OPTIMIZING SERVERLESS ARCHITECTURES FOR DATA-INTENSIVE ANALYTICS WORKLOADS

ANNA MARIA NESTOROV

Dissertation submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Architecture

Universitat Politècnica de Catalunya - BarcelonaTECH

March 2024

Anna Maria Nestorov: *Optimizing Serverless Architectures for Data-Intensive Analytics Workloads*
Dissertation submitted in partial fulfillment of the requirements for the Doctoral Degree in Computer Architecture, March 19, 2024.

SUPERVISORS:
Josep Lluís Berral
David Carrera

AFFILIATION:
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - BarcelonaTECH

LOCATION:
Barcelona

COLOPHON

The front cover picture shows the sunrise over one of the seven Rila lakes, Bulgaria. Unlicensed and free-to-use picture from WallPaperFlare.
This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".

The greatest glory in living lies not in never falling,
but in rising every time we fall.

Nelson Mandela

ABSTRACT

In recent years, serverless computing has gained significant attention from academia and industry. Its execution paradigm allows provisioning resources on-demand, leaving the tedious work of infrastructure and operational concerns to the cloud provider. This allows users to entirely focus on their core business logic, decomposing their jobs into small stateless functions to be managed independently and updated flexibly. In addition, serverless computing attracts many tenants in today's cloud environments also for its fundamental principles, including the 'pay-per-use' cost model with fine-grained charge granularity, greater flexibility, and transparent elastic resource auto-scaling all the way down to zero resources in the absence of requests allowing massive scalability.

Given the success in many simple real-world applications across multiple domains ranging from web microservices to Internet of Things (IoT) applications in a serverless environment, researchers in both academia and industry have begun to explore its applicability in more complex data-intensive analytics workloads. These workloads are usually resource-intensive, highly parallel, and are characterized by considerable inter-function communications. However, this shift comes with inherent challenges, making it essential to align the paradigm with this type of application's specific needs and constraints. This research area has gained significant interest and is currently considered one of the most compelling areas of study.

This thesis shows that it is possible to efficiently execute modern data-intensive analytics workloads, traditionally deployed in managed cloud clusters, within serverless computing environments using direct data inter-function communication and optimized performance-cost efficient resource allocation policies. To demonstrate this thesis, we first build a performance model for serverless workloads that considers how data is shared between functions, including the volume of data and the underlying communication technology employed. With a relative error of 5.52%, the proposed model allows us to evaluate the performance of a representative workload in serverless, analyzing task granularity and concurrency, data locality, resource allocation, and scheduling policies. We also explore possible solutions for the data-sharing problem, like using local memory and storage. Our results indicate that the performance of data-intensive analytics workloads in serverless can be up to 4.32× faster depending on how these are deployed. Furthermore, this characterization emphasizes the current state-of-practice centralized object storage inefficiencies and highlights the primary importance of efficient resource utilization.

We then present Floki, a data forwarding system addressing the centralized object storage bottleneck by proactively enabling direct inter-function communication between producer-consumer pairs of functions through utilizing established and widely adopted fixed-size communication mechanisms. Floki creates point-to-point data channels for intra-node and inter-node data transmission, allowing data to be transferred directly from producer to consumer functions in a fully transparent fashion, minimizing data copying over the network. Floki offers workflow-oriented data communication, increasing performance while minimizing resource requirements without imposing any constraint on function placement. Specifically, its flexibility and observability allow creating data channels adapting to the specific function scheduling of the underlying orchestration framework. Our experimental evaluation, focusing on the principal communication patterns in distributed systems, shows that Floki reduces the end-to-end time up to 74.95×, decreasing the most extensive data sharing time from 12.55 to 4.33 minutes, translating into nearly two-thirds of time-savings. Furthermore, leveraging the efficiency of the employed fixed-size communication mechanisms, Floki achieves up to $50,738\times$ of resource-saving, equivalent to a memory allocation of approximately 1.9MB, vastly outperforming state-of-practice object storage allocation of 96GB.

This thesis finally investigates how to achieve efficient resource utilization within modern serverless environments and proposes Dexter, a novel resource allocation manager, fully leveraging serverless computing elasticity. Dexter continuously monitors application execution, dynamically allocating resources at a fine-grained level to ensure performance-cost efficiency (optimizing total runtime cost). Dexter is novel in combining predictive and reactive strategies that fully exploit the resource elasticity of serverless to enhance the performance-cost efficiency for workflow executions. Unlike black-box Machine Learning (ML) models, Dexter quickly reaches a sufficiently good solution, prioritizing simplicity, generality, and ease of understanding. The proposed experimental evaluation, considering two industry-standard benchmarks and a real-world workload, demonstrates that our solution achieves a significant cost reduction of up to 4.65×, while improving resource efficiency up to 3.50×, when compared with the default serverless Spark resource allocation that dynamically requests exponentially more executors to accommodate pending tasks. Dexter also enables substantial resource savings, demanding up to 5.71× fewer resources. Finally, Dexter is a robust solution to new, unseen workloads, achieving up to 2.72× higher performance-cost efficiency thanks to its conservative resource scaling approach.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to all those who have contributed to completing this doctoral thesis. The Ph.D. has been a challenging journey, a formidable test of resilience, perseverance, and unwavering dedication. The support, invaluable guidance, and encouragement of all the people I have met along the way have been inestimable and made the Ph.D. journey a transformative and enriching experience, not only shaping me as a researcher but also as an individual.

First and foremost, I express my deepest gratitude and appreciation to my Ph.D. advisors, Dr. Josep L. Berral, and Dr. David Carrera, and previously also Dr. Jordà Polo, who have been a constant source of inspiration. Your expertise and constant encouragement have been fundamental in shaping the direction of my research. Your insightful feedback and constructive criticism have significantly contributed to refining my ideas and research. I am truly fortunate to have had you as my advisors, and I am immensely grateful for the invaluable lessons and skills I have gained under your mentorship. Also, thanks to Yolanda for being my tutor at UPC for some time and managing all the related bureaucracy.

I extend my heartfelt gratitude to my Ph.D. thesis committee members, including Dr. Xavier Martorell, Dr. Ramon Nou, Dr. Jordi Guitart, Dr. Marc Sanchez-Artigas, and Dr. Amirhosein Taherkordi. Thank you for accepting being part of this final step in my Ph.D. journey, for the time and effort to rigorously examine my research which I am sure it will provide constructive feedback and valuable suggestions that will significantly enhance the quality of this research.

Special thanks are also due to the supervisors and mentors I had the honor to work with in my collaboration with IBM during my stay in Barcelona and internship abroad. I want to thank Claudia Misale, Chen Wang, Alaa Youssef, Radu Stoica, and Bernard Metzler for the time invested in mentoring me, your dedication, and the positive impact you have had on my academic and professional development. I am privileged to have had the opportunity to work under your guidance.

I would like to say thank you to my colleagues and fellow researchers who provided stimulating discussions, insightful critiques, and a supportive community throughout my Ph.D. journey. First of all, the BSC Data-Centric Computing team made me feel at home from day one at BSC: Nicola, Felipe, Aaron, Alberto, Paco, Kiyana, Dani, Marcelo, David B., Claudia F., Ferran, Diego, Marc, Peini, Omar, Gonzalo, and Pol. I am proud to have been part of a team with someone as capable and dedicated as you. Your support has been my anchor, giving me the strength to overcome challenges and persevere. Many thanks also

to Beatrice, Hany, and Andrea for the valuable suggestions and the moments we shared when I was in Zurich that have made the experience unforgettable. It wouldn't have been the same without each of you. The Data-Centric Computing team, the colleagues in Zurich, and other colleagues at BSC, like Fabio, Guillem, Kilian, Filippo, Marta, Victor, and Noelia, made this experience not only academically enriching but also personally rewarding.

I am indebted to my family for their unwavering encouragement, patience, love, and understanding during the challenging phases of my Ph.D. journey. I am truly grateful for the sacrifices they have made and the support they have constantly provided me during the previous academic journey, which allows me to embark on this new chapter of my life with a Ph.D. in hand. Mum, particular thanks to you for always believing in me, instilling in me the values of determination and perseverance, and transmitting to me the willingness to be curious, to learn, and to improve myself everyday.

Enrico, I cannot thank you enough for your presence by my side; your encouragement and understanding have been my main source of strength and motivation. I am profoundly grateful that you always believe in me, even when complex and hard situations arise, and you have been a constant reassurance, offering not just a listening ear but also a shoulder to lean on. You have always been a constant pillar of support and point of reference.

Last but not least, I am incredibly grateful to all my friends. Especially, I want to thank Francesco, Jesica, Nicola, Thais, Federico, César, and Noemi. Thank you for the incredible support and warmth you have provided me, making you more than just friends but truly my family in Barcelona. Living abroad comes with unique challenges; having friends like you by my side has made all the difference. Big thanks to Alberto S., who has pushed me to pursue a Ph.D.. Your words of encouragement have been fundamental in my decision to embark on this challenging yet rewarding journey. Despite the physical distance between us, your support has bridged any gaps, making your friendship a crucial part of this experience.

In conclusion, I extend my deepest thanks to everyone who played a role, big or small, in completing this Ph.D. thesis. Thank you for being an integral part of this significant chapter in my academic and professional life.

CONTENTS

ACRONYMS

BDS	Boosted Decision Stumps
BRR	Bayesian Ridge Regression
CAGR	Compound Annual Growth Rate
CDF	Cumulative Distribution Function
CI/CD	Continuous Integration and Continuous Delivery
CNCF	Cloud Native Computing Foundation
CRD	Custom Resource Definition
DAG	Directed Acyclic Graph
DART	Dropouts Additive Regression Tree
DRL	Deep Reinforcement Learning
DSO	Distributed shared object
FaaS	Function as a Service
FIFO	First In First Out
GB	Gradient Boosting
IoT	Internet of Things
IPC	Inter Process Communication
JVM	Java Virtual Machine
LOOCV	Leave One Out Cross-Validation
LR	Linear Regression
LSTM	Long Short Term Memory
MAE	Mean Absolute Error
MILP	Mixed-Integer Linear Programming
ML	Machine Learning
NFS	Network File System
NN	Neural Network
OCR	Optical Character Recognition
OLAP	On-Line Analytical Processing
PaaS	Platform as a Service
PV	Persistent Volume
RDD	Resiliant Distributed Dataset
RF	Random Forest
ROI	Return Of Investment
RPC	Remote Procedure Call
SIMD	Single Instruction Multiple Data
3NF	Third Normal Form
TCP	Transmission Control Protocol
TIFF	Tagged Image File Format
VM	Virtual Machine

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In this chapter, we first provide the context of this thesis and the motivation underlying the work, highlighting the research challenges we are attempting to address and the thesis statement (Section 1.1). Then, we present the the main contributions of this thesis, specifying for each contribution the associated achievements (Section 1.2).

1.1 THESIS CONTEXT AND MOTIVATION

Representing the long-held dream of computing as a utility [1], the cloud computing model enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [2]. However, users still have to make decisions that can be challenging for many: instance types, cluster size, load balancing strategy, etc. In this context, serverless computing has received a significant uptick in attention over the last few years, both in academia and industry. Unlike the traditional cloud computing model, in the serverless paradigm, the operational concerns of the infrastructure are fully managed by the cloud provider, completely relieving the user from challenging management decisions. Serverless computing [3] attracts many tenants in today's cloud environments also for its fundamental principles, including transparent elastic resource scaling allowing massive scalability, 'pay-as-you-go' cost model with fine-grained charging granularity, and hassle-free management. While allowing tenants to focus on their core logic and run their applications as a set of functions on-demand, saving costs for unused computing resources, the serverless computing paradigm enables cloud providers to control the entire development stack and efficiently optimize and manage resources. Today, almost all major cloud providers offer services to deploy serverless functions through a Function as a Service (FaaS) computing platform. Examples include AWS Lambda [4], Google's Cloud Functions [5], and IBM's Cloud Functions [6] in commercial clouds, and Open-Lambda [7], Open-Whisk [8], and Knative [9] in open-source projects.

Given the success in many simple real-world applications across multiple domains ranging from web microservices to IoT applications, in a serverless environment, academia and industrial research have begun to explore its usage in more complex resource intensive and highly parallel applications, characterized by considerable inter-function communications. This research area has gained significant interest and is currently considered one of the most compelling areas of study. Researchers have demonstrated how the serverless paradigm can be successfully exploited in complex applications, including linear algebra tasks [10, 11], video encoding [12–14], deep learning training and inference [15–28], Monte Carlo simulations [29–32], as well as MapReduce-style [11, 33–36] and SQL-style analytics [37–40]. The serverless market share was valued at USD 8.93 billion in 2022 and is projected to reach USD 55.24 billion by 2030, with a Compound Annual Growth Rate (CAGR) of 22.45% [41].

Despite these successes, serverless computing exhibits limitations [23, 42–45]. The absence of network addressability and the lack of efficient data sharing between functions makes it challenging for a vast number of data-intensive analytics workloads to benefit from serverless. Not supporting direct inter-function communication, major serverless platforms [4–6, 46] take disaggregation to an extreme, imposing functions to exchange data only through shared object storage. Being data-intensive analytics workloads characterized by a considerable amount intermediate data transfers, shared object storage becomes a bottleneck for efficient inter-function communication due to its high-latency access. Furthermore, since data-intensive analytics workloads are typically resource-intensive and inherently parallel, they highly benefit from the elasticity and the virtually unlimited scalability offered by serverless platforms. In particular, since early 2022, a prominent effort has been made towards leveraging serverless platforms to efficiently run big data analytics frameworks, such as Apache Spark [47], which are traditionally deployed in managed cloud clusters. GCP Dataproc Serverless [48], Databricks Serverless [49], and IBM Analytics Engine [50] exemplify this trend. However, auto-allocating resources to complex workflows, such as big data analytics applications, is complex because the relationship between resource allocation and performance is complicated and changes over the application runtime [51].

As detailed in the remainder of this chapter, currently serverless comes with limitations, such as communication through shared object storage and the absence of an optimized resource auto-scaling mechanism. These limitations are posing several research challenges when porting data-intensive analytics workloads to serverless platforms.

> Research challenge 1: *While the desirability and potential advantages of enabling more complex workloads, such as data-intensive analytics workloads, in serverless environments are known, we need to fully understand its feasibility and efficiency. This involves identifying which are the key elements preventing these workloads to completely benefit from serverless environments. In particular, to understand whether new approaches are necessary, there is the need to analyze in-depth the performance impact of running more complex workloads on*

serverless platforms. This analysis should focus on various factors, including inter-function communication latencies, workload parallelisms, and resource requirements.

Although shared object storage enables the complete decoupling of storage and computation in serverless environments and offers a durable and low-cost data storage solution, it shows relatively high access latencies, leading to highly inefficient workload executions when running data-intensive analytics workloads. This brings a new research challenges:

Research challenge 2: *While object storage represents a good solution when there is the need to persist data, letting users to use them over time, we need to have an alternative data communication mechanism for scenarios where the long-term availability of data is not a requirement, allowing a high-performance volatile data sharing. Instead of buffering intermediate results in state-of-practice remote shared storage and exchange data between functions over the network, it is crucial to explore alternative communication mechanisms that facilitate rapid and direct intermediate data communication between functions.*

Finally, while serverless auto-scaling automatically increases or decreases the number of workload instances based on the number of incoming requests, when executing more complex data-intensive analytics workloads it is crucial to accurately adjust resources at a fine-grained level. This leads us to the final research challenge this thesis attempts to address.

Research challenge 3: *Given their typical resource-intensive and inherently parallel nature, it is fundamental to comprehend how to dynamically scale the amount of resources (i.e., parallelism/scale level), and at which granularity level, to fully leverage the virtually unlimited scalability offered by serverless platforms. Specifically, we need to dynamically adapt to variations by setting the level of parallelism to maximize the resource utilization and to achieve a balance between performance and cost, thereby minimizing the overall monetary cost while ensuring acceptable runtime performance.*

Due to these reasons, this thesis wants to demonstrate the feasibility of the following statement: **It is possible to efficiently execute modern data-intensive analytics workloads, traditionally deployed in managed cloud clusters, in serverless computing environments by using direct data inter-function communication and optimized performance-cost efficient resource allocation policies.**

1.2 THESIS CONTRIBUTIONS

To prove the aforementioned thesis statement, we divided the work into the following three major research contributions which are summarized as follows:

C1: Workload Scalability and Performance Analysis

C2: Direct Inter-Function Communication Enablement

C3: Automatic Performance-Cost Efficient Resource Allocation

The comprehensive analysis performed in the first contribution identifies and underscores research opportunities aimed at optimizing the execution of data-intensive analytics workloads in serverless environments. The analysis provides strong motivations for the subsequent second and third contributions, as illustrated in Figure 1.1. The second and third contributions are independent since they tackle two different problems, but they are complementary as they are part of one incremental work towards an efficient execution of data-intensive analytics workloads in serverless environments.

1.2.1 *Contribution 1: Workload Scalability and Performance Analysis*

This first contribution aims to fully characterize a proper use case data-centric analytics workload and to assess its performance when executed in current serverless platforms. Focusing on data communication, we deeply study and model all memory and storage data exchange mechanisms available in current open-source serverless platforms. In particular, in this analysis, we consider various function replication scenarios for embarrassingly parallel functions to optimize the balance between data communication and computation delays. The workload is then used to analytically demonstrate that directly using existing serverless platforms would lead to extremely inefficient executions of such workloads. This analysis allows us to understand which are the most critical key elements preventing data-intensive analytics workloads to completely benefit from serverless environments, and thereby indicating whether new approaches are necessary.

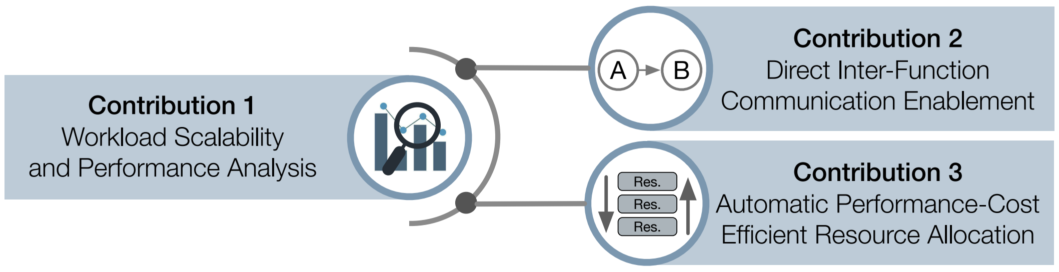


Figure 1.1: Thesis contributions

The achievements of this contribution are:

– Workload characterization in terms of execution times, scalability, and data movements.

– Identification of the serverless functions and the parallelism opportunities, showing a methodology to extract the necessary components to translate a traditional monolithic application to serverless.

– Development of a performance model capturing the performance of serverless workloads involving data exchanges, supporting different configurations and data exchange mechanisms.

– Evaluation of workload performance impact concerning task granularity and concurrency, data locality, resource allocation and scalability, and scheduling policies.

1.2.2 *Contribution 2: Direct Inter-Function Communication Enablement*

The second contribution has the scope to enable the runtime management of direct inter-function communication essential to efficiently run data-centric analytics workloads in open-source serverless environments. Utilizing established and widely adopted communication mechanisms, we introduce a new point-to-point data sharing communication mechanism capable to handle arbitrarily complex data structures. This data sharing mechanism enables proactive and fast data transfers without imposing any constraint on function placement. In this contribution we address the centralized storage bottleneck by proposing a system facilitating direct communication between functions, where data exchanges are orchestrated on a producer-consumer functions pair level, minimizing data copying over the network. Directly sharing data between functions implies two main advantages. First, the number of concurrent read/write operations is reduced since resources are shared among a lower number of functions, i.e., the ones co-placed on a given node, or of exclusive use. Second, leveraging multi-threading capabilities, a producer function can concurrently send the same data to multiple consumer functions, while a consumer function can simultaneously receive data from multiple producer functions.

The achievements of the contribution are:

– Build a proactive workflow-based data forwarding system enabling point-to-point data transfers without incurring additional overheads, such as state-of-practice storage overheads.

– Design of a full communication stack between functions, allowing non-colocated functions to seamlessly exchange data as they are hosted on the same node through local read and write operations.

- Build an in-memory mechanism tailored for transferring volatile data, capable of dealing with arbitrary intermediate data sizes efficiently and scalable on the data volume.

1.2.3 *Contribution 3: Automatic Performance-Cost Efficient Resource Allocation*

Finally, the third contribution concerns resource scaling of the different functions composing a workload. We investigate the performance enhancements and cost implications achieved by exploiting function parallelism. We leverage the elasticity of serverless computing by automatically adapting the number of allocated resources at the smallest granularity level, ensuring optimal performance-cost efficiency. This is achieved through continuous monitoring of application execution, allowing for adaptive responses to changes in performance and cost. In this last contribution, we provide a sufficiently good solution in real-time, prioritizing simplicity, generality and ease of understanding, combining predictive and reactive strategies. This design choice, allows us to feature small enough training delay to seamlessly execute model retraining on the fly, even within a multi-tenant cloud-like scenario, in response to a significant workload variation.

The achievements for this last contribution are:

- A performance and cost characterization across various granularities, highlighting the impact of scaling out and scaling in on execution duration and corresponding monetary cost.

- Fine-tune the scaling level at the smallest granularity considering at every optimization step the performance-cost trade-off, enabling more efficient executions in terms of total runtime cost.

- Provide a standalone an pluggable module that automatically reacts to deviations in performance-cost efficiency, fully integrated in one of the most prominent big data processing frameworks.

2

BACKGROUND

This chapter gives to the reader all the background required for understanding this thesis. Specifically, we first present the serverless computing paradigm (Section 2.1). We then introduce the Kubernetes container orchestration framework this thesis is based upon, along with its different layers introducing event-driven and serverless capabilities, namely Knative and Tekton (Section 2.2). We present Apache Spark data processing framework, describing its architecture, resource allocation policies, and scheduling mechanisms (Section 2.3). Then, we briefly introduce the optimization and learning methods used in this thesis (Section 2.4). Finally, we describe the data analytics applications have been used to evaluate the thesis work (Section 2.5).

2.1 SERVERLESS COMPUTING

The serverless paradigm, also known as FaaS, takes advantage of hardware disaggregation, using the data center as a pool of independent resources connected through high-speed networks, allowing millisecond latency between computing and storage resources. Disaggregation provides enhanced flexibility and adaptability within the data center infrastructure. In a disaggregated configuration, when a new application needs to be executed, the data center dynamically pools the necessary resources into an available node, executes the application, and upon application completion, the resources are freed, making them available for a new allocation to another node. This approach solves the lack of resources on individual nodes, as all nodes can access all the resources distributed across the entirety of the data center. Breaking the traditional tight coupling between a process and the underlying physical resources allows abstract serverless functions to span the data center transparently and scale. Since functions span across the data center, there is no guarantee that functions will be executed on the same physical machine. Whenever functions are placed on different physical nodes, fully-managed disaggregated storage (e.g., Amazon S3 [52] and IBM COS [53]) is used to share the function state and to store data persistently. This naturally leads to a "*Move data to compute*" model, where data and state have to be delivered to functions.

A *serverless function* is a small, discrete unit of code with short duration, typically minutes, having entirely stateless logic. In this computing paradigm, functions are executed in response to triggers, represented by events or HTTP requests. Serverless workflows are represented as Directed Acyclic Graphs (DAGs) in which nodes represent functions and edges represent input/output dependencies between pairs of nodes, determining the order of execution and the extent to which functions can be executed in parallel. The associated ephemeral compute environment is created only when an event, triggering the function, is received. After the function ends, the environment is left active for a short period of time in case another request arrives.

Generally, serverless computing can be described by three main principles: 1) No server to provision or manage, 2) Scales with usage, and 3) Pay for value. First, all the infrastructure management responsibilities are on the cloud provider, allowing the user to focus solely on designing, running, and managing the application functionalities with zero concerns about building and maintaining the back-end infrastructure. Second, the service scales automatically based on the number of received requests. More precisely, it can scale up when needed and scale all the way down to zero resources in the absence of demand, thus avoiding waste of resources. Finally, the user pays in proportion to resources used instead of resources allocated, implying a zero cost in case of no service requests.

'Serverless computing' does not mean servers are not used; instead, it means that users leave all the operational concerns of managing servers entirely to the cloud provider. This computing paradigm is attractive not only for consumers for the properties mentioned above but also for cloud providers. It promotes business growth since making the cloud easier to use draw in new consumers. Indeed, [54] has found that about 24% of serverless users were new to cloud computing. Moreover, it allows cloud providers to control the entire development stack and efficiently optimize and manage resource. For example, the cloud provider may decide to utilize previous generation machines, as the instance type is part of the infrastructure management responsibility.

Serverless computing and Platform as a Service (PaaS) backend architecture keep the entire backend invisible to users, allowing them to concentrate fully on writing and building applications. However, significant differences exist between these two architectures, particularly regarding scalability and pricing. Serverless architectures exhibit instant, automatic, and on-demand scalability without requiring extra user-provided configuration. In contrast, while users can configure PaaS-hosted applications to scale up and down based on demand, auto-scaling is not an inherent feature of this service. Serverless vendors employ a fine-grain, pay-as-you-go billing approach, charging users solely for the actual time the functions are running. Even though some PaaS vendors charge users for application uptime, billing is not as precise as it is for serverless. Typically, users are usually able to decide the computing

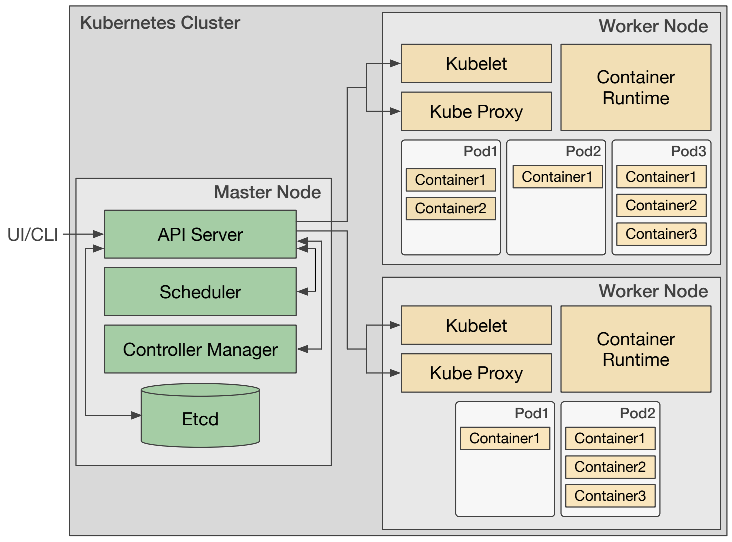


Figure 2.1: Kubernetes cluster architecture example.

power they are paying only in advance, without the possibility of increasing or decreasing the usage in real-time.

2.2 KUBERNETES ECOSYSTEM

While there are several available container orchestration frameworks, Kubernetes [55] has become the leading platform and *de-facto* cross-cloud standard for automatic management of containerized applications. Originally created by Google and then donated to the Cloud Native Computing Foundation (CNCF) in 2016, it provides a standardized and efficient way to deploy, manage, and scale containerized applications across diverse infrastructure environments. A Kubernetes cluster consists of a set of nodes, each one map on either a physical machine or a Virtual Machine (VM) and running a container runtime, like Docker [56] or containerd [57]. Figure 2.1 shows an example of kubernetes cluster with one master node and two worker nodes, the first running three pods and the second running two pods. The cluster is composed by at least one *master* node and multiple *worker* nodes. A master node is responsible for managing and coordinating the overall cluster. It includes several components: 1) the API server, 2) the scheduler, 3) the controller manager, and 4) the etcd datastore. The API server acts as the entry point for the Kubernetes control plane and is responsible for processing RESTful API requests. The scheduler assigns work (containers) to nodes based on resource availability and constraints. The controller manager maintains the desired state of the cluster by monitoring nodes and responding to changes. Etcd is a distributed key-value store that stores the configuration data of the entire cluster. Differently, worker nodes run applications in containers and each worker node has the following components: 1) the kubelet, 2) the container runtime, and 3) the kube-proxy. The

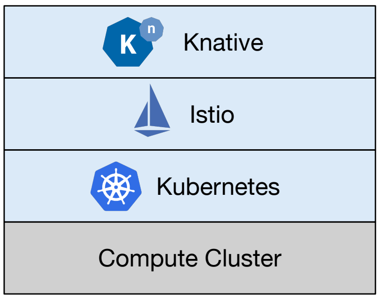


Figure 2.2: Kubernetes ecosystem enabling event-driven and serverless executions.

kubelet is the agent responsible for communication between the worker and the master(s) . Moreover, it ensures that one or more containers are running and healthy in a *pod*, the smallest deployable unit in Kubernetes, representing a single instance of a running process. Containers in the same pod share the same network namespace and storage, allowing them to communicate and share data easily. The container runtime is responsible for loading container images from a repository, monitoring local system resources, isolating system resources for use of a container, and managing container lifecycle within the Kubernetes environment. The kube-proxy communicate with the API server and maintains network rules to enable communication between Pods across the cluster.

Knative [9], recently accepted as a CNCF incubating project, introduces event-driven and serverless capabilities for Kubernetes clusters. It provides a set of building blocks for building, deploying, and managing modern serverless workloads. More specifically, it abstracts the complexity of dealing with the underlying infrastructure by managing stateless services by reducing the users effort required for autoscaling, networking, and rollouts, and it routes events between on-cluster and off-cluster components, allowing users to fully focus on their core logic. Representing the serverless standard, as shown in Figure 2.2, Knative has been built on top of Kubernetes thanks to the introduction of Istio service mesh [58], which simplifies the deployment of microservices by providing a language-agnostic way to connect, secure, manage, and monitor microservices.

Knative communication infrastructure follows the CloudEvents [59] specification and is based on a switchboard model connecting events to interested services. Knative offers two communication patterns: a more basic Channels and Subscriptions pattern, and a more advanced Broker and Trigger pattern. The first decouples the event producer and consumer, allowing the producer to send an event to a specific Channel which than forwards the event to all consumers that are subscribed to it. Differently, in the Broker and Trigger based pattern, events are sent to Brokers, which represent the event delivery system controlling accesses and ingress policies and storing events in Channels, and then the Trigger filters and delivers the events to the consumers. The Channel messaging layer, used in both communication patterns, is optimized for small messages of about 1KB in size, even though up to some MB messages are allowed by modifying some configuration options. Knative offers a default channel mechanism aiming to ease the usability of the system, but it also supports other channel types, such as Apache Kafka [60] and Google Cloud Pub/Sub [61].

Knative is composed by two key components: Serving and Eventing. The Serving component is responsible for deploying and serving containerized applications, called services, on Kubernetes. It enables autoscaling, handles traffic splitting, and supports revisions of your applications. The Eventing component allows the user to build event-driven architectures by providing a set of primitives. Currently, this component provides a sequential and a parallel functions invocation. The former defines an in-order list of functions that will be invoked at each event, while the latter defines a list of branches, each one receiving the same event.

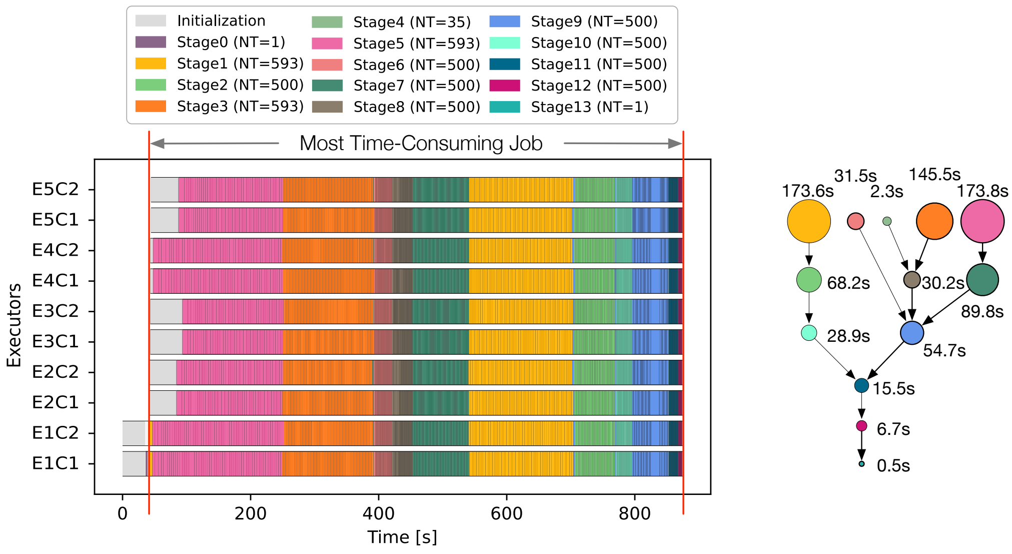
Knative currently lacks a more complete pipeline structure implementation. To fill this gap, Tekton [62] has been proposed to allow users to design and run event-driven workflows in an automated way. Even though Tekton has been originally developed for Continuous Integration and Continuous Delivery (CI/CD) building chains, it has been demonstrated its value also in other domains [63]. Tekton *pipeline* object is represented by a set of tasks. Each *task* is a single unit of work and is composed of a certain number of sequential steps. A *step* is the smallest unit of work in a task and performs a specific actions, such as building a container, running tests, or deploying an application. Steps exchange data either through parameters exploiting the underlying *channel* messaging layer or through Kubernetes volumes. Each task is deployed as a single pod, being a Pod the smallest deployable units of computing. Tekton *workspaces* automatically mount volumes within the tasks containers and only supports two types of volumes, namely Persistent Volumes (PVs) and emptyDir volumes. The first is backed persistently on one of the cluster nodes, representing a good choice for intra-tasks communications, while the second holds a temporary storage space that only lives for the task execution time, making it more suitable for intra-steps communication. By default, the Tekton pipeline controller, namely the Affinity Assistant, enforces the co-placing of all the tasks sharing a workspace. Data dependencies and user predefined execution orders are used to define the pipeline workflow in the form of a DAG.

2.3 APACHE SPARK

Apache Spark [47] represents one of the most prominent multi-purpose, multi-language, in-memory big data processing frameworks. It is designed to process and analyze large-scale datasets and supports various data processing workloads, like batch processing, real-time streaming, machine learning, and graph processing. To manage and coordinate distributed processing, Spark follows a *Master-Worker* model: while the master coordinates the cluster, the workers perform the actual data processing. Spark distributes data across a cluster of machines relying on its fundamental Resiliant Distributed Dataset (RDD) data structure. An *RDD* represents an immutable, distributed collection of objects that can be processed in parallel. Each RDD is divided into logical partitions across the cluster and thus can be operated in parallel, on different nodes of the cluster.

When a Spark parallel data processing application is submitted, the resource manager starts the *driver*, running the `main()` method of the application and generating both the logical and physical execution plans. More precisely, Spark converts the application into one or more *jobs* based on the presence of actions in the code. At a high level, each job is represented by a *logical execution plan* abstracting all the transformation steps, encoded as a DAG. A DAG consists of a set of 1) nodes representing computational *stages*, delimited by operations requiring data shuffling, comprising by one or more parallel *tasks* that apply the same operation over a different data partition, and 2) edges representing input/output dependencies between pairs of nodes, determining the order of execution and the extent to which stages can be executed in parallel. Many Spark applications often consist of tens or hundreds of stages characterized by a different number of parallel tasks with high volatility in duration and amount of data that they process. From the logical execution plan, Spark driver derives the the *physical plan*, a plan describing how the logical plan is going to be executed on the cluster. The driver directly coordinates with the cluster manager to request executors, representing computing processes, and schedule the execution of one or multiple tasks on each executor. The cluster manager usually spawns the executors within containerized Java Virtual Machines (JVMs) on the worker nodes. During the initialization phase, known as *cold start*, each executor's setup involves the following steps: 1) Downloading the container image if not present on the designated node, 2) Launching a new container, and then 3) Performing all the JVM-related operations, such as bytecode loading, heap allocation, and internal threads creation. The first task is launched as a thread once the JVM is ready.

The driver includes several components, including the *DAG Scheduler*, the *Task Scheduler*, and the *Backend Scheduler*, all responsible for translating user-written code into jobs that are executed on the cluster. The default allocation strategy is to assign all available executors to stages in a First In First Out (FIFO) manner: given a queue of runnable stages, the first stage gets priority on all available resources, then the second stage gets priority, and so on. The number of executors held by each stage depends on the number of stage's tasks. If a stage has fewer tasks than available executors, the scheduler allocates part of the available executors, allowing multiple independent stages to be executed concurrently. Contrarily, if a stage has more tasks than there are executors, all executors are assigned to the stage, and tasks are executed in multiple "waves". The Spark scheduler faces two main scheduling decisions: 1) Deciding which stage(s) to run next from a set of runnable stages and 2) Deciding how many executors to assign to each runnable stage. Then, upon completion of a stage, the dependent child stages becomes runnable and are considered for scheduling. With



(a) (b)

Figure 2.3: Application (a) sample execution schedule with dynamic allocation over a maximum of 5 executors, featuring two vCPUs each with a one-to-one core-task mapping (cold start are depicted in gray and the number of tasks are reported in parentheses), and (b) most time-consuming job DAG reporting the average job's stages duration on 5 sequential runs with different circle sizes.

default allocation, during the course of application execution, the framework requests in real-time exponentially more executors in case of pending tasks waiting to be scheduled. If the queue of schedulable tasks is not drained in one second (default backlog timeout), then new executors are requested. If the queue persists for another second, then more new executors are requested, and so on. At each round the number of requested executors increases exponentially from the previous round until the upper bound, defined by the autoscaling property, is reached. The rationale for the exponential increase is twofold. First, executors should be initially added slowly in case the number of extra executors needed is small. Otherwise, more executors than the necessary ones could be requested, possibly leading to some executors not performing any work. Second, executors should be added quickly over time in case the maximum number of executors is very high. Otherwise, ramping up under heavy workloads will take a long time. While this exponential increase can facilitate the completion of an unexpected number of tasks, a notable latency exists in responding to additional executor requests, often requiring multiple requests before attaining the intended executor allocation. This circumstance raises the risk of either belated allocation or an exponential surplus beyond the required resources. Additionally, exponentially requesting new executors implies an increasing number of parallel application image downloads, which can lead to a lower image transfer speed due to increased network load.

An example of application execution schedule, with the default dynamic allocation scheduling stages in FIFO fashion, is illustrated in Figure 2.3a. Initially, the scheduler assigns 1 executor (minimum set) and then scales out to 5 executors (maximum set) when tasks become backlogged. Among the executors, while the remaining executors experience an actual cold start, E4 executor experiences a warm start since the application image is cached because the application driver has been previously placed on the same node. The application example comprises a small single-stage single-task job at the beginning, followed by a big time-consuming job (highlighted by vertical red lines in Figure 2.3a). As shown in Figure 2.3b, this most time-consuming job features a complex DAG composed of 13 stages with a variable number of tasks and duration.

2.4 OPTIMIZATION AND LEARNING METHODS

In this section we present the optimization and Machine Learning (ML) methods used in this thesis, helping the reader to understand the last thesis contribution (Chapter 5). We consider the hill climbing search method and four ML methods, namely Linear Regression (LR), Bayesian Ridge Regression (BRR), Boosted Decision Stumps (BDS), and Dropouts Additive Regression Tree (DART) [64], to embrace historical knowledge to improve the heuristic search. Since the last contribution aims to provide real-time responses, we use the aforementioned four methods since they provide good tradeoff between response time and predictive performance.

2.4.1 *Hill Climbing*

The hill climbing algorithm is a widely used local search optimization method. The idea behind the method is to start at an initial neighbour and to attempt to improve the current solution (climbing the hill) by iteratively moving to neighbours that enhance the current solution until an optima is reached. The initial neighbour can be generated randomly or chosen based on some heuristic. Once a local optima is reached, no further improvement can be made with the current set of moves. The selection of the initial neighbour affects both the solution quality and the number of search steps needed to find the local optima solution. Hill climbing only guarantees optimal solutions for convex problems. In contrast, only local optima is guaranteed for the rest of the problems. Even though, in the context of this thesis, hill climbing only guarantees local optima solutions, it is a simple, effective, and intuitive algorithm that is easy to understand and represents a good choice for problems where a sufficiently good solution is needed quickly.

2.4.2 *Linear Regression*

Linear regression is a fundamental statistical method used for modeling the relationship between quantitative variables. It identifies a linear relationship between one or more input variables (observed values) and the output variable (predicted value). The graphical intuition of linear regression is to find the best-fitting straight line through the known points that minimize the sum of the squared difference, called *residuals of error*, between the observed and predicted values. Thus, the model equation can be formalized as follows:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k \tag{2.1}$$

Here, y represents the output variable the model tries to predict, $\{x_1, \cdots, x_k\}$ are the input variables the model observes, β_0 is y-axis intercept, and $\{\beta_1, \cdots, \beta_k\}$ coefficients are known as the *regression coefficients* and they represent the contribution of each input variable to the output value y. In the case of a single input variable, Equation (2.2) simplifies to the following linear equation:

$$y = \beta_0 + \beta_1 x_1 \tag{2.2}$$

Linear regression is one of the simplest and most widely used techniques in predictive analysis in both ML and statistics.

2.4.3 *Bayesian Ridge Regression*

In traditional linear regression, the primary goal is to identify a set of parameters that minimize the sum of squared differences between the predicted and actual values. Thus, this method inevitably inherits the uncertainties in parameters estimation, potentially arising from data errors. BRR model addresses this inherent instability in the estimated coefficients and associated uncertainty in the model - commonly referred to as the *multicollinearity problem* - by incorporating a Bayesian approach. This model characterizes the distribution of the regression coefficients, typically opting for a Gaussian (normal) distribution, in contrast to the linear relationship between them assumed in classical linear regression. Consequently, the BRR model provides a more flexible and robust solution, particularly useful in scenarios with limited or noisy data.

2.4.4 *Boosted Decision Stumps*

Gradient Boosting (GB) [65] regression is an ensemble learning technique where multiple weak learners are combined to construct a strong learner. During the training phase, weak learners are added in an iterative process wherein the ensemble error guides the subsequent models' creation. In other words, further iterations of the ensemble try to correct previous errors to produce more accurate estimates. This process iterates until a maximum number of iterations (models) is reached. Differently from GB, which typically employs decision trees as the weak learners, the BDS model utilizes decision stumps [66]. A *decision stump* is a simplified form of a decision tree, constrained to a single decision node and two leaf nodes containing the output values. Each decision stump tests a single input feature against a threshold, deciding the output value based on the test result. The training of each weak learner occurs sequentially, with increasing importance on misclassified instances from the previous iteration. Each decision stump is associated with a weight throughout the training process, giving more influence to decision stumps that perform well (with higher accuracy) on the training data. The boosting process continues for a specified number of iterations or until a certain level of accuracy is achieved. The final model prediction is determined through a weighted voting scheme, where each decision stump contributes to the final prediction in proportion to its assigned weight.

2.4.5 *Dropouts Additive Regression Tree*

The DART model is an extension of GB model, designed to address the problem of over-specialization. This problem arises when trees added at later iterations tend to impact the prediction of only a few instances, thereby making a negligible contribution to the final prediction of all the remaining instances. This makes the ensemble very sensitive to the decision made by the few, initially added trees. To deal with the over-specialization problem, the DART model adapts the dropout technique [67], originally proposed in the context of learning deep neural networks to mute random neurons to prevent overfitting, to an ensemble of decision trees. Specifically, at training time, DART introduces randomness by muting a random subset of weak learners, dropping out at least one complete tree when creating the model. This randomness results in a more robust ensemble, effectively mitigating errors introduced by previous trees.

2.5 APPLICATIONS

To evaluate the thesis a diverse set of applications ranging from real-wold workloads to standard benchmarks that are representative for data analytics frameworks have been used.

2.5.1 *Tesseract*

The Tesseract Optical Character Recognition (OCR) engine [68] is a highly accurate open-source OCR solution developed by Google. Originally created by Hewlett-Packard in the 1980s and later open-sourced in 2005, Tesseract has emerged as one of the most popular and widely used OCR engines globally, representing the current open-source state-of-the-art. It is designed to recognize printed text in images and convert it into a machine-readable format. Input images include scanned documents, photographs, and screenshots including text even with complex fonts and layouts. Document digitalization, data extraction, archiving and document management, and automated text recognition are examples of Tesseract use cases. With support for a multitude of languages — over 100 and expandable through language-specific training data — Tesseract proves to be a versatile tool for text recognition across diverse linguistic contexts.

2.5.2 *TPC-H*

The TPC-H [69] is an industry-standard decision support benchmark, featuring a set of business oriented ad-hoc queries and concurrent data modifications. The 22 queries and the data populating the database aim for broad industry-wide relevance, showcasing decision support systems, such as data warehouses, database, and Big Data systems, handling large volumes of data volumes to address to critical business questions. The benchmark randomly generates eight relational tables with a schema reflecting a typical data warehouse involving sales, customers, and suppliers. The benchmark employs relatively straightforward queries, suitable for sharding, avoiding tables pre-join and aggregation, utilizing single-column indexes. Noteworthy, TPC-H has shortcomings include the linear scaling of non-fact tables, homogeneous data distribution, use of Third Normal Form (3NF) rather than a star schema (common in this type of analytical workloads [70]), and the non-comparability of its handcrafted queries with the auto-generated queries of actual workloads [71]. Despite these limitations, the TPC-H benchmark remains widely cited, with hundreds of annual reference in the literature.

2.5.3 *TPC-DS*

The TPC-DS [72] is another industry-standard decision support benchmark, involving much complex queries using much wider tables when compared to previous TPC decision support benchmarks, such as TPC-H. More precisely, its tables are composed of up to 39 columns with domains ranging from integer, float (with various precisions), char, varchar (of various lengths) and date. Combined with a large number of tables (total of 25 tables and 429 columns) the schema gives both the opportunity of a realistic and challenging query set as well as an opportunity for innovative data placement algorithms and other schema optimizations, such as complex auxiliary data structures. Furthermore, the introduction of NULL values into any column except the primary keys opens yet another dimension of complexity compared to TPC-H benchmarks. TPC-DS generalized query model utilizes 99 query templates testing the interactive and iterative nature of On-Line Analytical Processing (OLAP) queries, the extraction of queries for data mining tools, ad-hoc queries, and the more planned behavior of well known reporting queries. The benchmark queries feature advanced SQL features, extensive filter predicates and functions, and diverse data scans.

2.5.4 *Terasort*

Terasort [73] is a widely recognized benchmark used to assess the performance of sorting large volumes of data in Big Data processing frameworks, especially within distributed computing settings. It was introduced as part of the Apache Hadoop project and has since become a standard benchmark for evaluating the efficiency and scalability of systems handling massive datasets. Terasort is specifically designed for execution on distributed computing frameworks like Apache Spark. The sorting of substantial datasets is a foundational task in data processing. In Big Data scenarios, efficient sorting massive datasets is crucial for a range of analytical, data mining, and processing tasks. Terasort operates on semi-structured data, consisting of key-value pairs, where keys are structured as strings while values are often unstructured or irrelevant to the sorting operation. An example of use case is a large-scale e-commerce company generating massive amounts of log data daily, which needs to sort this data based on timestamps to gain insights into customer behavior.

3

WORKLOAD SCALABILITY AND PERFORMANCE ANALYSIS

This chapter describes the first contribution of this thesis. This contribution analyzes the data exchange mechanisms between functions in current open-source serverless platforms, considering function levels of parallelism, and proposes a performance model capturing the behavior of serverless workloads (Section 3.2). Furthermore, we present the use case and validation workload, along with its characterization, showing a methodology to extract the necessary components to translate a traditional monolithic application to a serverless application (Section 3.3). Then, we perform a bandwidth characterization of the different available communication mechanisms and evaluate the accuracy of the proposed performance model (Section 3.4). Finally, thanks to the flexibility of the proposed model, we investigate the performance impact on the use case workload of the following key factors: task granularity and concurrency, data locality, resource allocation, and scheduling policies (Section 3.5).

3.1 INTRODUCTION

The stateless nature of serverless computing initially limited its applicability to a subset of workloads, primarily encompassing simple real-world applications, e.g., web microservices and IoT applications. Nevertheless, the intrinsic resource-intensive and inherently parallel characteristics of data-intensive analytics applications suggest that this type of applications could highly benefit from the elasticity and the virtually unlimited scalability offered by serverless platforms. However, while the desirability and advantages of enabling more complex workloads in serverless environments are known, its feasibility and efficiency still remain unclear. There is a need to fully understand 1) data exchange mechanisms between functions in current serverless platforms, and whether new approaches are needed, and 2) data flows of these workloads, considering their levels of parallelism and resource requirements. Furthermore, being able to evaluate the data sharing impact on the workload execution and to choose the best storage configuration ahead of the actual porting, would save time and increase the overall workload performance. Aiming to understand the possible data exchange solutions and the impact of complex data flows in serverless environments, the main contributions of this work are:

– A performance model capturing the performance characteristics of serverless workloads with data exchanges, supporting different configurations and models of exchanging data.

– A complete characterization of a data-intensive workload, showing a methodology to extract the necessary components to translate a traditional application to serverless.

– A comprehensive evaluation of data-intensive workloads in serverless environments, exploring parallelism, data exchange, resource allocation, and scheduling policies.

3.2 PERFORMANCE MODEL

The goal of the proposed model is to estimate the end-to-end performance of a non-trivial workload with data dependencies between the pipeline tasks, as if it was running in a data center on top of a Knative-based serverless environment. Since there is a significant emphasis on data, the model allows for different exchange mechanisms, including local memory, local storage, and remote storage. While some of these may not be supported or are limited in current serverless implementations, they can still be used to evaluate alternative solutions. The inputs of the model are: 1) hardware platform description, 2) scheduling strategy, 3) workload input, 4) workload graph, and optionally 5) bandwidth characterization. The hardware platform description (1) input provides the necessary information about the cluster machines' memory and storage. For memory, it specifies the speed [MegaTransfers/s] and the data width [bits] necessary to compute the theoretical memory bandwidth (i.e., $speed \times data\_width/8$) [Mbps], while for storage, the maximum transfer rate [Mbps] is reported. We have retrieved this information from the datasheets or, in the case of memory, inspecting the underlying system from the CLI (e.g., Linux dmidecode command). The scheduling strategy (2) input specifies the task-node assignments. The workload (3) input defines the input dimension, the total execution time, and the amount of data it contains. The workload graph (4) input specifies for each task its name, predecessor tasks names, inputs/outputs object names, compute time or profiling real percentage, parallelisms object with their possible degree of parallelism. With *degree of parallelism* we denote the number of concurrent task executions for a specific parallelism. Finally, the optional bandwidth characterization (5) input provides, for a set of data dimensions, the median read/write bandwidths for the three accounted data exchange mechanisms. If this last input is provided, the model returns a more realistic estimation; otherwise, it provides a best-case prediction by using the theoretical bandwidths. Given a pipeline composed of n tasks, each i-th task total time is composed of: the time necessary to read the input data T_i^r, the compute time T_i^c, and the time necessary to write the output data T_i^w. Given the i-th task exploits m levels of parallelism, for each j-th level of parallelism, we denote with $x_{i,j}$ the number of sequential computations in case of no task replication and with $p_{i,j}$ the degree of parallelism. Therefore, $\frac{x_{i,j}}{p_{i,j}}$ represents the actual number of computations the task has to perform. In case of no concurrent execution, the model considers $p_{i,j} = 1$. Thus, we derive the end-to-end pipeline time T_{tot} as:

$$T_{tot} = \sum_{i=1}^{n} \left[\left(\prod_{j=1}^{m} \left\lceil \frac{x_{i,j}}{p_{i,j}} \right\rceil \right) (T_i^r + T_i^c + T_i^w) \right] \tag{3.1}$$

where:

$$T_i^r = \sum_{k=1}^{3} \left(\sum_{s=0}^{ni} \frac{D_s}{BW_k^r(D_s)} \right) \tag{3.2}$$

$$T_i^w = \sum_{k=1}^{3} \left(\sum_{u=0}^{no} \frac{D_u}{BW_k^w(D_u)} \right) \tag{3.3}$$

For example, in a text recognition workload, given a 6 pages input ($x_{i,1} = 6$) with an average of 40 text lines ($x_{i,2} = 40$) each, if the i-th task has to perform its computation for each input page text line, it inherits m=2 levels of parallelism. With a sequential execution, the i-th task total number of actual computations amounts to $(\frac{6}{1} \times \frac{40}{1}) = 240$, since $p_{i,1} = p_{i,2} = 1$. Differently, with a parallel execution computing 2 pages ($p_{i,1} = 2$) and 5 text lines ($p_{i,2} = 5$) concurrently, implying $2 \times 5 = 10$ i-th Task replicas, the model only considers the total number of actual computations of a single parallel replicas, i.e., $(\frac{6}{2} \times \frac{40}{5}) = 24$.

As shown in Equation (3.2) and Equation (3.3), to account for the three data exchange mechanisms, the model computes the read time T_i^r and the write time T_i^w as the sum of the ni inputs and no outputs partial communication latencies, computed by dividing the amount of data to be read/written D by the specific mechanism bandwidth BW. Bandwidths BW are functions of the amount of data D. If the bandwidth characterization input is given and the target data dimension D is not present in the set of benchmarked dimensions, the bandwidth BW is estimated by applying the linear interpolation method between the two data dimension points defining the interval containing the target data dimension, as described in Equation (3.4). More specifically, the method approximates the output bandwidth $\hat{BW}$ by generating a straight line between the bottom point (D_b, BW_b) and the top point (D_t, BW_t), with $D_b < D < D_t$, and taking the corresponding bandwidth value:

$$\hat{BW}(D) = BW_b + (D - D_b)\frac{BW_t - BW_b}{D_t - D_b} \tag{3.4}$$

3.3 USE CASE AND VALIDATION WORKLOAD

To validate the proposed performance model we characterize and study Google's Tesseract OCR engine, representing the current open-source state-of-the-art. It is an interesting use case study because it represents a more traditional workload with non-trivial computational and data graphs, not currently well supported in serverless environments. Tesseract features smaller data transfers compared to more I/O-intensive workloads; however, it involves a significant amount of data at scale. For example, financial institutions processing large amounts of documents.

3.3.1 *Description of Tesseract*

Given an input image, Tesseract requires converting the image to the Tagged Image File Format (TIFF) file format. The OCR engine output consists of a simple text file containing only the image text content. The computation follows a traditional step-by-step pipeline in which, at a high level, three different phases can be identified, namely pre-processing, processing and post-processing. The pre-processing phase involves image manipulation steps, allowing to extract text lines and words features from the input components. The processing phase exploits Long Short Term Memory (LSTM)-based Neural Network (NN) classifier to recognize the input words, whose architecture is illustrated in Figure 3.1. Given a text line as model input, the NN output matrix reports the specific language characters probabilities for each time-step, being a *time-step* a slice of the model input. The NN output matrix is then decoded into text by using the Beam Search algorithm. The processing phase proceeds as a two-pass process. In the first pass, each word is predicted through the classifier, and, in the case of satisfactory recognition, it is used by the classifier as training data. This allows the classifier to be more accurate on the next recognition. A second pass is run over the words that were not recognized well enough in the first pass. Our tests on the English language show that the number of words accessing this second pass is negligible. The final post-processing phase identifies paragraphs based on page margins and text indentation. Tesseract offers a multi-threaded execution with up to four threads used, while processing a page, to perform the NN computations based on Single Instruction Multiple Data (SIMD) instructions.

3.3.2 *Methodology*

Transforming a traditional application like Tesseract into a serverless workload requires understanding its internal computational graph, and splitting the application into smaller

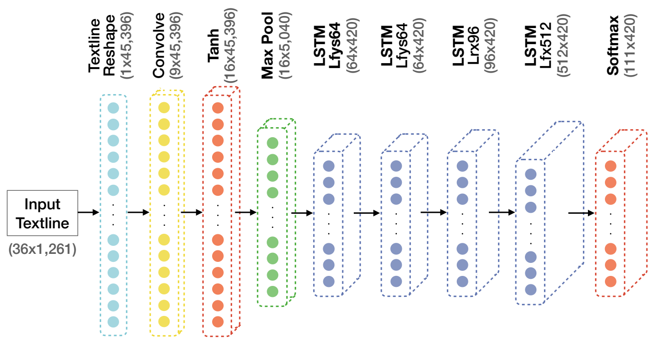


Figure 3.1: Tesseract LSTM-based NN architecture. The model takes in input a text line and returns all the associated time-steps language probabilities. The layer type and the related matrix size is reported above each layer.

computational units. Obtaining the computational graph involves profiling the application to extract a detailed *call-graph* representing the workload call chains using tools like the Valgrind profiler along with Callgrind [74]. The call-graph gives a clear view of the Tesseract computational paths, and it allows to identify the most time-consuming functions as well as possible levels of parallelism.

Since the code base of applications like Tesseract can be very large, it would not be feasible to map every application function into an independent serverless function. Hence we simplify the call-graph by grouping functions into computational units called *macro-nodes*. These macro-nodes can be further evaluated by instrumenting the application code, and using memory-profiling tools like Valgrind and Massif [74] to gather detailed information of the workload's footprint and the heap and stack memory allocations of each function. Even though Massif gives an in-depth view of the application's memory allocations, such tool do not provide any information on the correlation between memory allocations and variables/objects to which the memory belongs. To extract this correlation we build a set of custom tools to trace the average dimension of each macro-node's inputs and outputs. Finally, to highlight data exchanges, we create a *dataflow* graph where nodes represent macro-nodes and arcs represent data channels. Macro-nodes with small cost-times and trivial data movements are merged with neighboring macro-nodes.

3.3.3 *Tesseract Characterization*

To characterize Tesseract, we evaluate the execution time and the data dimensions of two single-page documents, *DataSetA* and *DataSetB*, that present different page structure and number of lines. These documents have features similar to financial documentation, with two columns, figures, tables, and text with different font sizes. To evaluate the scalability, datasets are replicated generating increasingly bigger inputs, from 1 to 128 pages with a power of two increment. For every dataset and input page combination, tests are repeated

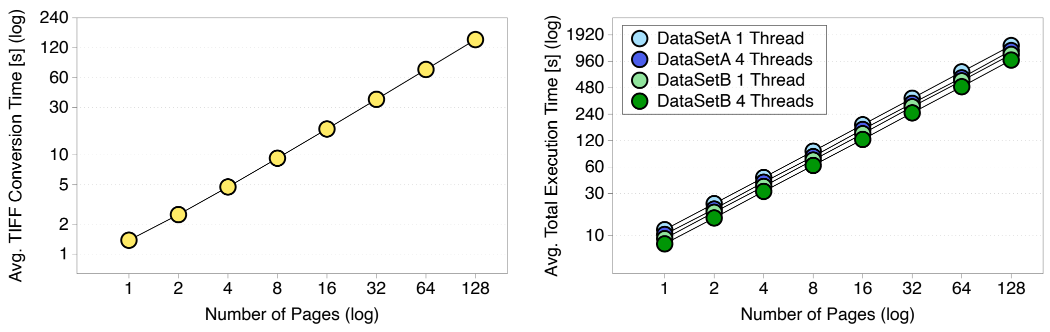
Table 3.1: Tesseract simplified call-graph

Macro-node	**Percentage [%]**		**Parallelism**	**Granularity**	
	Inclusive	**Real**		**CG**	**FG**
main	99.97	0.28	-	1	1
ProcessMultipageTIFF	99.69	1.56	-	1	1
Recognize	98.13	0.60	Page	2	2
FindLines	12.54	0.01	Page	2	2
ExtractThresholds	2.33	2.33	Page	2	2
SegmentPage	10.20	0.01	Page	2	3
AutoSegmentation	8.13	0.01	Page	2	3
SetupSegCorOrient	4.18	4.18	Page	2	3
FindBlocks	3.94	3.94	Page	2	4
MakeTextlinesWords	2.06	2.06	Page, Block	2	5
RecogAllWords	83.82	0.20	Page	3	6
LSTMRecogWords	83.62	0.01	Page, Textline	3	7
GetLineImage	2.60	2.60	Page, Textline	3	7
RecognizeLine	81.01	0.04	Page, Textline	3	8
LSTM-NN	79.65	79.65	Page, Textline	3	8
DecodeNNOutput	1.32	1.32	Page, Textline	3	9
DetectParagraphs	1.17	1.17	Page	4	10

ten times in single and multi-threaded configurations exploiting up to four CPU threads[1]. All the experiments are executed on an Intel Xeon Silver 4114 CPU running at 2.20GHz.

Table 3.1 presents the simplified Tesseract call-graph, where the macro-node real percentage is computed by subtracting the children inclusive percentages from its inclusive value. We emphasize that when a macro-node computation can be parallelized, all the macro-nodes belonging to the sub-tree having the given macro-node as root are parallelized as well. The three main Tesseract phases, namely pre-processing, processing and post-processing, are identified by the *FindLines*, *RecogAllWords* and *DetectParagraph* macro-nodes, taking 12.54%, 83.82%, and 1.17% over the total execution time, respectively. The processing phase is the most time-consuming, mainly due to the LSTM-based NN computation featuring an inclusive percentage of 79.65%. The workload is characterized by three levels of parallelism: at page, block, and textline-level. More precisely, the initial *Recognize* macro-node computation can be fully parallelized on the number of pages. Moreover, each concurrent page computation can be further parallelized in its pre-processing phase on the number of page blocks when searching for text lines and words in the macro-node *MakeTextlinesWords*, as well as in the processing phase on the number of words when computing the sub-tree with the *LSTM-RecognizeWords* macro-node as a root. This second parallelization opportunity is the most relevant since it embraces the NN computation. The last outlined level of parallelism, related to the NN output decoding, can not be exploited because there is a strong dependency between subsequent Beam Search algorithm time-step computations.

1 Tesseract offers a multi-threaded execution with up to four threads used, while processing a page, to perform the NN computations based on SIMD instructions.



(a) TIFF Conversion

(b) Text recognition

Figure 3.2: Scalability analysis of the (a) initial TIFF conversion phase and (b) text recognition phase.

Considering the initial TIFF conversion, we find that the amount of time necessary to convert the input scales linearly with the number of input pages and is strongly dependent on the page dimension, as shown in Figure 3.2a. In particular, a single PDF page in A4 format takes 1.19s to be converted. Similarly, as shown in Figure 3.2b, both single- and multi-threaded workload execution times scales linearly with the number of input pages. Figure 3.3 shows the average per page execution times of the different configurations. DataSetA execution times are approximately 2s slower than DataSetB ones. The multi-threaded implementation of both inputs datasets only achieves a 15% of performance improvement over the corresponding single-threaded implementation. This small performance gain is due to the way in which Tesseract benefits from multi-threading: differently from what was expected, i.e., computing one page per thread, multiple threads are used to perform the NN computation based on SIMD instructions.

Last, we analyze the workload memory footprint and we highlight the input and output dimensions of the initial conversion step and of each workload macro-node. The memory footprint is up to 150MiB, with a short peak of 166.9MiB, in the first two-thirds of the workload execution and up to 90MiB in the last one-third. Massif reveals a new macro-node representing the Tesseract initialization step, loading all the necessary data to perform the computation, such as the NN weights and biases. For the initial conversion step, a page of roughly 40KB is converted into a 8.7MB image, and grows linearly with the number of pages, so a 128 pages input takes 1.09GB. Figure 3.4 shows the dataflow graph, reporting the input and output data dimensions for each macro-node. By analyzing the dataflow graph, we observe that data movements are in the order of tens of . While this seems relatively small, data transfers could become prohibitive when exploiting the per-page, per-block, and textline-level of parallelism, highlighted with p_1, p_2, and p_3. Indeed, considering the coarse-grained scenario, computing n pages in parallel would increase by n times the number of read and write requests of the involved macro-nodes. The number of requests becomes even

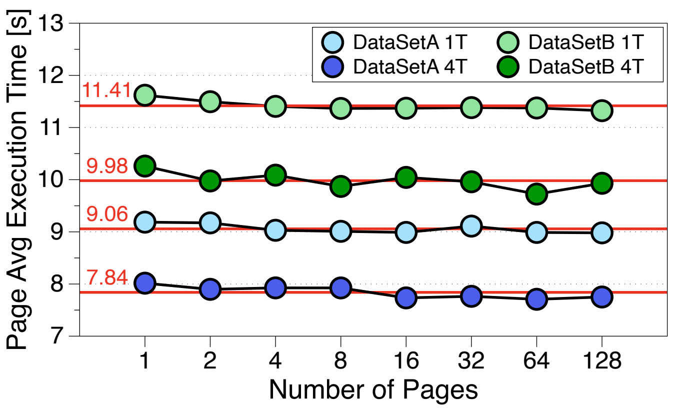


Figure 3.3: Tesseract per-page average execution times exploiting one and four threads (1T and 4T). The horizontal red lines identify the averages over all the number of pages.

higher when making use of the other levels of parallelism in the fine-grained scenario. In the previous example, computing m lines of text in parallel would imply $n \times m$ requests.

To evaluate Tesseract in serverless environments, in this work we provide two scenarios with different task *granularity*, which defines at what level code functions are grouped to become serverless tasks. Accounting with the initial TIFF conversion step, the Coarse-Grained (CG) scenario consists of *bigger* groups for a total of 5 tasks, while the Fine-Grained (FG) scenario consists of *smaller* groups for a total of 11 tasks. Table 3.1 shows the mapping between macro-nodes and tasks of the two different scenarios.

3.4 EVALUATION OF THE MODEL

In this section, we first perform a bandwidth characterization of the different communication mechanisms considered in this work. Then we conduct end-to-end experiments to conduct an accuracy analysis of the proposed performance model.

3.4.1 *Experimental Setup*

Experiments are executed on a virtualized Kubernetes cluster composed of one master, representing the Kubernetes control-plane, and three worker nodes on which Tasks are deployed, along with an Network File System (NFS) server inside the same infrastructure. The master runs in a PV with 32GB of memory and 16 vCPUs, while the three workers and the NFS server run in a PV with 32GB of memory, 8 vCPUs, and 50GB of disk space each. The Kubernetes cluster and the NFS server are mapped on five physical nodes residing in the same rack and featuring an Intel®Xeon Silver 4114 CPU running at 2.20GHz connected to four Seagate®ST2000NM0033-9ZM via a Fujitsu PRAID EP400i Controller, and a network interface composed of an Emulex Engine™(XE)100 Series NIC and an Intel®82599ES 10Gbps

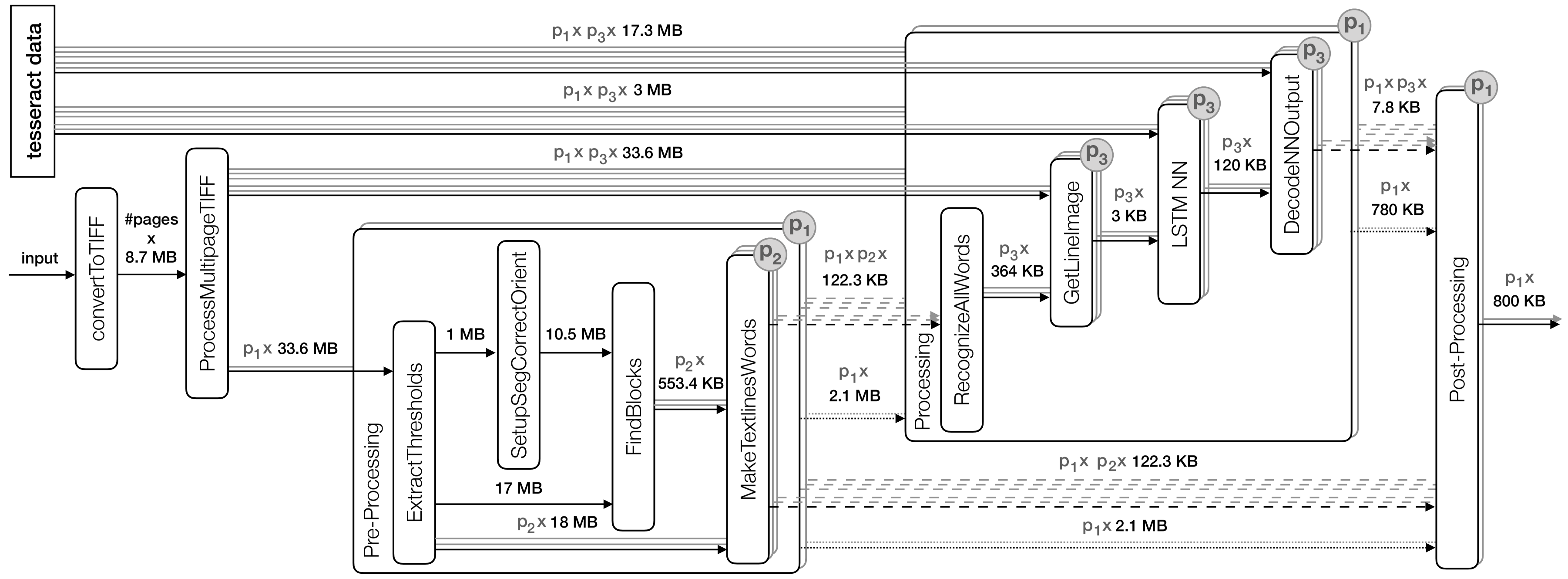
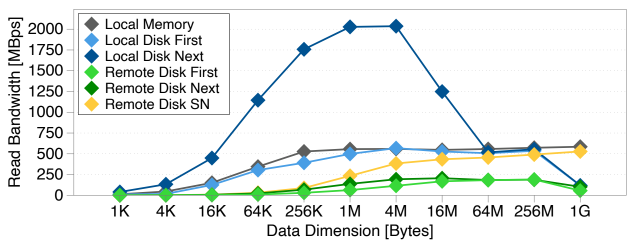
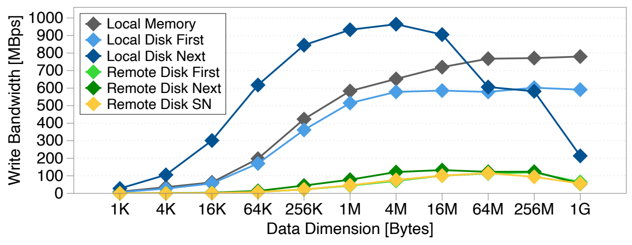


Figure 3.4: Tesseract dataflow graph reporting the macro-nodes data movements on the arcs. The different workload levels of parallelism p_i, i.e., page, block and text line, are highlighted on both macro-nodes and arcs. The dotted and dashed arcs are mutually exclusive: the dotted arcs are used with the coarse-grained workload computation, while the dashed arcs are used with the fine-grained workload computation.



(a) Read



(b) Write

Figure 3.5: Memory and disk characterization. First and Next keywords highligh the first and subsequent local/remote disk bandwidths, while Remote Disk SN refers to remote operations with producer and consumer placed on the same node.

Ethernet Controller. Physical nodes are connected through a 10Gbps Brocade VDX6740 network switch. The memory, the raid controller, and the disk feature 17GBps, 1.6GBps, and 175MBps of maximum transfer rate, respectively.

3.4.2 *Model: Bandwidth Characterization*

One of the model inputs is the characterization of the different communication mechanisms, i.e., local memory, local storage, and remote storage. In this characterization we want to capture the performance of the same complete stack that can be found in a serverless platform, and not just the performance of the devices. The experiments are based on two kinds of tasks: *producer* tasks writing data, and *consumer* tasks reading data. Focusing on solutions inside the same infrastructure, data can be shared by either using a local PV (local memory and storage), or a remote PV map on the NFS server (remote storage). The latter can also be used to simulate an environment similar to that of major public serverless providers. While it would be possible to run directly against a Google Cloud Storage or Amazon S3 bucket, it is not considered in this characterization because the focus is on storage solutions inside the same infrastructure. We evaluate data exchanges of different sizes, ranging from 1KB to 1GB with a 4× increment. Figure 3.5 shows the sequential bandwidths of the considered data exchange mechanisms. For both read and write operations, the local memory and the local disk (First) solutions exhibit similar performance. Thanks to the *disk buffering* computing infrastructure optimization, allowing to buffer data in main memory *buffer cache* space, the local disk read and write bandwidths reach a maximum of

535MBps and 602MBps, which are much higher than the 175MBps disk maximum transfer rate reported by the vendor. When a Task reads or writes data on a PV map on local disk, data are buffered in main memory *buffer cache* space, allowing to benefit from higher bandwidths compared to the disk ones. The actual dump on the disk is done periodically in background from the operating system. Differently, the remote data requests are handled by communicating directly with the disk. The only exception is when two tasks are co-placed: the producer task writes the data on the remote disk, while the consumer task reads from the memory buffer cache. We notice that in both local and remote solutions, when a task reads/writes multiple times sequentially on the same PV, e.g., a task collecting multiple data in input from various producers, the first operation and the following ones exhibit different bandwidths. More precisely, the subsequent local and remote read and write bandwidths are up to 4.41× and 2.14×, and 3.34× and 1.88× higher when compared to the first operations bandwidths, accordingly. The local disk bandwidth increase is due to caching, while, being the data read/written sequentially, the subsequent remote operations take advantage of the sequential locality at the hard disk level.

Since the model is meant to handle many tasks running and sharing data at the same time, we also evaluate the performance degradation with concurrent requests. We characterize concurrent read and write bandwidths by manually enforcing the synchronization of the executed tasks. As expected, by increasing the number of concurrent requests, the single task bandwidth decreases, especially with high data dimensions. More specifically, with 30 concurrent tasks, the read and write bandwidth experience the highest performance degradation with a decrease of up to 4.60× and 5.98×, and to 26.72× and 38.55×, for local and remote operations, accordingly.

3.4.3 *Model: Accuracy*

To validate and measure the accuracy of the proposed model, we compute the average relative error of the inter-function I/O time prediction with respect to the actual communication overhead of 100 Tesseract-like deployment runs over two different input datasets, featuring 1 and 10 pages, respectively. Following a classical black-box approach, with *Tesseract-like* deployment we mean a deployment where each task performs the same data exchanges described in 3.3.3, and sleeps during the specified time. To be fairer in the results, since compute time is constant and significantly larger than inter-function I/O time, we consider only the I/O time in the measurements. We run the Tesseract-like computations in two configurations: the first co-places all the tasks on a single node following the default Knative/Tekton strategy, while the second maps the tasks on the worker nodes in a round-robin fashion. These two configurations are referred to as *local* and *remote*.

Table 3.2: Model accuracy.

	KPI	Coarse-grained		Fine-grained	
		1 page	10 pages	1 page	10 pages
Local	Avg I/O Time [s]	0.324	2.890	0.670	7.362
	Pred I/O Time [s]	0.319	2.753	0.750	7.055
	Avg Rel Error [%]	3.09	4.21	7.70	5.64
Remote	Avg I/O Time [s]	1.189	9.303	5.388	49.938
	Pred I/O Time [s]	1.099	9.283	5.579	53.813
	Avg Rel Error [%]	6.90	3.64	5.61	7.42

To validate our model, we only consider sequential executions, representative of several real-world serverless applications [75, 76]. We run exhaustive experiments with varying data dimensions and number of operations, and compare the I/O times with the model predicted values. In particular, we run the coarse-grained and fine-grained Tesseract-like workloads exploiting both local and remote storage configurations. As shown in Table 3.2, our model predicts communication latency with an average relative error of 5.52%, which we believe is accurate enough to analyze the I/O time impact when porting a workload to Knative/Tekton.

3.5 EVALUATION OF DATA-AWARE SERVERLESS

Thanks to the flexibility of the proposed model, we investigate the performance impact on the use case workload of the following key factors: task granularity and concurrency, data locality, resource allocation, and scheduling policies.

3.5.1 *Task Granularity and Concurrency*

Task granularity and concurrency are two relevant factors that could highly increase or reduce performance in terms of number of deployed task-instances and overall end-to-end time. Finding the optimal task granularity and concurrency becomes crucial when moving to serverless. Figure 3.7 shows the effect of different granularities and concurrencies on the number of deployed task-instances and the end-to-end time in local and remote deployments on a 10 pages input with 100 text lines each. As shown in Figure 3.6, given the three levels of parallelism, i.e., at per-page, per-block, and per-textline level, we find that the page parallelism always improves the end-to-end time, while the block parallelism negligibly impacts the performance and the text line parallelism have similar end-to-end times for concurrency ranging from 25 to 100. For this reason, while the coarse-grained (CG) solution exploits page parallelism, the fine-grained (FG) solution exploits different degrees of parallelism at per-page level, no block parallelism, and up to a degree of parallelism of 25%

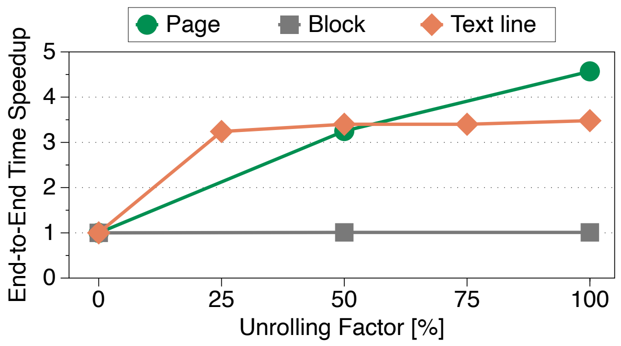


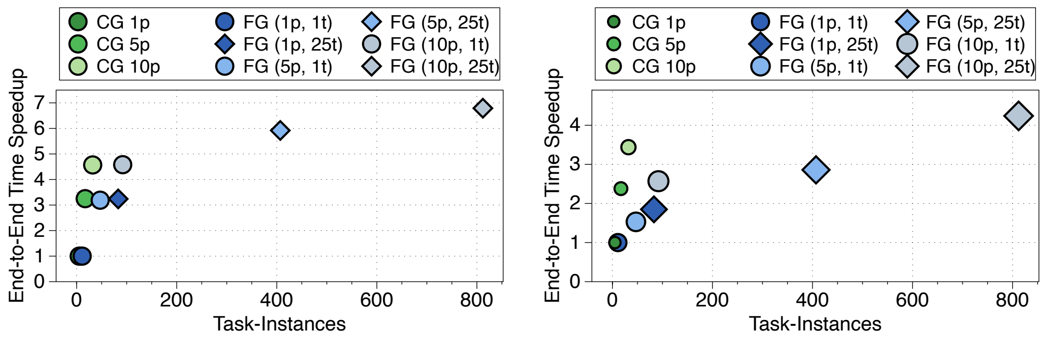
Figure 3.6: End-to-end time speedup versus the sequential execution, achieved by applying the three different levels of parallelism.

for the text line parallelism. The fully-parallelized coarse-grained deployment increases the number of task-instances from 5 to 32 and achieves an end-to-end time speedup of 4.57× and 3.44×, for local and remote deployments, respectively. The fully-parallelized fine-grained deployment significantly increases the number of task-instances from 11 to 812, achieving a speedup of 6.79× and 4.24×, accordingly. For similar configurations (depicted as circles in Figure 3.7), coarse-grained deployment outperforms fine-grained deployment, especially in the remote configuration where it achieves up to 1.55× of performance increase. On the other hand, fine-grained executions achieve higher speedups, but at the expense of number of tasks and I/O. This needs to be considered because the number of tasks may be limited in local configurations, and I/O may become a bottleneck in remote configurations. In the remote configuration, the network load changes substantially: with fully-parallelized coarse-grained deployment it reaches 1.4GB, while it grows up to 15.6GB with fully-parallelized fine-grained deployment.

3.5.2 *Data Locality*

The local deployment outperforms its equivalent remote deployment in both coarse-grained and fine-grained scenarios, as shown in Figure 3.7. However, fully local deployments are not realistic for many applications, particularly in large clusters where remote deployments may be more flexible and improve overall throughput and resource usage.

A shortcoming of existing serverless frameworks is that tasks that are co-placed on the same node and share the same inter-function data (e.g. functions that read the same input) do not currently benefit from data reuse. That is, each instance of a function on the same node will fetch the same inter-function input data. Enabling data-intensive serverless workloads will likely require some optimizations in this regard. Creating a local memory or disk buffer holding the shared input data might be a simple way to benefit from data locality, decreasing requests for remote data, and improving its performance.



(a) Local

(b) Remote

Figure 3.7: End-to-end time speedups versus the sequential execution, achieved by exploiting page- and textline-level of parallelism (p, t), and Task-instances number. (a) highlights the local deployment, while (b) highlights the remote deployment reporting network load impact with increasingly bigger symbol sizes.

3.5.3 *Resource Allocation*

In an ideal serverless environment, the user should not need to provide any resource configuration. However, existing frameworks still allow customizing expected resource usage for improved performance and cost. In case of no customization, frameworks usually default new tasks with predefined amount of CPU and memory (i.e., 100m of CPU and 128MiB of memory), leaving the Task to use more resources if available, up to the default limits (i.e., 2CPU and 1GiB of memory). With the default resource allocation, tasks under-utilizing the assigned resources could prevent other tasks from being scheduled, leading to worse overall system throughput. This is particularly relevant for many data-intensive workloads since tasks belonging to the same pipeline may have significantly different resource requirements, and these can also change depending on the input.

For example, focusing exclusively on memory for the sake of simplicity, we get approximately 31GiB of allocatable memory in each one of the worker nodes in the evaluation environment. Each Tesseract execution amounts to 150MiB on average, with a peak of 167MiB, in the first two-thirds of its execution and 90MiB in the last one-third. In a coarse-grained scenario, and allocating the same peak memory of 167MiB (see Section 3.3.3) to all 5 tasks, we would be able to fit up to 38 Tesseract deployments per node. With a more accurate allocation of 150MiB, 167MiB, 150MiB, 90MiB, 90MiB for each one of the five tasks, we would be able to fit up to 49 Tesseract deployments.

3.5.4 *Scheduling Policies*

Most of the schedulers proposed in the literature consider properties such as CPU usage, memory utilization, job execution time, and job deadline. Another essential factor to consider when optimizing data-intensive task placement is the effect on the network. By leveraging on the proposed model, we highlight the primary importance of data locality in reducing communication overheads by showing the impact on the overall end-to-end time of four different scheduling strategies: (1) random, (2) round-robin, (3) consolidating, and (4) data-centric. The first two are self-explanatory. The *consolidating* strategy places tasks at per-node level in a sequential fashion, saturating the resources available for each node. The *data-centric* strategy aims to reduce inter-node data movements, considering the number and amount of data to be shared and exchanged between functions. For the use case workload, the data-centric strategy maps for the coarse-grained scenario the *processMultipageTIFF*, *Pre-Processing* and *Processing* tasks on the same node. For the fine-grained scenario, instead, the set of tasks computing the same page and belonging to the pre-processing and processing phase are co-placed, while spreading the remaining ones randomly. This allows to reduce the communication overhead by exploiting the higher local bandwidths for the most relevant data movements, and to reduce the number of remote data exchange.

Figure 3.8 shows the performance achieved by the four strategies when simulating the execution of 50 pages input on a 100 nodes cluster, with a degree of parallelism of 50% and 100%. It is noticeable that the random and round-robin performance are similar, and that they are outperformed by both the consolidating and data-centric strategies. In particular, the consolidating strategy achieves an I/O time speedup of 4.32× and 3.40× in the coarse-grained scenario, and of 1.49× and 1.16× in the fine-grained scenario, for 50% and 100% degrees of parallelism, respectively. The data-centric solution gains an I/O time speedup of 2.62× and 2.35× in the coarse-grained scenario, and of 1.84× and 1.48× in the fine-grained scenario, for 50% and 100% degree of parallelism accordingly. The coarse-grained workload deployment is composed of 77 and 152 tasks for 50% and 100% degrees of parallelism, respectively; its consolidating placement performs better than the data-centric solution because it relies on fewer nodes, i.e., one or two. Differently, the fine-grained workload deployment comprises a much higher number of tasks, i.e., 2027 and 4052; therefore, the data-centric solution impact on the network load is lower than the consolidating solution. More specifically, the data-centric achieves a 1.23× and 1.27× of performance improvement versus the consolidating strategy, for 50% and 100% degrees of parallelism, respectively. While the consolidating solution sequentially maps tasks, the data-centric one efficiently places all the tasks characterized by the page-level parallelism as a unit, reducing the communications over the network.

3.6 RELATED WORK

A big challenge when deploying data-intensive workloads on serverless computing platforms is efficiently sharing data between tasks. Several works in the literature have proposed

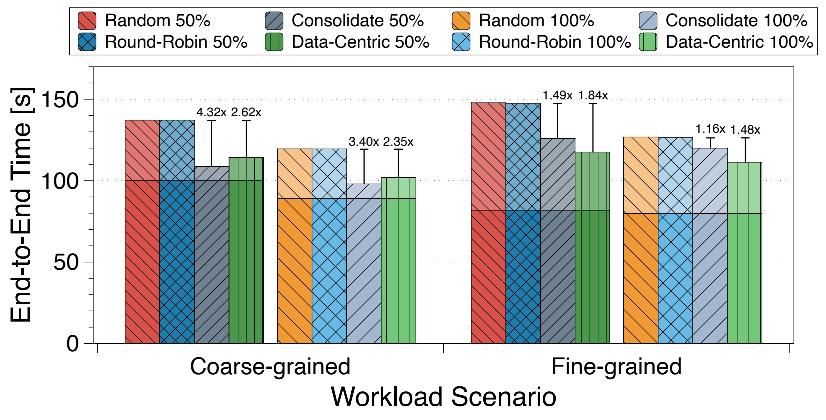


Figure 3.8: Comparison of the considered scheduling policies when simulating the execution of a 50 page input on a cluster with 100 nodes. Horizontal bars divide computational time (bottom) and inter-function I/O time (top).

relaxing disaggregation to favor performance. [77] proposes the *"fluid multi-resource disaggregation"* concept, where the platform can fall back on disaggregation as needed, but can also move resources around to enhance proximity, meaning co-locating code with the data it accesses. Similarly, [78] proposes the so-called *"fluid code and data placement"* in which the infrastructure should have the ability to selectively co-locate code and data on the same side of a network boundary, whether done via caching/prefetching data near computation or pushing computation closer to data. The same work claims that this feature is not provided in a meaningful way by today's serverless frameworks.

In the past years, there have been many proposals to orchestrate distributed computing running on serverless [79–83], but only a few focusing on storage [37, 84, 85]. The majority of the works present in the literature are built on top of AWS Lambda, as the two discussed in the following, which more relate to the proposed work. In [37], the authors propose Locus, a serverless model for the most general all-to-all shuffle scenario that combines cheap but slow storage with fast but expensive storage, achieving 4×-500× performance improvements over baseline, and at the same time being close to or beating Spark's query completion time by up to 2×. The work shows that, with only slow storage, sorting 100TB of data, a hash-based shuffle would be 500× slower than the current record, while with only fast storage, supporting a much higher throughput, the cost would become prohibitive. Locus accurately predicts shuffle performance, with an average error of 15.9%. When compared to Locus, being much more general, our model predicts, with higher accuracy, the performance of an entire workload deployment, exploiting fast local storage as much as possible. [84] highlights the relevance of efficiently communicate data between functions via a shared data store in analytics workloads on serverless platforms. The work analyzes three different storage systems, i.e., a disk-based managed object storage service, an in-memory key-value store, and a Flash-based distributed storage system. In particular, the authors show that S3 has significant overhead, particularly for small requests, achieving a throughput of up to

70MBps for requests of 10MB or larger. Similarly, by characterizing an NFS-based remote shared storage, we show that it also introduces significant overheads for small requests while achieving higher throughput at scale. To be in line with the serverless abstraction, in this work, we only account for fully-managed storage solutions, as in-memory key-value stores and flash-based storage would require the user to either select instance types with the appropriate memory, compute and network resources, or to manually configure and scale their storage cluster resources, respectively.

To the extent of our knowledge, our work is the first analyzing the storage solutions and modeling the data-sharing performance within the serverless Kubernetes-base ecosystem.

3.7 SUMMARY

This work analyzes in detail the feasibility and efficiency of running more complex workloads, such as data-intensive analytics workloads, within serverless environments. More precisely, we fully understand data exchange mechanisms between functions in current open-source serverless platforms, emphasizing that communication latency through shared object storage represents the main bottleneck in scenarios involving more complex workloads. To address this, we introduce a performance model accurately predicting the data-sharing performance with an average relative error of 5.52%. This performance model accounts for how data is shared between functions, considering not only the amount of data but also the underlying technology employed. Such model enables the assessment of data-intensive analytics workloads in terms of task granularity and concurrency, data locality, resource allocation, and scheduling policies. Our evaluation shows that task granularity and concurrency are pivotal factors that could highly reduce or increase performance in terms of overall end-to-end time and number of deployed task-instances. For similar configurations, coarse-grained deployment outperforms fine-grained deployment, achieving an end-to-end time speedup of up to $1.55\times$. On the other hand, when considering varying parallelism configuration, fine-grained executions achieve higher speedups (from up to $4.57\times$ to up to $6.79\times$), but at the expense of number of tasks (from 11 to 812 tasks) and I/O (from 1.4GB to 15.6GB). We also show that, when fixing the amount of resources, a more accurate resource allocation strategy allows to fit 49 workload instances per worker node, an increase of 11 instances compared to the default resource allocation. Finally, we show that the data-sharing time for such complex workloads can reach a speedup of up to $4.32\times$, depending on how the workload is deployed and scheduled. This work gave us valuable insights on research directions towards optimizing the performance of serverless environments when executing complex data-intensive analytics workloads.

3.8 PUBLICATIONS

The contents of this contribution are summarized in the publication:
[86] **Anna Maria Nestorov**, Jordà Polo, Claudia Misale, David Carrera and Alaa S. Youssef, "Performance Evaluation of Data-Centric Workloads in Serverless Environments." In Proceedings of the *14th International Conference on Cloud Computing (CLOUD)*. IEEE, Chicago, IL, USA, 2021, pp. 491-496, 10.1109/CLOUD53861.2021.00064

4

DIRECT INTER-FUNCTION COMMUNICATION ENABLEMENT

This chapter describes the second contribution of this thesis. This contribution tackles the problem of centralized object storage, leading to highly inefficient workload executions when directly using serverless platforms to run data-intensive analytics workloads. To cope with this, we propose Floki, a system enabling direct data communication between non-colocated functions, allowing to efficiently run data-centric analytics workloads in open-source serverless environments (Section 4.2). We prototype our proposed solution and our experimental evaluation (Section 4.3) shows the effectiveness of the proposed solution to overcome state-of-practice shared object storage high latency accesses and to execute data-intensive analytics workloads efficiently in terms of end-to-end latencies and resource usage impact.

4.1 INTRODUCTION

Major serverless platforms [4–6, 46] take disaggregation to an extreme, imposing functions to exchange data only through shared object storage. In the context of data-intensive analytics workloads, represented as DAGs and characterized by a considerable intermediate data transfers, shared object storage becomes a bottleneck for efficient inter-function communication due to its high-latency access. Indeed, as demonstrated in [37], directly using a serverless platform for data-intensive workloads leads to highly inefficient executions. The slow data transfers between functions make the CloudSort benchmark [87] to be up to 500× slower when executed on AWS Lambda [4] with S3 instead of on a cluster of VMs.

Recent studies tackle this problem by implementing optimized exchange operators [4, 38], using multi-tier storage combining slow with fast storage or solely remote in-memory storage [37, 85, 88], exploiting per-node caches [89, 90], co-locating functions on a single container [91–94], handling external storage on long-running VMs [16, 95, 96], or circumventing the network constraints [97]. However, these methods either use domain-specific optimizations, require two copies of data over the network, are not fully transparent to the user, break the advantage of fine-grained scaling, or use non-serverless components.

In this work, we present *Floki*, a system enabling direct inter-function communication in Kubernetes-based environments. It creates point-to-point data channels exploiting conventional volumes, pipes and Transmission Control Protocol (TCP) sockets, for intra-node and inter-node data transmission, allowing data to be transferred directly from producer to consumer functions in a fully transparent fashion, minimizing data copying over the network. Floki offers workflow-oriented data communication, increasing performance while minimizing resource requirements without imposing any constraint on function placement. Specifically, its flexibility and observability allow creating data channels adapting to the specific function scheduling of the underlying orchestration framework. We envision Floki to be leveraged by container-based platforms and users for high-performance volatile intermediate data exchange, as an alternative solution for message passing.

In summary, the main contributions of this work are:

– A proactive workflow-based data forwarding system enabling point-to-point data transfers without additional overheads, such as state-of-practice storage overheads.

– The design of a full communication stack, allowing non-colocated functions to share data as they are hosted on the same node through local read-write operations.

– An in-memory mechanism for transferring volatile data, capable of dealing with arbitrary intermediate data sizes efficiently and scalable on the data volume.

– A complete benchmark of Floki on the principal communication patterns in distributed systems, i.e., one-to-one, fan-out, fan-in, and all-to-all, with data transfers between 1MB and 16GB. Floki improves end-to-end time performance up to $74.95\times$, reducing the largest data sharing time from 12.55 to 4.33 minutes, while requiring up to $50,738\times$ fewer disk resources, with up to roughly 96GB disk space release.

The remainder of this chapter is organized as follows: Section 4.2 describes the proposed architecture. Section 4.3 reports the experimental setup and evaluates the results. Section 4.4 discusses potentials, limitations, and future research directions. Section 4.5 reviews the main related work. Finally, Section 4.6 summarizes the work.

4.2 SYSTEM DESIGN

As introduced in Section 4.1, the shared remote storage represents the main bottleneck when deploying data-intensive workloads in serverless environments. In this section, we tackle the shared object storage bottleneck problem by presenting Floki, a system that proactively enables faster point-to-point data sharing by exploiting local resources and TCP socket connections. The system requires two inputs: First, the DAG describing the workflow where nodes represent functions and arcs represent data dependencies between functions; Second,

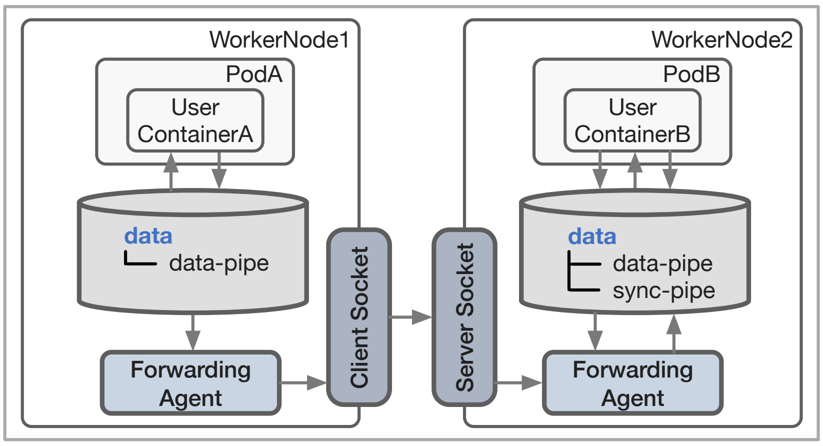


Figure 4.1: Floki's architecture.

the mapping between functions and cluster nodes. In the current version of Floki, we assume the two inputs are given, and the user containers read/write data sequentially. We further discuss these assumptions in Section 4.4. In Floki, we achieve the following design goals: 1) Deal with arbitrary complex data structures, 2) Proactive data transfer between functions, 3) Fast and direct inter-function communication, 4) No constraint on functions placement. We highlight how we achieve these specific design goals in the remainder of this section.

Floki's architecture transmits data in a fully volatile manner relying on pipes and TCP sockets, widely used for Inter Process Communication (IPC) for their efficiency, for intra-node and inter-node data transmission, respectively. Figure 4.1 provides an overview of Floki's architecture allowing a volatile sharing of data. In Floki functions run concurrently, while with the naïve shared object storage communication functions run sequentially based on data dependencies (the consumer function is deployed only when all the producer functions end writing intermediate data). Data is transmitted and stored as byte arrays, allowing Floki to deal with arbitrary complex data structures independently of the programming language (Goal 1). Floki's architectures solve the centralized storage bottleneck by offering direct communication between functions, where data exchanges are managed on a producer-consumer functions pair level, minimizing data copying over the network. Direct communication between functions implies the following advantages. First, the number of concurrent read/write operations is reduced as resources are shared among a lower number of functions, i.e., the ones co-placed on a given node, or of exclusive use. Second, the I/O and CPU usage are lower. Finally, thanks to multi-threading, a producer function data can be sent in parallel to multiple consumer functions, and a consumer function can receive concurrently multiple data.

4.2.1 *Key Components*

Floki's architectures consider five key components: the *data-* and *sync-pipe*, the *client* and *server sockets*, and the *forwarding agent*. To proactively transfer data between producer-consumer function pairs, based on the specific workload DAG and function-node mappings, Floki creates all the necessary components and relative connections immediately after workload submission and before its deployment (Goal 2).

4.2.1.1 *Data- and sync-pipe*

The two pipes, exposed on a local PV, allow exchanging data between the user container and the local *forwarding process*. While the *data-pipe* represents the data communication channel, the *sync-pipe* synchronizes Floki with the user container letting the *forwarding agent* to write on the *data-pipe* only when the user container is ready to receive. This prevents the write operation from receiving a broken pipe signal when the read file descriptor referring to the pipe read end is not opened.

4.2.1.2 *Client and server sockets*

These are the key components in charge of transmitting data between pairs of nodes. We choose to implement TCP sockets guaranteeing features such as error checking, ordered data delivery, and enabling uniquely identified connections between two endpoints, i.e., combining client and server sockets.

4.2.1.3 *Forwarding agent*

This component, instantiated into a process in the host namespace, represents Floki's architecture core component. At a high level, the primary purpose of this component is to drive inter-node communication, forwarding data directly from the producer to the consumer function (Goal 3). More precisely, its role is threefold. First, it creates and sets up the required TCP connections. Second, it supplies the necessary input data to the user container. Third, it forwards the data produced by the user container to the following functions in the chain. Since the *forwarding agent* mainly performs write/read memory buffers operations, we expect its overhead to be negligible. To proactively set up the communication infrastructure for the specific workflow, it internally stores the function name to which it refers and the ordered lists of data object names to receive/send for each of the previous/following functions in the workflow. In addition, it stores the mapping between the following functions in the workflow and the IP address of the nodes to which they have been scheduled to account for the underlying scheduler functions placement

Algorithm 1 Floki's *forwarding agent* algorithm.

```
 1: procedure RECVDATA(sSocket, listObjsToRecv)
 2:     AquireLock(dataPipe)
 3:     for all objName ∈ listObjsToRecv do
 4:         dataSize = RecvAndWriteSize(sSocket, dataPipe)
 5:         RecvAndWriteData(sSocket, dataPipe, dataSize)
 6:     end for
 7:     ReleaseLock(dataPipe)
 8: end procedure
 9: procedure SENDDATA(cSocket, listObjsToSend)
10:     for all objName ∈ listObjsToSend do
11:         if currentObjName == objName then
12:             SendDataSize(cSocket, sizeBuffer)
13:             for k = 1 to $\lceil outDataSize/packetSize \rceil$ do
14:                 SendDataPacket(cSocket, dataBuffer)
15:             end for
16:         end if
17:     end for
18: end procedure
19: procedure FORWARDINGAGENT( )
20:     producers = GetSocketsProducersNames(sSockets)
21:     SendOwnName(cSockets)
22:     WaitReady(syncPipe)
23:     for i = 1 to #sSockets do
24:         thsIn[i] = thread(RECVDATA, sSockets[i], listObjsToRecv)
25:     end for
26:     WaitThreadsEnd(thsIn)
27:     for j = 1 to #cSockets do
28:         thsOut[j] = thread(SENDDATA, cSockets[j], listObjsToSend)
29:     end for
30:     for all outObjName ∈ outObjsNames do
31:         currentObjName = outObjName
32:         outDataSize = ReadDataSize(dataPipe, sizeBuffer)
33:         for k = 1 to $\lceil outDataSize/packetSize \rceil$ do
34:             ReadDataPacket(dataPipe, dataBuffer)
35:         end for
36:     end for
37:     WaitThreadsEnd(thsOut)
38: end procedure
```

(Goal 4). Based on this information, it automatically derives the necessary number of*server* and *client socket* connections, i.e., *#sSockets* and *#cSockets*.

Algorithm 1 shows the pseudo-code of the *forwarding agent*, whose top-function is represented by the ForwardingAgent procedure. Once the *#sSockets server* and *#cSockets client socket* connections are opened and set up, the *forwarding agent* receives producer functions' names getting the correspondence with the *server socket* connections (line 20). Storing for each producer the list of data object names to receive allows the *forwarding agent* to know the number and the names of the data objects transmitted on each *server socket*. Then, the *forwarding agent* sends the related function name on all *client sockets* (line 21). Since the *forwarding agent* starts before the workflow is deployed, to respect the coordination of the pipe operations, the *forwarding agent* waits for the user container ready signal eventing that it is running and ready to read data from the *data-pipe* (line 22). Once received, the *forwarding agent* creates the *#sSockets* input threads (lines 23-25). The input threads alternately write on the *data-pipe*, sending first the data size (line 4) and then the data content read in a packet-based fashion from the corresponding *server socket* (line 5). The input threads' *data-pipe* write operations follow the order declared in the stored list of data objects to receive. To handle *data-pipe* contention, the threads' write operations are synchronized by acquiring (line 2) and releasing (line 7) a lock. When all input threads finish (line 26), the output threads, sending the user container-produced data on the *client sockets*, are created (lines 27-29).

For each produced data object, the *forwarding agent* reads the data object size $outDataSize$ (line 32) and the data object content from the *data-pipe* (lines 33-35). To guarantee output threads access to both data object size and content, the *forwarding agent* read operations store them in $sizeBuffer$ and $dataBuffer$ shared memory buffers. Finally, each output thread sends the buffers on the *client socket* (lines 12-15) if the current received data object, i.e., $currentObjName$, belongs to its list of objects to send (line 11).

4.2.2 *Execution Lifecycle*

Figure 4.2 illustrates how Floki works step-by-step with a simple example of a two functions workflow. The first function reads the workflow input stored in the centralized object storage and creates the intermediate output *outA*. In contrast, the second function reads the intermediate data object *outA* and computes the workflow output *out*, saving it in the object storage. For data transferred in a packet-based fashion, in Figure 4.2 we highlight the operations performed multiple times with circular arrows. First, the two *forwarding processes* are created and started, and then the two functions, represented by *PodA* and *PodB*, are deployed and started concurrently. While the first function reads the workflow input from the object storage (step 1), the second function sends the ready signal on the *sync-pipe* to the local *forwarding agent* (step 2), eventing it is up and running and waiting to read data

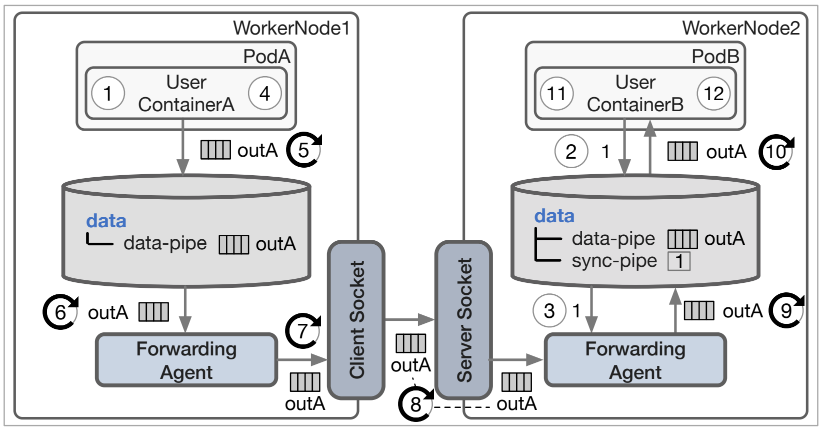


Figure 4.2: A step-by-step example of Floki.

on the local *data-pipe*. During the intermediate data object *OutA* computation (step 4), the *forwarding agent* on the second node reads the ready signal (step 3) from the *sync-pipe* and waits for the first packet on the *server socket*. Once the first function ends to compute the intermediate data object *outA*, it first writes *outA* size packet and then iteratively writes *outA* content in packets on the local *data-pipe* (step 5). The local *forwarding agent* reads the packets from the *data-pipe* (step 6) and sends them to the *client socket* (step 7). On the consumer side, packets are read from the *forwarding agent* (step 8) and written to the local *data-pipe* (step 9). Finally, packets are read from the second function (step 10), which, once received *outA*, computes the workflow output *out* (step 11) and stores it in the object storage (step 12).

4.3 EXPERIMENTAL RESULTS

To evaluate our approach, we first analyze the impact of different pipe and socket buffer sizes on data communication latencies to find the optimal sizes balancing performance and resource usage. Then, we evaluate Floki in terms of performance and resource usage impact, considering four of the most common distributed systems communication patterns, i.e., one-to-one, fan-out, fan-in, and all-to-all.

4.3.1 *Experimental Setup*

To prevent us from benchmarking cloud vendors' specific environments, the experiments are run on an on-premise cloud-prepared environment. Experiments are executed on a virtualized Kubernetes cluster composed of one master, representing the Kubernetes control-plane, and 7 worker nodes on which functions are deployed. Given the current absence of a built-in pipeline structure within Knative, as introduced in Section 2.2, we leverage Tekton for serverless capabilities. A MinIO server [98], a widely used high-performance

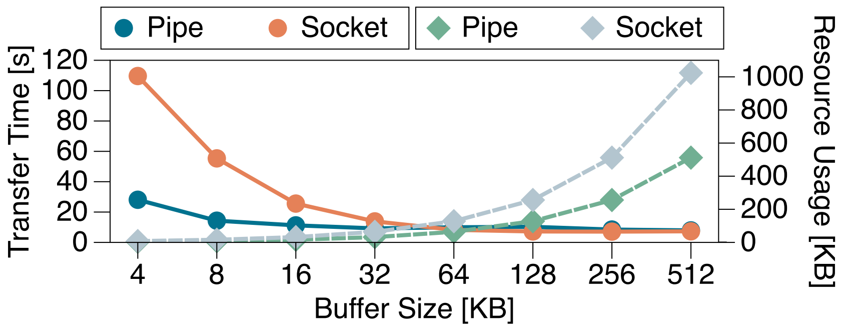


Figure 4.3: Average transfer time (left-hand side legend) and resource usage (right-hand side legend) for 16GB of data with pipe and socket buffer sizes ranging from 1 to 128 pages (4KB to 512KB).

object storage, outside the Kubernetes cluster but inside the infrastructure, is considered in the experiments as shared storage. In-memory key-value stores, such as Redis [99] and Memcached [100], are not considered since they require users to select instance types in terms of network, compute, and memory resources to satisfy their application requirements, and to explicitly provision resources. This requirement breaks the serverless hassle-free management fundamental principle. Furthermore, these in-memory storage solution usually re-introduce always-on infrastructure, severely limiting the benefits of serverless. Finally, while cloud providers offer in-memory storage instances based on Memcached or Redis, they are not fault tolerant and do not auto-scale as do serverless computing platforms (not matching the real flexibility demands of such platforms [3, 82]. The master runs in a Linux-based VM with 32GB of memory, 16 virtual cores, and 100GB of disk space, while the workers run in a Linux-based VM with 128GB of memory, 16 virtual cores, and 200GB of disk space each. The MinIO server runs bare-metal on a node featuring an Intel®Xeon E5-2620 CPU running at 2.00GHz, interfacing with two 1.6TB Intel®DC P3608 SSDs through NVMe. The VMs are synchronized in the millisecond range. The Kubernetes cluster is mapped on 8 physical nodes residing in the same rack and featuring either an Intel®Xeon Silver 4114 CPU running at 2.20GHz or an Intel®Xeon E5-2630 v4 CPU running at 2.20GHz. In the first case, nodes are connected to four Seagate®ST2000NM0033-9ZM; in the second case, nodes are connected to four Seagate®ST2000NM0055-1V4104. Physical nodes are connected through a 10Gbps Brocade VDX6740 network switch. Memories and disks feature 17GBps, 19GBps, 175MBps, and 249MBps of maximum transfer rate, respectively. The experimental evaluation reports average results computed over 10 sequential runs for reliability.

4.3.2 *Pipe and Socket Buffer Sizes Analysis*

In this analysis, we want to analyze the impact of different buffer sizes on fixed-size data communication over pipes and sockets and find the optimal buffer sizes. The experiments are based on two types of functions: a producer function writing data in packets on a channel and a consumer function reading data in packets from a channel. The evaluation considers different buffer sizes, ranging from 1 system page, i.e., 4KB, to 128 system pages, i.e., 512KB, with the kernel imposed constraint of a power-of-two increment[1]. The lower range limit, i.e., 1 system page, represents the size for which the kernel guarantees pipe writes operations to be atomic. Within Floki, a *data-pipe* is of exclusive use of a single producer at a time; thus, the pipe buffer size can be increased without affecting writes operations atomicity.

Figure 4.3 shows the average transfer times of a 16GB data object with different pipe and socket buffer sizes, and the related resource usage. Note that the socket resource usage is always twice the corresponding buffer size: TCP allocates twice the requested buffer size and uses the extra space for administrative purposes and internal kernel structures. While the difference between the pipe and socket transfer times is significant for small buffer sizes, the transfer times are comparable for big buffer sizes. In particular, as highlighted in Figure 4.3, 16 system pages buffer size, i.e., 64KB, represents the optimal size, reducing and balancing the pipe and socket transfer times. Increasing the buffer size would provide comparable transfer times while using more resources. Therefore, the following experiments and evaluations consider a buffer size of 16 system pages.

4.3.3 *Performance Evaluation*

We conduct a series of experiments to evaluate the performance of Floki in terms of end-to-end times. Targeting data-intensive analytics workloads, the evaluation considers data sizes ranging from 1MB to 16GB with a $2\times$ increment. The object storage end-to-end time, representing the baseline, considers two time components: the function time to write data from the object storage, and the function time to read data from the object storage. Differently, Floki end-to-end time considers three time components: the function time to write on the local buffer on the producer side, the time to transmit the data on the network, and the function time to read data from the local buffer on the consumer side. To measure the end-to-end times, we register the timestamp before each producer function starts to write data and the timestamp after each consumer function finishes reading data. Thus, given N producers and K consumers functions with TS_{p_i} and TS_{c_j} as their timestamps, we derive the end-to-end time T_{E2E} as:

$$T_{E2E} = max(TS_{c_1}, .., TS_{c_K}) - min(TS_{p_1}, .., TS_{p_N}) \tag{4.1}$$

1 A power-of-two increment is a constraint set by the kernel when allocating the pipe buffer size: given a requested buffer size x, the kernel allocates the next higher power-of-two page-size multiple of x.

As Equation (4.1) shows, the end-to-end time T_{E2E} accounts for possible not fully concurrent operations by considering the minimum of the producers timestamps TS_{p_i} and the maximum of the consumers timestamps TS_{c_j}.

Analyzing Floki end-to-end time components, i.e., write, network, and read times, we find that the network component dominates the end-to-end time. More specifically, local write and read times are in the tens of milliseconds range, while the network times are in the range of hundreds of milliseconds. As writers and readers functions are sequentially executed, the network component is highly affected by the underlying orchestrator overhead of creating, scheduling, and initializing the readers functions pods. With the data size increase, this almost constant overhead becomes negligible because the read and write time components, ranging from tens to thousands of seconds, tend to dominate the end-to-end time. While the overhead is visible in the one-to-one pattern, it is hidden in the remaining patterns by the object storage performance degradation. With multiple writers and readers, in the proposed volume-based solution, each writer/reader exclusively operates on its local PV; instead, with object storage, the writers/readers access the shared storage

Contrarily to a naïve solution relying on shared object storage, producers and consumers functions are deployed and run concurrently. The benefits of the volatile data share are more visible with small data sizes, i.e., from 1MB to 256MB, for which a higher performance increase is obtained. It is worth noting that, since data objects are read sequentially from the consumer functions, with multiple producers, functions using Floki gain smaller performance than those achieved with a single producer. Floki reduces the end-to-end time up to: 74.95× in the one-to-one pattern; 25.34×, 15.83×, and 24.83× in the fan-out pattern; 10.11×, 10.18×, and 7.49× in the fan-in pattern; 9.99× and 8.11× in the all-to-all pattern. Overall, considering the impact of Floki in terms of end-to-end time, the most significant time-savings are reached with a data size of 16GB, featuring the largest data transfer latency. In particular, the higher time reductions are achieved in the 1to6 pattern, where communication latencies are reduced from 753s to 260s on average. In other words, Floki allows saving 8.22 minutes on data sharing latency over the object storage baseline requiring 12.55 minutes.

4.3.4 *Resource Usage Evaluation*

Resource usage is crucial in serverless environments, where resources are billed with a pay-as-you-go model. We want to estimate and compare the resource usage of Floki to the object storage solution, representing the baseline, in the four considered patterns, i.e., one-to-one, fan-out, fan-in, and all-to-all. To assess the gap of resource requirements between varying data sizes, we choose two extreme cases just for comparison, i.e., 1MB and 16GB. In the following, DS represents the data size, T the total amount of functions composing the

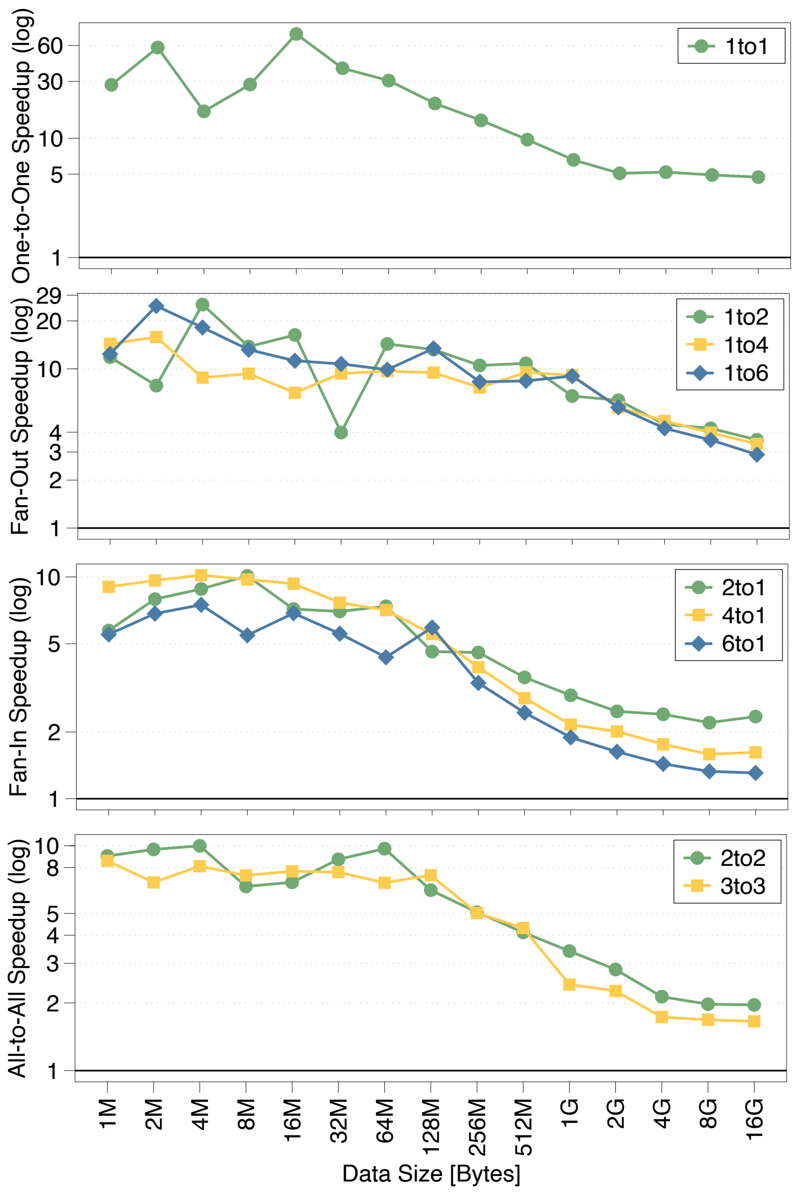


Figure 4.4: Floki's end-to-end speedups over the object storage solution in the analyzed distributed systems patterns.

specific pattern, P and C the number of producers and consumers functions, and PBD and SBD the pipe and socket local buffer sizes (i.e., 64KB and 128KB), accordingly. The object storage resource usage estimation does not consider the necessary internal buffers to write and read the data object/file since their sizes are negligible compared with the analyzed data sizes. Being the object storage shared among the different functions, we derive the related resource usage $RU_{ObjStorage}$ as:

$$RU_{ObjStorage} = (P * DS)_D \tag{4.2}$$

The required disk space is proportional to the number of producers functions P. Therefore, from a resource usage perspective, there is no difference among the patterns composed

Table 4.1: Resource usage and saving on 16GB data size.

Num. Producers	Num. Consumers	Object Storage	Floki	
		Usage [GB]	Usage [KB]	Saving
1	1	16	384	43,691
1	2	16	704	23,831
1	4	16	1,344	12,483
1	6	16	1,984	8,456
2	1	32	704	47,663
4	1	64	1,344	49,932
6	1	96	1,984	50,738
2	2	32	1,280	26,214
3	3	48	2,688	18,725

of the same number of producers and a different number of consumers. For instance, the fan-in pattern with three producers and one consumer would require the same disk space as the all-to-all pattern with three producers and three consumers. Differently, Floki represents a significantly less expensive resource usage solution. Considering only the memory space needed to hold the local buffers to perform the pipe and socket operations, we derived Floki resources usage RU_{Floki} as:

$$RU_{Floki} = (T * PBD + 2 * P * C * SBD)_M \tag{4.3}$$

We evaluate Floki resource usage following the presented analysis. By applying Equations (4.2) and (4.3), the resource-saving is evaluated by dividing the resource usage of the object store baseline for the Floki resource usage. For example, when sharing 1MB, Floki saves $\frac{1MB}{384KB} = 2.67\times$ of resources compared to the baseline. As Table 4.1 shows, Floki always demands a significantly lower amount of resources compared to the object storage solution. More precisely, each function composing the workflow only requires 64KB of memory for the pipe buffer (PBD) and 128KB for each client/server socket buffer (SBD), allowing significant resource-saving scaling linearly with the data size increase. For example, considering the simple one-to-one pattern ($P = 1$ and $C = 1$) on 16GB, while the object storage requires 16GB of disk space, Floki only requires $2 * 64KB + 2 * 1 * 1 * 128KB = 384KB$ of memory space, allowing a $16GB/384KB = 43,691\times$ of resource-saving. The achieved resource-saving on 16GB differs from the corresponding 1MB saving by a factor of $16,384\times$, equivalent to the difference between the two data sizes. Overall, Floki achieves up to $50,738\times$ of resource-saving, translating into a memory allocation of roughly 1.9MB instead of an object storage allocation of 96GB.

4.4 DISCUSSION

Envisioning Floki as part of Knative and Kubernetes-based workflows frameworks, we discuss the made assumptions and their integration limitations. In particular, the missing features relate with interfaces, fault-tolerance, multi-tenancy, data recovery, and integration with such frameworks.

4.4.1 *Assumptions*

Floki requires two inputs: the functions-nodes mapping and the workload DAG describing the workload pipeline. In the currently available computation frameworks, the workload DAG is provided by the user in the form of a configuration file (e.g., JSON file) [101, 102], in the form of a high-level description [103], or derived by the framework [104–106]. In our scenario, an immediate solution requires the user to supply the workload DAG. However, this could be a solution for coarse-grained pipelines with a small number of functions but challenging for complex pipelines with a large number of functions. Further steps are required to either facilitate the DAG specification or to remove the user *from the loop*. Instead, the mapping between functions and cluster nodes can be easily retrieved by inspecting the Kubernetes scheduler.

4.4.2 *Interface and Application Data Endpoints*

Floki currently requires the application to write data into a local binary "file" or a pipe. Given the current POSIX *write* function implementation, writing a file in a local file system is a straightforward process for the application, while writing a pipe requires reading and writing data in batches. Adapting an application would require substituting such *write* functions with *looped writes*, where a simple library function provided as API could interface such change without changing the programming model. Furthermore, processing data batches is performed sequentially, while enhanced *write* functions could perform the operations both in a sequential and a parallel fashion, increasing throughput. Finally, in scenarios where data is processed as streams, additional *write* functions could be provided for transmitting and computing concurrently, processing batches upon arrival at the consumer.

4.4.3 *Fault-Tolerance, Multy-Tenancy, and Data Recovery*

Fault-Tolerance, multi-tenancy, and data recovery represent crucial attributes of cloud and serverless computing. Floki needs more effort to be fault-tolerant. Hard multi-tenancy, enforcing strict tenant isolation, in Kubernetes environments can be achieved through complex namespaces, resource quotas, access control, and virtual cluster configuration, and Floki indirectly guarantees security in a multi-tenant environment. Concerning data recovery, in case of a component failure, intermediate data must be re-computed by re-running the entire workflow. However, in data-intensive workloads, re-computing data would imply an additional overhead, which scales linearly with the function computation complexity. Re-computing only failed functions could lower the overhead; thus, we believe per-function data recovery deserves further investigation.

4.4.4 *Porting on Kubernetes-based Workflow Frameworks and Knative*

Even though Floki is currently at its first maturity stage, we target to port the proposed solution to Kubernetes-based workflows frameworks and Knative. To integrate Floki with Knative, two main features are required. First, following the underlying orchestration platform feature, it is necessary to enable Knative functions to mount local volumes. Second, on top of the existing Knative Custom Resource Definitions (CRDs) providing sequential and parallel functions invocations, a more general workflow CRD has to be built. While porting Floki to Knative is more complex, the porting on Kubernetes-based workflows frameworks, e.g., Argo, would only require to automatically create the system components.

4.5 RELATED WORK

As we previously introduced, efficient intermediate data sharing between functions represents a key challenge for chained function execution, especially when dealing with data-intensive workloads [7, 16, 19, 23, 85, 95].

In recent years, different approaches have been proposed to optimize data exchange in serverless workflows. Lambada [34] and Starling [38] are serverless distributed data processing frameworks reducing the remote shared storage overhead by implementing exchange operators specifically optimized for cloud object storage. However, these works focus solely on database analytics, using domain-specific optimizations.

Locus [37] is an analytics system combining cloud object storage with managed in-memory storage to overcome the performance limitations of shared storage while remaining cost-efficient. The system applies a performance model to guide users in selecting the type and the amount of storage to achieve the desired cost-performance trade-off. Pocket [85] provides specialized and autoscaled distributed data stores for intermediate data sharing combining three storage tiers: a DRAM, a Flash, and an HDD tier. The authors show how the key goals of low-latency/high-throughput, storage resource sharing, and resource elasticity can be achieved by strictly decoupling control, metadata, and data planes. However, technologies like Locus and Pocket leverage per-workload resource demand information, provided by the user at submission time, to allocate and reserve resources for the workload lifetime. Different from these works, our pipe-based solution does not need any information on resource demand. As shown in [37, 88, 107], intermediate data sizes can consistently vary during the workload execution, resulting in the well-understood problem of potential performance degradation and/or resource under-utilization [108, 109]. To cope with this, Jiffy [88] is an elastic far-memory system allocating memory resources at the granularity of small fixed-size memory blocks—multiple memory blocks store intermediate data for individual tasks within a job. All these works involve indirect communication, demanding two serial data copies over the network in the critical path: one from producer function to shared storage and one from shared storage to consumer function. Contrarily, Floki always requires only one data copy for each inter-function communication (from source to destination node).

Crucial [89] improves inter-function data sharing in highly-concurrent applications by building a Distributed shared objects (DSOs) layer implemented on top of a modified Infinispan [110] in-memory data grid. Cloudburst [90] achieves logical disaggregation and physical co-location of computation and state by exploiting per-node data caches interacting with a specialized key-value store service [111, 112]. Unfortunately, the authors do not provide details on how the node caches are provisioned and sized, which indicates the necessity of manual/static provisioning of per-node cache resources. Similarly, OFC [113] exploits the co-location of functions and states to benefit from per-node caches and proposes an in-memory caching system distributed over the cluster nodes, leveraging RAMCloud [114]. However, it relies on a storage backend service and focuses on objects of 10MB or less. SAND [91] reduces data sharing latency by executing multiple functions in a workload as separate processes within long-lived containers where data are shared using a message-passing service exploiting data locality. SONIC [115] is a data-passing manager selecting the optimal data sharing method for each inter-function communication. It adopts a hybrid approach by sharing files within a VM (*VM-Storage*), copying files across VMs (*Direct-Passing*), or sharing files through shared storage (*Remote-Storage*). Even though our volume-based and SONIC's *Direct-Passing* data-passing methods are similar, we proactively start copying data from source to destination node immediately after producer function termination, while SONIC waits for consumer function to be scheduled to start copying data.

While works mentioned above allow workload functions to span across multiple nodes, Faasm [92] and Nightcore [93] co-locate workload functions on a single container to minimize data sharing latency thanks to shared memory access. The difference between these two works relates to the isolation strategy they leverage to provide private memory partitions. Following the same approach, Faastlane [94] also executes workload functions as threads within a single process of a container instance. It further identifies opportunities for function parallelism and concurrently executes parallel functions in fork processes or new container instances. While OFC, SAND, SONIC, and Faastlane represent fully transparent solutions, Crucial, Cloudburst, Faasm, and Nightcore require modifications at the application level. Even though container sharing systems improve data sharing efficiency, memory over-provisioning is necessary to ensure containers run multiple functions and extra services for concurrent executions during peak usage. Moreover, posing restrictions on function placement affects the advantage of fine-grained scaling of serverless.

Other works [16, 95, 96] rely on long-running VMs to handle external storage. For example, Cirrus [16] is an end-to-end framework specialized for ML training in serverless cloud infrastructures using large VM instances to run a custom storage backend through which intermediate results are shared. These works require additional infrastructure using non-serverless components and add complexity and cost to scale the VMs to match elasticity and function parallelism.

Finally, the work which most relates to Floki is Boxer [97]. Boxer has recently improved Lambada [37] by enabling inter-function direct network communication using conventional TCP connections. The authors overcome the limit preventing functions to accept incoming connections by deploying with the function code the Boxer sub-systems, executed alongside each function, establishing connections by exploiting the sequential and parallel TCP hole punching protocols [116, 117]. Differently, Floki establishes TCP connections on the host namespace; therefore, it does not require deploying additional connection-specific components in the serverless platform. Besides, Boxer sends buffers asynchronously, while Floki sends buffers synchronously.

4.6 SUMMARY

In this work, we presented Floki, a data forwarding system tackling the shared object storage problem by implementing direct inter-function communication between non-coplaced functions. Floki proactively establishes point-to-point communication channels between producer-consumer pairs of functions through fixed-size memory buffers, pipes and sockets. The designed memory mechanism's resource utilization scales linearly with the data volume increases, allowing to effectively deal with arbitrary data sizes within a paradigm where resources are billed with a pay-as-you-go model. Even though Floki is currently in its initial stage of maturity, with this work we demonstrate that direct inter-function communication is crucial for executing data-intensive analytics workloads in serverless platforms. We benchmark Floki on the principal distributed systems communication patterns, i.e., one-to-one, fan-out, fan-in, and all-to-all, considering data transfers from 1MB to 16GB. The evaluation reveals remarkable improvements achieved by Floki over the object storage solution. In particular, Floki shows up to 74.95× of end-to-end time performance increase, reducing the largest data sharing time from 12.55 to 4.33 minutes. Moreover, Floki requires up to $50,738\times$ fewer disk resources, which translates into a memory allocation of roughly 1.9MB instead of an object storage disk space allocation of 96GB. These findings underscore the potential of Floki to significantly enhance the end-to-end time performance and resource utilization efficiency when executing data-intensive analytics workloads in serverless environments.

4.7 PUBLICATIONS

The contents of this contribution are summarized in the publication:
[118] **Anna Maria Nestorov**, Josep Lluís Berral, Claudia Misale, Chen Wang, David Carrera, and Alaa Youssef. 2022. "Floki: a proactive data forwarding system for direct inter-function communication for serverless workflows." In Proceedings of the *8th International Workshop on Container Technologies and Container Clouds (WoC '22)*. Association for Computing Machinery, New York, NY, USA, 13–18. 10.1145/3565384.3565890

5 AUTOMATIC PERFORMANCE-COST EFFICIENT RESOURCE ALLOCATION

This chapter describes the third and last contribution of this thesis. This contribution analyses the impact, at different granularities, of dynamically scaling resources on performance and monetary cost metrics (Section 5.2). Furthermore, we propose a resource allocation manager, called Dexter, specifically designed to constantly monitor each application execution and automatically allocate just the right amount of resources at the smallest fine-grained level, guaranteeing performance-cost efficiency in terms of total runtime cost (Section 5.3). We implemented our proposed solution in the form of a pluggable module, toughly coupled with Spark, one of the most prominent multi-purpose, multi-language, in-memory big data processing frameworks (Section 5.4). Our experimental evaluation (Section 5.5) demonstrates the efficiency of our solution in reacting to variability of number of tasks and duration and the capability to trade-off performance and cost, significantly increasing the number of spawned workload instances, when compared with the current default serverless Spark dynamic resource allocation.

5.1 INTRODUCTION

Being typically resource-intensive and inherently parallel, data-intensive analytics applications highly benefit from the elasticity and the virtually unlimited scalability offered by serverless platforms. In particular, since early 2022, a prominent effort has been made towards leveraging serverless platforms to efficiently run large-scale data analytics frameworks, such as Apache Spark, which are traditionally deployed in managed cloud clusters. GCP Dataproc Serverless [48], Databricks Serverless [49], and IBM Analytics Engine [50] exemplify this trend. One key factor for both tenants and cloud providers is efficient utilization of resources: even minor improvements in utilization can save millions of dollars at scale [119]. However, auto-allocating resources to complex workflows, such as big data analytics applications, is challenging because the relationship between resource allocation and performance is complicated and changes over the application runtime [51].

To optimize their resource utilization and enhance performance-cost efficiency, it is crucial to set the right level of parallelism (how many tasks can run in parallel) for each stage. This involves allocating more resources to highly parallelizable stages with large-chunk input data, while limiting resources to stages with little inherent parallelism or running on small-chunk input data. In current serverless Spark frameworks, dynamic resource allocation, with FIFO mappings between stages and resources, represents the default behavior. The framework constantly monitors the number of pending tasks and scales resources, in case of backlog of pending tasks, up to a predefined upper bound. For instance, in the case of Dataproc Serverless, the default maximum number of resources is set to 1000, and the user has the option to control it by setting the respective autoscaling property. If the user does not set this upper bound, with small tasks the default setting can: 1) waste a lot of resources due to executor allocation overhead, as some executor might not even perform any work, and 2) see diminishing returns or even performance degradation at a higher cost. However, with fewer parallelizable tasks, the default setting can: 1) Waste a lot of resources due to executor allocation overhead, as some executors might not even perform any work, and 2) See diminishing returns or even performance degradation at a higher cost. The authors in [51] show that 75% of applications are over-provisioned (even at their peak), with 20% of them with over $10\times$ more additional resources than necessary. On the other hand, setting the appropriate amount of resources is non-trivial even for an expert user. Mis-configuration can lead to severe performance and cost issues due to resource under- or over-provisioning [51].

Given the serverless fine-grained elasticity and the pivotal role of cluster resource allocation managers in modern computing environments, the specific research problem we target in this work is: *Given a highly parallel data analytics workload, how should we dynamically scale the amount of resources (i.e., parallelism/scale level) at per-stage granularity to balance the performance-cost tradeoff, minimizing the overall cost while providing acceptable runtime performance?*

Currently, cluster schedulers rely either on general heuristics, such as simple packing strategies, or on modern and complex ML techniques [39, 120–122]. These approaches diverge in their fundamental principles, prioritizing opposite objectives. The former prioritizes generality and straightforward understanding, making them widely used in practice, but neglect potential performance optimization based on inherent workload characteristics. The latter captures cluster- and workload-specific features, but sacrifices interpretability and explainability and usually require a time-consuming training phase (may need millions of samples and take dozens of hours), making their adoption in real-world cloud clusters challenging. To bridge the gap between the two approaches, previous work designed more sophisticated heuristics built on performance models considering cluster and workload characteristics [51, 123–128]. Yet the trade-off between performance and costs is not considered in these heuristics.

The efficient use of resources in modern serverless big data processing systems poses several challenges. With serverless workloads, we potentially have limited or no historical data available. While approximately 40% of applications in production environments are recurring [129], conducting an initial profiling phase is not always feasible as some applications may not be used often enough to accumulate a sufficient number of samples for analysis. Second, abstracting the high diversity and scale in typical production environments, where hundreds of thousands of applications per day are executed over petabytes of data, is challenging. Third, existing resource allocation approaches targeting the cloud computing paradigm do not fit well with serverless Spark. Most literature focuses on optimizing resource allocation within a fixed-size cluster, aiming to optimize applications' resource sharing effectively. Differently, serverless environments are characterized by dynamic resource availability; therefore, each compute unit can scale independently. Ultimately, the performance of the same application may vary over time, influenced by changes in input or intermediate data sizes, or external factors like a surge in shared storage access.

In this work, we present Dexter aimed to increase resource efficiency in serverless data analytics through better resource management. Dexter monitors each stage execution, and it allocates resources to reach the performance saturation point, accounting for the correlated monetary cost. The proposed solution dynamically copes with performance-cost changes by automatically adapting the number of allocated resources at the fine-grain level, guaranteeing resource efficiency. Dexter is motivated by the lack of a serverless Spark solution, incorporating both performance and cost in the decision-making when allocating resources at the per-stage level. We integrated Dexter with Spark, and our evaluation shows that, compared with the current default serverless Spark dynamic resource allocation, our solution achieves a significant cost reduction, while improving performance-cost efficiency. Furthermore, Dexter enables a substantial resource saving, allowing to place a significantly higher number of workload instances on a fixed amount of resources. Finally, the proposed work provides an accurate and robust solution to new unseen workloads, achieving higher performance-cost efficiency thanks to its conservative resource scaling approach.

In summary, the main contributions of this work are:

– Through a performance and cost characterization study, we highlight the effects, at different granularities, of scaling out and scaling in on the execution duration and the corresponding cost, showing that it is crucial to tune resource allocation at the smallest fine-grained granularity, i.e., per stage level.

– We design Dexter, a resource allocation manager enabling more efficient executions, in terms of total runtime cost, for serverless data analytics. Dexter leverages historical knowledge to identify an initial scale level and, considering the price-performance trade-off, explores different scale levels during stage execution, converging to the optimal scale level.

- To demonstrate the applicability of Dexter, we fully integrate it as a standalone and pluggable module in Spark, building a resilient and fault-tolerant system.
- We extensively evaluate Dexter to assess its effectiveness on two industry-standard benchmarks, i.e., TPC-H and TPC-DS. Results show that Dexter is an accurate and robust solution, reducing cost up to $4.65\times$ while providing reasonable performance. Dexter improves performance-cost efficiency up to $3.50\times$, allowing up to $5.71\times$ resource savings, enabling a higher number of deployed workload instances.

The remainder of this chapter is structured as follows: Section 5.2 explains Dexter's design goals motivating this work. Section 5.3 describes Dexter's architecture. Section 5.4 describes the implementation details, including the integration with Spark. Section 5.5 presents the system evaluation, where we validate Dexter through state-of-practice benchmarks and real-world workloads, assessing its effectiveness for fine-grained resource allocation in serverless data analytics. Section 5.6 details the main related work to the resource allocation problem in the context of big data analytics. Finally, Section 5.7 summarizes Dexter.

5.2 GRANULARITY AND PARALLELISM LEVELS ANALYSIS

Automatically adjusting resources to meet the application's needs represents one of the key advantages of serverless computing. Making benefit of this key feature, dynamically adapting application resource parallelism levels in real-time ensures greater resource allocation flexibility, as well as higher resource efficiency. When an application is given more executors, while at the beginning performance sees a significant boost, after a saturation point, adding more resources gives either similar or even lower performance. As also observed by recent works [130, 131], once the saturation point is reached, increasing parallelism causes significant overheads due to higher serialization and de-serialization operations, garbage collection, and intensive shuffle operations on the network. These overheads generate a higher variance in the task durations and increase the probability of suffering from the well-known straggle[1] problem, delaying the entire stage execution.

Figure 5.1 illustrates how different queries, from TPC-H and TPC-DS benchmarks, scale differently with parallelism level when running on Spark with an input dataset of 100GB and each executor featuring two virtual cores and 16GB of memory. Results reveal that TPC-H q2, q9, and q21 queries exhibit strongly different scalabilities: q2 sees efficient returns of investment up to 5 executors, while q9 achieves marginal returns with no more than 11

1 A *straggler* task has been defined by previous works as a task characterized by an execution time lying above the 75th percentile value among all the tasks in the stage.

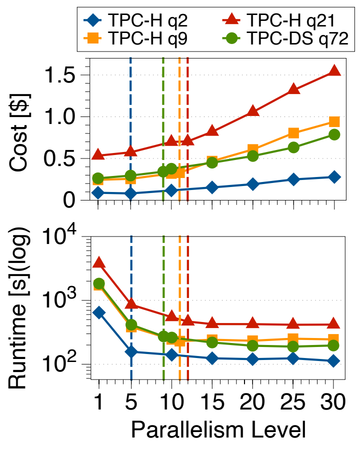
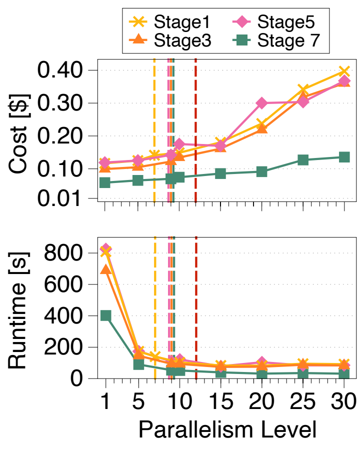


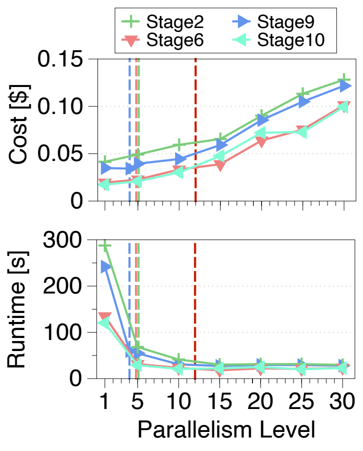
Figure 5.1: Influence of scaling parallelism level on performance and cost at per-query level. Queries runtimes (bottom) and costs (top). Saturation points are reported with dashed vertical lines.

executors, and q21 only needs 12 executors to be performance-cost efficient. Additionally, the TPC-DS q72 query shows almost the same scalability of TPC-H q9, but its saturation point occurs at a parallelism level of 9 executors. It is noteworthy that 1) The saturation point tends to shift towards higher parallelism levels as the application runtime increases, 2) Even if two applications show similar curves, slight variations in the respective saturation points can be observed. All the queries show a common trend in the curves, reflecting the fundamental characteristic of parallel computing: as the parallelism level increases, the marginal gain in performance, i.e., runtime, decreases while the cost increment rises. Focusing on queries and stages runtime, curves follow an "elbow" trend where two different regions can be distinguished: at low parallelism levels (steepy curve region), runtime exhibits a significant variation for a small change in the parallelism level, whereas at high parallelism levels (flattened curve region), increasing the parallelism level reflects in a small or no return of investment. It is noteworthy to highlight that, even though the runtime can be assumed to be a monotonically decreasing function, there could be cases where, at high parallelism levels, a third region with an increasing runtime emerges. However, in all cases, the saturation points from a performance-cost trade-off perspective, for both the user and the service provider, stay between the first and second regions.

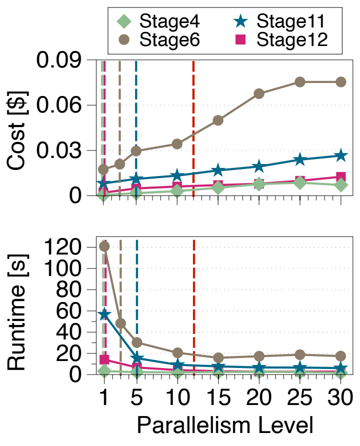
When moving to the smaller per-stage granularity, similar behaviors and remarks can be observed. Figure 5.2 shows the runtime and cost curves for the long, medium, and short running stages of TPC-H q21 query. It is interesting to note that per-stage saturation points always differ from the one found at application level, i.e., 12 executors. For example, allocating more than 7 executors to Stage1 yields marginal returns. This difference is more pronounced in medium time-consuming stages: Stage9 requires at maximum 4 executors, representing a third of the application-level saturation point. When analyzing the short running stages, the difference is even more pronounced: Stage4 and Stage12 corresponding



(a) Long running stages.



(b) Medium running stages.



(c) Short running stages.

Figure 5.2: Influence of scaling parallelism level on performance (bottom) and cost (top) at per-stage level. Red dashed vertical line reports the TPC-H q21 query saturation point, while the remaining lines reports its stages saturation points.

performance-cost saturate at 1 executor, representing a twelfth of the application-level saturation point.

Finally, we conduct a task runtime analysis for the most time-consuming stages, to investigate how the inherited overheads affect runtime when scaling the number of executors. Figure 5.3 depicts the Cumulative Distribution Function (CDF) of individual tasks for one of the most time-consuming stages, i.e., Stages1 of TPC-H q21 query. We make two fundamental observations. First, scaling the number of resources leads to a significant task runtime increase: while with 1 executor 90% of the tasks complete in less than 3.1s with a maximum of 6.9s, with 30 executors 90% of the tasks finish within 13.5s, reaching a maximum of 32.4s. When scaling-out, the shuffle operations and the garbage collection increase, degradating performance in terms of latency. Moreover, increasing the number of executors implies a higher number of tasks belonging to the first wave, which usually show a higher task runtime due to higher transfers of data shared across all stage's tasks (e.g., broadcast variables and task binaries), possible initializations and loading of additional libraries. Second, the task runtime variance significantly increases with the number of executors.

Therefore, fine-tuning the amount of resources at per-stage level is crucial to avoid under- and over-provisioning, thus preventing waste of resources for negligible performance gains. This enhances the overall resource efficiency, increasing the number of workload instances deployable on a fixed amount of resources.

To highlight the effects of varying parallelism levels on runtime and cost and to motivate the necessity of a fine-grained per-stage resource allocation, we conducted an analysis of the impact of increasing resources, i.e., number of executors and therefore number of parallel tasks, at different granularities: at application (Figure 5.1), stage (Figure 5.2), and task (Figure 5.3) levels. During this analysis, we incrementally compared the runtime and

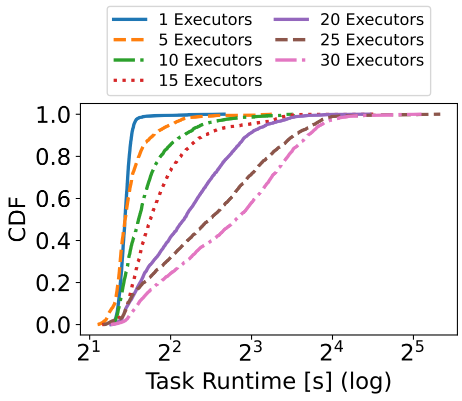


Figure 5.3: Influence of scaling parallelism level on performance at per-task level.

cost at sequential parallelism levels and identified the saturation points, or optimal scale factors, where runtime speedup turns out to be lower than the respective cost increase beyond the assigned resources.

5.3 SYSTEM DESIGN

This section details Dexter, a resource allocation manager constantly monitoring each application stage execution and automatically allocating the right amount of resources at a fine-grained level, guaranteeing efficient resource usage. Dexter is a significant departure from the majority of the works present in the literature, which rely on either reactive or predictive approaches. Instead, Dexter combines these approaches to tune the parallelism level of each application stage independently. Figure 5.4 shows the block diagram of the proposed system architecture. At a high level, Dexter continuously monitors each stage's performance and cost throughout its execution. We emphasize that Dexter is designed to be performance-cost-centric. Targeting a good Return Of Investment (ROI), if the stage performance-cost tradeoff deviates, Dexter adjusts the resource allocation, optimizing for total runtime cost. Furthermore, to increase resource utilization, once the agent assigns resources to all stages, if there are still free resources, the system reassigns them to the running stages in order of stage priority.

In what follows, we first detail Dexter's major components (Section 5.3.1), then we describe the lifecycle of an application in our optimized serverless Spark system (Section 5.3.2), and finally we discuss the system fault tolerance (Section 5.3.3).

5.3.1 *Key Components*

Dexter consists of three major components: the *historical module*, the *search module*, and the *custom executor allocation manager*. While the first two compose the Dexter's *scaling agent*,

Table 5.1: Models prediction error and error distribution.

KPI	BDS	DART	LR	BRR
MAE	**1.60**	1.64	1.61	1.62
Max Error	**6**	**6**	7	7
Error ≤ 2	72.77%	70.50%	**75.00%**	74.32%
Error ≤ 1	**56.59%**	53.18%	55.36%	56.45%

which is external to Spark, the last one extends Spark to dynamically allocate and remove executors based on the *scaling agent*'s decisions. The proposed enhanced Spark architecture, highlighting Dexter's components, is depicted in Figure 5.4.

5.3.1.1 *Historical Module*

This section details our approach for the *historical module* as an ML regression problem for estimating the initial amount of resource allocations for a given application stage. The key observation for this module is that different Spark application stages may present similar resource scalability needs (as highlighted in Section 5.2), enabling the use of ML to learn from past executions to predict the initial resources for the current stage.

The *historical module* does not require user-provided information and only relies on stage attributes known up front to the stage execution (compile-time) to optimize the stage performance cost. We aim for 1) Providing a good enough estimate so the *search module* can refine it with minimum iterations (minimizing the total runtime) and 2) Bounded response times in the order of a few milliseconds for each sample. In this section, we compare four regression models needing low computation to achieve good enough predictive performance, namely LR, BRR, BDS, and DART (briefly presented in Section 2.4).

The input to the model is a set of features describing a stage (e.g., total count of tasks, size of all RDD in bytes, and number of each RDD type). We collected a dataset of 182 input samples by running all the 22 queries composing the TPC-H benchmark, and we log features for each query stage along with the optimal number of allocated executors as determined by Algorithm 2. After cleaning constant input features, the resulting dataset contains the following six informative features: Number of tasks, Input RDDs total size, Number of MapPartitionsRDDs, Number of ShuffleRowRDDs, Number of ZippedPartitionsRDDs, and Number of FileScanRDDs.

Model error drives the model selection since a high error in initial resource allocation estimations may disturb the *search module* convergence. We split data at the query level and use Leave One Out Cross-Validation (LOOCV) to compare the error of each model with different data partitions. Each cross-validation iteration leaves one query out for testing and trains the model on the remaining 21 queries. The process is repeated 22 times and ensures the collection of diverse results for each model. As shown in Table 5.1, the evaluated models

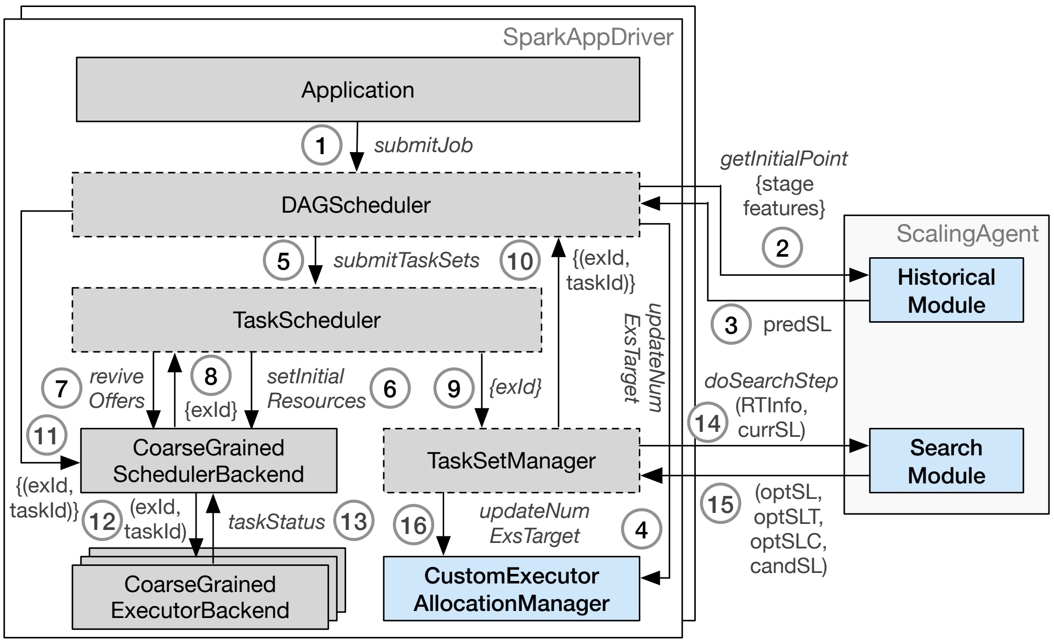


Figure 5.4: Dexter's system architecture showing new components in light blue and modified components with dotted rectangles.

demonstrate comparable performance in terms of Mean Absolute Error (MAE). Given the limited discriminative power of this metric in model selection, we further examined the maximum error and the proportion of cases with errors at or below 2 and 1. Despite LR and BRR achieve better rates of errors ≤ 2, we prefer models with lower maximum error since they make the *search module* deviate less from the optimal. Comparing BDS and DART, BDS provides slightly better percentages of errors ≤ 1. Thus, we deployed BDS to provide initial resource estimations to the *search module*, selecting the model tested on TPC-H q8 query since it represents the query with the highest number of stages.

5.3.1.2 *Search Module*

The *search module* is responsible for handling imperfect resource predictions of the *historical module* by fine-tuning each stage parallelism level. Specifically, its main target is twofold. First, it handles imperfect resource prediction of the *historical module*, potentially leading to either under- or over-estimation of resources. Second, it constantly monitors each stage's performance and cost metrics throughout its execution, and, in case the performance-cost tradeoff deviates, it adapts the number of allocated resources towards the new optimal scaling level. Algorithm 2 shows the pseudo-code of the proposed *search module*, with the top-function represented by the DOSEARCHSTEP function. Each TaskSetManager invokes the *search module* on a periodic cycle, calling the DOSEARCHSTEP function, potentially adjusting the stage allocated resources at each cycle.

During each function call, the *search module* compares the optimal solution found so far (`optSL`) with the current solution (`currSL`). More precisely, the module initially estimates the expected runtime and cost for the current scaling level (lines 22 and 23). Given `nMissTasks` missing tasks to compute for a given stage and `SL` available executors, each one computing $\mathtt{Ex}_{\mathtt{vCPUs}}$ tasks in parallel, the number of missing task waves `tWaves` is determined as:

$$\mathtt{tWaves} = \left\lceil \frac{\mathtt{nMissTasks}}{\mathtt{SL} \times \mathtt{Ex}_{\mathtt{vCPUs}}} \right\rceil \tag{5.1}$$

The module derives the expected time $\mathbb{E}[T]$ by multiplying the number of missing task waves, computed using Equation (5.1), by the average task runtime observed so far (line 2). The expected cost is then determined by adding up the two costs associated with the two different components defining each Spark executor, i.e., the allocated amount of vCPUs $\mathtt{Ex}_{\mathtt{vCPUs}}$ and memory $\mathtt{Ex}_{\mathtt{GBs}}$ resources (line 8). Specifically, to compute each cost component (lines 6 and 7), the module multiplies the expected time (expressed in hours) by the pricing per hour for the serverless Spark components, the number of resources, and the number of executors assigned to the stage. If Dexter is performing the first search step, the *search module* gets the the current optimal solution (set to the current scaling level due to reassignment of free and active executors), the optimal solution predicted by the *historical module* and the current solution, and returns a new candidate solution, along with the optimal information (lines 24-27). Otherwise, it computes the performance (line 28) and cost variation values (line 29) achieved by the current solution over the optimal one found so far. At this point, in case we are increasing resources (lines 31-37), the doSearchStep function checks if a good return of investment characterizes the current solution by comparing the performance and cost variation values (line 32). Suppose the current solution represents a better neighbouring solution than the currently known optimum solution (condition in line 32 is true). In this case, the current solution becomes the new optimum solution (line 33), and the module updates its expected runtime and cost information (lines 34 and 35). Otherwise, if the current solution does not represent an improvement, it is rejected, and the optimum solution remains unchanged. Contrarily, in case we are decreasing resources (lines 37-45), we compute the performance slowdown and the corresponding cost reduction (lines 38 and 39). If the performance decrease is smaller than the cost reduction (condition in line 40 is true), the module sets the new optimum solution to the current solution (line 41) and updates its expected runtime and cost information (lines 42 and 43). Then, based on the current optimum solution (`newOptSL`), the module picks a new candidate neighbour (line 46). When selecting the new candidate, we consider the possible reassignment of free and active executors aimed at increasing the overall system resource utilization. Specifically, when we are decreasing the current amount of allocated resources (condition in line 12 is true), we search in the lower range with respect to the current solution towards the optimal solution (lines 13-14).

Algorithm 2 Dexter's *search module* algorithm.

```
1: function computeExpTime(currSL, nMissTasks, avgTaskTime)
2:     $\mathbb{E}[T] = \left\lceil \frac{\texttt{nMissTasks}}{\texttt{currSL} \times \texttt{Ex}_{\texttt{CPUs}}} \right\rceil \times \texttt{avgTaskTime}$
3:     return $\mathbb{E}[T]$
4: end function
5: function computeExpCost(currSL, $\mathbb{E}[T]$)
6:     $\mathbb{E}[C_{CPU}] = \frac{\mathbb{E}[T]}{3600} \times \texttt{Price}_{\texttt{1vCPU/h}} \times \texttt{Ex}_{\texttt{CPUs}} \times \texttt{currSL}$
7:     $\mathbb{E}[C_{Mem}] = \frac{\mathbb{E}[T]}{3600} \times \texttt{Price}_{\texttt{1GB/h}} \times \texttt{Ex}_{\texttt{GBs}} \times \texttt{currSL}$
8:     $\mathbb{E}[C] = \mathbb{E}[C_{CPU}] + \mathbb{E}[C_{Mem}]$
9:     return $\mathbb{E}[C]$
10: end function
11: function getCandidate(newOptSL, optSL, currSL)
12:     if $|\texttt{currSL} - \texttt{optSL}| > \texttt{stepSize}$ and optSL < currSL then
13:         lowerBound = getMax(1, newOptSL - stepSize)
14:         searchRange = $[\texttt{lowerBound}, \texttt{newOptSL} - 1]$
15:     else
16:         searchRange = $[\texttt{newOptSL} + 1, \texttt{newOptSL} + \texttt{stepSize}]$
17:     end if
18:     cand = selectRandValIn(searchRange)
19:     return cand
20: end function
21: function doSearchStep(nMissTasks, avgTaskTime, optSL, $\mathbb{E}[T]_{opt}$, $\mathbb{E}[C]_{opt}$, currSL)
22:     $\mathbb{E}[T]_{curr}$ = computeExpTime(currSL, nMissTasks, avgTaskTime)
23:     $\mathbb{E}[C]_{curr}$ = computeExpCost(currSL, $\mathbb{E}[T]_{curr}$)
24:     if $\mathbb{E}[T]_{opt} == 0.0$ then
25:         candSL = getCandidate(currSL, optSL, currSL)
26:         return (optSL, $\mathbb{E}[T]_{curr}$, $\mathbb{E}[C]_{curr}$, candSL)
27:     end if
28:     perfVar = $\frac{\mathbb{E}[T]_{opt}}{\mathbb{E}[C]_{curr}}$
29:     costVar = $\frac{\mathbb{E}[T]_{curr}}{\mathbb{E}[C]_{opt}}$
30:     newOptSL = optSL
31:     if optSL <= currSL then
32:         if perfVar > costVar then
33:             newOptSL = currSL
34:             $\mathbb{E}[T]_{opt}$ = $\mathbb{E}[T]_{curr}$
35:             $\mathbb{E}[C]_{opt}$ = $\mathbb{E}[C]_{curr}$
36:         end if
37:     else
38:         $\texttt{perfSlowDown} = 1 - \texttt{perfVar}$
39:         $\texttt{costRed} = \texttt{costVar} - 1$
40:         if $\texttt{perfSlowDown} < \texttt{costRed}$ then
41:             newOptSL = currSL
42:             $\mathbb{E}[T]_{opt}$ = $\mathbb{E}[T]_{curr}$
43:             $\mathbb{E}[C]_{opt}$ = $\mathbb{E}[C]_{curr}$
44:         end if
45:     end if
46:     candSL = getCandidate(newOptSL, optSL, currSL)
47:     return (newOptSL, $\mathbb{E}[T]_{opt}$, $\mathbb{E}[C]_{opt}$, candSL)
48: end function
```

We then select and return a random value in the selected search range (lines 18 and 19). This condition is verified when, to increase resource usage, executors previously used in parent stages that have become free are allocated to the current stage. Contrarily, we randomly select a value in the upper range (lines 16). This design decision aims to increase resource utilization: if at the previous search step we have set the candidate to N executors, testing a smaller number of executors M, i.e., $M < N$, would imply leaving $N - M$ resources idle, lowering the overall resource utilization. Finally, the module returns all information related to the current optimum solution, along with the newly candidate solution (line 47).

5.3.1.3 *Custom Executor Allocation Manager*

This component dynamically allocates and removes executors based on the *scaling agent* decisions. In particular, it maintains a changing target number of executors, periodically syncing to the underlying cluster manager. Every running stage TaskSetManager checks the equivalence between the agent-suggested candidate and the current scaling level, and it requests to increase or decrease the target if necessary. More precisely, increasing the target number of executors happens when the candidate exceeds the currently allocated executors. When the target number of executors increases, the amount of requested resources to the cluster manager can vary. If there are enough available active executors, there is no need for new resources, and the *custom executor allocation manager* assigns the missing executors to the given stage from the pool of free and active executors. Otherwise, it assigns all the active executors, if any, to the stage and requests the missing ones to the cluster manager. Differently, decreasing the target number of executors happens when the candidate scaling level is more than the currently allocated resources, meaning that fewer resources are sufficient to handle the current load. In this case, even though the target number is lowered, none of the executors are killed immediately. The executor is killed only if it has been idle for a certain amount of time, by default 60s, meaning that the current running stages run efficiently with the currently allocated resources.

5.3.2 *Execution Lifecycle*

When a new application is deployed, Spark launches the application driver on one of the cluster nodes. The driver assumes the pivotal role of coordinating the entire application's execution across the worker nodes within the cluster.

Among the driver's components, the DAGScheduler is a crucial element since it represents the high-level scheduling layer. When presented with a new application (step 1), the DAGScheduler computes the DAG of stages, keeping track of RDDs, tasks, and stage outputs, and submitting stages as TaskSets. Each TaskSet contains a collection of fully independent tasks computing the same function on different data partitions. Before submitting a stage, the DAGScheduler sets its initial scale level by inspecting the *historical module* via a gRPC call, i.e., *getInitialPoint*, forwarding the stage main features (step 2), identified as detailed in Section 5.3.1.1. It receives the predicted optimal scaling level based on historical knowledge (step 3). It then sets the stage scaling level to the predicted optimal scaling level and automatically requests resources to the *custom executor allocation manager*before the TaskSet is run (step 4). In turn, the *custom executor allocation manager* checks the number of free and active executors and forwards the request for new resources to the cluster manager only if there are not enough free resources. Then, the DAGScheduler submits the stage TaskSet to the TaskScheduler (step 5).

Upon the submission of a new TaskSet with specified stage parallelism N, the TaskScheduler assesses the number of free and active executors and takes one of the following two actions: 1) Assigns N active executors to the TaskSet if there are more than N available, or 2) In case there are fewer than N active executors, it assigns the available ones (if any) and submits a request for new executors to the underlying resource manager through the CoarseGrainedSchedulerBackend (step 6). For every new schedulable TaskSet, and whenever the TaskScheduler has tasks ready for scheduling, such as upon task completion, the TaskScheduler requests the list of active executors (steps 7 and 8) to the CoarseGrainedSchedulerBackend, charged to coordinate and communicate with the underlying resource manager. If only one active TaskSet exists, the TaskScheduler makes all active executors available to that TaskSet. Conversely, if multiple active TaskSets are enqueued, the TaskScheduler allocates executors following the default stage priority order (step 9).

Every TaskSetManager schedules the tasks within a single TaskSet across the allocated executors, closely monitoring every task's progress until all tasks have been completed successfully, regardless of the number of task attempts. Specifically, it schedules the set of tasks, assigning them to the set of active executors, and returning pairs in the form of (taskId, executorId). These pairs mappings are then returned to the CoarseGrainedSchedulerBackend through the DAGScheduler (steps 10 and 11), which is notified that the subset of tasks has been started. Finally, the CoarseGrainedExecutorBackend launches tasks on executors as specified in the pairs mappings (step 12). After the task has been launched, it receives every task status update from the CoarseGrainedExecutorBackend (step 13).

Throughout the entire stage execution, using a dynamic time interval, function of the current stage average task runtime, the TaskSetManager calls the proposed *search module* through a *doSearchStep* gRPC call, exploring the search space with an additional step (step 14). The *search module* carries out this search step and returns the optimal scaling level, with its related expected runtime and cost information, as well as a new candidate scaling level (step 15). At this point, the TaskSetManager resets the stage task average runtime to zero

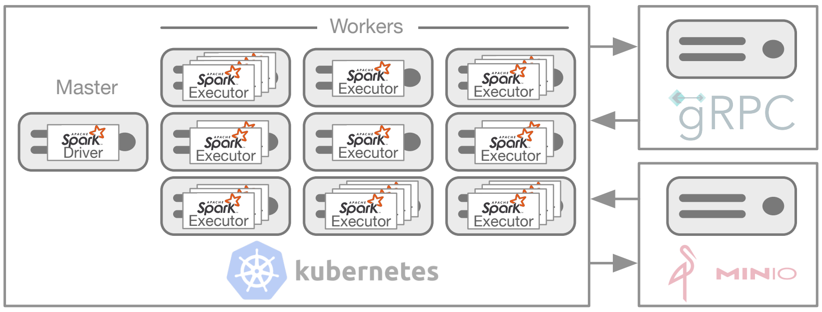


Figure 5.5: System implementation overview.

and the *search module* is not called again until all the new executors are up and running and, starting from this point in time, the optimal time interval is not expired. This allows the system to collect accurate tasks information about the tasks execution over the new amount of resources. In case the candidate scaling level, $candSL$, exceeds the current scaling level, $currSL$, it contacts the *custom executor allocation manager*, which sets the overall target number of executors accordingly and requests $currSL - candSL$ new executors to the underlying resource manager (step 16). After resetting the task average runtime, the TaskSetManager does not call again the search module until all new executors are up and running and within the optimal time interval, ensuring accurate task execution information collection across the expanded resources. This allows indirectly Dexter to consider both provisioning latencies and executors' cold start times in its decision-making.

5.3.3 *Resiliency and Fault Tolerance*

Failures in large and complex distributed systems are inevitable and can happen for many reasons, e.g., network unreliability, so fault tolerance plays a crucial role in preventing system-wide outages. Because Dexter is based on Remote Procedure Call (RPC) offering built-in features that allow clients to retry failed calls automatically, Dexter is by design resilient to failures. Therefore, the system is capable of continuing to operate in the face of outages. Specifically, RPC offers multiple patterns to overcome RPC failures: 1) Retrying a failed RPC, 2) Rerouting traffic to a healthy service, and 3) Adding a fallback path. We have instructed the Spark *scaling agent* client to automatically retry failed gRPC calls with a maximum number of call attempts of ten, including the original attempt, and a maximum backoff delay between retry attempts of five seconds.

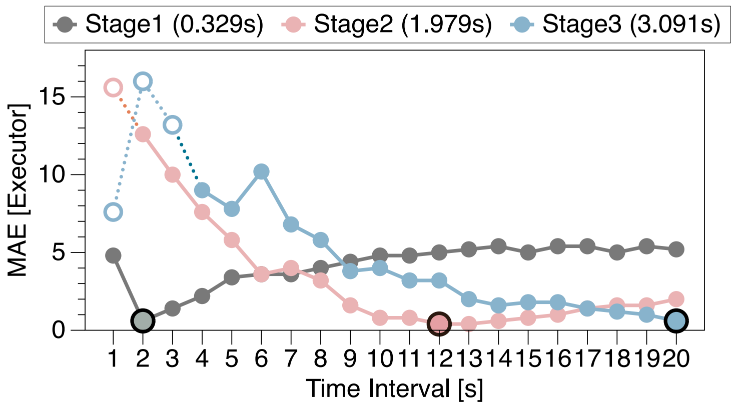


Figure 5.6: Time intervals performance of different stages, with varying average task runtime (within round brackets), when scaling intervals from 1s to 20s. Optimal values are reported with a black circle, while not valid time intervals (lower than the average task runtime) are reported with not filled circles.

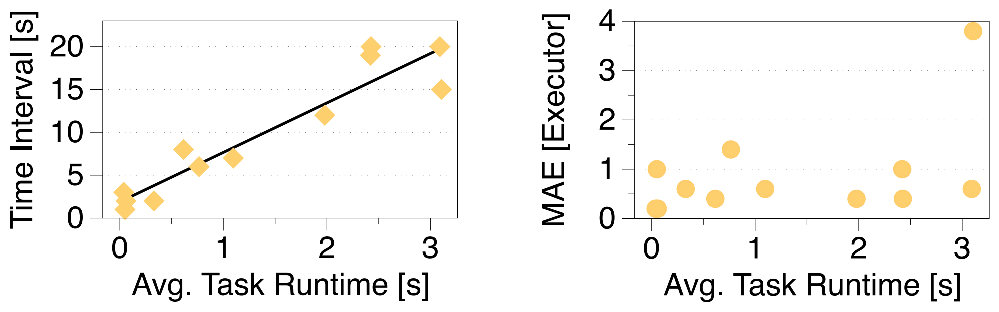
5.4 SYSTEM IMPLEMENTATION

We have implemented a complete Spark system integrating Dexter with all components detailed in Section 5.3.1. Figure 5.5 shows the overall system implementation overview. Kubernetes, representing the de facto standard for deploying and managing containerized applications in cloud environments, serves as underlying resource manager for Spark. Our *scaling agent* is implemented relying on gRPC[132], a modern, high-performance, open-source universal RCP framework, accepted to CNCF in 2017.

The *scaling agent* server side was implemented in approximately 150 lines of Python, and the client side required roughly 30 lines of Scala. Additionally, the protobuf [133] file defines the protocol buffer messages and services in around 20 lines. Adaptations to Spark's DAGScheduler, TaskScheduler, TaskSetManager, and Stage required roughly 50, 100, 150, and 50 lines of Scala, respectively. About 1500 lines of Scala were needed to implement the *custom executor allocation manager*. To store input and output data, we have deployed a MinIO server [98], a high-performance Kubernetes-native object storage tailored for large-scale systems.

5.5 SYSTEM EVALUATION

This section presents a comprehensive evaluation of Dexter. After presenting the baseline algorithms (Section 5.5.1) and the experimental setup (Section 5.5.2), we first analyse the impact of different search time intervals on the final solution accuracy (Section 5.5.3). Second, we discuss the end-to-end performance of Dexter, highlighting the performance-cost tradeoff and the obtained resource usage efficiency (Section 5.5.4). Third, we analyse the accuracy



(a) Optimal time intervals for varying average task runtimes and their linear relationship.

(b) Minimum MAEs achieved by the optimal time intervals.

Figure 5.7: Optimal time intervals, achieving minimum MAEs, when varying the task runtime.

of our results by picking the real optimal values and comparing them with our results (Section 5.5.5). Finally, we discuss Dexter's adaptability to unseen workloads (Section 5.5.6) and overheads (Section 5.5.7).

5.5.1 *Baselines Algorithms*

During our evaluation, we compare Dexter's performance to the following baseline algorithms. First, Spark's default dynamic allocation scheduling stages in FIFO manner, where stages are enqueued based on their order of arrival, and they get priority on all available resources while they have tasks to launch. Second, Spark's default dynamic allocation with FAIR scheduling gives an equal share of resources to the different runnable stages in a round-robin fashion.

5.5.2 *Experimental Setup*

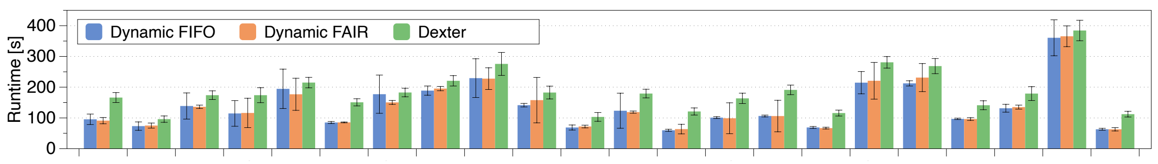
The system has been evaluated on an on-premise cloud deployment to prevent us from benchmarking cloud vendors' specific environments. To run Apache Spark v3.3.0, modified as described in Section 5.4, we use a virtualized Kubernetes v1.26 cluster, consisting of one master and 9 worker nodes. The master runs in an Ubuntu 22.04 VM with 16 vCPUs, 120GB of memory, and 500GB of disk, while the workers run in an Ubuntu 22.04 VM with either 18 or 32 vCPUs, 120GB of memory, and 400GB of disk. While each worker node can accommodate up to 7 executors featuring 4 vCPUs, 16GB of RAM, and 32GB of disk, the master node is the control plane, and it accommodates the driver featuring 4 vCPUs, 16GB of memory, and 128GB of disk. The Kubernetes cluster is mapped on 10 physical nodes residing in the same rack and featuring either an Intel®Xeon Silver 4114 CPU running at 2.20GHz or an Intel®Xeon E5-2630 v4 CPU running at 2.20GHz.

We configured Spark to launch executors with the aforementioned amount of resources to mimic the default configuration of serverless Spark environments. Thus, we can deploy a maximum of 54 executors on the available worker nodes due to a lack of further resources. In the remainder of this section, we consider the round number of 50 executors as the upper bound resource limit. We deploy A MinIO server *v2023-10-14T01-57-03Z*, a widely used high-performance object storage, as shared storage. It runs bare-metal on a node featuring an Intel®Xeon E5-2620 CPU running at 2.00GHz, interfacing with two 1.6TB Intel®DC P3608 SSDs through NVMe. In-memory key-value stores, such as Redis and Memcached, are not considered since they break one of the serverless advantages by requiring users to select instance types in terms of network, compute, and memory resources to satisfy their application requirements. The proposed scaling agent is a gRPC server v1.59.2, implemented in Python v3.11.1 using *pickle-mixin* v1.0.2 and *lightgbm* v4.1.0, and it is deployed in an Ubuntu 22.04 VM with 16 vCPUs, 32GB of memory, and 100GB of disk. The VM runs on an Intel®Xeon E5-2630 v4 CPU @2.20GHz. The VMs and the resource allocation agent machine are synchronized in the millisecond range. Physical nodes are connected through a 10Gbps Brocade VDX6740 network switch. The experimental evaluation reports average results computed over 10 sequential runs for reliability and considers the current IBM Analytic Engine pricing plan of 0.154 USD/Virtual processor core hours and 0.0146 USD/Gigabyte hours.

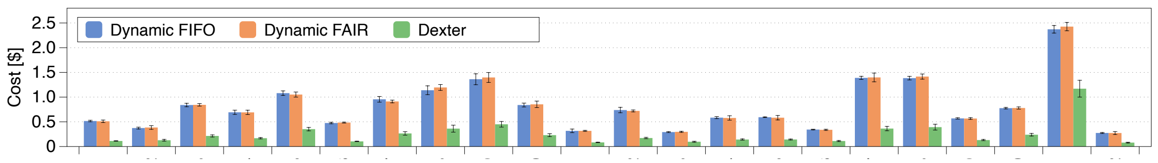
5.5.3 *Dynamic Time Interval Analysis*

In this analysis, we want to assess the influence of varying time intervals on the search accuracy and determine the optimal time interval for invoking Dexter's *search module*, denoted as the time window between two consecutive calls. This analysis is based on the observation that the use of static intervals, was sub-optimal within the context of this work. The average task runtime is a key feature in exploring the search space; therefore, it is important to select a time interval large enough to guarantee the execution of enough tasks to get accurate information. In addition, since the average task runtime shows significant variability not only across stages but also within a given stage due to increasing parallelism overheads, as shown in Section 5.2, there is the need for an adaptive time interval (varying accordingly).

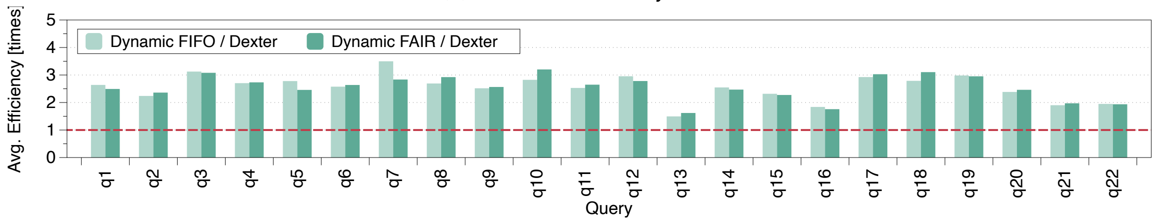
The evaluation of the optimal time interval considers different task runtimes, ranging from some milliseconds up to some seconds, and different statically defined time intervals, varying from 1s up to 20s with a 1s increment. Figure 5.6 illustrates the MAEs for three distinct stages characterized by an average task runtime of 0.329s, 1.979s, and 3.091s, respectively. Notably, each stage exhibits a different optimal time interval - 2s, 12s, and 20s - achieving an error of 0.6, 0.4, and 0.6 executors, respectively. To define the dynamic time interval as a



(a) Queries end-to-end runtimes.



(b) Queries monetary costs.



(c) Dexter's efficiency, in terms of total runtime cost, over the two baselines (red dashed horizontal line).

Figure 5.8: End-to-end performance over the two baselines when running TPC-H benchmark.

function of the average task runtime, we include a higher number of stages ranging from some milliseconds up to roughly 3s. Figure 5.7a and Figure 5.7b depict the optimal time intervals and their corresponding MAEs, respectively. Although optimal time intervals result in MAEs lower than 1.4 executors, there is a notable exception, where the MAE reaches 3.8 executors. Closer analysis reveals that the executors in this stage are affected by multiple long cold starts; therefore, the stage does not have enough time to scale appropriately. In contrast, another stage with an almost equivalent average task runtime shows a MAE of 0.6 executors. The relationship between the average task runtime and the optimal time interval, i.e., the one achieving minimum MAE, can be modeled by a linear regression model, as highlighted in Figure 5.7a. Specifically, given an average task runtime ATR, the dynamic time interval DTI can be modeled as the linear relationship defined as:

$$DTI(ATR) = 5.77 \times ATR + 1.89 \tag{5.2}$$

In the following, each stage TaskSetManager calls periodically Dexter with a dynamic time window defined following Equation (5.2).

5.5.4 *End-to-End Performance-Cost Evaluation*

In this evaluation, we assess the performance-cost efficiency, and effectiveness of Dexter's improved resource utilization compared to the default Spark dynamic resource allocation

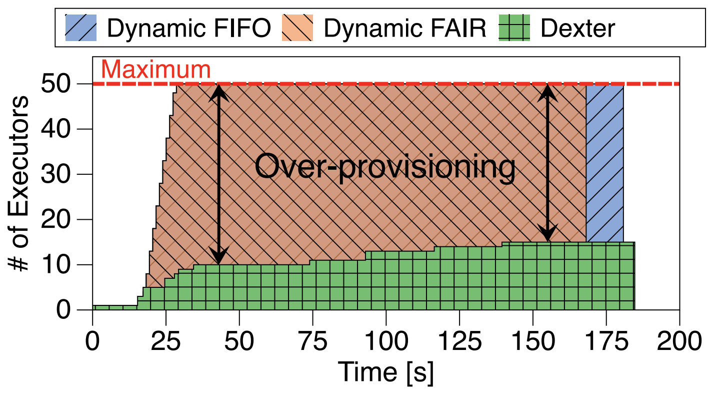


Figure 5.9: Resource allocation patterns for TPC-H q7 query.

strategies assigning resources in a FIFO and FAIR fashion. Differently from other works in the literature, focusing on optimizing the sharing of a fixed amount of resources among a set of workloads, we study Dexter's performance-cost tradeoff under the assumption that every incoming workload can independently scale. This assumption stems from the serverless paradigm, where each application can leverage a theoretically unlimited pool of resources. In our custom implementation of serverless Spark, the gRPC server and the MinIO object storage represent the two potential primary system bottlenecks. While the gRPC server can efficiently deal with high loads, distributing the requests optimally across a set of server instances, object storage has been proved to be a practical but powerful approach for serverless data analytics when considering optimized versions [37, 134–137]. Therefore, since implementing an optimized object storage is out of the scope of this work, in this evaluation, we assume that neither the gRPC server nor the MinIO object storage imposes constraints on system performance.

We assess the effect of dynamically adjusting the amount of resources at each stage by adopting a hybrid approach that combines predictive with reactive approaches. Precisely, for each query, we measure two primary metrics: 1) The end-to-end runtime, defined as the temporal span between the timestamp preceding the first executor request and the timestamp immediately following the shutdown of the last executor (including executors' cold start time), and 2) The associated monetary cost, calculated following the serverless paradigm, accounting solely for executors uptimes while not accounting for cold start. To derive an unbiased evaluation of the overall performance-cost tradeoff, we introduce a single composite metric denoted as *efficiency*, defined as:

$$Efficiency = \frac{Runtime_{Baseline} \times Cost_{Baseline}}{Runtime_{Dexter} \times Cost_{Dexter}} \tag{5.3}$$

By incorporating both runtime and cost variables, the efficiency metric avoids biased solutions by reflecting any variation in terms of latency, cost, or a combination of the two. Figure 5.8 illustrates the results obtained from benchmarking the TPC-H dataset.

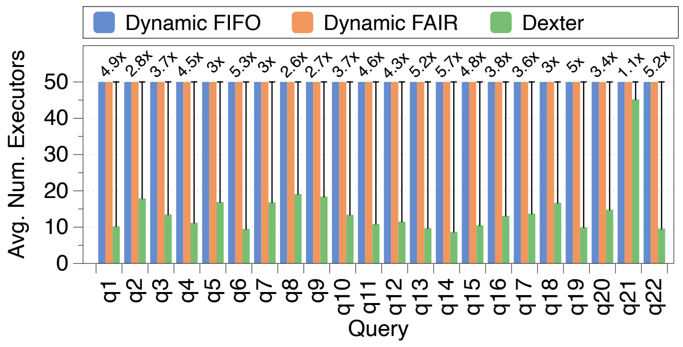


Figure 5.10: Resource provisioning of the two baselines and Dexter, highlighting the resource savings achieved by Dexter.

Notably, as Dexter exhibits a less aggressive resource scaling compared to the two baselines, the end-to-end runtime achieved by Dexter is generally comparable or higher in all queries, increasing the latency by a factor ranging from 1.03× to 2.02× (median: 1.45×, interquartile range: 1.27× to 1.68×), and from 1.05× to 1.90× (median: 1.47×, interquartile range: 1.24× to 1.72×) for the two baselines, respectively. Differently, from the pure monetary cost perspective, Dexter always costs significantly less than the two baselines, achieving a cost reduction ranging from 2.02× to 4.59× (median: 3.79×, interquartile range: 3.15× to 4.13×) and from 2.07× to 4.65× (median: 3.71×, interquartile range: 3.19× to 4.07×) on each baseline. For all analyzed queries, Dexter demonstrates improved performance-cost efficiency, ranging from 1.50× to 3.50× (median: 2.75×, interquartile range: 2.41× to 2.93×) with respect to the FIFO baseline and from 1.62× to 3.20× (median: 2.66×, interquartile range: 2.45× to 2.88×) for the FAIR baseline. An example of resource allocation patterns for the TPC-H q7 query, showing the maximum efficiency, are illustrated in Figure 5.9. Dexter has a more conservative approach on scaling resources, achieving comparable or slightly higher runtime, while using 3.33× fewer resources.

Scaling at fine-grained granularity, i.e., at the per-stage level, allows to allocate just the right amount of resources for each stage, thereby leaving available resources for concurrent workload executions. The resource usage of each considered solution is depicted in Figure 5.10. In contrast to the two baselines, which always use the maximum of 50 executors, Dexter consistently achieves resource savings ranging from 1.13×to 5.71× (equivalent to 4.8 and 41.3 free executors on average, respectively). Figure 5.9 presents the resources allocated by the three analyzed solutions in the case of TPC-H q7 query, reaching a resource saving of 3.33×, which translates in using only 15 executors, thus leaving 35 executors free.

Finally, we analyse the number of deployable instances under various constraints, leveraging the three different resource allocation strategies. We constrain three aspects: 1) Amount of available resources, 2) Time availability, and 3) Monetary budget. Notably, on query q14, representing the query achieving the most significant resource saving, Dexter enables the deployment of $\lfloor \frac{100}{9} \rfloor = 11$ concurrent instances with 100 executors, compared to a maximum of $\lfloor \frac{100}{50} \rfloor = 2$ concurrent instances achievable by the baselines, which translates in $5.5\times$ more deployable workload instances. In the worst case with query q13, our solution deploys $\lfloor \frac{3600}{59.81} \rfloor - \lfloor \frac{3600}{120.99} \rfloor = 60 - 29 = 31$ $(2.02\times)$ and $\lfloor \frac{3600}{63.99} \rfloor - \lfloor \frac{3600}{120.99} \rfloor = 56 - 29 = 27$ $(1.90\times)$ less instances compared to the baselines. Therefore, when focusing only on the runtime metric, Dexter deploys fewer instances than the two baselines. This is expected since we are scaling resources conservatively.

Lastly, when considering a monetary budget constraint of 100$, due to the significantly higher cost efficiency, Dexter yields a considerable increase in deployable instances. For the most cost-efficient q1 query, Dexter enables $\lfloor \frac{100}{0.1124} \rfloor - \lfloor \frac{100}{0.5161} \rfloor = 889 - 193 = 696$ $(4.60\times)$ and $\lfloor \frac{100}{0.1124} \rfloor - \lfloor \frac{100}{0.5108} \rfloor = 889 - 195 = 694$ $(4.55\times)$ more instances compared to the baselines.

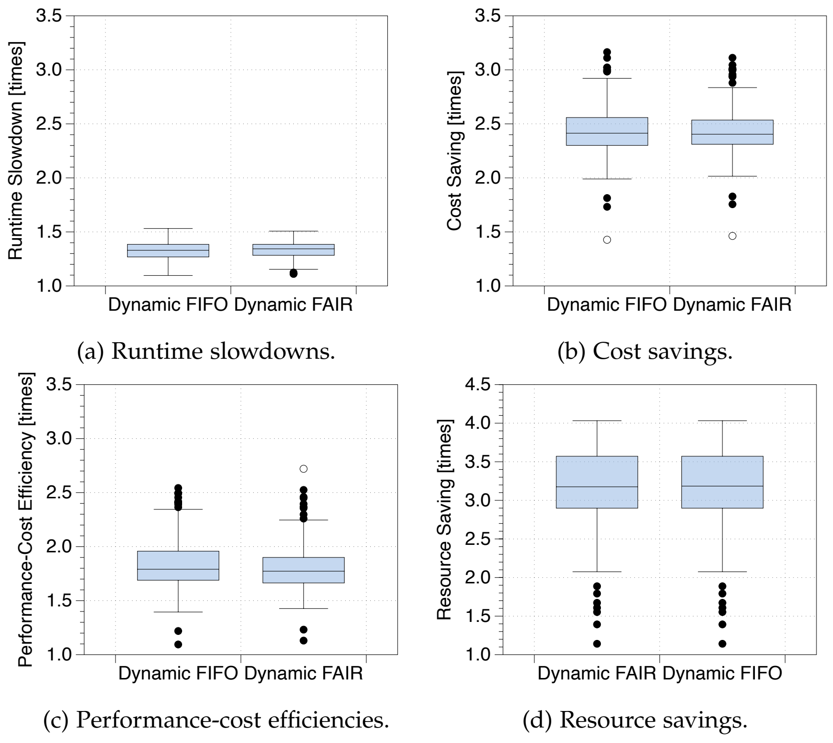
In summary, our fine-grained per-stage scaling approach significantly outperforms the baselines when constraining the amount of resources and the monetary cost. In contrast, the baselines perform better when constraining time availability since they scale more aggressively.

5.5.5 *Parallelism Accuracy Analysis*

While the individual accuracy of the *historical* and *search module* has been provided in Section 5.3.1.1 and Section 5.3.1.2, we now evaluate the overall system accuracy by analyzing the absolute error distribution. The analysis reveals a median absolute error of 4, with the interquartile range spanning from 1 to 8 executors. The minimum and maximum absolute error are 0 and 18.5 executors, respectively. The reason behind such a maximum absolute error is due to the free executors' reassign policy we apply, aiming at increasing resource utilization. Under this policy, stages get all free resources in priority order. However, this reassignment speeds up the stage completion, hindering Dexter's efficient resources scaling and preventing the search process from converging towards the actual stage optimal parallelism. Given this distribution of absolute errors, we can conclude that Dexter gives good accuracy while achieving significantly higher efficiency, as reported in Section 5.5.4.

5.5.6 *Input Workload Sensitivity Analysis*

We now conduct a sensitivity analysis on the robustness of Dexter when presented with new unseen input workloads, simulating a workload change. The research question we pose is: *Is our agent trained on the TPC-H benchmark capable of achieving efficient resource allocations for previously unseen Spark workloads?* If the agent provides inefficient resource allocation for unseen queries, the work would lack of generality. To address this question, we validate Dexter's generality by benchmarking TPC-DS and Terasort on 100GB input data. While TPC-



(a) Runtime slowdowns.

(b) Cost savings.

(c) Performance-cost efficiencies.

(d) Resource savings.

Figure 5.11: Dexter's (a) runtime performance, (b) cost savings, (c) performance-cost efficiencies, and (d) resource savings when running the 99 TPC-DS queries with 100GB input data.

DS involves more complex queries compared to TPC-H ones with advanced SQL features, extensive filter predicates, and diverse data scans, Terasort represents a real-world workload used by most of the existing serverless analytics systems [37, 89, 135, 136, 138]. Furthermore, given the notable diversity in datasets often observed in serverless environments, we extend our analysis to include Dexter's adaptability to dataset variations.

Regarding TPC-DS, Figure 5.11 depicts the distribution of Dexter's runtime performance, cost savings, performance-cost efficiencies, and resource savings in comparison to the two baselines. As shown in Figure 5.11a, on the one hand, Dexter presents a median runtime slowdown of 1.33× (interquartile range: 1.27× to 1.38×) compared to the dynamic FIFO baseline. Similarly, when compared to dynamic FAIR baseline, Dexter exhibits a median runtime slowdown of 1.34× (interquartile range: 1.28× to 1.38×). On the other hand, as shown in Figure 5.11b, Dexter demonstrates notable cost savings compared to the two baseline. More precisely, when compared to the dynamic FIFO baseline, Dexter enable a median cost saving of 2.41× (interquartile range: 2.30× to 2.56×). High cost savings are achieved also in the case of dynamic FAIR baseline, where Dexter allows a median cost saving of 2.40× (interquartile range: 2.31× to 2.54×). When considering the unbiased composite performance-cost efficiency metric presented in Figure 5.11c, Dexter achieves a median efficiency increase of 1.79× (interquartile range: 1.69× to 1.96×). When compared with the dynamic FAIR baseline, Dexter presents a median efficiency increase of 1.77× (interquartile

Table 5.2: Runtime and cost of the five most compute-intensive queries from TPC-DS benchmarks.

Query	Dynamic FIFO			Dynamic FAIR			Dexter		
	Runtime[s]	Cost[$]	#Exs	Runtime[s]	Cost[$]	#Exs	Runtime[s]	Cost[$]	#Exs
q14a	140.89	0.5339	50	131.47	0.5521	50	169.56	0.2324	22.5
q23b	153.92	0.6366	50	139.18	0.6283	50	184.91	0.2953	23.3
q67	189.34	0.6717	50	188.09	0.6482	50	207.68	0.2546	14.9
q72	133.94	0.5091	50	124.24	0.4925	50	154.45	0.2459	20.5
q78	126.91	0.5666	50	125.53	0.5621	50	170.72	0.2138	16.7

range: $1.67\times$ to $1.90\times$). While increasing significantly performance-cost efficiencies, Dexter allows a median resource saving of $3.40\times$ (interquartile range: $2.99\times$ to $3.73\times$), as presented in Figure 5.11d. We selected the top 5 most time-consuming TPC-DS queries, namely, q14a, q23b, q67, q72, and q78. Table 5.2 shows the end-to-end runtime and cost when running the five most time-consuming queries with the two considered strategies. The corresponding performance-cost tradeoff and the achieved resource savings are shown in Table 5.3. While Dexter slightly increases the runtime by a factor ranging from $1.10\times$ to $1.34\times$, it achieves monetary savings ranging from to $2.08\times$ to $2.65\times$. Similarly to what we have done with TPC-H queries in Section 5.5.4, we analyze the benefit in terms of additional deployable instances under various constraints. With 100 executors, the proposed solution deploys from $2.15\times$ up to $3.36\times$ more query instances than the baseline dynamic resource allocation with FIFO scheduling, resulting in a maximum of $\lfloor \frac{100}{19.9} \rfloor = 6$ deployable instances instead of $\lfloor \frac{100}{50} \rfloor = 2$. When the time window is fixed at one hour, our solution deploys, on average, $1.20\times$ fewer instances, i.e., 4 less instances, compared to the baseline. Finally, when constrained by a monetary budget of 100$, Dexter deploys $2.37\times$ more instances of average, which translates in 233 more workload instances.

When running Terasort, Dexter demonstrates a remarkable enhancement in cost efficiency, achieving a cost reduction of up to $2.39\times$. Moreover, the performance-cost efficiency is significantly increased by up to $1.93\times$, alongside notable resource savings of up to $2.3\times$. These improvements are achieved while observing a slight increase in runtime, with a maximum factor of $1.26\times$.

Dataset changes affect runtime and cost, which are considered in Dexter's decision-making process. This is evidenced by differences in TPC-H and TPC-DS benchmarks, using different datasets (8 and 22 tables), with TPC-DS queries exhibiting significant variability in data scan ranges. Moreover, experiments with Terasort further underscore Dexter's adaptability to dataset changes, including shifts in data models, such as from structured relational data models to semi-structured ones.

Our sensitivity analysis underlines the Dexter adaptability to both workload and dataset changes, establishing its usefulness in optimizing resource allocation for previously unseen workloads.

Table 5.3: Efficiency and resource savings evaluation of the five most compute-intensive queries from TPC-DS benchmarks.

Query	Dynamic FIFO / Dexter		Dynamic FAIR / Dexter	
	Efficiency	**Resource Saving**	**Efficiency**	**Resource Saving**
q14a	$1.91\times$	$2.22\times$	$1.81\times$	$2.22\times$
q23b	$1.79\times$	$2.15\times$	$1.61\times$	$2.15\times$
q67	$2.41\times$	$3.36\times$	$2.28\times$	$3.36\times$
q72	$1.80\times$	$2.44\times$	$1.60\times$	$2.44\times$
q78	$1.97\times$	$2.99\times$	$1.93\times$	$2.99\times$

5.5.7 *System Overhead Evaluation*

In this evaluation, we discuss the system overhead in terms of time necessary to fit the *historical module* model and to perform the model inference when determining the initial stage parallelism level. We also discuss the time taken to perform one search step when fine-tuning the parallelism level during stage computation. The reported times in the remainder of this evaluation represent end-to-end latency: we compute the latency as the difference between the registered timestamp prior to each agent call and the timestamp immediately following the reception of the corresponding response.

Regarding the initial parallelism level prediction, the time needed to fit the parameter model utilizing BDS on a batch of 182 stages known at compile-time is, on average, 19.12ms with a single thread. Aiming at understanding the impact of training delay in a multi-tenant cloud-like scenario, with a dataset scaled up to 18,200 rows (two orders of magnitude larger), the training would take an average of $19.12\text{ms}\times 10^2 = 1,912\text{ms} \approx 2\text{s}$.

We want to emphasize that the training delay is small enough to seamlessly execute model retraining on the fly, even at per-minute granularity. The time required for the one-time per-stage prediction of the initial parallelism level averages to 15.87ms. For a single row, the prediction complexity of using BDS amounts to $O(n)$, where n is the number of decision stumps, since each decision stump is evaluated in constant time c ($O(n * c) = O(n)$). In the *search module*, the average delay for each search step amounts to 5.65ms. Since the overall search time for a given stage strictly depends on the number of performed search steps, which in turn depends on the average task runtime, to assess the impact of search delay we consider the three different stages discussed in Section 5.5.3, featuring an average task runtime of 0.329s, 1.979s, and 3.091s, respectively. During stage execution, the agent is invoked once, twice and four times, translating into a median total search latency of 5.5ms, 6.5ms, and 26.5ms, accordingly.

Therefore, we assert that Dexter's performance-cost effective resource allocation imposes negligible overhead on the final stage execution time.

5.6 RELATED WORK

Resource allocation in modern cloud data centers has been extensively investigated in the literature. However, directly applying existing work on VMs to serverless Spark is challenging. The key factors preventing this translation are: 1) The number of instances spawned is significantly higher in serverless than the one for VMs (10s for VMs, 100s for serverless), and 2) Fitting the executors following a consolidating strategy goes against the concept of hardware disaggregation, at the base of the serverless paradigm. Existing cloud service providers' production systems, such as Google Borg [139], and open source systems, such as Google Omega [140], typically provide configuration parameters to control the resource allocation process. Since the manual configuration setting requirement breaks the free-of-management serverless principle, we do not consider these solutions in this chapter.

Focusing on the resource allocation problem for big data analytics frameworks, in the following, we discuss the work related to Dexter, classifying existing solutions into three main families: analytical model-based, heuristic-based, and ML-based schedulers. A detailed summary of the existing work is presented in Table 5.4.

5.6.1 *Analytical Model-based Schedulers*

OptEx [123] builds a closed-form performance model that decomposes the application completion time into smaller, logically distinct phases, leveraging profiling information. Based on the performance models, the authors identify the application's cost-optimal resource allocation required to satisfy a specific deadline with the minimum number of resources. Justice [124] estimates the resources that applications need to meet deadlines from historical application execution traces and implements admission control and resource allocation. It also automatically adapts to workload variations to provide sufficient resources for each job to meet the deadline.

5.6.2 *Heuristic-based Schedulers*

Quasar [125] is a cluster manager that maximizes resource utilization while meeting user-defined performance and QoS high-level specifications. Given the cluster state at any point in time, combining a small profile of information from the application itself with historical data on previously scheduled applications, it applies collaborative filtering to determine the least amount of available resources necessary to meet such user specifications. In [51], the authors present Morpheus, a system designed to automatically derive periodic application SLOs and resource models from resource-usage historical data, relying on recurrent reservations and packing algorithms, to place containers, minimizing cluster resource usage cost. Moreover, Morpheus dynamically adapts to changing conditions at runtime and reprovisions applications accordingly to mitigate inherent performance variance due to changes, for example, in application input or failures. AutoPath [126] leverages the application DAG information to identify parallel paths and adaptively allocates computing resources to each path during runtime based on the estimated demand. Resource allocation is triggered every time a new stage is launched. The authors estimate the total execution time of each path by analyzing runtime statistics of the average transformation time between RDDs in every parallel path, adjusting resources accordingly. 3Sigma [127] observes the importance of relying on full distributions of relevant runtime history for each job, rather than just a point estimate derived from it, to make more robust decisions. Based on this observation, the authors propose a cluster scheduling framework that employs Mixed-Integer Linear Programming (MILP) for bin-packing applications on available resources. AdaptiveConfig [128] is a runtime configuration tuning framework for cloud cluster schedulers. The framework automatically adapts to workload changes by comparing performance discrepancies of different configurations and identifying the best configuration to satisfy the latest mix of jobs' resource demands.

5.6.3 *ML-based Schedulers*

ML models interact with the environment to learn the optimal behavior without a priori knowledge and can solve complex problems, such as resource allocation, accounting with uncertainty. Decima [120] makes use of Deep Reinforcement Learning (DRL) to automatically learn application-specific scheduling algorithms. Specifically, it employs a graph neural network to convert the states' information concerning the application DAG and cluster nodes' status into embedding vectors. These embedding vectors are then used as input to a policy network to concurrently select the next stage to schedule and the related application parallelism level. Noteworthy is the authors' design decision to abstain from exploiting fine-grained stage-level parallelism. This decision stems from the need to trade-off between the granularity of control and model training complexity. By restricting parallelism control at the application level, Decima reduces the action space of scheduling policies that must be explored and optimized during training. AutoToken [121] predicts the peak resource usage of recurring big data applications. It first groups applications based on their recurring signatures and then makes use of different per-signature linear models, one for each recurring job group identified, to scale resources for each application appropriately. AutoExecutor [39] predicts the executor counts based on a parametric model analyzing the price-performance trade-off, trained using Random Forest (RF) regression algorithms. The features used for the parametric model include characteristics of both the application

Table 5.4: Overview of related work and comparison with this work.

	Work	Year	Objectives		Cluster		All Workload	DAG Knowledge	Deadline Awareness	Profiling Free	Training	Frequency	Exp. Setup	Granularity Level	Parallelism Level
			Perf.	Cost	Nodes	Unbound Size									
A.M.	[123]	2016	✓	✓	heterogeneous	✓	✗	✗	✓	✗	-	up front	cluster	application	static
	[124]	2017	✓	✗	homogeneous	✗	✓	✗	✓	✗	-	up front	simulation	application	static
Heuristics	[125]	2014	✓	✓	heterogeneous	✗	✓	✗	✓	✗	-	up front, perf. not met	cluster	application	dynamic
	[51]	2016	✓	✓	homogeneous	✓	✗	✗	✓	✓	-	up front, perf. not met	cluster	application	dynamic
	[126]	2017	✓	✗	homogeneous	✗	✓	✓	✗	✓	-	up front	cluster	stage	dynamic
	[127]	2018	✓	✗	heterogeneous	✗	✓	✗	✓	✓	-	periodic (fix) [s]	cluster, simulation	application	dynamic
	[128]	2019	✓	✗	homogeneous	✗	✗	✓	✗	✓	-	up front, periodic (fix) [m]	clsuter	stage	dynamic
ML Models	[120]	2019	✓	✓	homogeneous	✗	✓	✓	✗	✓	offline	set runnable stages changes	cluster, simulation	application	dynamic
	[121]	2020	✓	✗	homogeneous	✗	✗	✗	✗	✗	offline	up front	cluster	application	static
	[39]	2021	✓	✓	homogeneous	✗	✗	✗	✓	✓	offline	up front	cluster	application	static
	[122]	2021	✓	✓	homogeneous	✓	✓	✗	✓	✗	online	up front	cluster	application	static
	This	2023	✓	✓	homogeneous	✓	✓	✓	✗	✓	offline	periodic (dyn) [s]	cluster	stage	dynamic

plan and the application inputs. Differently from the majority of the works present in the literature using runtime statistics and similarly to Dexter, AutoExecutor relies only on features available at compile-time and before running the application. Differently from the majority of the works present in the literature using runtime statistics, AutoExecutor relies only on application features available at compile-time. In a similar fashion, TASQ [122] also centers its predictions only on compile-time application features. It leverages these features to train and use graph neural networks to predict the application performance characteristic curve trend and the optimal application amount of resources. Autoexecutor, TASQ, and Dexter share a common characteristic: they rely on compile-time features, known before running the application. The authors introduce a novel data augmentation technique to tackle training data sparsity. This technique effectively simulates the entire application performance characteristic curves using solely the information from a single application run with a single resource instance.

5.7 SUMMARY

This work investigates the effect of resource scaling on execution duration and associated monetary cost, highlighting critical importance of setting the optimal scaling level at the smallest fine-grained granularity, specifically at the per-stage level. This implies allocating more resources to highly parallel stages with large-chunk input data, while limiting resource for stages with little inherent parallelism or processing small-chunk input data. To this aim, we present Dexter, a robust resource allocation manager designed to continuously monitor application execution, automatically assigning an optimal amount of resources at a fine-grained level to maximize resource utilization. Notably, when the stage performance-cost tradeoff deviates during workload execution, Dexter dynamically adjusts resource allocation to balance the performance-cost tradeoff, minimizing the overall cost while providing acceptable runtime performance. Our experimental evaluation shows that on different analytics workloads, compared with the default serverless resource allocation, our solution achieves a significant cost reduction of up to $4.65\times$, while improving resource efficiency up to $3.50\times$ and substantially reducing resource saving up to $5.71\times$. This work shows that dynamically adjusting resources to meet the application's needs at fine-grain in real-time ensures greater resource allocation flexibility, as well as higher resource efficiency. Consequently, we prove that fine-grained per-stage resource allocation significantly enhances the performance of executing complex data-intensive analytics workloads in serverless environments.

5.8 PUBLICATIONS

The contents of this contribution are summarized in the following publication:

– *[Submitted]*. **Anna Maria Nestorov**, Diego Marron, Alberto Gutierrez-Torre, Chen Wang, Claudia Misale, Alaa Youssef, David Carrera, and Josep Lluís Berral. "Dexter: A Performance-Cost Efficient Resource Allocation Manager for Serverless Data Analytics". In: *25th International Middleware Conference (Middleware '24)*.

6 SUMMARY AND CONCLUSIONS

In recent years, serverless computing has gained significant attention from both academia and industry. Its fundamental principles, including transparent elastic resource scaling allowing massive scalability, 'pay-as-you-go' cost model with fine-grained charging granularity, and hassle-free management, have attracted many tenants in today's cloud environments. However, directly using serverless platforms in more complex data-intensive analytics applications lead to highly inefficient executions and its adoption comes with inherent challenges. Given the resource-intensive and highly parallel nature of these applications, these challenges pose significant obstacles for these applications to benefit from the elasticity and the virtually unlimited scalability offered by serverless platforms. The dynamic availability of resources in modern serverless systems further introduce several challenges, making an efficient usage of resources a key requirement.

In this thesis, we address the aforementioned challenges by exploring several research paths. Firstly, we comprehensively characterize a proper use case data-intensive analytics workload and access its performance when executed within serverless platforms. By deeply studying this use case workload we identify the most critical factors preventing the efficient execution these complex workloads to fully benefit from serverless environments. In particular, we find that communication via shared object storage and application-level scalability are two essential factors limiting the efficient execution of data-intensive analytics workloads within serverless platforms. Building upon this findings, we introduce a data forwarding system for direct inter-function communication, called Floki, to overcome the high data sharing latencies through object storage. Our results demonstrate that Floki significantly reduces inter-function communication latencies, thereby enhancing the end-to-end time performance and improving overall resource utilization efficiency. Finally, we examine different scalability granularities, and we propose Dexter as a solution to fine-tune at real-time the scaling levels at the smallest application granularity. Dexter enables more efficient usage of resources, in terms of total runtime cost, fine-tuning resource allocations at the smallest granularity, prioritizing simplicity, generality, and ease of understanding. It dynamically adapts to performance-cost changes by automatically adjusting the allocated resources.

In this thesis we demonstrate that it is possible to execute efficiently complex data-intensive analytics workloads, traditionally deployed in managed cloud clusters, in serverless platforms.

6.1 SUMMARY OF RESULTS

The main results and contributions of this thesis can be summarized as follows.

6.1.1 *Workload Scalability and Performance Analysis*

Thanks to the flexibility of the proposed model, we accurately estimate the performance impact on a use case workload, the Google's Tesseract OCR engine, of the following key factors: task granularity and concurrency, data locality, resource allocation, and scheduling policies. Concerning task granularity and concurrency, our investigation reveals distinct saturation points for the three workload parallelisms, beyond which increasing the assigned resources yields negligible performance gains. When varying the task granularity, the coarse-grained deployment outperforms the fine-grained deployment for similar configurations, particularly in remote scenarios, achieving a performance boost of up to 1.55×. However, when considering all application parallelisms and fully unrolled deployments, fine-grained executions demonstrate higher speedups, moving from 4.57× to 6.79× locally and from 3.44× to 4.24× remotely. This performance increase comes at the cost of significantly higher number of tasks, rising from 32 to 812, and higher I/O, growing from 1.4GB to 15.6GB. Regarding resource allocation, a more accurate resource allocation strategy increases the number of deployed coarse-grained workload instances from 38 to 49, fitting 11 more instances on the same amount of resources. Finally, we show that the data-sharing time for such complex workloads can be reduced by a factor of up to 4.32× by efficiently placing functions considering data locality, thereby minimizing the communication over the network, and grouping functions with the same parallelisms.

6.1.2 *Direct Inter-Function Communication Enablement*

We overcome the shared remote storage bottleneck problem by proposing Floki, a system that proactively enables point-to-point data sharing, reducing the end-to-end time up to 74.95× in the one-to-one pattern, up to 25.34× in the fan-out pattern, up to 10.18× in the fan-in pattern, and up to 9.99× in the all-to-all pattern. The higher time-saving is reached with the maximum analyzed data size of 16GB, where communication latency is reduced from 12.55 to 8.22 minutes, saving more than one-third of the time. From the resource usage perspective, Floki represents a significantly less expensive resource usage solution compared to the state-of-practice shared object storage. More precisely, Floki requires 64KB of memory for the pipe buffer and 128KB for each client/socket buffer for each function composing the workload, allowing resource-saving to scale linearly with the increase in data size. Differently, the shared object storage solution requires a disk space equal to the total data size. For example, in the case of the simple one-to-one pattern, the 16GB saving differs from the 1MB saving by a factor of $16,384\times$, equivalent to the difference between the two data sizes. Overall, Floki achieves up to $50,738\times$ of resource-saving, translating into a memory allocation of roughly 1.9MB instead of an object storage allocation of 96GB.

6.1.3 *Automatic Performance-Cost Efficient Resource Allocation*

We propose Dexter, a robust resource allocation manager designed to continuously monitor application execution, automatically assigning an optimal amount of resources at a fine-grained level to maximize resource utilization. Dexter outperforms the default serverless resource allocation on different analytics workloads by achieving a significant cost reduction of up to $4.65\times$, while providing reasonable performance. Moreover, thanks to its conservative approach on scaling resources, Dexter shows comparable or slightly higher runtime, while using up to $5.71\times$ fewer resources. The significant resource usage reduction allows to place allows to place up to $5.5\times$ and up to $4.60\times$ more workload instances when constraining the available amount of resources and the monetary budget. Differently, when focusing on the the pure runtime metric, Dexter deploys up to $2.02\times$ less workload instances since the baseline scales resources more aggressively in time while Dexter follows a more conservative approach. The absolute error distribution analysis reveals that Dexter represents an accurate solution, achieving a median absolute error of 4 executors. Our sensitivity analysis underlines that Dexter is highly adaptable to workload variation and provides performance-cost efficient resource allocations, with an average monetary cost saving of up to $2.65\times$ while slightly increasing the runtime by a factor up to $1.34\times$. Finally, Dexter imposes negligible overheads on the stage execution time since the one-time per stage initial parallelism level prediction requires on average 15.87ms, while the average delay for each search step amounts to 5.65ms. The time needed to fit the parameter model utilizing a batch of 182 stages known at compile-time averages to 19.12ms with a single thread. Projecting this analysis to a multi-tenant cloud-like scenario, with a dataset scaled up to 18,200 rows (two orders of magnitude larger), the training would take $\approx$ 2s on average. This delay is small enough to seamlessly execute model retraining on the fly, even at per-minute granularity.

6.2 FUTURE WORK

In the following, we provide an overview of the research directions we consider promising within the context of the contributions discussed in this thesis.

Exploring Data Locality and Scheduling Policies

The first contribution of this thesis has shown that task granularity and concurrency, data locality, resource allocation, and scheduling policies are pivotal factors that could significantly enhance the performance of executing complex data-intensive analytics workloads in serverless platforms. While task granularity and concurrency, as well as resource allocation, have been investigated in the third contribution, data locality and scheduling policies represent two relevant research directions for future work. In large-scale clusters, co-placing producer-consumer function pairs featuring the most relevant data movements would reduce the network load and the communication overhead thanks to the higher local bandwidths. In this direction, the research community has recently begun to propose solutions to alleviate the expensive data movement overhead of shared remote storage through data locality enhancement [141–143]. However, co-placing functions restrict the resource allocation flexibility, a core characteristic of the serverless paradigm. Therefore, finding the right balance between function co-location and spreading is fundamental. As a result, novel scheduling policies exploiting data locality can be explored to increase further the performance of data-intensive analytics workloads in serverless platforms.

Expanding Floki with Checkpoints

The second contribution of this thesis presented Floki, a system enabling direct inter-function communication within Kubernetes-based serverless environments by proactively forwarding data based on the workload pipeline. The architecture has been designed to be leveraged by serverless platforms and users for high-performance volatile intermediate data exchange. Even though Floki prototype is completely implemented, we aim to fully integrate Floki with Knative. Moreover, adopting a fully volatile data transmission approach significantly enhances end-to-end time performance but at the cost of no data persistency. In the final integration, in the event of a failure, the absence of data persistence imposes the re-computation of intermediate data by re-running the entire pipeline. This re-computation process introduces additional overhead, particularly in data-intensive analytics workloads, where the overhead scales linearly with the computational complexity of functions. Thus, to mitigate this overhead, per-function data recovery deserves further investigation. To enhance resilience and fault tolerance, we envision expanding Floki into a more robust backup and recovery system capable of recovering reliably from failures or faults. One potential direction is developing a consistent "point in time" persistence mechanism of function states. This extension aims to contribute to the overall efficiency of Floki in executing data-intensive analytics workloads in serverless environments.

Expanding Dexter with Critical Stages Knowledge and User Objective
The third contribution of this thesis presented Dexter, a resource allocation manager, leveraging serverless computing elasticity, that continuously monitors the execution of applications, dynamically allocating resources at a fine-grained level to guarantee performance-cost efficiency. Although our evaluation shows that the proposed solution can achieve higher efficiencies than the current serverless Spark frameworks (see Section 5.5), Dexter can be further optimized. Specifically, leveraging DAG knowledge could enable the prioritization of critical stages during resource allocation. This approach involves allocating more resources to critical stages initially and subsequently adapting the resources for remaining parallel stages based on the critical stages estimated time. Adapting dynamically resource allocation prioritizing critical stages increases further resource utilization, freeing resources by assigning the minimum amount to non-critical stages (possibly even less than the optimal ones). To this aim, we are currently working towards enhancing Dexter's architecture with a new *priority module*, coordinating with the existing *historical module* and *search module* to prioritize critical stages. Moreover, expanding Dexter to account with a user-defined objective represents another interesting research direction. Enabling the user to specify performance and cost priority factors would guide the optimal scaling factor search towards his/her preferred objective, rather than maintaining a balanced performance-cost tradeoff. This last extension aims to increase the system's flexibility to adapt to diverse user requirements.

6.3 PUBLICATIONS

The content of this thesis has been published in the following conference papers:

– [86] **Anna Maria Nestorov**, Jordà Polo, Claudia Misale, David Carrera and Alaa S. Youssef, "Performance Evaluation of Data-Centric Workloads in Serverless Environments." In Proceedings of the *14th International Conference on Cloud Computing (CLOUD)*. IEEE, Chicago, IL, USA, 2021, pp. 491-496, 10.1109/CLOUD53861.2021.00064

– [118] **Anna Maria Nestorov**, Josep Lluís Berral, Claudia Misale, Chen Wang, David Carrera, and Alaa Youssef. 2022. "Floki: a proactive data forwarding system for direct inter-function communication for serverless workflows." In Proceedings of the *8th International Workshop on Container Technologies and Container Clouds (WoC '22)*. Association for Computing Machinery, New York, NY, USA, 13–18. 10.1145/3565384.3565890

– *[Submitted - Notification Late February 2024]*. **Anna Maria Nestorov**, Diego Marron, Alberto Gutierrez-Torre, Chen Wang, Claudia Misale, Alaa Youssef, David Carrera, and Josep Lluís Berral. "Dexter: A Performance-Cost Efficient Resource Allocation Manager for Serverless Data Analytics". In: *25th International Middleware Conference (Middleware '24)*.

We are currently involved in the advance of our third future work, described in Section 6.2, extending Dexter's architecture by incorporating a new module accounting with critical stages in parallel stage computation. As an extension of Dexter, our plan is to submit this work to a journal in the upcoming months.

– *[In progress]*. **Anna Maria Nestorov**, Diego Marron, Alberto Gutierrez-Torre, Chen Wang, Claudia Misale, Alaa Youssef, David Carrera, and Josep Lluís Berral. "Dexter: A Priority-Aware Performance-Cost Efficient Resource Allocation Manager for Serverless Data Analytics". In: *Future Generation Computer System International Journal*.

The following publication has been made in addition to the aforementioned publications. This publication is not directly included in this thesis because it is not aligned with the thesis topic. It is an independent study that helped me to understand if to consider heterogeneous computing during the Ph.D.. The contribution was limited to providing ideas and helping with the write-up.

– [144] Nils Voss, Tobias Becker, Simon Tilbury, Georgi Gaydadjiev, Oskar Mencer, **Anna Maria Nestorov**, Enrico Reggiani, and Wayne Luk. 2020. Performance Portable FPGA Design. In Proceedings of the *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 324. https://doi.org/10.1145/3373087.3375362

6.4 CODE REPOSITORIES

Aside of the aforementioned publications, this thesis has produced two open-source code repositories with the implementation of the Floki and Dexter components, as well as the performance evaluation scripts. While Floki code repository is public, the Dexter repository is currently private since the related paper is currently in the peer review process. Specifically, the two code repositories are:

– Floki: The public repository contains the Floki codebase and is hosted on Gitlab (https://gitlab.bsc.es/datacentric-computing/floki-prototype)

– Dexter: The private repository contains the Dexter codebase, as well as our custom version of Apache Spark, integrating the client side of the Dexter *scaling agent*. The source code is available in Gitlab (https://gitlab.bsc.es/datacentric-computing/dexter).

6.5 projects

This thesis was financed by the EU-HORIZON programme (grant agreement GA.101092646); the EU-HORIZON MSCA programme (grant agreement GA.101086248); the Spanish Ministry of Science (MICINN), the Research State Agency (AEI), and European Regional Development Funds (ERDF/FEDER) (grant agreements PID2021-126248OB-I00, MCIN/AEI/10.13039/501100011033/FEDER, UE); the Generalitat de Catalunya (AGAUR) (grant agreement 2021-SGR-00478); the EU-H2020 programme (grant agreement GA.952179); IBM T.J.Watson Research Center in Yorktown Heights, New York (USA), and IBM Research Laboratory in Zurich (Switzerland).

6.6 collaborations and visits

During the whole Ph.D., I have collaborated with IBM T.J.Watson Research Center in Yorktown Heights, New York (USA). In particular, the collaboration involves IBM's Container Cloud Platform team and the Cloud and Cognitive Solutions team. Unfortunately, I was not able to fulfill the initially scheduled stays (at least 3 months per year) at IBM T.J.Watson Research Center in Yorktown Heights due to unforeseen circumstances related to the ongoing COVID-19 pandemic. For this reason, to overcome the limitations posed by the COVID-19 pandemic, the collaboration has been held by implementing a consistent and structured schedule of virtual meetings. These regular virtual meetings have allowed us to stay connected, discuss important matters, and continue our collaborative efforts seamlessly. After the COVID-19 pandemic started to subside, I started to look for internship opportunities in Europe, to contribute my skills and gain practical knowledge in an industrial setting. Fortunately, I found a 6-months internship position in the Hybrid Cloud/Infrastructure-Software team at the IBM Research Laboratory in Zurich (Switzerland), where I could apply my expertise and learn from experienced professionals in the field of infrastructure and resource management in the cloud.

BIBLIOGRAPHY

[1] D. Parkhill. *The Challenge of the Computer Utility*. US: Addison-Wesley Educational Publishers Inc., 1966.

[2] Peter M. Mell and Timothy Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, USA, 2011.

[3] Eric Jonas et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing." In: *CoRR* abs/1902.03383 (2019). arXiv: 1902.03383. URL: http://arxiv.org/abs/1902.03383.

[4] AWS Lambda Functions. URL: https://aws.amazon.com/lambda/?nc1=h_ls.

[5] Google Cloud Functions. URL: https://cloud.google.com/functions.

[6] IBM Cloud Functions. URL: https://cloud.ibm.com/functions/.

[7] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Serverless Computation with OpenLambda." In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[8] Apache OpenWhisk: Open Source Serverless Cloud Platform. URL: https://openwhisk.apache.org.

[9] Knative: Serverless Containers in Kubernetes Environments. URL: https://knative.dev.

[10] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. "Serverless Linear Algebra." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, 281–295. ISBN: 9781450381376. DOI: 10.1145/3419111.3421287. URL: https://doi.org/10.1145/3419111.3421287.

[11] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. "Serverless Data Analytics in the IBM Cloud." In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, 1–8. ISBN: 9781450360166. DOI: 10.1145/3284028.3284029. URL: https://doi.org/10.1145/3284028.3284029.

[12] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. “Sprocket: A Serverless Video Processing Framework.” In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, 263–274. ISBN: 9781450360111. DOI: 10.1145/3267809.3267815. URL: https://doi.org/10.1145/3267809.3267815.

[13] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. “Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads.” In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[14] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. “From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers.” In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488. ISBN: 978-1-939133-03-8. URL: http://www.usenix.org/conference/atc19/presentation/fouladi.

[15] Amazon. URL: https://aws.amazon.com/it/blogs/machine-learning/code-free-machine-learning-automl-with-autogluon-amazon-sagemaker-and-aws-lambda/.

[16] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. “Cirrus: A Serverless Framework for End-to-End ML Workflows.” In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, 13–24. ISBN: 9781450369732. DOI: 10.1145/3357223.3362711. URL: https://doi.org/10.1145/3357223.3362711.

[17] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. “Towards Demystifying Serverless Machine Learning Training.” In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. Virtual Event, China: Association for Computing Machinery, 2021, 857–871. ISBN: 9781450383431. DOI: 10.1145/3448016.3459240. URL: https://doi.org/10.1145/3448016.3459240.

[18] Anirban Bhattacharjee, Yogesh Barve, Shweta Khare, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. “Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks.” In: *2019 USENIX Conference on Operational Machine Learning (OpML 19)*. Santa Clara, CA: USENIX Association, May 2019, pp. 59–61. ISBN: 978-1-939133-00-7. URL: https://www.usenix.org/conference/opml19/presentation/bhattacharjee.

[19] Joao Carreira. "A Case for Serverless Machine Learning." In: *Workshop on Systems for ML and Open Source Software at NeurIPS*. 2018.

[20] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. "Exploring Serverless Computing for Neural Network Training." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 334–341. DOI: 10.1109/CLOUD.2018.00049.

[21] Marc Sánchez-Artigas and Pablo Gimeno Sarroca. "Experience Paper: Towards Enhancing Cost Efficiency in Serverless Machine Learning Training." In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, 210–222. ISBN: 9781450385343. DOI: 10.1145/3464298.3494884. URL: https://doi.org/10.1145/3464298.3494884.

[22] Hao Wang, Di Niu, and Baochun Li. "Distributed Machine Learning with a Serverless Architecture." In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2019, pp. 1288–1296. DOI: 10.1109/INFOCOM.2019.8737391.

[23] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. "Serverless Computing: One Step Forward, Two Steps Back." In: *CoRR* abs/1812.03651 (2018). arXiv: 1812.03651. URL: http://arxiv.org/abs/1812.03651.

[24] Vipul Gupta, Swanand Kadhe, Thomas A. Courtade, Michael W. Mahoney, and Kannan Ramchandran. "OverSketched Newton: Fast Convex Optimization for Serverless Systems." In: *CoRR* abs/1903.08857 (2019). arXiv: 1903.08857. URL: http://arxiv.org/abs/1903.08857.

[25] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. "INFless: a native serverless system for low-latency, high-throughput inference." In: Feb. 2022, pp. 768–781. DOI: 10.1145/3503222.3507709.

[26] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. "BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–15. DOI: 10.1109/SC41405.2020.00073.

[27] Anirban Bhattacharjee, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun, Aniruddha Gokhale, and Gabor Karsai. "BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services." In: *CoRR* abs/1904.01576 (2019). arXiv: 1904.01576. URL: http://arxiv.org/abs/1904.01576.

[28] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E. Gonzalez, and Joseph M. Hellerstein. "Optimizing Prediction Serving on Low-Latency Serverless

Dataflow." In: *CoRR* abs/2007.05832 (2020). arXiv: 2007.05832. URL: https://arxiv.org/abs/2007.05832.

[29] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "Serving deep learning models in a serverless platform." In: *CoRR* abs/1710.08460 (2017). arXiv: 1710.08460. URL: http://arxiv.org/abs/1710.08460.

[30] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. "Stateful Serverless Computing with Crucial." In: *ACM Trans. Softw. Eng. Methodol.* 31.3 (2022). ISSN: 1049-331X. DOI: 10.1145/3490386. URL: https://doi.org/10.1145/3490386.

[31] Mariano Ezequiel Mirabelli, Pedro García-López, and Gil Vernik. "Bringing Scaling Transparency to Proteomics Applications with Serverless Computing." In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. WoSC'20. Delft, Netherlands: Association for Computing Machinery, 2021, 55–60. ISBN: 9781450382045. DOI: 10.1145/3429880.3430101. URL: https://doi.org/10.1145/3429880.3430101.

[32] Michael Kiener, Mohak Chadha, and Michael Gerndt. "Towards Demystifying Intra-Function Parallelism in Serverless Computing." In: *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. WoSC '21. Virtual Event, Canada: Association for Computing Machinery, 2021, 42–49. ISBN: 9781450391726. DOI: 10.1145/3493651.3493672. URL: https://doi.org/10.1145/3493651.3493672.

[33] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. "Caerus: NIMBLE Task Scheduling for Serverless Analytics." In: *Symposium on Networked Systems Design and Implementation*. 2021.

[34] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, 115–130. ISBN: 9781450367356. DOI: 10.1145/3318464.3389758. URL: https://doi.org/10.1145/3318464.3389758.

[35] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. "Occupy the Cloud: Distributed Computing for the 99%." In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, 445–451. ISBN: 9781450350280. DOI: 10.1145/3127479.3128601. URL: https://doi.org/10.1145/3127479.3128601.

[36] Youngbin Kim and Jimmy Lin. "Serverless Data Analytics with Flint." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 451–455. DOI: 10.1109/CLOUD.2018.00063.

[37] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure." In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[38] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. "Starling: A Scalable Query Engine on Cloud Function Services." In: *CoRR* abs/1911.11727 (2019). arXiv: 1911.11727. URL: http://arxiv.org/abs/1911.11727.

[39] Rathijit Sen, Abhishek Roy, and Alekh Jindal. "Predictive Price-Performance Optimization for Serverless Query Processing." In: *CoRR* abs/2112.08572 (2021). arXiv: 2112.08572. URL: https://arxiv.org/abs/2112.08572.

[40] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. "Astrea: Auto-Serverless Analytics Towards Cost-Efficiency and QoS-Awareness." In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 3833–3849. DOI: 10.1109/TPDS.2022.3172069.

[41] Verified Market Research. URL: https://www.verifiedmarketresearch.com/product/serverless-architecture-market/.

[42] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research." In: *CoRR* abs/1708.08028 (2017). arXiv: 1708.08028. URL: http://arxiv.org/abs/1708.08028.

[43] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. "Serverless Computing: An Investigation of Factors Influencing Microservice Performance." In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pp. 159–169.

[44] Eric Jonas et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing." In: (Feb. 2019).

[45] Pedro García López, Marc Sánchez Artigas, Simon Shillaker, Peter R. Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, and Ana Juan Ferrer. "ServerMix: Tradeoffs and Challenges of Serverless Data Analytics." In: *CoRR* abs/1907.11465 (2019). arXiv: 1907.11465. URL: http://arxiv.org/abs/1907.11465.

[46] Microsoft Azure Functions. URL: https://azure.microsoft.com/en-us/services/functions/.

[47] Apache Spark: Unified engine for large-scale data analytics. URL: https://spark.apache.org.

[48] Dataproc Serverless. URL: https://cloud.google.com/dataproc-serverless/docs.

[49] Databricks SQL Serverless. URL: https://www.databricks.com/blog/announcing-general-availability-databricks-sql-serverless.

[50] IBM Analytics Engine. URL: https://cloud.ibm.com/docs/AnalyticsEngine?topic=AnalyticsEngine-getting-started.

[51] Sangeetha Abdu Jyothi et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 117–134. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi.

[52] Amazon Simple Storage Service (S3). URL: https://aws.amazon.com/s3.

[53] IBM Cloud Object Storage. URL: https://www.ibm.com/cloud/object-storage.

[54] Andrea Passwater. *2018 Serverless Community Survey: huge growth in serverless usage*. URL: https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/. Accessed: 05.07.2020.

[55] Kubernetes (K8s): Production-Grade Container Orchestration. URL: https://kubernetes.io.

[56] Docker. URL: https://www.docker.com.

[57] Containerd. URL: https://containerd.io.

[58] Istio. URL: https://istio.io.

[59] CloudEvents. URL: https://github.com/cloudevents/spec/blob/master/spec.md.

[60] Apache Kafka. URL: https://kafka.apache.org.

[61] Google Cloud Pub/Sub. URL: https://cloud.google.com/pubsub.

[62] Tekton Cloud Native CI/CD. URL: https://tekton.dev.

[63] Kubeflow Pipelines with Tekton. URL: https://github.com/kubeflow/kfp-tekton.

[64] Rashmi Korlakai Vinayak and Ran Gilad-Bachrach. "DART: Dropouts meet Multiple Additive Regression Trees." In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Guy Lebanon and S. V. N. Vishwanathan. Vol. 38. Proceedings of Machine Learning Research. San Diego, California, USA: PMLR, 2015, pp. 489–497. URL: https://proceedings.mlr.press/v38/korlakaivinayak15.html.

[65] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine." In: *Annals of statistics* (2001), pp. 1189–1232.

[66] Wayne Iba and Pat Langley. "Induction of One-Level Decision Trees." In: *Machine Learning Proceedings 1992*. Ed. by Derek Sleeman and Peter Edwards. San Francisco (CA): Morgan Kaufmann, 1992, pp. 233–240. ISBN: 978-1-55860-247-2. DOI: https://doi.org/10.1016/B978-1-55860-247-2.50035-8. URL: https://www.sciencedirect.com/science/article/pii/B9781558602472500358.

[67] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Improving neural networks by preventing co-adaptation of feature detectors." In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580.

[68] R. Smith. "An Overview of the Tesseract OCR Engine." In: *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*. ICDAR '07. USA: IEEE Computer Society, 2007, 629–633. ISBN: 0769528228.

[69] TPC-H Benchmark. URL: https://www.tpc.org/tpch/.

[70] Raghunath Othayoth Nambiar and Meikel Poess. "The making of TPC-DS." In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, 1049–1058.

[71] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. "Get Real: How Benchmarks Fail to Represent the Real World." In: *Proceedings of the Workshop on Testing Database Systems*. DBTest'18. Houston, TX, USA: Association for Computing Machinery, 2018. ISBN: 9781450358262. DOI: 10.1145/3209950.3209952. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3209950.3209952.

[72] TPC-DS Benchmark. URL: https://www.tpc.org/tpcds/.

[73] Owen O'Malley Yahoo! "Terabyte Sort on Apache Hadoop." In: (2008). URL: http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf.

[74] Valgrind. URL: https://valgrind.org/info/tools.html.

[75] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. "GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303958. URL: https://doi.org/10.1145/3302424.3303958.

[76] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. *Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement*. 2018. arXiv: 1811.09721 [cs.DC].

[77] Z. Al-Ali, S. Goodarzy, E. Hunter, S. Ha, R. Han, E. Keller, and E. Rozner. "Making Serverless Computing More Serverless." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 456–459. DOI: 10.1109/CLOUD.2018.00064.

[78] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. arXiv: 1812.03651 [cs.DC].

[79] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. "Cirrus: A Serverless Framework for End-to-End ML Workflows." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing Machinery, 2019, 13–24. ISBN: 9781450369732. DOI: 10.1145/3357223.3362711. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3357223.3362711.

[80] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. *Occupy the Cloud: Distributed Computing for the 99%*. 2017. arXiv: 1702.04024 [cs.DC].

[81] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads." In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, 363–376. ISBN: 9781931971379.

[82] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020). DOI: 10.1145/3318464.3389758. URL: http://dx.doi.org/10.1145/3318464.3389758.

[83] Y. Kim and J. Lin. "Serverless Data Analytics with Flint." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 451–455. DOI: 10.1109/CLOUD.2018.00063.

[84] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. "Understanding Ephemeral Storage for Serverless Analytics." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 789–794. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless.

[85] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics." In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, 427–444. ISBN: 9781931971478. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[86] Anna Maria Nestorov, Jordà Polo, Claudia Misale, David Carrera, and Alaa S. Youssef. "Performance Evaluation of Data-Centric Workloads in Serverless Environments." In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021, pp. 491–496. DOI: 10.1109/CLOUD53861.2021.00064.

[87] Cloud Sort Benchmark. URL: http://sortbenchmark.org.

[88] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. "Jiffy: Elastic Far-Memory for Stateful Serverless Analytics." In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, 697–713. ISBN: 9781450391627. DOI: 10.1145/3492321.3527539. URL: https://doi.org/10.1145/3492321.3527539.

[89] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures." In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, 41–54. ISBN: 9781450370097. DOI: 10.1145/3361525.3361535. URL: https://doi.org/10.1145/3361525.3361535.

[90] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. "Cloudburst: Stateful Functions-as-a-Service." In: *CoRR* abs/2001.04592 (2020). arXiv: 2001.04592. URL: https://arxiv.org/abs/2001.04592.

[91] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. "SAND: Towards High-Performance Serverless Computing." In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, 923–935. ISBN: 9781931971447.

[92] Simon Shillaker and Peter Pietzuch. "FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing." In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[93] Zhipeng Jia and Emmett Witchel. "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021).

[94] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. "Faastlane: Accelerating Function-as-a-Service Workflows." In: *USENIX Annual Technical Conference*. 2021.

[95] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. "Narrowing the Gap Between Serverless and Its State with Storage Functions." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '19. Santa Cruz, CA, USA: Association for Computing

Machinery, 2019, 1–12. ISBN: 9781450369732. DOI: 10.1145/3357223.3362723. URL: https://doi.org/10.1145/3357223.3362723.

[96] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads." In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, 363–376. ISBN: 9781931971379.

[97] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. "Boxer: Data Analytics on Network-enabled Serverless Platforms." In: *CIDR*. 2021.

[98] MinIO: Multi-Cloud Object Storage. URL: https://min.io.

[99] Redis: The open source in-memory data store. URL: https://redis.io.

[100] Memcached: A high-performance, distributed memory object caching system. URL: https://redis.io.

[101] Fission Workflows. URL: https://github.com/fission/fission-workflows.

[102] HyperFlow. URL: https://github.com/hyperflow-wms/hyperfl.

[103] Apache Airflow. URL: https://airflow.apache.org.

[104] Apache Hadoop Yarn. URL: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.

[105] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing." In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 2.

[106] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks." In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, 59–72. ISBN: 9781595936363. DOI: 10.1145/1272996.1273005. URL: https://doi.org/10.1145/1272996.1273005.

[107] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis." In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: Association for Computing Machinery, 2012. ISBN: 9781450317610. DOI: 10.1145/2391229.2391236. URL: https://doi.org/10.1145/2391229.2391236.

[108] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. "Building an Elastic Query Engine on Disaggregated Storage." In: *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*. NSDI'20. Santa Clara, CA, USA: USENIX Association, 2020, 449–462. ISBN: 9781939133137.

[109] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. "Dynamic Query Re-Planning Using QOOP." In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, 253–267. ISBN: 9781931971478.

[110] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[111] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. "Anna: A KVS for Any Scale." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 401–412. DOI: 10.1109/ICDE.2018.00044.

[112] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. "Autoscaling Tiered Cloud Storage in Anna." In: *Proc. VLDB Endow.* 12.6 (2019), 624–638. ISSN: 2150-8097. DOI: 10.14778/3311880.3311881. URL: https://doi.org/10.14778/3311880.3311881.

[113] Djob Mvondo et al. "OFC: An Opportunistic Caching System for FaaS Platforms." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, 228–244. ISBN: 9781450383349. DOI: 10.1145/3447786.3456239. URL: https://doi.org/10.1145/3447786.3456239.

[114] John Ousterhout et al. "The RAMCloud Storage System." In: *ACM Trans. Comput. Syst.* (2015). ISSN: 0734-2071. DOI: 10.1145/2806887. URL: https://doi.org/10.1145/2806887.

[115] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. "SONIC: Application-aware Data Passing for Chained Serverless Applications." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 285–301. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/mahgoub.

[116] Bryan Ford, Pyda Srisuresh, and Dan Kegel. "Peer-to-Peer Communication Across Network Address Translators." In: *CoRR* abs/cs/0603074 (2006). arXiv: cs/0603074. URL: http://arxiv.org/abs/cs/0603074.

[117] Jeffrey Eppinger. "TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem." In: (Jan. 2005).

[118] Anna Maria Nestorov, Josep Lluis Berral, Claudia Misale, Chen Wang, David Carrera, and Alaa Youssef. "Floki: A Proactive Data Forwarding System for Direct Inter-Function Communication for Serverless Workflows." In: *Proceedings of the Eighth International Workshop on Container Technologies and Container Clouds*. WoC '22. Quebec, Quebec City, Canada: Association for Computing Machinery, 2022, 13–18. ISBN: 9781450399296. DOI: 10.1145/3565384.3565890. URL: https://doi.org/10.1145/3565384.3565890.

[119] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The datacenter as a computer: An introduction to the design of warehouse-scale machines." In: *Synthesis lectures on computer architecture* 8.3 (2013), pp. 1–154.

[120] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. "Learning Scheduling Algorithms for Data Processing Clusters." In: *CoRR* abs/1810.01963 (2018). arXiv: 1810.01963. URL: http://arxiv.org/abs/1810.01963.

[121] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. "AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft." In: *Proc. VLDB Endow.* 13.12 (2020), 3326–3339. ISSN: 2150-8097. DOI: 10.14778/3415478.3415554. URL: https://doi-org.recursos.biblioteca.upc.edu/10.14778/3415478.3415554.

[122] Anish Pimpley, Shuo Li, Anubha Srivastava, Vishal Rohra, Yi Zhu, Soundararajan Srinivasan, Alekh Jindal, Hiren Patel, Shi Qiao, and Rathijit Sen. *Optimal Resource Allocation for Serverless Queries*. arXiv. 2021. URL: https://www.microsoft.com/en-us/research/publication/optimal-resource-allocation-for-serverless-queries/.

[123] Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay. "OptEx: A Deadline-Aware Cost Optimization Model for Spark." In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2016, pp. 193–202. DOI: 10.1109/CCGrid.2016.10.

[124] Stratos Dimopoulos, Chandra Krintz, and Rich Wolski. "Justice: A Deadline-Aware, Fair-Share Resource Allocator for Implementing Multi-Analytics." In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 233–244. DOI: 10.1109/CLUSTER.2017.52.

[125] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management." In: *SIGPLAN Not.* 49.4 (2014), 127–144. ISSN: 0362-1340. DOI: 10.1145/2644865.2541941. URL: https://doi.org/10.1145/2644865.2541941.

[126] Han Gao, Zhengyu Yang, Janki Bhimani, Teng Wang, Jiayin Wang, Bo Sheng, and Ningfang Mi. "AutoPath: Harnessing Parallel Execution Paths for Efficient Resource Allocation in Multi-Stage Big Data Frameworks." In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. 2017, pp. 1–9. DOI: 10.1109/ICCCN.2017.8038381.

[127] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. "3Sigma: Distribution-Based Cluster Scheduling for Runtime Uncertainty." In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190515. URL: https://doi.org/10.1145/3190508.3190515.

[128] Rui Han, Chi Harold Liu, Zan Zong, Lydia Y. Chen, Wending Liu, Siyi Wang, and Jianfeng Zhan. "Workload-Adaptive Configuration Tuning for Hierarchical Cloud Schedulers." In: *IEEE Transactions on Parallel and Distributed Systems* 30.12 (2019), pp. 2879–2895. DOI: 10.1109/TPDS.2019.2923197.

[129] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. "Re-Optimizing Data-Parallel Computing." In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 21.

[130] Hanfei Yu, Hao Wang, Jian Li, and Seung-Jong Park. "Harvesting Idle Resources in Serverless Computing via Reinforcement Learning." In: *CoRR* abs/2108.12717 (2021). arXiv: 2108.12717. URL: https://arxiv.org/abs/2108.12717.

[131] Simon Kassing, Ingo Müller, and Gustavo Alonso. *Resource Allocation in Serverless Query Processing*. 2022. arXiv: 2208.09519 [cs.DB].

[132] gRPC. URL: https://grpc.io.

[133] Protocol buffers. URL: https://protobuf.dev;https//github.com/protocolbuffers/protobuf.

[134] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data using Serverless Cloud Infrastructure." In: *CoRR* abs/1912.00937 (2019). arXiv: 1912.00937. URL: http://arxiv.org/abs/1912.00937.

[135] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. "Primula: A Practical Shuffle/Sort Operator for Serverless Computing." In: *Proceedings of the 21st International Middleware Conference Industrial Track*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, 2020, 31–37. ISBN: 9781450382014. DOI: 10.1145/3429357.3430522. URL: https://doi.org/10.1145/3429357.3430522.

[136] Marc Sánchez-Artigas and Germán T. Eizaguirre. "A Seer Knows Best: Optimized Object Storage Shuffling for Serverless Analytics." In: *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. Middleware '22. Quebec, QC, Canada: Association for Computing Machinery, 2022, 148–160. ISBN: 9781450393409. DOI: 10.1145/3528535.3565241. URL: https://doi.org/10.1145/3528535.3565241.

[137] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. "On Data Processing through the Lenses of S3 Object Lambda." In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. 2023, pp. 1–10. DOI: 10.1109/INFOCOM53939.2023.10228890.

[138] Jonatan Enes, Roberto R. Expósito, and Juan Touriño. "Real-time resource scaling platform for Big Data workloads on serverless environments." In: *Future Generation Computer Systems* 105 (2020), pp. 361–379. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.11.037. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19310015.

[139] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-Scale Cluster Management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. URL: https://doi.org/10.1145/2741948.2741964.

[140] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: Association for Computing Machinery, 2013, 351–364. ISBN: 9781450319942. DOI: 10.1145/2465351.2465386. URL: https://doi.org/10.1145/2465351.2465386.

[141] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. "FaaSFlow: enable efficient workflow execution for function-as-a-service." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, 782–796. ISBN: 9781450392051. DOI: 10.1145/3503222.3507717. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3503222.3507717.

[142] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. "Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing." In: *CoRR* abs/2010.07268 (2020). arXiv: 2010.07268. URL: https://arxiv.org/abs/2010.07268.

[143] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. "SAND: Towards High-Performance Serverless Computing." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

Boston, MA: USENIX Association, July 2018, pp. 923–935. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[144] Nils Voss, Tobias Becker, Simon Tilbury, Georgi Gaydadjiev, Oskar Mencer, Anna Maria Nestorov, Enrico Reggiani, and Wayne Luk. "Performance Portable FPGA Design." In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, p. 324. ISBN: 9781450370998. DOI: 10.1145/3373087.3375362. URL: https://doi.org/10.1145/3373087.3375362.

DECLARATION

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Barcelona, March 2024

Anna Maria Nestorov