that the message will not be able to execute the pipeline on time.

We assume that the execution of the pipelines is repetitive but aperiodic. Thus, the execution time of a pipeline $totalEstimate_p$ can be estimated either via previous executions of the pipeline or by using a small number of invocation runs during a profiling period at the initialization of the pipeline. Note, that, the execution time of a pipeline depends on multiple factors, including (i) the scheduling policy, (ii) the arrival pattern of the incoming messages, as well as (iii) the scaling decisions. For example under overload conditions at the edge cluster or poor container scheduling due to orchestration policies of the serverless environment, the average process time of the pipeline will increase as shown in the experimental evaluation. Thus, once an estimate is computed, we constantly monitor and adjust this estimate during run-time. Our approach is to dynamically compute the laxity value for the 90^{th} percentile of messages in each PNode inside an *epoch*; this will be subsequently used to trigger our scaling algorithm and select the appropriate number of replicas for each PNode. Our ultimate goal is to meet the pipelines' deadline targets.

4.3 Holt's prediction for incoming rates

If we select to scale according to the current λ_i we may underestimate the number of containers that we need to instantiate and then

we will have to wait for the next *epoch* to scale up again. As a result, especially if the rate is high, we may end up accumulating a large number of messages in the queues. Also, due to the known cold start problem in serverless environments [9], the actual containers will not start immediately after a scale up decision. Thus, it is crucial to reduce the number of scale ups to avoid excessive time overhead due to communication with the underlying container orchestrator and the containers start-up period.

One approach is to predict the λ_i for the next *epoch* and to scale up or down according to the demand. For this purpose, we use Holt's Model [10], a trend forecasting technique that applies a double exponential smoothing process that captures the changing trends in data and predicts a data value for a future period. Holt's Model works better compared to other techniques [24] such as weighted KNN where a regression model is built by exploiting the similarity of the features, or random forests where the predictions are performed by taking the mean of outputs from each decision tree, as these techniques cannot accurately capture the trend of changes in the demand when there are frequent oscillations.

In our approach, we use the Holt's model to estimate the incoming demand based on a time-series of the incoming rates of the previous *N epochs*. Holt's model is considered as an extension of the moving average technique. It attempts to remove the inherent lag associated to Moving Averages by placing more weight on recent values. The first step to find the predicted incoming rate is the calculation of the smoothed value of $S_{\lambda_i^t}$. The computation process takes into consideration the current λ_i^t, the previous smoothed value $S_{\lambda_i^{t-1}}$, and the previous trend tr_i^{t-1} (shown in Equation 5).

Equation (6) suggests that the estimation of the trend of λ_i^t (whether it will increase or decrease and at which degree) is based on previous smoothed values of λ_i^{t-1} and helps better capture its behaviour. The values of α (7) and γ (8) are tunable parameters and represent the data smoothing factor and the trend smoothing factor respectively. These are chosen by sampling and checking the mean square error for the different values to identify the best ones.

From this procedure, we estimate the future traffic and we use this as input to the scaling algorithm to determine the appropriate number of instances to adapt to future traffic.

$$S_{\lambda_i^t} = a \cdot \lambda_i^t + (1-a) \cdot (S_{\lambda_i^{t-1}} + tr_i^{t-1}) \tag{5}$$

$$tr_i^t = \gamma \cdot (S_{\lambda_i^t} - S_{\lambda_i^{t-1}}) + (1-\gamma) \cdot tr_i^{t-1} \tag{6}$$

$$\alpha (0 \leq \alpha \leq 1) \tag{7}$$

$$\gamma (0 \leq \gamma \leq 1) \tag{8}$$

4.4 Recomputing the Execution Times

In our approach, we dynamically compute each PNode's 90$^{\text{th}}$ percentile laxity value which we subsequently use to determine whether a PNode is a bottleneck for a pipeline.

Serveless functions can be hosted on edge clusters that are limited in that regard and are more sustainable to throttling when their resource capacity is reached. As mentioned by Lopez and Spillner in [25] the increase in performance during a scale up is proportional to the number of replicas and not linear. During our run-time experiments, we confirmed this assumption by observing a significant degradation of our deployed functions service rate as our cluster reached its full resource capacity. In the experimental evaluations, we observed that the average execution time of one replica increases up to 30% as the machine that hosts the function container reaches its full capacity. For these reasons to mitigate that problem, we need to recalculate the average execution time as the pipelines run and scale according to the updated demand.

4.5 AMESoS Scaling Algorithm

AMESoS scaling objective is two-fold: (i) determine the number of replicas for each PNode function in an effort to minimize the number of missed deadlines as the input rate varies dynamically, (ii) keep the number of replicas low by not over-utilizing resources when they are not needed. This procedure is performed by exploiting the metrics collected by the Pipeline Manager. Algorithm 1 describes our procedure which consists of 3 steps: (i) *Rate prediction with the Holt's model*, (ii) *Replica calculation* and (iii) *Scaling Procedure*.

In the first step, we use Holt's Double Exponential Smoothing Timeseries prediction method to predict the future λ_i value for the next *epoch* as described in section 4.3.

In the second step, we estimate the number of replicas for each PNode. By using queuing theory we calculate the total number of containers c_i' that we need to adapt to the predicted traffic in the next *epoch* by solving Eq. (4). μ_i is the service rate of each PNode that we want to satisfy. By solving (4) we can find the c_i' that can satisfy our predicted rate. ρ_i is a tunable parameter that can affect the number of predicted replicas. As ρ_i tends to 1 the algorithm will predict a lower number of replicas. In our experiments we choose ρ_i's value to be 0.7 but this can be tuned based on the application settings.

In the final step we decide and perform the actual scaling. We obtain the c_i' from the *Replica calculation* step and check the 90$^{\text{th}}$ of the messages with the lowest laxity in the current epoch. If that laxity is below a user-specified *threshold* and the number of the predicted replicas is higher than the current ones, then the algorithm invokes a scale up request to the Gateway. Alternatively, if the laxity is higher than a threshold and the predicted number of replicas is lower than the current number, then a scale down request is invoked. In the special cases when $laxity_m < threshold$ and $c_i' < c_i$ or $laxity_m > threshold$ and $c_i' > c_i$ then we do not perform any action and wait until the next *epoch*. If the Algorithm results in a scale request, once the scale completes, the Scaler informs the Pipeline Manager for the change to update the size of the PNodes connection pool accordingly.

In our experimental evaluation we demonstrate that our scaling algorithm works well with a very large number of serverless functions, making the use of the queueing model practical in edge clusters.

5 AMESOS IMPLEMENTATION

We implemented all AMESoS components in Java 11 with a total of around 5000 lines of code. All components can run as standalone applications or can be packaged and delivered as Docker Images.

Container Orchestration We use Mesosphere Marathon [27] just to start function containers on Apache Mesos cluster. The function containers are created using the OpenFaas Watchdog and Of-Watchdog templates. The Watchdog contains an embedded HTTP

Algorithm 1: AMESoS algorithm for finding function replicas needed for serverless Graphs

Input: $PNode's$ current replicas c_i , average execution time $avgProcessTime_i$ and incoming rate λ

Scaling Procedure `Scaler()`

- $\lambda_{pred} \leftarrow Holts(\lambda\ of\ last\ N\ rounds)$
- **foreach** $PNode$ **do**
  - $c'_i \leftarrow$ `replicaCal(`λ_{pred},$avgProcessTime_i$,ρ_i`)`
  - ScaleTo(c'_i)

Replica Calculation

`replicaCal(`λ_{pred},$avgProcessTime_i$,ρ_i`)`

- $\mu_i \leftarrow \frac{\lambda_{pred}}{\rho_i}$
- $c'_i \leftarrow \lceil \mu_i * avgProcessTime_i \rceil$
- **return** c'_i

Scaling Procedure `scaleTo(`c'_i,c_i,$laxity_{i,90}$`)`

- **if** $laxity_{i,90} < threshold\ \&\ c'_i > c_i$ **then**
  - `/* Scale Up */`
  - scaleFunction(c'_i)
  - updateConnectionPool(c'_i)
  - **return**
- **else if** $laxity_{i,90} > threshold\ \&\ c'_i < c_i$ **then**
  - `/* Scale Down */`
  - scaleFunction(c'_i)
  - updateConnectionPool(c'_i)
  - **return**
- **else**
  - `/* No Scale Needed */`
  - **return**

server that receives function invocations as HTTP requests, calls the actual function code within the container and returns the function result with the same HTTP call. The deployed containers run in Mesos Agents. Each container listens to port 8080 and Mesos maps that port to a random port of the host Agent. Then by sending an HTTP Post request to *http://<agentIp>:<mappedPort>* we can invoke each function.

Gateway We have developed a custom Gateway for talking to Marathon using its provided API in order to deploy a new function or scale up/down existing ones. Also it acts as a Proxy for function invocations. It receives HTTP function invocations and propagates them to the deployed function containers. Although a function container can process more than one requests concurrently we chose to limit the processing to a single request at a time due to unexpected behaviour of the functions under heavy load. For this reason the Gateway keeps track of the in-flight requests to the deployed replicas and proxies the new requests only to available containers. If there are no available containers at that time it returns a *429 Too Many Requests* HTTP Status code or it can block the request and wait until some container becomes available.

Moreover, the Gateway listens to events from Marathon that announce the deployment or termination of containers and immediately updates its state. This way, it can begin to proxy traffic to the newly added replicas before the entire scale up of the function has completed.

Message Delivery We use Apache Kafka as the message delivery system. Kafka is one of the most widely utilized Publish Subscribe (pub/sub) systems available. Kafka has emerged as a state-of-the-art pub/sub system due to its high performance and attractive features of scalability and fault-tolerance. Linkedin uses Kafka to propagate more than 7 trillion of messages per day across its micro-services.

In our system the Producers send messages to Kafka topics by utilizing the Kafka's Producer API. We use one topic per Pipeline Flow, so all the messages for one specific Pipeline arrive at the same topic. Each message that is produced is in JSON format and includes the timestamp that is originally created by the Producer, a deadline time and its payload that is the actual Pipeline invocation. Kafka guarantees at least once processing schematics for the messages. The Connector Component uses the Kafka Java Client to subscribe to specified topics and then periodically polls the topic for new messages and propagates them to the Pipeline Manager.

Pipeline Manager The Pipeline Manager receives the description of the Pipeline with a JSON Formatted file. It maintains a set of N_i Threads for each PNode i that are responsible for the HTTP communication with the Gateway by utilizing the Gateway's provided API. The number of active HTTP connections per function should be the same with the number of the deployed replicas. This is kept synchronized between the Pipeline Manager and the Gateway by periodically polling the Gateway at the endpoint *\replicas* and giving as parameter the name of the function. Periodically, the Pipeline Manager reports back to Kafka the offsets of the messages that have been successfully processed. In this way we achieve at-least-once delivery semantic.

Monitoring system The Pipeline Manager collects and maintains a set of metrics as defined in section 3.1. Additionally, we use Prometheus [30] and Grafana [17] to periodically get the metrics from the manager and visualise them in a custom built Grafana dashboard. These tools are used only for monitoring and visualisation of the experiments and are not crucial components of the system.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

Hardware. We conducted our experiments in our local small edge cluster comprising 5 nodes. Each node has an Intel i7-7700 3.6GHz processor with 4 physical cores and 8 threads and 16GB of RAM. All of the nodes are interconnected with 1GBps Ethernet.

Software. All of the nodes run on Ubuntu 20.04 LTS. We run Apache Mesos 1.9 [4] as our serverless platform and we use Marathon 1.5 [27] in order to deploy Docker containers on top of Mesos. Marathon is used only to start/stop the containers, no other features of Marathon are utilized. With the current hardware, Mesos reports in total 40 CPUs and 80GB of RAM available. In this cluster we deployed our AMESoS system. We deployed Prometheus[30] and Grafana [17] for collecting performance statistics during the execution. In the experiments we also used a separate REDIS cluster as our cache for keeping each PNode's intermediate results.

Functions. We used OpenFaas Python templates in order to create our functions. We organized our functions into three different Pipelines as shown in Table 1 and Figure 3. The first pipeline creates signed PDF documents and consists of three different functions:

SHA256, which generates the digital signature of the text that will be in the PDF by calculating its checksum with SHA256, Qrcode which generates a QRCode image that contains the signed text with the signature and finally, Pdf which combines the signed text and the generated Qrcode in a single PDF file. Each function stores its output in our local REDIS cluster and returns a unique UUID of that object. Then this UUID is passed as argument to the next function that polls the object from REDIS and the cycle continues until the final function which returns the UUID of the final PDF. This pipeline is expected to depict high load that could easily exceed 100 requests/sec.

The second Pipeline consists of three functions that perform AES Encryption/Decryption of a message. The third Pipeline consist of six functions that perform a series of Matrix Multiplication functions. The latter one also splits the workflow into two branches and then merges them back together. These pipelines differ in the number of encrypt/decrypt cycles and martix sizes for multiplication which result in different execution times.

The average execution time for a single request in the Document Creation Pipeline is around 50ms and we set the deadline for those experiments to 500ms. For the Synthetic Workflow Pipeline the average execution time is 700ms and the selected deadline was 1.5s. Finally for the Matrix Multiplications Pipeline the average execution time is close to 1.3s with a deadline of 3.5s.

Rate. We used Microsoft's Azure traces [[8]], [[39]] in order to set the request inter-arrival time. We created a Custom Traffic Generator that generates new Pipeline requests in time-windows and in each time-window the Generator creates Pipeline requests by following a Poisson Distribution with a given λ. We used a predefined *seed* for the Poisson Distribution for repeatability across the experiments so in each experiment we had exactly the same rate as shown in Fig. 4.

6.2 Evaluation Method

We have compared AMESoS with the following approaches: **(a) Request Per Second(RPS) Scaler**. RPS is a scaling method that is used in two popular serveless environments, OpenFaas [29] and KNative [22] as their default auto-scaling method. RPS periodically obtains the incoming rate on each function and then calculates the desired number of replicas as follows: $desiredReplicas = \frac{currentLoad}{targetLoadPerReplica}$. RPS requires from the user to explicitly set a target number of requests per replica for each function by using a benchmarking tool such as Hey [20] to stress a single replica of the function. The benchmark might be affected by the current resource allocation of the container, so the user has to perform several tests with different allocations. We used Hey for benchmarking our functions before the experiment to estimate the average requests per replica for the RPS experiment. We developed an RPS Scaler which calculates the rate in a period of 2.5 seconds.

(b) Deadline With Prediction (DwP) Scaler. For the DwP Scaler we used the same replica calculation method as described in 4.5 where we partition time into *epochs* with a period of 2.5 seconds and we monitor the total number of messages that missed their deadline within each epoch. The Scaler makes scale up/down decisions as follows: (i) if more than 10% of the messages lost their deadline within the same window we trigger a Scale up based on the computed number of replicas. (ii) While, if no or very few invocations miss their deadline and our replica calculation method computes the desired replicas to be fewer than the currently running ones, then a scale down is triggered.

(c) LaSS. We used LaSS[35]'s scaling method provided in their GitHub repository. We focused only on their scaling methodology as the resource reclamation methods was not relevant to our experiments.

AMESoS. We have implemented our scaling approach as we described in section 4. We set our algorithm 1 to run every 2.5s. For our Laxity based scaling method we chose the threshold to be 200ms for the Document Creation, 1300s for Workflow #1 and 500ms for Workflow #2.

Experiment monitoring. In our experiments we use Grafana to monitor the execution of the different methods.

6.3 Evaluation Results

We evaluate our approach using the following metrics: (i) *Size of Queues*, for the duration of the experiment. (ii) *Number of Replicas*, for the duration of the experiment. (iii) *Deadline misses*, the percentage of pipeline messages that miss their deadlines. In Figure 4 we show the input rates we used in the experiments.

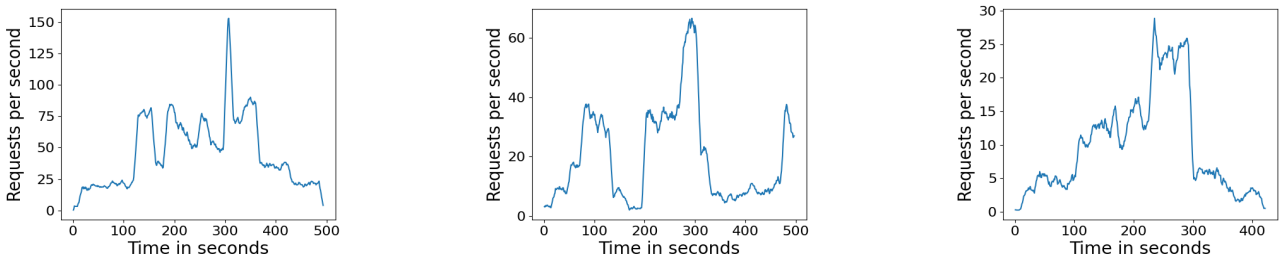
6.3.1 Number of Replicas. In Figures 5, 6, 7 we report each PNode's replica size during runtime.

For all of our Workflows tested we observe that the deployed replicas of each PNode have similar size for AMESoS and DwP approaches. This is expected as AMESoS and DwP use the same rate prediction algorithm. The small differences in replica deployments between them occur only during the traffic spikes and do not last long. Compared to DwP we see that for all the experiments AMESoS takes scaling decisions slightly faster. This is caused by the different time that each algorithm triggers a scale up command. This difference is because the DwP scales only when a portion of the messages have missed their deadline while, AMESoS predicts that, by exploiting the laxity metrics of the requests at each PNode, it can decide to scale without having to wait for them to expire. Because of this difference DwP decides to scale later than AMESoS with an additional decision delay that is equal, in worst case, with the whole Pipeline's execution time.

In contrast, RPS deploys fewer replicas during incoming rate spikes for the Document Creation Workflow. The reason for this is because RPS only scales according to the current measured rate unlike AMESoS and DwP that make a prediction for the future rate for their scaling. This can be also seen in the other two Workflows where RPS scales in steps to the needed replicas as rate keeps increasing while the other approaches scale immediately for that rate value with the help of Holt's prediction. Finally, RPS scales one function after the other in the Pipeline while the other approaches scale the entire Pipeline's functions simultaneously. This behaviour can be explained since we consider the traffic only in the first node of the pipeline and we assume that the rest of the nodes will eventually have the same input rate after the scale up of the first node, while RPS monitors each PNode's rate independently. Similarly to RPS, LaSS calculates the replicas needed by exploiting the current

Table 1: Experimental Pipelines

Workflow	Description	Number of Functions	Branches	Average Execution Time	Deadline
Document Creation	Generates a PDF and stores it in REDIS	4	No	50ms	500ms
Synthetic Workflow	Performs AES Encryption/Decryption	3	No	700ms	1500ms
Matrix Multiplications	Matrix Multiplication functions	6	Yes	1300ms	3500ms



(a) Document Creation **(b) Synthetic Workflow** **(c) Matrix Multiplications**

Figure 4: Pipeline input rates.

measured rate for each PNode. At the Document Creation and Synthetic Workloads Fig.5, 6 LaSS uses more replicas than AMESoS and does not scale down for prolonged periods.

6.3.2 Size of Queues. In Figures 8, 9, 10 we observe each PNode's queue size during the experiment. Even though all approaches create queues we can see that for all the experiment our AMESoS approach has lower queue sizes compared to DwP and RPS. For the Document Creation Workflow, AMESoS has lower queue size that is up to 20% lower compared to both DwP and RPS. Although, AMESoS presented higher peaks compared to LaSS the queue draining time for both was similarly fast. Due to the scrape interval of Prometheus, that we use for monitoring and data gathering, in combination with the fast incoming rate of the messages we cannot measure the accurate queue size peak for these cases. For the Synthetic Workflow, AMESoS performed 30% better than DwP, 70% better than RPS and 50% better than LaSS in terms of queue sizes. Finally we observe that in the last Workload, although AMESoS experienced a queue peak that is up to 20% more than the other approaches, it was removed faster and also it was the only queue that was created in the entire experiment. For RPS and LaSS we see that when a queue is created it is not cleared immediately like AMESoS and DwP but it is transferred from one PNode to another and it requires additional time to clear. This can be especially seen in Figure 9(c),(d).

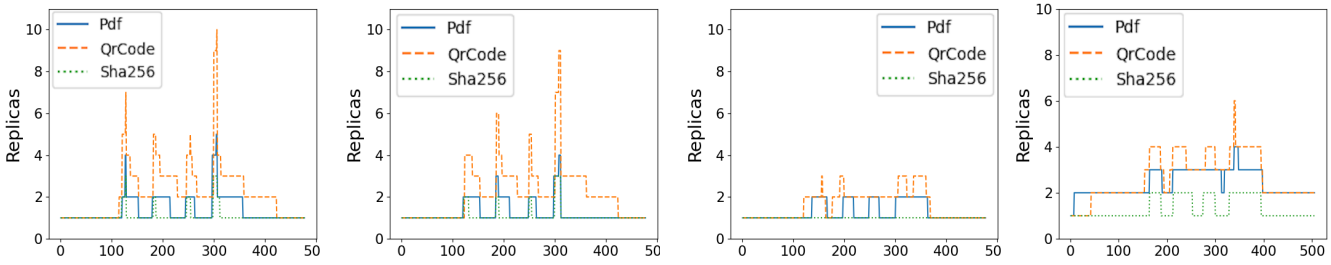
6.3.3 Deadline misses. As we observe in Figure 11, AMESoS outperformed RPS by 70% in the Deadline misses. This indicates that by monitoring only the current input rate of the functions in a pipeline is not enough to take scaling decisions for the entire pipeline and thus a prediction for the near future traffic is needed.

Comparing AMESoS with DwP, even though the differences are smaller, AMESoS has a 60% fewer deadline violations at the Matrix Multiplication Workflow where the execution time of the Pipeline is the highest. This occurs because by monitoring the laxity of the messages as they get processed by the functions, AMESoS scales as soon as it predicts that some messages might miss their deadlines. Considering also that the functions do not scale immediately, the sooner we schedule a scale up command the faster our system will become stable. Without the laxity indication, the scaler would order a scale up command only after some messages get expired. This means that the scaling decision will be delayed at most as much as the execution delay of the whole pipeline. Compared to LaSS AMESoS experienced slightly fewer deadline misses for Document Creation and Matrix Multiplication Workflows and presented considerably better performance, 22% fewer deadline misses, in the Synthetic Workflow.

In Fig 12 we look closely at the accumulative resources required by AMESoS and LaSS during the three scenarios. We observe that LaSS uses more replicas than AMESoS and does not scale down for prolonged periods for the Document Creation and the Matrix Multiplication pipelines. Due to this behaviour AMESoS uses up to 28% less resources. We observe that AMESoS uses 13% more resources in the Synthetic Workload but results in 22% fewer deadline misses.

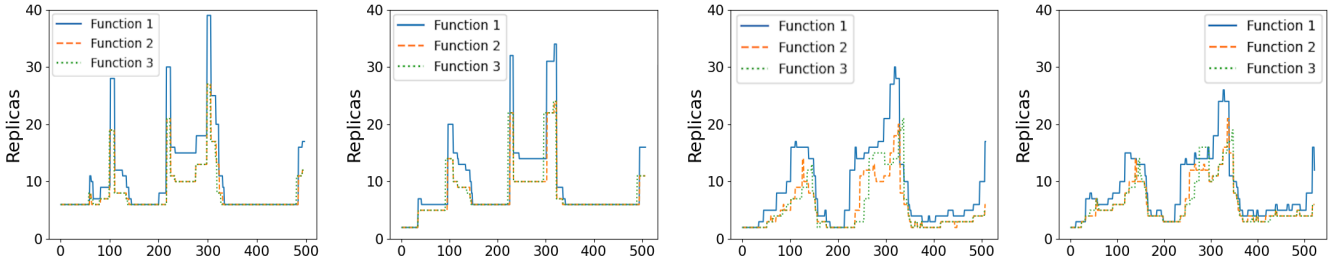
7 RELATED WORK

Pipeline Workflows. The need for supporting data pipelines has come up in a variety of edge cluster settings. For example, Apache Storm[7] is a distributed realtime computation system and provides reliable real-time processing on streams of data. Apache Spark[6] is an open-source analytics engine for large-scale data processing by providing an interface for programming clusters with implicit data parallelism and fault tolerance which can be used for both batch and stream pipelines processing. Apache Flink[2] is a distributed processing engine for stateful computations over unbounded and bounded data streams. EdgeWise[15] which is implemented on top of Apache Storm is incorporating a congestion-aware scheduler and a fixed-size worker pool into an edge friendly Streaming process environment.



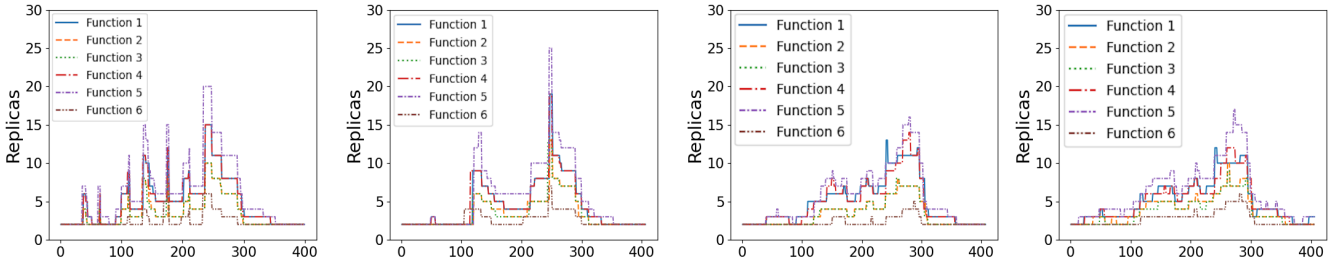
(a) AMESoS (b) Deadline with Prediction (c) RPS (d) LaSS

Figure 5: Document Creation Replicas



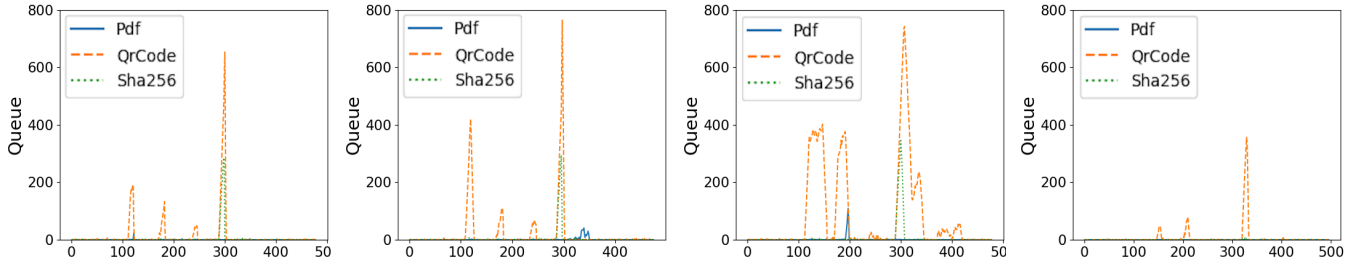
(a) AMESoS (b) Deadline with Prediction (c) RPS (d) LaSS

Figure 6: Synthetic Workflow Replicas



(a) AMESoS (b) Deadline with Prediction (c) RPS (d) LaSS

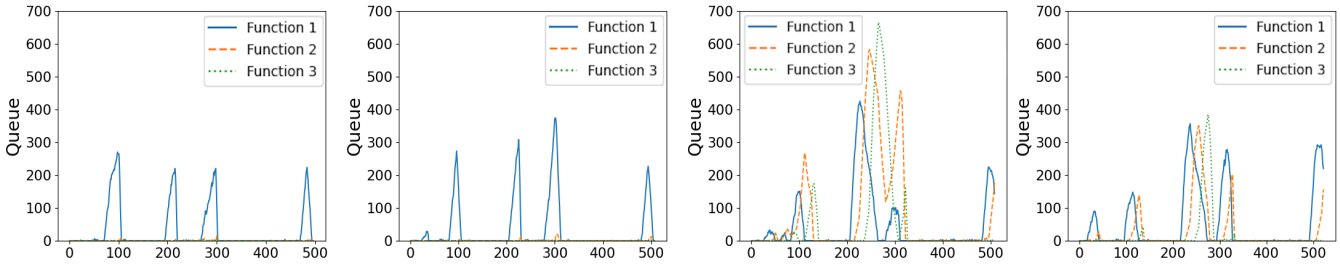
Figure 7: Matrix Multiplications Replicas



(a) AMESoS (b) Deadline with Prediction (c) RPS (d) LaSS

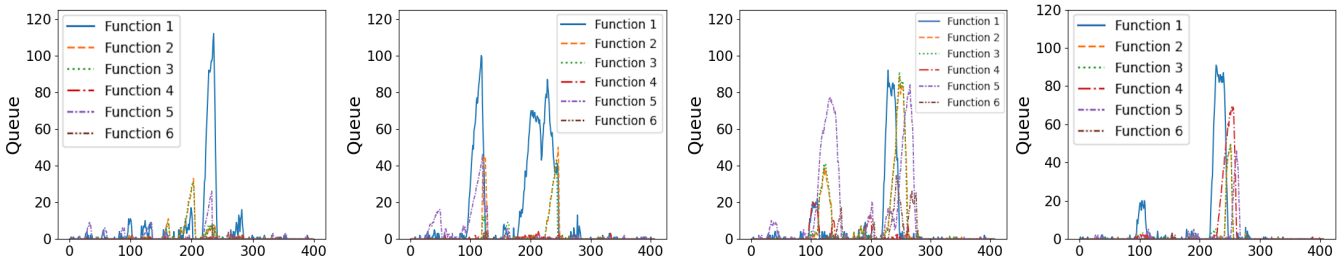
Figure 8: Document Creation Queue sizes

GrandSLAm[21] estimates the completion time of requests that propagate through a set of application stages implemented as individual microservices. It leverages this estimation to dynamically batch and reorder requests at each microservice to meet the respective target deadline. Magic-Pipe [11] which is a self-optimizing



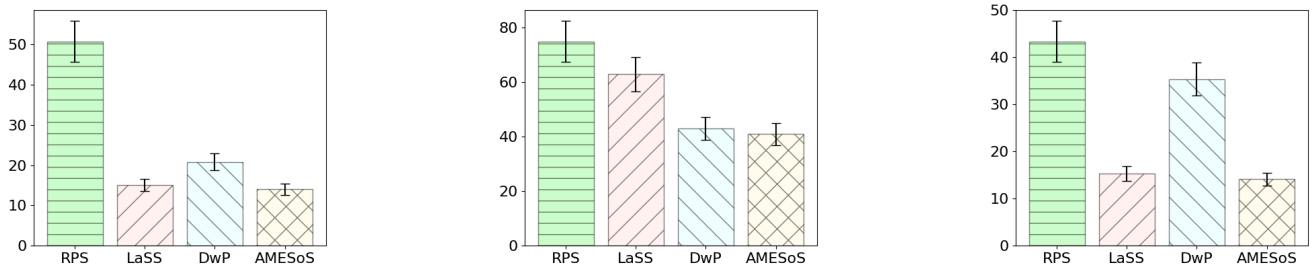
(a) AMESoS (b) Deadline with Prediction (c) RPS (d) LaSS

Figure 9: Synthetic Workflow Queue sizes



(a) AMESoS (b) Deadline with Prediction (c) RPS (d) LaSS

Figure 10: Matrix Multiplications Queue sizes



(a) Document Creation (b) Synthetic Workflow (c) Matrix Multiplications

Figure 11: Percentage of deadline misses in the different pipelines.

video analytic pipeline that leverages AI techniques to periodically self optimize. They use State-Action-Reward-State-Action (SARSA) Reinforcement Learning in order to perform some actions based on the current state of the environment. This approach is limited to video analytics pipelines and needs re-training for different pipelines. Although the aforementioned systems provide pipeline processing they are not designed to meet the serverless environment characteristics. gg [14] is a framework that enables jobs to be expressed as a pipeline of individually transient containers. It deploys these containers in a serverless environment and takes care of their instantiation and the flow of data across the containers. Fifer[18] deals with over-provisioning resource utilization that occurs in serverless environments during workload fluctuations. This is done by efficiently bin packing jobs to fewer containers.

Scheduling and workflow elasticity. Lopez and Spillner in [26] studied microservices scalability. They propose a replica count determination method by exploiting a pre-calculated combinations matrix of different number of replicas for each microservice in a specific architecture with predictable behaviour under certain conditions. Although their method is effective it requires for the system administrators to acquire and maintain that set of combinations for each architecture in advance. This is very limiting for serverless environments where the deployments can last for a very small amount of time and thus it's not efficient to test and pre-calculate the combination matrix. The scheduler proposed in TetriSched [34] aims to prevent tasks from being sent to a sub-optimal set of resources as resources are being held by earlier jobs, but it limited as it only supports per-operation targets, not an end-to-end pipeline latency that we target in our approach. In our previous work [38], [37] we have proposed scalable distributed top-k join queries in topic-based pub/sub systems and looked at the problem of maximizing determinism under latency constraints. In [32] they formulate the

elasticity on heterogeneous resources as a Markov Decision Process and solve this process with linear function approximation and tile coding to efficiently compute the elasticity policies at run-time.

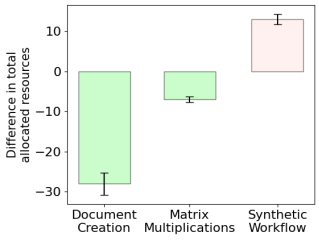


Figure 12: Comparison in resource allocation between AMESoS and LaSS for the three Pipelines

Serveless scheduling Recent research has been conducted in the field of auto-scaling and elasticity of serverless environments and its earlier form of microservices. Francisco Romero et. al in Llama[31] present a framework for heterogeneous and serverless auto-tuning video pipelines, which are represented as DAGs of operations. By giving an end-to-end deadline Llama calculates the latency target for each operation invocation and dynamically assigns configurations across heterogeneous hardware that best meet the calculated per-invocation latency target. AMesos on the other hand, is designed for latency sensitive streaming data and takes into consideration and predicts the traffic bursts that may occur. Wang et. al in LaSS[35], presents a model-driven approach for running latency-sensitive serverless computations on edge resources. LaSS uses queuing-based methods to select an allocation for each hosted function and auto-scales the allocated resources in response to workload dynamics. It uses a fair-share allocation approach that guarantees minimum allocated resources for each function. The authors propose a resource reclamation method based on container deflation to reallocate resources from over-provisioned functions to under-provisioned ones. FaaSRank[36], is a scheduling service for serverless platforms based on metrics gathered from servers and deployed functions. FaaSRank uses Reinforcement learning and and Neural Networks to learn scheduling policies. However, LaSS FaaSRank are not configured for pipeline optimization. Serveless in the wild[33] proposes a resource management policy that reduces the number of function cold start, while spending fewer resources by using observations gathered from the entire FaaS workload of Azure Functions.

8 CONCLUSION

In this paper, we presented AMESoS, our scalable and elastic framework that aims to meet the urgency demands on latency-sensitive pipelines and a lightweight event-driven scaling algorithm that adapts dynamically in the presence of overloads. We have implemented AMESoS on our edge cluster and illustrate that it achieves as much as 70% improvement in Deadline misses compared to state-of-the-art systems.

Acknowledgment. This research has been supported by the H2020 LAMBDA Project 734242, the EU ICT-48 2020 project TAILOR (No. 952215), the H2020 AutoFair project (No. 101070568).

REFERENCES

[1] Amazon. https://aws.amazon.com/lambda/.
[2] Apache Flink. https://flink.apache.org/.
[3] Apache Kafka. https://kafka.apache.org/.
[4] Apache Mesos. https://mesos.apache.org/.
[5] Apache OpenWhisk. https://openwhisk.apache.org/.
[6] Apache Spark. 2022. https://spark.apache.org/.
[7] Apache Storm. https://storm.apache.org/.
[8] Azure. https://github.com/Azure/AzurePublicDataset.
[9] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. *Using Application Knowledge to Reduce Cold Starts in FaaS Services.* 134–143.
[10] C. Chatfield. 1978. The Holt-Winters Forecasting Procedure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 27, 3 (1978), 264–279.
[11] Giuseppe Coviello, Yi Yang, Kunal Rao, and Srimat Chakradhar. 2021. Magic-Pipe: Self-Optimizing Video Analytics Pipelines. In *Middleware* (Québec city, Canada).
[12] Docker Swarm. https://docs.docker.com/engine/swarm/.
[13] Fission. 2022. https://fission.io/.
[14] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *USENIX ATC*. Renton, WA, 475–488.
[15] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. 2019. EdgeWise: A Better Stream Processing Engine for the Edge. In *USENIX ATC*. USENIX Association, Renton, WA, 929–946.
[16] Google. 2022. https://cloud.google.com/functions.
[17] Grafana. 2022. https://grafana.com/.
[18] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C. Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. 2020. Fifer: Tackling Resource Underutilization in the Serverless Era. In *Proceedings of the 21st ACM International Middleware Conference* (Delft, Netherlands). 280–295.
[19] HashiCorp Nomad . 2022. https://www.nomadproject.io/.
[20] Hey. https://github.com/rakyll/hey.
[21] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *14th EuroSys* (Dresden, Germany).
[22] Knative. https://knative.dev/docs/.
[23] Kubernetes. https://kubernetes.io/.
[24] J. Liu, L. Sun, W. Chen, and H. Xiong. 2016. Rebalancing bike sharing systems: A multi-source data smart optimization,. In *SIGKDD 2016*. San Francisco, USA, 1005–1014.
[25] Manuel Ramírez López and Josef Spillner. 2017. Towards Quantifiable Boundaries for Elastic Horizontal Scaling of Microservices. In *ACM UCC 2017 (Companion Proceedings)* (Austin, Texas, USA). New York, NY, USA, 35–40.
[26] Manuel Ramírez López and Josef Spillner. 2017. Towards Quantifiable Boundaries for Elastic Horizontal Scaling of Microservices. In *UCC 2017 (Companion Proceedings)* (Austin, Texas, USA). New York, NY, USA, 35–40.
[27] Marathon. https://mesosphere.github.io/marathon/.
[28] Microsoft. 2022. https://azure.microsoft.com/en-us/services/functions/.
[29] OpenFaas. https://www.openfaas.com/.
[30] Prometheus. https://prometheus.io/.
[31] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *Proceedings of the ACM SoCC* (Seattle, WA, USA). 1–17.
[32] Gabriele Russo, Valeria Cardellini, and Francesco Lo Presti. 2019. Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources. In *13th ACM DEBS* (Darmstadt, Germany). 31–42.
[33] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX ATC*.
[34] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. 2016. TetriSched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Eleventh ACM European Conference on Computer Systems* (London, United Kingdom).
[35] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. 2021. LaSS: Running Latency Sensitive Serverless Computations at the Edge. In *30th ACM HPDC* (Sweden).
[36] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. 2021. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *2021 ACSOS*. Washington, DC, USA, 31–40.
[37] Nikos Zacheilas, Dimitris Dedousis, and Vana Kalogeraki. 2018. Scalable Distributed Top-k Join Queries in Topic-Based Pub/Sub Systems. In *BigData 2018*.
[38] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, M. Papatriantafilou V. Gulisano, and P. Tsigas:. 2017. Maximizing Determinism in Stream Processing Under Latency Constraints. In *DEBS 2017* (Barcelona, Spain).
[39] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *ACM SOSP*. 724–739.