UNIVERSITAT
ROVIRA i VIRGILI

Towards Enhanced State Management for Serverless Computation

DANIEL BARCELONA PONS

DOCTORAL THESIS
2022

Daniel Barcelona Pons

Towards Enhanced State Management for Serverless Computation

Doctoral Thesis

Supervised by

Dr. Pedro García López

Department of Computer Engineering and Mathematics

UNIVERSITAT ROVIRA i VIRGILI
Tarragona
2022

UNIVERSITAT
ROVIRA i VIRGILI

Departament d'Enginyeria
[DΣIM]
Informàtica i
Matemàtiques

FAIG CONSTAR que aquest treball, titulat "Towards Enhanced State Management for Serverless Computation", que presenta Daniel Barcelona Pons per a l'obtenció del títol de Doctor, ha estat realitzat sota la meva direcció al Departament d'Enginyeria Informàtica i Matemàtiques d'aquesta universitat i que compleix els requeriments per poder optar a la Menció Internacional.

HAGO CONSTAR que el presente trabajo, titulado "Towards Enhanced State Management for Serverless Computation", que presenta Daniel Barcelona Pons para la obtención del título de Doctor, ha sido realizado bajo mi dirección en el Departamento de Ingeniería Informática y Matemáticas de esta universidad y que cumple los requisitos para poder optar a la Mención Internacional.

I STATE that the present study, entitled "Towards Enhanced State Management for Serverless Computation", presented by Daniel Barcelona Pons for the award of the degree of Doctor, has been carried out under my supervision at the Department Computer Engineering and Mathematics of this university and that it fulfills all the requirements to be eligible for the International Doctorate Award.

Tarragona, 15 de Novembre/15 de Noviembre/November 15, 2022

El director de la tesi doctoral
El director de la tesis doctoral
Doctoral thesis supervisor

Dr. Pedro García López

Abstract

Cloud computing is evolving with new technologies and abstractions that make it more accessible and fitting for many applications. In this line, and with fervent interest from research and industry, there is the concept of serverless. In short, a serverless service hides the existence of servers from users, so that they can focus on their applications. In the trend to exploit the impressive elasticity of serverless computing, distributed parallel computations such as analytics, machine learning, and data processing have raised special attention from researchers.

However, current serverless services are unfitted out-of-the-box to support parallel applications. Broadly, the issues come from a lack of state management, coordination, and predictability. In consequence, several research lines open in this context; we discuss three: 1) It is unclear how well can current serverless services support distributed parallel computing. Parallel applications require execution simultaneity and performance consistency. Current platforms, however, do not ensure that, and users may find extreme variability between services. 2) Current serverless computing exclusively offers stateless ephemeral workers with no direct communication. Applications built on top of these services struggle to manage their global state, specially to mutate it at fine granularity. Furthermore, it is impossible to efficiently coordinate execution with precision. 3) Intermediate data generated during data processing workloads is forcibly transferred between workers and disaggregated storage. This data-shipping model generates expensive data movement that also impacts application performance.

In this thesis, we present three novel core contributions to tackle the challenges that manifest in the above lines of research. First, we categorize the main serverless computing platforms from a parallel computation point of view. We carefully investigate their architectural design and empirically evaluate them to find the platforms that best fit to these applications. Second, we explore novel methods to code stateful distributed applications in serverless. For this, we present the cloud thread abstraction and build a shared objects layer that allows serverless workers to share and mutate common state and easily coordinate their execution. Finally, we study a solution to the serverless data-shipping problems through in-storage ephemeral stateful computation. We build a storage system with integrated computation that allows to offload data-bound tasks from serverless workers and significantly reduce data transfers in data processing pipelines.

Acknowledgements

During my time at the Cloud and Distributed Systems Lab (CloudLab) research group, where this dissertation has been slowly being crafted, I have met many people. All of them deserve my thanks for what they have given me these last years. First of all, I would like to send a special "thank you" to my advisor Dr. Pedro García López for his help and advice in writing this thesis, but also for his open mind towards doing research, patience, clear ideas, and the almost-philosophical discussions we have sporadically found ourselves into. Without him, this work would not have been possible. I also thank all the people who I have shared some time with at the CloudLab, formerly AST, research group. Specially, Marc Sánchez and Gerard París, who were a part of my daily life at the lab for several years. Thank you for all those moments.

I would also like to thank all the people at the Cloud Data Platforms group at IBM Research Europe in Switzerland for accepting me as an intern and for their help and insights. I appreciate many people I met during the months spent there, as well as everything I learnt; not only professionally, but also personally.

And most importantly, my deepest gratitude is to my family, for their love and support. Specially my parents, whose hard work allowed my brothers and I to be where we are now.

Daniel Barcelona Pons
Gandesa, October 2022

This work has been partially funded by the European Union Horizon 2020 Framework Programme, in the context of the project CloudButton: Serverless Data Analytics Platform (825184), and by the Spanish Ministerio de Ciencia, Innovación y Universidades (Grant PID2019-106774RB-C22).

Thesis Publications

- 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, Dec. 2019, pp. 41–54. ISBN: 978-1-4503-7009-7. DOI: `10.1145/3361525.3361535`. CORE A [34].

- Daniel Barcelona-Pons. “State Support for Serverless Cloud Services”. In: *6th URV Doctoral Workshop in Computer Science and Mathematics.* Ed. by Carme Julià and Aïda Valls. Tarragona, Spain: Publicacions URV, Apr. 2020, pp. 9–12. ISBN: 978-84-8424-865-1 [30].

- Daniel Barcelona-Pons and Pedro García-López. “Benchmarking parallelism in FaaS platforms”. In: *Future Generation Computer Systems* 124 (Nov. 2021), pp. 268–284. ISSN: 0167-739X. DOI: `10.1016/j.future.2021.06.005`. JCR Impact Factor: 7.307, Q1 [31].

- 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 (Mar. 2022). ISSN: 1049-331X. DOI: `10.1145/3490386`. JCR Impact Factor: 3.685, Q1 [35].

Other publications

- Daniel Barcelona-Pons, Álvaro Ruiz-Ollobarren, David Arroyo-Pinto, and Pedro García-López. “Studying the feasibility of serverless actors”. In: *Proceedings of the European Symposium on Serverless Computing and Applications, ESSCA@UCC 2018.* Ed. by Josef Spillner. Vol. 2330. CEUR Workshop Proceedings. Zurich, Switzerland: CEUR-WS.org, 2018, pp. 25–29 [33].

- Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. “Comparison of FaaS Orchestration Systems”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion).* 2018, pp. 148–153. DOI: `10.1109/UCC-Companion.2018.00049` [66].

- Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. “FaaS Orchestration of Parallel Workloads”. In: *Proceedings of the 5th International Workshop on Serverless Computing*. WOSC ’19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 25–30. ISBN: 978-1-4503-7038-7. DOI: `10.1145/3366623.3368137` [32].

Table of Contents

List of Figures

List of Tables

List of Abbreviations

μ**s**	Microsecond
AOP	Aspect Oriented Programming
API	Application Programming Interface
AUC	Area under the curve
AWS	Amazon Web Services
BSP	Bulk Synchronous Parallelism
CDF	Cumulative Distribution Function
CLI	Command-Line Interface
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
DRAM	Dynamic Random Access Memory
e.g.	Exempli gratia
EC2	Amazon Elastic Compute Cloud
EMR	Amazon Elastic MapReduce
FaaS	Function as a Service
FPGA	Field-programmable gate array
GB	Gigabyte (10^9 bytes)
Gbps	Gigabits per second
GCP	Google Cloud Platform
GHz	Gigahertz
GiB	Gibibyte (2^{30} bytes)
HDD	Hard Disk Drive
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol

i.e.	Id est
I/O	Input/Output
IaaS	Infrastructure as a Service
ID	Identifier
IP	Internet Protocol
JDK	Java Development Kit
JVM	Java Virtual Machine
KB	Kilobyte (10^3 bytes)
KiB	Kibibyte (2^{10} bytes)
KVM	Kernel-based Virtual Machine
MB	Megabyte (10^6 bytes)
Mbps	Megabits per second
MHz	Megahertz
MiB	Mebibyte (2^{20} bytes)
min	Minute
ML	Machine Learning
MPI	Message Passing Interface
MPI	Message Passing Interface
ms	Millisecond
NAT	Network Address Translation
NIC	Network Interface Card
NVM	Non-Volatile Memory
OOB	Out of bag score
OOP	Object Oriented Programming
OS	Operating System
PaaS	Platform as a Service
POJO	Plain Old Java Object
RDMA	Remote Direct Memory Access
REST	REpresentational State Transfer
RPC	Remote Procedure Call
RTT	Round-trip time
s	Second
S3	Amazon Simple Storage Service
SDK	Software Development Kit
SLA	Service Level Agreement
SLO	Service Level Objective
SLOC	Source Lines of Code
SMR	State Machine Replication
SNS	Amazon Simple Notification Service

SQS	Amazon Simple Queue Service
SSD	Solid State Drive
TB	Terabyte (10^{12} bytes)
TCP	Transmission Control Protocol
TiB	Tebibyte (2^{40} bytes)
vCPU	Virtual CPU
VM	Virtual Machine
VPC	Virtual Private Cloud

Chapter 1

Motivation and Challenges

Nowadays, the cloud is the first option for many companies and startups to build their services and store their data. The ability to request resources and scale their systems without needing to invest in hardware enables companies of any size, and even particular users, to run solutions at scale from a simple laptop. Although the cloud started offering simple virtual machines on demand (IaaS), it did not take long for more specific services to appear, such as managed storage solutions or communication brokers. The appeal of these services and what makes them thrive is the responsibilities assumption by the cloud providers. With more system management being dealt by vendors, cloud users can focus on developing their applications and business logic. Through high-level abstractions, a modest company with few resources can run and maintain large-scale services that are automatically kept available and secure; a task that otherwise would require a team of engineers. Aware of this, cloud vendors constantly strive for new services and abstractions that facilitate cloud development to lure more users into their products. One of the hottest topics currently in this matter is serverless.

The concept of serverless begun its wide usage with serverless computing to define Function-as-a-Service (FaaS) platforms and quickly arose a fierce interest in both industry and research. The ideas that uphold the term *serverless* are simple: a serverless service should abstract any idea of a server from its users. This means that the service should provide a useful abstraction (such as running a piece of code, sending a message, or storing some data) that the user can use at any time, at any scale, and pay only and accurately for its usage. In other words, the cloud provider scales the service automatically on demand, keeps it available and secure, and offers it under a just-right billing plan.

These properties allow many services to be serverless. In general, we can classify them into three categories: computation, storage, and integration tools. Serverless computation services are those that allow users to run code by simply providing it. In essence, users deploy their code by uploading the source into the service and selecting the desired runtime. Then they configure the conditions for its execution, i.e., a set of rules or triggers, or enable execution through direct invocation requests. FaaS, or serverless functions, is the foremost example of serverless computing, enabling the ephemeral execution of stateless functions in response to events. This is why we will usually use serverless computing and serverless functions interchangeably when referring to FaaS services. Nonetheless, we have seen other options explored with different abstractions [24, 50] and container virtualization [12, 110]. Serverless storage dates from before the term *serverless* gained popularity. Cloud-managed object stores and databases have been part of vendors' lineup for many years. A storage service should be serverless if it offers an interface for accessing and handling data and keeps capacity management away from its users. Finally, serverless integration tools include communication services like distributed queues or event brokers, orchestration managers in different flavors (e.g., state machines, DAGs, and other workflows), connectors to other services, and any type of management tools that allow cloud users to employ them without explicitly dealing with resources.

1.1 Problem statement

In this thesis, we take special interest in serverless computation, explore its limitations, and find ways of expanding its application. Previous research has put the limitations of serverless computing under study [83, 99]. We find several recurring aspects about the issues of the model that can be summarized in three pillars: state management, coordination/communication, and execution consistency/predictability. Serverless functions are ephemeral and, thus, stateless, which makes it difficult to support distributed applications that share execution state and increases remote data transfers. Similarly, distributed applications need coordination to progress in execution, and this is very difficult to achieve in a model that bans direct communication between functions. And thirdly, the startup latency of serverless functions varies unpredictably, which may affect applications that require low latency or consistent performance to correctly exploit parallelism. For instance, cold starts in functions do not only depend on runtime creation, but also on environment initialization, dependency loading, and application set-up.

Among the many types of applications that have taken an interest in serverless computing, there is one that raises special attention in research: large-scale distributed parallel computation. This area of application includes hot topics like big data analytics [65, 98, 109, 141] and machine learning [42, 43, 93], and presents important research challenges in the three pillars introduced above. Indeed, these distributed programs generate and process huge amounts of data, require coordinating their multiple tasks, and suffer from variability in performance.

Given all these high-level challenges that arise in a general analysis, we are first interested in the characteristics and performance that existing FaaS platforms can provide to parallel computation. A survey of the literature shows multiple studies and benchmarks of FaaS platforms that qualify and compare basic properties like configuration options, available runtimes, invocation latency, resources provided, and more. However, none of these works focus on the traits that could directly affect parallel computation with serverless functions, nor there is a detailed discussion on such topic. And this leads to our first research question:

Question I: *What are the benefits and restrictions that serverless computing architectures provide to parallel computing?*

While basic FaaS properties are certainly important towards answering this question, we believe further study is needed to understand parallel computing on serverless functions. Parallel applications usually run tasks that closely collaborate on the computation. This means that they usually need to communicate, synchronize, and share state. For this behavior to occur efficiently, simultaneity is a key quality for success. Additionally, tasks are usually compute-intensive, which require resources to be correctly isolated and consistent for best collaboration. It is important to highlight that function concurrency, as advertised by services and investigated in the literature, is not enough for these applications. To achieve the best levels of simultaneity without performance degradation full-fledged parallelism is required. In other words, functions must be ensured to run on correctly isolated resources to avoid interferences, the service must provide them eagerly to reach the necessary scale, and resources must be spawned quickly to provide the essential simultaneity. All these properties, however, have not yet been evaluated in detail and remain an open question.

The uncertainty in these properties affirms on the third core limitation of FaaS platforms (predictability), and already makes evident that building parallel applications for serverless computing is far from trivial. However, these applications also suffer from the other two pillars: shared state and coordination. Parallel computation, like most traditional computing applications, is often complex and stateful, requiring tasks to share global state and coordinate their progress in real time. The complexity of implementing such applications on top of serverless computing raises a fundamental challenge that we convey with our second research question:

Question II: *Can we efficiently use serverless computing for applications with mutable shared state and complex coordination requirements?*

Currently, there is no serverless system that effectively addresses these issues. Since it is not possible to communicate functions directly, the common practice is to use remote storage [98]. Mostly, existing solutions use object storage, which is too slow for small application state. And while some propose using faster in-memory stores to improve this matter [141], these systems are still unfitted to handle mutable state at fine granularity. Furthermore, function coordination is still very limited. FaaS orchestration services only allow coarse coordination patterns [32, 66] and existing communication services are too slow to handle fine-grained coordination between tasks. All in all, fine-grained state sharing is a clear open issue and there is no general way to coordinate functions in arbitrary patterns to support the requirements of many traditional applications, including parallel computing.

The issues of serverless computing with data, however, do not stop with the modest-sized mutable shared state. Distributed parallel applications usually ingest huge amounts of data from a separate storage system for processing. Even in traditional compute clusters, such a model puts big pressure on the network and becomes a bottleneck. This issue is known as the data-shipping problem. Then again, parallel applications divide computation into several stages, which need to transfer more data from one to the next. While traditional clusters can keep these data locally, the ephemeral nature of serverless computation forces it to be relayed through remote storage, worsening the repercussions of data-shipping. Which brings us to propose the following research question:

Question III: *Can we improve the data-shipping problem in serverless computing without hindering the advantages of serverless functions?*

Data-shipping is still an open problem in serverless computing. There are no services available that help diminish its effects. Existing research mostly focuses on function orchestration tools that still require huge data movements. Others have tackled data locality in serverless functions, but this requires modification of the FaaS platform. The main hindrance of this approach is its impact on basic beneficial properties of serverless computing. Indeed, it places restrictions on function scheduling and placement, relaxes resource isolation, and heavily hinders the elasticity that makes FaaS attractive. Furthermore, these solutions are limited to small amounts of data and are not fit for holding intermediate results in multi-stage computations.

A solution that has helped against data-shipping previously is co-locating computation within the storage to offload specific tasks and significantly reduce data transfers. This has yet to be explored in a serverless context and its application is not trivial. Among several challenges that arise for this task, we focus our work on the following: (i) compute and storage co-location may degrade performance if not correctly handled; (ii) data-shipping in serverless is tougher and requires storage computation that is stateful, something inexistent in the literature; and (iii) storage computation should provide improved management of large intermediate data.

1.2 Contributions of this thesis

In view of the above open research questions, we describe here the three main contributions of this thesis that aim to provide an answer to each of them.

Contribution I: *Categorize FaaS platforms for parallel computation.*

Our first contribution is a detailed benchmark of parallel computation on serverless functions. For this, we carefully investigate the architectures and performance indicators of the four major cloud FaaS services. In particular, these are AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. Differently from other benchmarks, we place special emphasis on those traits that create a good environment for highly parallel computation. New to the literature, we first describe and study the architectural approach of each service to management and execution of functions. Furthermore, we build a comparative framework to examine virtualization technologies, scheduling models, resource management, and other components. Next, we empirically evaluate parallelism of FaaS platforms in a specifically defined experiment. The experiment is designed to run a controlled distributed parallel simulation and draw a detailed view of the execution timeline that evidences the simultaneity of tasks and performance achieved along with indications of its causes. Finally, we put both studies into consideration to feed a discussion with the objective to understand the execution of parallel applications in current serverless computing, visualize their performance, and spot potential problems. All in all, to help users choose the best platform for their applications.

Contribution II: *Enable stateful serverless computing and coordination.*

For the second contribution, we explore a simplified way to code stateful distributed applications in serverless. We combine two ideas to this end. First, we present a cloud thread abstraction that maps the traditional notion of thread to a serverless function for an easy orchestration of tasks. Second, cloud threads may share global state thanks to a distributed shared objects layer. Shared objects provide fine-grained mutation of state through the implementation of arbitrary methods. Access to shared objects is transparent and strongly consistent out-of-the-box, which smoothens development. Additionally, the layer includes different primitives and communication collectives to enable fine-grained function coordination. We then propose Crucial, a framework that incorporates these ideas. We design a very simple programming model with conventional multi-threaded abstractions. This makes it possible to transform existing concurrent stateful programs to serverless with just a few annotations and constructs. Although the range of applications is very wide, we show these model in several applications that are very complex to run on serverless computation right away. With limited programming involvement, our framework allows us to quickly scale to serverless traditional parallel computations, specialized ML training jobs, and complex concurrency task coordination.

Contribution III: *Enable serverless ephemeral storage computation.*

Our third contribution consists in the formulation of computational storage as a solution to data-shipping in serverless computing. In particular, we define a serverless model for in-storage ephemeral stateful computation. It is a multi-tenant storage service that grows elastically as a companion to serverless computing. Its key novel feature is its ability to offload computation to it and confront the issues of data-shipping. Importantly, we define storage spaces to scale computation and storage separately and not compromise performance isolation while still provide improved synergy between them. At the same time, the main computation tier (e.g., FaaS platforms) is unaffected so that it can be fully exploited for its elasticity. Our contributions go beyond previous computational storage research with storage actions. Actions are integrated within the storage namespace and enable stateful computation beyond traditional stateless data interceptors. This novel abstraction eases development and allows to solve more complex applications. Furthermore, we keep large intermediate data in mind, and provide actions with a streamed I/O interface to speed up data processing in computing pipelines. Finally, we provide different examples of applications that demonstrate the applicability of our solution and its benefits, including a reduction of data transfers in size and quantity, while improving the overall performance.

1.3 Outline of this dissertation

Here we provide a summary of the thesis chapters:

Chapter 2: State of the Art This chapter discusses the literature around the research areas related to the thesis contributions. It includes modifications of FaaS platforms and different solutions for stateful serverless computing and coordination.

Chapter 3: Studying Parallelism in FaaS This chapter presents an empirical study of FaaS platforms with special emphasis on parallel computation. The conclusions throw light into different FaaS architectures and evidence the benefits and caveats of using each platform for such demanding computation.

Chapter 4: Serverless Stateful Computation This chapter explores a novel method for the programming of stateful serverless applications. With a disaggregated object layer, programmers can combine FaaS functions with remote objects to coordinate complex computation and share state.

Chapter 5: Serverless Ephemeral Computational Storage This chapter presents an innovative serverless ephemeral computational storage model through storage actions. Actions alleviate the effects of data-shipping present in serverless computing by reducing far data movements.

Chapter 6: Conclusions and Future Work This chapter collects the conclusions evinced from this work and several lines for continuation, as well as some discussion on future evolutions.

Chapter 2

State of the Art

This chapter reviews the state of the art relative to parallel stateful serverless computation and coordination with the aim to bring the reader closer to the research problems that motivate this thesis. The previously stated research questions and contributions can be included in different areas of research. We tackle each of them as follows:

For **Contribution I** (*Categorize FaaS platforms for parallel computation*), we collect information from several benchmarking works and reviews of serverless services. We study its properties and put especial emphasis on their affinity to distributed computation and big data.

For **Contribution II** (*Enable stateful serverless computing and coordination*), we review the current state of research and industry for stateful serverless computing. This includes works on FaaS platform improvements, optimizations, and other modifications; serverless orchestration and coordination systems, as well as other programming frameworks; serverless storage systems and its applications for intermediate data serving and coordination; and other novel complex systems for stateful serverless computation.

For **Contribution III** (Enable serverless ephemeral storage computation), we extend the previous information with a focus on the problem of data-shipping, computation close to the data, and active storage systems.

2.1 Serverless computing under study

The recent popularity of serverless computing has triggered the appearance of several analysis, benchmarks, and studies tackling FaaS platforms and other serverless services. An important concern when analyzing these services is that most of them, including the most popular ones, are cloud services with proprietary code, architecture, and infrastructure. This forces the community to explore the services from a black-box perspective, which only allows for a high-level user point of view. The lack of detailed information on how the systems work internally hinders the ability of users to understand the affinity of each service to the requirements of their applications, which can lead to dreadful, unexpected behavior in their business solutions. In this section, we review the different benchmarks around FaaS services that the community has engaged in since their appearance.

2.1.1 Performance evaluation

The most analyzed aspect of FaaS platforms is perhaps their cold start latency and how it changes based on function configuration, such as memory size, runtime, or trigger type. Indeed, there are research papers, blogs, and even dedicated websites that either include cold start benchmarking or are specially focused on it [127, 165]. Extending that, many benchmarks in the literature also extensively explore invocation latency in general (beyond cold starts) and CPU performance [152, 177]. For instance, there are studies that compare the CPU performance in a function against traditional VMs and analyze how it evolves when varying the function resources, usually by configuring memory. Other recent benchmarks [127] extend this evaluation at service level with the function invocation throughput as perceived by users. Also, another interesting quality studied in this line is the function invocation cost and how it compares against other cloud computing services.

In terms of scale, different works explore function invocation concurrency [49, 139, 159, 177]. These benchmarks perform large-scale experiments with multiple concurrent function invocations and analyze the behavior of the FaaS platform from a high-level point of view. Their objective is to collect information on maximum invocation concurrency supported by the platform and compare it against service specifications and the requirements of several applications. In this line, the measurements on a recent paper [168] regarding the QoS of different platforms also show special emphasis on their concurrency and explore several issues with resource allocation and function scheduling.

A very interesting topic in FaaS benchmarks is service elasticity. A long discussed feature and one of the most important qualities in serverless technologies is precisely their capacity to adapt their scale on demand by providing resources to users as they are needed and release them afterwards for precise utilization and billing. Hence, several benchmarks [112, 113, 116, 139] investigate this property. To that end, they usually generate a dynamic workload of many invocations and observe how the platform behaves under different demand patterns. This provides information to analyze the capacity of each system to accept incoming requests rapidly. In sum, these benchmarks evaluate the limits of FaaS platforms in number of running function invocations, performance degradation, and quickness in adapting to demand. A recurrent conclusion is the evident performance variation across platforms that shows in their results. The causes for these issues and variations, however, have yet to be explored.

Beyond FaaS performance studies, a few papers explore the affinity of the model to more complex applications. A common type of application extensively evaluated on serverless functions is batch computing for big data analytics. An implementation of MapReduce [70] is evaluated for large computations on AWS Lambda following the observations of PyWren [98] and ExCamera [65]. *gg* [64] and Pocket [108, 109] also perform several analysis of large computations atop this particular platform. AWS Lambda proves to be a good fit for the task, but other platforms are left with very little experimentation. Other benchmarks perform their evaluation from an even higher level and focus on orchestration tools atop FaaS [32, 66]. They show that some platforms do not achieve good parallelism, but do not explore the causes in detail.

2.1.2 Architectural analysis

The internals and architectures of commercial FaaS platforms are not studied in detail, with just some comments in a paper [177]. This aspect is, however, certain to affect the performance of the platform. The lack of information is not surprising given the nature of the FaaS services in public clouds, which are proprietary platforms with little documentation available about their internal structure. The case of open-source FaaS platforms presents a very different situation, for which we can find extensive analysis [133, 155], or even reference architectures [173].

A recent paper from Azure [156] offers some insights on how Azure Functions works and a platform usage analysis from the perspective of the cloud provider. However, the paper focuses its exploration on optimizing latency in cold starts and reduce resource waste, and it disregards application performance and other system properties such as concurrency, elasticity, isolation, or parallelism.

2.1.3 Discussion

Several works leverage serverless functions for data analytics, linear algebra, machine learning, big data processing, and many other batch computing jobs. We see a lot of interest in running these kinds of highly parallel computations on serverless services. This makes sense due to the quick and accurate spawning of thousands of workers in FaaS, that provides the required computing power for each stage of a job without over- or under-provisioning resources.

All these highly parallel applications, however, require that the FaaS platform provides enough parallelism for them to achieve good performance that competes with traditional computing services. Nonetheless, as we have seen in our literature review, parallelism in FaaS platforms is not investigated carefully enough.

Closely associated with parallelism, we discussed some benchmarks that evaluate system elasticity. However, the fact that a system is able to handle changes in the amount of function invocation requests does not mean that these requests will be handled appropriately to perform as expected by parallel applications (i.e., achieving execution simultaneity without performance degradation). These papers do not evaluate parallelism. In addition, the workloads utilized in these works are usually I/O-bound, like reactive web applications, which do not represent the highly parallel computations we evaluate in this dissertation.

Question I places its emphasis on compute-intensive workloads and parallel computing. These applications need strong guarantees on execution parallelism and resource isolation to achieve good performance, but this has not yet been evaluated. Existing benchmarks do not differentiate between resolving invocations concurrently and actually handling them in parallel to bring the required performance. One objective in Contribution I is to explore the behavior of each platform with deeper detail to evaluate these characteristics.

The architecture of a FaaS platform is very important to understand its performance for parallel applications. The system design greatly determines its ability to run invocations in parallel and isolate performance. While the results of the surveyed benchmark papers evidence performance variation across platforms, they usually disregard its causes and do not explore the architecture of each system for properties or patterns that affect performance. A few papers partially dig into the internals of platforms [177], but do not study its effects on performance and parallelism.

Another discovery of our FaaS benchmark survey is the general focus on AWS Lambda. Most articles in the literature that explore serverless parallel computation [65, 98] prove their ideas on Amazon's cloud. In these works, AWS Lambda seems a good fit for batch computations. However, to conclude that FaaS is a good fit for parallel applications we still need to study and verify that other platforms and their designs match these results.

In sum, none of the existing works investigates the architecture design of the different FaaS platforms, and how it affects their performance for different types of applications. Specially, there is a lack of detailed evaluation for highly parallel computations, which have become popular in the literature.

2.2 Stateful serverless computing

Serverless computing has gained much traction and many works have been proposed in this area. In this section, we survey runtimes (FaaS platforms), programming frameworks, and storage systems specialized in serverless computing. To finish, we include a review of (serverless and non-serverless) solutions to the problem of stateful distributed computation. A summary of our findings appears in Table 2.1, where we compare the most relevant serverless solutions that address the problems of state sharing and coordination against the optimal objectives presented in Question II. The comparison is made along five dimensions:

- *Storage.* This category describes what storage media/service the system uses to keep the intermediate state of an application. The storage type determines the access latency for I/O operations. It ranges from object stores, which exhibit high latency, to in-memory storage designed for fast access and high throughput.
- *Mutability.* This category indicates how the system handles updates to the shared state (e.g., fine-grained updates to arbitrary mutable data, append-only semantics, etc.).
- *Coordination.* Here, we detail how coordination between multiple functions is achieved and, most importantly, at which granularity. For instance, fine-grained coordination allows functions to coordinate with well-known synchronization primitives. On the contrary, coarse-grained coordination, such as in the BSP model, only allows functions to progress in lock step.
- *Durability.* Some systems enable the shared state to survive system failures. This dimension categorizes the methods employed to achieve such a property, if available. Sometimes, application state is ephemeral and the benefits of fast access outweigh the cost of making it durable [109].
- *Consistency.* Since concurrent accesses to the mutable shared state can hit stale data, the system should provide a consistency criterion for the programmer. Here, we list the existing guarantees offered by each system.

Table 2.1: Serverless solutions for state sharing and coordination.

System	Storage	Mutability	Coordination	Durability	Consistency
PyWren	object store	—	coarse-grained	replication	weak
ExCamera	rendezvous server	—	rendezvous server	—	—
Ripple	object store	—	high-level dataflow	—	weak
Beldi	key-value store	transactions	—	replication	strong
Pocket	multi-tiered	append-only	coarse-grained	ephemeral	—
Cloudburst	FaaS + cache	lattice data structures	coarse-grained	replication	weak: repeatable read and causal
Faasm	hierarchical: local shared memory + global tier (Redis)	byte-array level	coarse-grained	—	variable; strong via explicit locking
Optimal	in-memory store	fine-grained	fine-grained	optional	strong

2.2.1 Serverless runtimes

Serverless computing has appealing characteristics based on simplicity, high scalability, and fine-grained execution. It has seduced both industry [14, 25, 74] and academia [84]. This enthusiasm has also led to a blossom of open-source systems, e.g., OpenWhisk [4], Kubeless [111], OpenFaaS [138], OpenLambda [84], and Fission [62], to name a few.

The most common form of serverless computing systems are FaaS platforms or services, which we review as serverless runtimes. At core, a serverless runtime is in charge of maintaining the user-defined functions, executing them upon request. It must ensure strong isolation between function instances and deliver fast startup times to enhance the critical path of function execution. Many works propose to tackle these two central challenges.

Micro-kernels [128] offer a solid basis to quickly start a function, even achieving sub-millisecond startup time. Catalyzer [59] introduces the `sfork` system call to reuse the state of a running sandbox. Similarly, Firecracker [2] makes containers more lightweight and faster to spawn. SOCK [137] is a serverless-specialized system that uses a provisioning mechanism to cache and clone function containers. SAND [3] exploits function interaction in FaaS to improve startup time and resource efficiency. The system achieves these properties by relaxing isolation at the application level, enabling functions from the same application to share memory and communicate through a hierarchical message bus. Faasm [160] offers similar guarantees using a language-agnostic runtime. Fast function initialization is achieved thanks to a lightweight execution mechanism built atop the software-fault isolation (SFI) facilities of WebAssembly. For data sharing between functions, Faasm offers a two-tier architecture: the local tier provides in-memory data sharing for co-located functions, while the global tier supports distributed access across the whole system.

Rather than playing out with isolation guarantees for better performance, Contribution II pursues to provide an efficient substrate for handling *mutable state* and *coordination* at fine granularity over existing platforms (e.g., AWS Lambda), which the above runtimes do not support in place. For instance, Faasm's global state tier is implemented with a distributed Redis instance, which is inefficient for complex operations as we will see in Chapter 4. Also, two recent works coincide that existing runtimes do not support mutable shared state and coordination across cloud functions. Hellerstein et al. [83] underline that the serverless computing model is a data-shipping architecture that imposes indirect communication and hinders coordination. Jonas et al. [99] highlight the lack of adequate storage for fine-grained operations and the inability to coordinate functions at fine granularity.

Contribution II seeks a solution to these challenges that may run atop any FaaS platform. This is however not a straightforward task since FaaS platforms do not follow a common standard and necessitate users to rewrite entirely the scripts for the deployment and invocation of functions for a new targeted platform. This issue is common to many serverless applications. The dependency to the FaaS platform causes a "vendor lock-in" and reduces code portability. Several projects try to address this concern. The Serverless Framework [154] offers plug-ins to simplify the deployment and execution of serverless functions over multiple clouds and FaaS environments. RADON [44] targets the whole development stack with the goal of providing a cloud-agnostic serverless programming experience. Similarly, Lithops [147] hides under a common interface the deployment and execution of serverless functions for different cloud settings.

2.2.2 Programming frameworks

Several works that address the challenges of mutable shared state and coordination confront them from a function composition perspective: a scheduler orchestrates the execution of stateless functions and shares information between them.

Several cloud services support function compositions. AWS allows creating state machines with Step Functions [22]. The Amazon States Language (JSON-based) is, however, ill-suited to express complex workflows. IBM Composer [5] offers a similar solution. In this case, function compositions are written in JavaScript and then transformed into state machines. As before, the expressiveness of IBM Composer is bound to a small set of constructs. Google Cloud Composer [73], built on Apache Airflow, allows to create and run a DAG of tasks. In addition to a poorer expressiveness than state machines, it requires to deploy multiple components in Google Kubernetes Engine before the execution of a workflow, similar to an on-premises deployment. Finally, Azure Durable Functions [25] enables to programmatically coordinate function calls. It is the most complete solution among all, allowing to write imperative code. Asynchronous calls to functions are expressed in C♯, permitting to explicitly wait prior results. The major downside of the above services is their poor performance for running highly parallel compositions [32, 66].

To sidestep the limitations in function coordination, PyWren [98] pioneered the idea to use FaaS for bulk synchronous parallel (BSP) computations. The paper shows the elasticity and scalability of FaaS and demonstrates with a base Python prototype how to run MapReduce workloads. PyWren uses a client-worker architecture where stateless functions share state through slow cloud storage. IBM-PyWren [149] evolves the PyWren model with new features and a broader support to run fully-fledged MapReduce tasks. Further, Locus [141] extends PyWren to support shuffling with a good cost-performance ratio. Tailored to linear algebra, NumPyWren [157] manages a pool of stateless workers that run small tasks built on the fly as the mathematical computation progresses. ExCamera [65] is another system atop FaaS, more focused on video encoding and low latency. Its computing framework (*mu*) is designed to run thousands of threads as an abstraction for cloud functions. It handles inter-thread communication through a rendezvous server. *gg* [64] continues *mu*'s line for running serverless parallel threads, but targeting a broader audience. Finally, Ripple [100] is a programming framework to enable single-machine applications to benefit from the ample parallelism of FaaS platforms. It provides a simple interface of eight functions for programmers to express the dataflow of their applications. Also, it automates resource provisioning and handles fault tolerance by eagerly detecting stragglers. Before the full computation run, the framework performs a series of dry runs to test and find the best resource provisioning for the job.

Jangda et al. [96] propose an operational model for serverless platforms (named λ_λ), a simplified semantics, and an extension for stateful functions. The simplified semantics is equivalent to λ_λ when the cold and warm states of a function produce the same result. The stateful extension adds a (global) transactional key-value store that serverless functions may call. Extending serverless computing with transactions is also the path taken in Beldi [184]. Compared with the previous model [96], the one of Beldi does not serialize accesses to the data store through a central lock.

Fault tolerance is a key concern when programming in a serverless environment. The λ_λ model captures the fact that the FaaS platform may start multiple instances to answer a request, yet use a single one to reply. The bisimulation result by Jangda et al. [96] indicates when this is equivalent to executing the request exactly once (that is under the simplified semantics). Serverless cloud vendors warn programmers that serverless functions must be idempotent. Yet they do not precise what does this mean, neither what to do when computation is stateful. In their paper, Sreekanti et al. [163] introduce a layer that interposes between the FaaS platform and the storage engine to ensure read atomicity when functions access multiple data items.

2.2.3 Storage

Many frameworks focus on cloud function scheduling and coordination, while using disaggregated storage to manage data dependencies. In particular, they opt to write shared data to slow, highly-scalable storage [98, 149, 157]. To hide latency, they perform coarse-grained accesses, resort to in-memory stores, or use a combination of storage tiers [141].

Pocket [109] is a distributed data store that scales out and in on demand to match the storage needs of serverless applications. It leverages multiple storage tiers and right-sizes them offline based on the application requirements. Crail [166] presents the NodeKernel architecture with similar objectives. These two systems are designed for ephemeral data, which are easy to distribute across a cluster. They do not use a distributed hash table that would require data movement when the cluster topology changes, but instead use a central directory. Both systems scale in to zero when computation ends.

InfiniCache [175] is an in-memory cache built atop cloud functions. The system exploits FaaS to store objects in a fleet of ephemeral cloud functions. It uses erasure coding and a background rejuvenation mechanism to maintain data availability despite the continuous fluctuations in the pool of cloud functions as they are reclaimed by the provider. Similar to a traditional distributed in-memory cache, InfiniCache has been designed to provide fast access to read-only objects but not to mutate them. Sion [185] follows a similar idea.

The above works do not allow fine-grained updates to mutable shared state. Such a feature can be abstracted in various ways. However, we need to look beyond serverless systems and prototypes to find solutions for it. Furthermore, our second challenge raises interest in systems that enable dealing with state by keeping the inherent simplicity of serverless computing.

Existing systems such as Memcached [63], Redis [142], or Infinispan [129] cannot be readily used for such purpose. They either provide too low-level abstractions or require server-side scripting. Coordination kernels such as ZooKeeper [89] can help synchronizing cloud functions. However, their expressiveness is limited, and they do not support partial replication [57, 101]. We explore these problems in Section 4.6.

Creson [167] presents the concept of callable objects on top of Infinispan. The system enables a simplified way of accessing shared state as remote objects and keeps the well-understood semantics of linearizability. This brings Creson close to the objectives of Question II, but it is not ready to resolve the concrete issues present in serverless computing in regard to function coordination, access transparency, and control over data durability.

Contribution II targets strong consistency in storage with the objective to simplify programming. Some systems [158, 162, 169] rely instead on weak consistency, trading ease of programming for performance. Weak consistency has been used to implement distributed stateful computation in FaaS, as detailed in the next section.

2.2.4 Distributed stateful computation

This section reviews different complete systems that are dedicated to enable distributed stateful computation, going beyond the frameworks discussed above. We start with two proposals in the serverless space.

Cloudburst [164] is a stateful serverless computation service. State sharing between cloud functions is resolved by building a custom FaaS platform atop Anna [181], an autoscaling key-value store that supports a lattice put/get CRDT data type. Cloud functions run coupled with the store for local access to shared state. Cloudburst offers repeatable read and consistent snapshot consistency guarantees for function composition; something that is not achievable, for instance, when using AWS Lambda in conjunction with S3 (i.e., computing $x + f(x)$ is not possible if x mutates).

Cirrus [42] is a machine learning framework that leverages cloud functions to efficiently use computing resources. It specializes in iterative training tasks and asynchronous stochastic gradient descent. Cirrus relies on a distributed data store that does not allow custom shared objects and/or simple mutability options. Furthermore, distributed workers cannot coordinate at fine granularity.

Besides serverless systems, there exist many frameworks for machine clusters that target stateful distributed computation.

Ray [135] is a recent specialized distributed system mainly targeting AI applications (e.g., Reinforcement Learning). It offers a unified interface for both stateless tasks and stateful actor-based computations. Ray motivates for the need of a specialized system that combines stateful and stateless computations. This is in line with Contribution II. However, Ray couples both models in the same system and is built for a provisioned resource environment where stateless tasks and actors live co-located.

Other systems with a focus on stateful computations, such as Dask and PyTorch, usually build on low-level technologies (e.g., MPI) to communicate between nodes. These frameworks rely on clusters with known topology and fail to quickly scale out or in to match demand. Such a design is also at odds with the FaaS model, where functions are forbidden to communicate directly. Specialized distributed big data processing frameworks, such as MapReduce, are available as a service in the cloud (e.g., AWS EMR).

2.2.5 Discussion

We have seen multiple attempts in the literature to utilize serverless functions to execute distributed stateful applications at scale. However, very few focus on resolving an efficient management of shared application state.

Some works focus on improving the FaaS platform or runtime itself, usually to reduce invocation latency, refine scheduling, or enhance isolation and/or performance. Others modify the runtime to include data storage on the platform itself and allow several functions to share memory. This, however, complicates function scheduling and the ability of functions to scale rapidly on demand. Contribution II chooses to keep FaaS unchanged to preserve its full properties, and seeks for a specialized system that harmonizes with functions to bring them a layer of shared state and facilitate coordination.

The different frameworks reviewed present several ways to facilitate development and deployment of serverless applications. None, however, provide a complete solution to the joint problem of fine-grained updates and coordination (see Table 2.1). To wit, state sharing in PyWren, and others [141, 149], is too coarse-grained and/or with weak consistency guarantees. Similarly, Ripple shares intermediate results using Amazon S3, which is slow and provides an ill-suited interface for tasks with fine-grained data sharing needs. ExCamera requires a long-lived relay server to share state between workers with a low-level interface. For certain operations such as `AllReduce`, the message-passing architecture can become a bottleneck. The challenge presented with Question II is to keep communication complexity low. Analogous concerns can be raised about coordination in the surveyed systems.

In the case of storage systems, we find no options that are readily usable for the objectives of our second research challenge. Most of the systems available in the cloud are slow object storage with weak consistency guarantees, that are ill-suited for interactive state sharing in stateful applications and even worse for fine-grained coordination. Traditional in-memory stores, on the other hand, need heavy user management and are not prepared for complex state mutations, requiring programmers to resolve complex communication patterns.

Lastly, traditional systems for distributed stateful computation cannot be simply used for serverless functions. This is mainly due to the lack of direct communication between functions and their ephemeral nature. Attempts in the serverless space either modify the FaaS platform (with the caveats we already discussed) or they are too application-specific to enable general stateful computation.

2.3 Data-shipping in serverless computing

The problem of data-shipping in the serverless computing model has been mentioned multiple times [83, 99, 105, 164]. The data processing pipeline is split into several stages of ephemeral workers and the intermediate data they generate creates a problematic amount of network traffic. Its causes and effects have been studied and categorized [83, 99, 108] with various works pointing out its challenges and taking different paths to solve or palliate its consequences. We see three main approaches to data management in distributed computation (from left to right in Figure 2.1): A) a complete disaggregation of compute and storage with separation of concerns, B) a combination of storage capacity within a FaaS platform or vice-versa, and C) a disaggregated model with task offloading to computation-enabled storage systems. In this section we review these models within the literature and see how they fall short to efficiently confront the challenges of serverless data-shipping.

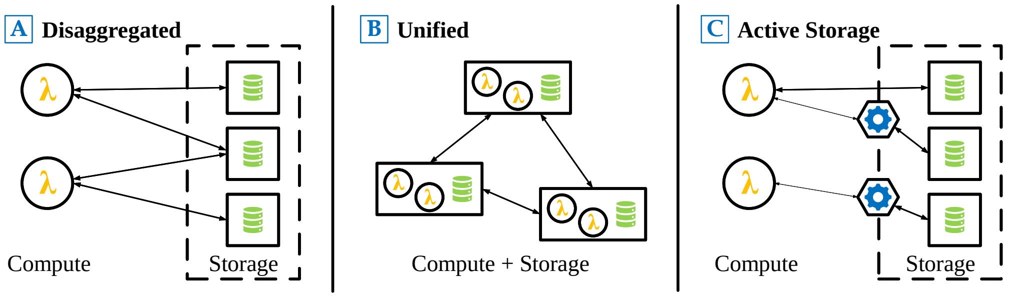


FIGURE 2.1: Architectural approaches to data-shipping.

2.3.1 Complete disaggregation

The basic approach to data processing in serverless is to exploit the fast elasticity of serverless functions to obtain massive parallelism [20, 98]. However, this kind of computation generates lots of intermediate data that must be transferred between computation stages, i.e., between serverless functions. To this end, the ephemeral nature of serverless workers requires this information to be stored on disaggregated storage, as drawn in model A in Figure 2.1. The common choice is to use cloud object storage due to its high bandwidth. Nonetheless, the large amount of data generated during data processing workloads quickly becomes a network bottleneck in this model [83, 99].

This has raised the development of new solutions as a countermeasure. Some of them explore the optimization of the services available in the cloud. For instance, Primula [150] focusses on the MapReduce model and studies the problems of serverless computing with disaggregated object storage. The authors create a specialized model to improve the performance of applications. Other projects search a solution by replacing the object storage with a more performant system, or by adding an intermediate cache layer. Locus [141] creates a model to optimize the use of storage in shuffle operations by combining slow (object storage) and fast (in-memory cache) storage systems. ExCamera [65] uses a rendezvous server to handle function communication and speed up state sharing. Other systems [52] try to overcome these problems by enforcing direct function to function communication through NAT hole punching. This last method is, however, limited and unreliable due to the ephemeral nature of functions. All these examples, among others in the literature, show that data-shipping is very present and users struggle with the insufficient tools available in the cloud.

Due to the lack of direct support in the cloud, we see some projects that build serverless storage solutions to efficiently handle the state of serverless applications. An instance of this is Pocket [109], a multi-tiered storage system based on the NodeKernel architecture of Apache Crail [166], that is built for serverless ephemeral data with multi-tenant capabilities and the ability to scale elastically per application. These properties make Pocket more efficient as a FaaS companion than available cloud storage systems such as object storage. Jiffy [105] improves on Pocket's ideas and builds a scalable remote memory system for serverless functions. Functions can easily use this storage system as a shared space that automatically grows or shrinks on demand and at fine granularity by adding or removing small-sized blocks of memory.

Despite these many efforts, none of these works confront the fundamental challenges of serverless data-shipping: reduce the amount of data being transferred during computation. By fully following a complete disaggregation of compute and storage, data must be irremediably moved back and forth between tiers, which becomes an important bottleneck for serverless data processing workloads.

2.3.2 Unified systems

Understanding the problems of data-shipping, several projects combine storage and computation in a unified system. Their goal is to co-locate computation and its data (mostly by caching it) to avoid far network transfers as much as possible. Generally, this model can be depicted as in diagram B in Figure 2.1. There are several trends to this approach.

One interesting line of research opts to modify the FaaS platform to enable functions to directly share data or exploit locality in some way. A first example of this idea is SAND [3], that enables functions to directly share memory by relaxing isolation. Similarly, Faasm [160] uses the software-fault isolation (SFI) facilities of WebAssembly to share memory between functions. This is limited to functions that can be co-located jointly, necessitating an external storage solution for global distributed state access. SONIC [126] optimizes application performance and cost with a data-passing manager that selects the optimal method for each communication in a workflow and implements communication-aware function placement.

Differently, some projects implement a FaaS platform on top of a storage system. Cloudburst [164] brings serverless functions onto a distributed, autoscaling cache. Functions are run on the same system where they can store shared data and exploit locality for improved performance. Shredder [186] also moves the FaaS platform into the storage system server. It runs functions as WebAssembly programs isolated in the storage cluster to enable faster access to data.

Another trend is to exploit existing serverless platforms to build cache stores on the function resources themselves. InfiniCache [175] and Sion [185] create in-memory caches by leveraging vanilla functions in commercial FaaS services. They use a fleet of functions and exploit the fact that providers keep function instances warm to construct a pay-per-access serverless in-memory storage. Faa$T [146] is an auto-scaling serverless distributed cache that is built within the FaaS platform (implemented atop Azure Functions) and allows functions to store recurring data co-located with the computation. Consequently, cache size and capacity scales with the serverless function application, since they share resources. Similarly, OFC [136] exploits the commonly underutilized memory resources in FaaS functions to build an opportunistic memory cache distributed over the application workers themselves.

All these projects fully couple storage and computation in shared resources, which has been proved to be a problem in the past [46, 148]. In particular, this approach creates resource contention and performance interferences between the storage and compute features. Also, managing the scale of both components jointly is usually inefficient for one, or both, of them. Consequently, computation cannot scale freely like it usually does in dedicated FaaS platforms (which is one of their most appealing advantages) and their storage capacity is very limited, which is unfitting for large intermediate data.

2.3.3 Computation-enabled storage

Running computation close to the data to exploit data locality and improve performance is a long-explored idea nowadays. This concept has been applied from the hardware level and all the way up to the highest software abstractions in cloud services. Computational storage is a term mostly used for close to the data computation applied to hardware components. The most relevant cases are *active disks* [1, 104, 172] and, more recently, SmartNICs [60, 61, 87]. The former refers to storage drives (HDD, SSD, etc.) with computational capacity, while SmartNICs are Network Interface Cards with programmable packet-processing capabilities. These concepts enjoy recent interest thanks to the advances in technologies like FPGAs [117]. Close to the data computation has also been applied from a software perspective in distributed systems. For instance, databases expose interfaces to install and run stored procedures [82, 143] and coprocessors [6, 174].

For distributed storage systems, the concept of close to the data computation has been studied in the past as active storage [145]. Over the years, active storage has been researched for many different storage systems [95, 125, 178, 183]. A couple recent examples are Scoop [132] and Lamda-Flow [80]. These systems utilize the computing resources in the storage systems to enable analytics frameworks such as Apache Spark to offload operations to an active storage layer. From a general perspective, this model resembles diagram C in Figure 2.1. The main computation cluster (represented as serverless workers in the diagram) may trigger the storage operation when objects are uploaded or downloaded to or from the storage system. These computations are simple stateless interceptions that execute in the data path. Scoop allows to offload Spark SQL selections to OpenStack Swift object storage, while Lamda-Flow focuses on the automatic identification of Spark dataflow operators that can be pushed to an active storage system. The results of active storage research demonstrate huge data transfer savings between compute and storage tiers and an effective way to counteract data-shipping in cluster computing.

However, this has some drawbacks that become of special relevance in the context of the cloud. An important development in this research field is the awareness and management of resource contention in active storage systems. As an instance of this, Chen et al. [46] study the impact of resource contention in these systems and propose an architecture to mitigate the effects of the problem. In the evaluation, the authors conclude that resource contention is a critical problem for active storage systems. This problem is further explored by Zion [148], which brings the issues to cloud object storage. The cloud setting requires special attention due to differences in data access and management. But more importantly, the multi-tenant nature of cloud services places special emphasis in resource contention issues. As a solution, Zion proposes an architecture with an active storage layer correctly isolated from the storage resources. Object accesses can still be intercepted by user-provided stateless functions in a model that resembles serverless functions in storage.

This is, in fact, the only approach that has been adopted, to some extent, by the cloud. Some cloud object storage services now include extensions that exploit these ideas. This is the case of Amazon S3 Select [18], that enables to filter data read from objects using simple SQL select queries through HTTP requests. But we also have the recent S3 Object Lambda [17], an alternative S3 endpoint that intercepts all object accesses with AWS Lambda functions that the user can code freely. Functions in S3 Object Lambda allow to preprocess data from S3 (e.g., filtering, compressing, etc.) and, differently from normal Lambda functions, include streamed data transfer between function and client. While S3 Select runs queries in the storage system, Object Lambda is still a separate service and cannot fully exploit data locality.

All these solutions, however, are not enough to satisfy the demands of data processing workloads in serverless. Traditional systems that integrate computation into an object store suffer from resource contention and performance interference. Solutions like Amazon S3 Select, that try to control this with predefined operations, are too limited in versatility to support all the needs of serverless data processing. The active storage layer in Zion allows to scale storage and computation separately in the same system, but it essentially works like a serverless functions platform invoked from the object storage proxies that intercept the data access path. S3 Object Lambda adopts a very similar approach where serverless functions intercept S3 data operations. In both cases, computation is not fully integrated within the storage system, and it is encapsulated in ephemeral, stateless functions. Stateless computation promotes moving data, and it is limited in use cases. Serverless data processing generates more intermediate data than traditional cluster computing due to the ephemeral nature of their workers, which cannot communicate between them nor keep state throughout stages. Stateless storage interception is not enough to resolve these situations that are unique to serverless computing. An active storage solution that enables stateful computation is currently non-existent. And lastly, temporary data requires a specialized data management approach that object stores do not provide [166]. In this sense, there is no active storage to efficiently handle ephemeral data.

2.3.4 Discussion

We have seen many works in industry and research that employ serverless computing (particularly FaaS services) to run large scale, distributed applications such as data processing workloads. The inherent limitations of current platforms enforce a data-shipping model to implement these applications. Since functions are ephemeral and anonymous, any data they produce or consume must be handled in a disaggregated storage. This puts significant pressure on network transfers, which usually becomes the bottleneck in a model where workers tend to have modest network links [108, 177].

Researchers quickly identified this issue and started to propose solutions to optimize data transfers. We review several of these projects. A first idea was to exploit the elasticity of FaaS to reach high aggregated bandwidth; thousands of parallel functions compensate their limited individual network bandwidth. This is sometimes not enough or simply too expensive, thus some works optimize the number of data partitions and workers to explore a performance to cost tradeoff. Another solution for certain applications is to simply replace the storage for a faster one. For example, using an in-memory store to relay information between functions instead of slower object stores or combine them intelligently to achieve a better cost-performance relation. In the end, however, these solutions still follow a data-shipping architecture and, even after optimization, are hindered by the problems it creates. Importantly, huge data transfers thwart application performance and can quickly become economically expensive.

For this reason, some projects especially tackle the issues of data-shipping and aim to minimize its effects. Most of these works try to merge a FaaS platform with a storage system either way. As we have seen in our review of active storage research, a complete combination of compute and storage is unadvised [46, 148]. Such an approach creates interferences and contention between both elements, which is of special relevance in multi-tenant services. Furthermore, it thwarts system elasticity, since storage and compute typically undergo very different demand from applications. A better approach is to decouple computation within the storage and manage them separately while still keeping them close.

Precisely, traditional active storage systems are built on top of object storage. In these solutions, computation scales with the storage in a sub-optimal way. Object stores handle persistent data and, consequently, the process of scaling them is heavy. This does not match with the quickly changing demand of ephemeral computation that is necessary to process data in data processing workloads.

Moreover, serverless functions generate extra intermediate data due to their lack of direct communication. This hinders computational stages that require caching data or simple aggregations with multiple, expensive data transfers. Traditional active storage typically implements stateless data processors that execute in the data path. This model is not enough to resolve these situations, which would require stateful computational elements.

This situation sets up our third research question, where we ask for a solution that reduces the effects of the data-shipping model but does not hinder the properties and performance of serverless functions and storage. In sum, there is no current solution to counteract serverless data-shipping for large temporary data.

Chapter 3

Studying Parallelism in FaaS

A MYRIAD OF WORK has explored Function-as-a-Service (FaaS) for highly-parallel computing jobs. However, are FaaS platforms a good fit for parallel computation? Most of the literature focuses on specific platforms and convey that their ideas can be extrapolated to any other. However, not all FaaS systems follow the same architecture and none detail specific targets towards such kind of applications.

In this chapter, we explore the architectures of four cloud FaaS offerings with special emphasis on parallel performance and conclude that not all of them provide the necessary means to host highly parallel applications. We validate these findings with an extensive empirical experiment.

The results of this chapter have been published in an article [31].

3.1 Introduction

Function-as-a-Service (FaaS) has picked the interest of many applications due to its simplicity. One of such applications is highly parallel computing jobs. Elastic scale and on-demand resource availability look like a good substrate to run embarrassingly parallel tasks at scale. Consequently, it motivated the appearance of several research and industry projects that adopt FaaS to run highly parallel jobs. On a first take, the "Occupy the cloud" [98] and ExCamera papers [65] demonstrated inspiring results from using FaaS for data analytics applications. On their basis, several works [35, 64, 141, 157] evolved on the idea of running compute-intensive parallel workloads on cloud functions and showed interesting results against traditional IaaS cluster computation. Some of the literature [67, 83, 99] analyzes these efforts and focuses on the challenges and viability to run data analytics workloads on FaaS platforms. Their conclusions show enticing results despite some issues. E.g., they discuss open challenges such as cost efficiency and statefulness. In sum, they convey that FaaS platforms are a good fit for data-processing parallel applications [98].

3.1.1 Scope and challenges

In parallel computing, many compute-intensive tasks or processes are executed simultaneously. Simultaneity is important since these tasks usually collaborate. Data analytics jobs, linear algebra, and iterative machine learning training algorithms are some examples. This requires a set of very specific properties in terms of resources, scale, and latency that allow to run all tasks at the same time without interleaving for compelling performance. Indeed, the information presented on the above papers shows that parallel applications on FaaS only make sense when the platform provides the necessary properties to enable their parallelism. However, they do not investigate them.

Simple function concurrency is not enough if each function invocation does not get its full isolated resources (we refer to each unit of resources as function instance). Otherwise, computation faces throttling and resource interference and becomes too slow and expensive compared to traditional clusters. In fact, most works on parallel computing atop FaaS presuppose that function invocations will run simultaneously, each on isolated resources [35, 65, 98, 141]. Consequently, the FaaS service must be able to provide enough resources at low enough latency to run all invocations in parallel.

However, all the aforementioned works base their arguments exclusively on the performance of AWS Lambda. While AWS seems to provide compelling values for the discussed properties [98, 113], they are not included in the simple FaaS definition of cloud functions that we presented above [51]. The properties are, in fact, particular of each implementation of the FaaS model and usually detailed on each platform's documentation. Still, cloud-offered FaaS platforms do not guarantee any of them: there are no service-level agreements (SLA) for these properties. More so, while function resources, timeouts, or even concurrency are clearly described by every platform, parallelism is not carefully addressed by any of them.

This raises an important question: **do current FaaS platforms fit parallel computations?** And also: what makes some FaaS platform a better fit for parallel applications than the others? Several benchmarking works [32, 113, 127, 177] compare different FaaS platforms. These papers indicate that indeed not all services provide the same properties. Unfortunately, existing literature approaches FaaS platforms from a high-level user perspective. They tackle use cases that resemble the I/O-bound, reactive applications FaaS is prepared for, and focus on properties such as latency, cold start, cost, and configuration capabilities. Some go beyond and explore elasticity. However, they tackle it as the ability to quickly handle dynamic workloads and disregard actual parallelism and the implications of the FaaS platform architecture. While this methodology is logical due to the black-box nature of the platforms, it does not allow to evaluate their suitability for highly parallel, compute-intensive applications. Indeed, understanding why each platform behaves as it does when they deal with parallelism requires a deeper knowledge of their architecture. And existing literature does not provide a detailed view of the architectures and management approaches of each platform, neither any of them tackle parallel computations in detail.

3.1.2 Contributions

To address the above challenges, in this chapter we carefully investigate the parallel performance of the four major cloud FaaS platforms. Namely, we analyze the architecture and performance of AWS Lambda (AWS), Azure Functions (Azure), Google Cloud Functions (GCP), and IBM Cloud Functions (IBM). We especially focus on details that would affect the ability of the services to provide a good substrate for highly parallel computations. First, we describe and analyze the design of each service based on available information. We are interested in how functions are managed, the virtualization technology used, how invocations are scheduled and their approach to scale, the management of resources, and other components that directly affect parallelism. To organize all these traits, we build a comparative framework that helps the description and posterior discussion on the differences between platforms. Second, we perform an experiment that allows to clearly visualize the parallelism of executions in a FaaS platform.[1] The experiment runs a job split into several function invocations (tasks) and produces plots with their execution timeline, drawing a complete view of the parallelism achieved. Combined with the information from their architectures, this visualization allows us to understand when new resources (function instances) are allocated to process function invocations, and whether resources are used simultaneously to handle different invocations in parallel. We can also see if this scheduling and resource management affects the performance of parallel tasks, such as by throttling invocations or by sharing resources across invocations (interleaving them).

Our objective is hence to understand their performance, and be able to spot bottlenecks, limitations, and other issues that can severely influence applications. In sum, we want to categorize characteristics of each service that must be considered and may help users understand the different platforms to choose the one that better fits their needs.

This chapter describes the following contributions:

- We present a detailed architectural analysis of the four major FaaS platforms: AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. We categorize their design through a comparative framework with special focus on parallelism. Two traits importantly influence parallelism of the platforms: *virtualization technology* and *scheduling approach*.

- We perform a detailed experiment to reveal invocation scheduling and parallelism on each platform. The experiment consists in running several function invocations concurrently and gather as much information as possible to draw a comprehensive timeline of the execution. This visualizes the parallelism achieved and reveals issues.

- We analyze the information gathered for the different platforms and their affinity to parallel computations. Generally, lighter virtualization technologies and proactive scheduling improve parallelism thanks to faster elasticity and finer resource allocation. Thus, platforms like AWS and IBM resolve parallel computations more satisfactorily than Azure, where our experiment only reaches a parallel degree of 11%.

[1] The experiment code and results for all platforms, including extra plots, are accessible at `https://github.com/danielBCN/faas-parallelism-benchmark`.

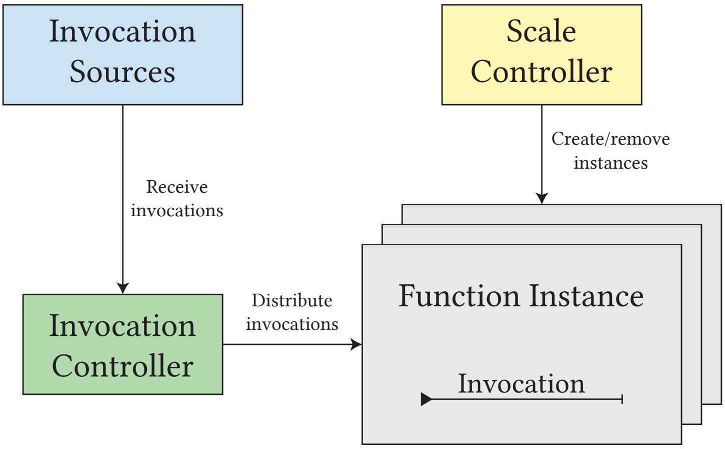


FIGURE 3.1: Abstract FaaS architecture.

3.2 Architecture analysis

In this section we describe the architecture of each FaaS platform. For an easy comprehension of the differences between services, we first create a comparative framework. We use it to outline the general organization, configuration possibilities, and documented limitations, and we put them in context with a description of their deployment model. Our interest is specially focused on *resource provisioning and scalability* to meet on-demand requests. Thus, we make emphasis on work distribution in terms of concurrency and parallelism. The descriptions on this section are all based on official information available online, unless indicated otherwise.

Figure 3.1 shows an abstract FaaS architecture with the main components we analyze in this section: function instances and invocations, the scale controller, the invocation controller, and invocation sources. We draw this schema based on open-source platforms and the literature [4, 133, 155, 173].

There is an important distinction in a FaaS platform: *function invocations* and *function instances*. Invocations are each one of the function executions in response to a request. Instances refer to the resource units that are provided to run invocations. If two function invocations are run on the same resource entity, we consider they run on the same function instance. This can happen by reusing a container, or by running several invocations in the same VM. While an invocation is easily identified on all platforms, each service manages instances differently. As we will see, function instances are usually determined by the virtualization used on each architecture.

The scale controller represents the logic that decides when to create or remove instances. The invocation controller is the logic that decides where to run each invocation that comes from invocation sources. In practice, these components may be merged into a single one; or be part of another component.

3.2.1 Comparative framework

For a handy comparison between FaaS platforms, we design a comparative framework to collect the most relevant characteristics of each one. It explores two items: (1) the general model of function deployment and management, and (2) the architectural approach to scale and resource management. Our focus is specially on the second one, since it conditions scalability and parallelism for each service, while the first provides important context. In this sense, we expand the second item by reviewing the following six traits:

Technology. In this category we discuss the virtualization technology used to build *function instances*. Instances need to be isolated resource units to provide multi-tenant properties. This is usually achieved with virtualization, but the chosen technology is very important for the design of a FaaS platform. Traditional VMs are heavier than containers, what makes the latter better for the irregular, low-latency FaaS scaling. But we also have microVMs, light as containers but with kernel-based virtualization. Some providers may combine technologies to efficiently handle isolation and performance.

Approach. This category analyzes the job of the invocation controller logic, i.e. the scheduling approach used to distribute work (invocations) across resources (instances). In particular, we categorize two kinds: push-based and pull-based. We refer as push-based to architectures that follow a proactive policy where a control plane takes the role of the invocation controller: the controller pushes invocations to instances. A pull-based architecture is more loose and reactive; the invocation controller logic is delegated to instances, which obtain work from the event sources: instances pull invocations from queues.

Scaling. This describes the scale controller. The scheduling approach heavily influences this component: push-based architectures usually merge the scale and invocation controller logic to balance load on demand, while pull-based ones use a dedicated scale controller to manage the instance pool. Here we also focus on the decisions of this component. For example, when does the controller create or remove instances?

Resources. Most platforms let users configure the resources that each function gets. We determine the *minimum guaranteed resources* for a single invocation with

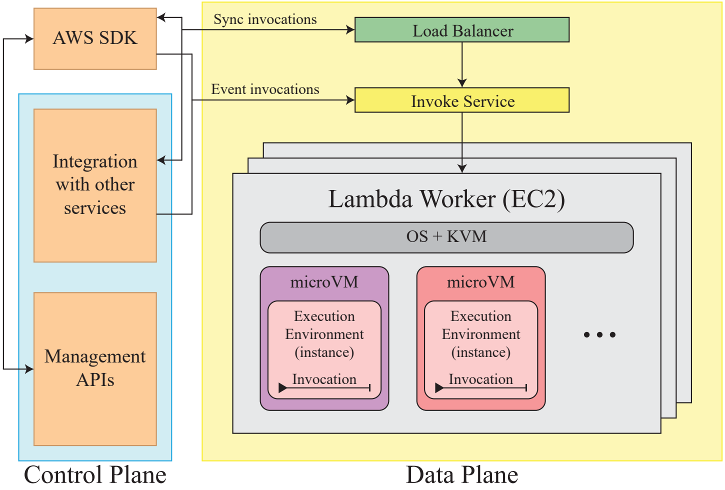


FIGURE 3.2: AWS Lambda architecture.

a particular function configuration. This is a product of the platform architecture and the tuning set by the provider. On one hand, how an architecture manages resources may introduce interferences across invocations. On the other hand, the service provider may set up some limits on the system that affect this category. For instance, resources could be restricted to ensure the proper functioning of the system or the economic viability of the service.

Parallelism. This category analyzes all information relative to function concurrency and parallelism. Particularly, we want to quantify the *maximum amount of parallelism* that a platform can achieve. It is important to remember that this is just an imposed limit, and the service does not guarantee (through an SLA) to reach such parallelism.

Rate limits. Providers protect their systems with use rates that block excessive request bursts and can limit parallelism. We illustrate it with the number of invocations per second the system accepts, but also discuss other limits related to parallelism.

3.2.2 Architecture of AWS Lambda

All AWS Lambda specification, configuration, and limitations are described in its documentation [10, 16]. Additionally, a recent AWS whitepaper [19] sketches its internals with more detail. An architecture overview is depicted in Figure 3.2. The service is split into the Control and Data planes. The Control plane handles the management API, such as creating or updating functions, and also includes integrations with other cloud services (e.g., forwarding S3 events or polling SQS queues). The Data plane manages resources and function invocations. The Invoke service is the main control component taking the logic of the invocation and scale controllers. Event-triggered invocations go directly to the Invoke, where they may be queued; synchronous invocations, which need extra management to respond to callers, are handled by a load balancer.

Function deployment

In AWS Lambda, the user deploys functions individually. The Management API enables function creation and configuration (e.g., runtime and memory). The function code is uploaded to the service in compressed packages and the configuration is updated with HTTP requests. Functions may be invoked with an HTTP request, but the usual approach is to bind them to function triggers. Triggers set up links with other cloud services that produce events and allow enabling invocations in response to those events. Configuration includes other features, such as limiting function concurrency and pre-provisioning resources.

Resources and scale

Technology. Lambda uses several virtualization levels in its architecture [19] (see Figure 3.2). The general structure changed recently with Firecracker [2], which enhances performance and management. We focus on the new model. The first level contains Lambda Workers, which are metal EC2 machines running a Firecracker hypervisor. This technology allows to populate Lambda Workers with microVMs that are quick to spawn and provide strong isolation. MicroVMs draw tenant boundaries, being each of them exclusive to a user. Within a microVM, the service creates *execution environments* to run the invocations. Execution environments are the function instances, created with the help of `cgroups` and other container technologies. Each of them is created especially for a function deployment, containing the appropriate runtime and function code, and can be reused for subsequent invocations. MicroVMs are not tied to a single function deployment and may hold several execution environments of the same user. With Firecracker, each microVM only contains a single execution environment at a time.

Approach. An official AWS whitepaper [19] depicts Lambda following a push-based scheduler. The Invoke service proactively designates the instance for each invocation. Upon a request, this component creates an execution environment (instance) inside a microVM or chooses an existing idle one. To perform such decision, this component must monitor all system resources. Then, the service pushes the invocation payload to the instance, where it is run.

Scaling. The Invoke service controls the scale at a multi-tenant level. It identifies the instance for each invocation among the cluster of Lambda Workers, which is common for all users. Since multi-tenancy is achieved at microVM level, the service can easily fill Lambda Workers. If the user performing a new invocation has no microVM available, the Invoke service finds the resources for a new one in the cluster. If there is already a microVM running, it can be reused for two cases: the existing execution environment is for the same function that is being invoked (and it is simply unfrozen and run with the new payload), or it is for another function (and a new container is created).

Resources. Users configure function memory from 128 MiB to 10 GiB. Then, instances will grant exactly that much memory for each invocation. To achieve so, a function instance only processes one invocation concurrently. With memory, Lambda scales other resources proportionally. In particular, 1792 MiB corresponds to the equivalent to one vCPU [10].

Parallelism. The service imposes a limit of 1000 concurrent executions per user—which can be increased under request [16]. Since there is no per-instance concurrency, the achievable parallelism shares this limit.

Rate limits. The request per second rate is very ample: 10 times the concurrent executions limit for synchronous and unlimited for asynchronous invocations. However, instance creation is controlled by a burst limit [15]. Depending on the region, the service creates from 500 to 3000 instances without any limitation in a burst phase. Reached that point, the number of instances created is limited to 500 each minute.

3.2.3 Architecture of Azure Functions

An architecture overview of Azure Functions is shown in Figure 3.3. A description of it is available in its documentation [28]. In the service, a set of function instances run invocations in response to events from different sources. The scale of this set is regulated by a long-running component that monitors the state of the service: the Scale Controller.

Function deployment

In Azure Functions, a Function App is the general unit of management and deployment. Each Function App works as a bundle that may contain many function

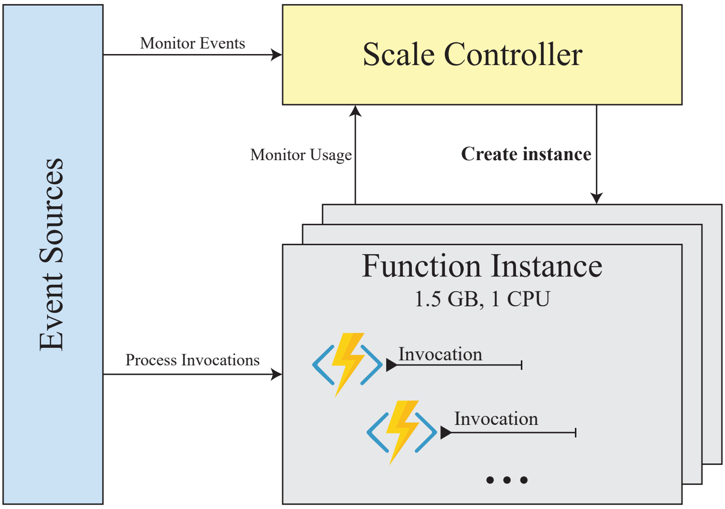


Figure 3.3: Azure Functions architecture.

definitions and manages a pool of resources (function instances). The application package the user uploads includes their code, dependencies, and configuration. Each function definition is a piece of code correctly annotated as an Azure Function. The next part of configuration is function triggers and bindings, which define the events that will result in function invocations and enable functions to operate input and output streams. Advanced configuration parameters allow to tune some extra features.

Resources and scale

Technology. Azure Functions is built atop Azure WebJobs, a Web App PaaS service that auto-scales a VM cluster based on load. Function instances are therefore VM hosts with fixed resources and the whole Function App deployment package installed. The set of instances is managed by Azure WebJobs within each Function App. Differently from all other platforms, Azure uses Windows hosts by default, instead of Linux.

Approach. The documentation of Azure Functions depicts a pull-based scheduling approach [28]. Function instances poll event sources to process function invocations. When an instance finds an unprocessed request in one of its bound triggers, it runs it. An instance can run any function definition in the Function App and several invocations may be taken by the same instance concurrently. This means that different invocations (same Function App) may share resources.

Scaling. The Scale Controller manages the number of Function Instances in a Function App. This component monitors the event rates and instance usage to determine when to create or remove VMs. The actions of this manager are dictated by a set of internal policies. For example, it only creates one instance per second if invocations are by HTTP request [28].

Resources. The available resources on each instance depend on the plan the Function App is deployed on: Consumption plan or Premium plan [28]. We focus on the Consumption plan since it is the serverless one. It presents the typical FaaS properties of fine-grained pay-per-use and scale to zero. But differently from other platforms, resources are not configurable, and all instances have 1.5 GiB of memory and one CPU. This means that an invocation may get *at most* these resources. Remember that each instance may take several invocations concurrently, so there are no guaranteed resources for each invocation. The Premium plan allows increased performance by pre-provisioning resources. The user defines a lower and upper limit to the number of instances, that do not scale to zero.

Parallelism. The documentation of Azure Functions depicts the service clearly not focused on parallelism. The number of instances per Function App is limited to 200 and cannot be increased [28]. However, it seems to be built for small I/O-bound tasks that benefit from concurrency. A single instance may choose to fetch several invocations from the event sources at the same time, allowing unlimited invocation concurrency by sharing instance resources, a good fit for I/O operations. The actual parallelism is thus limited to the number of instances since they only have one CPU each. To avoid resource interference, a necessity for compute-heavy tasks, concurrency can be configured by the user by setting a per instance limit [27]. There is a different limit for each trigger type, and they are managed by the instances autonomously. For example, HTTP requests have a default limit of 100 concurrent invocations per instance, which, after scaling to the maximum 200 instances, could offer 20K concurrent invocations. This does not improve parallelism.

Rate limits. There is no service limit on the number of invocations processed per second. It depends on the functions themselves (user code) and how many of them the available instances can process following the service polices (at which rate they pull from queues). Note that there is a limit on the instance creation rate: one per second based on HTTP trigger load, and one every 30 seconds for other triggers [28].

3.2.4 Architecture of Google Cloud Functions

The general concepts of the architecture of Google Cloud Functions are detailed in its documentation [75]. However, it does not specify its internal components with clarity, such as which component runs the invocation and scale control logic. Consequently, we do not present an overview scheme for this platform. This only affects the scheduling approach and scaling categories of our comparative framework. The documentation gives enough information for the other categories.

Function deployment

In Google's FaaS, the unit of deployment is a single function. The system manages each function separately, even if deployed on the same package, and scales them individually. To deploy a function, the CLI uploads the code directory and detects functions based on project structure conventions. The configuration is updated through HTTP calls to the service API. Functions may always be invoked with HTTP requests, but the user may also associate them with triggers to generate invocations in response to events from other services.

Resources and scale

Technology. To isolate executions across tenants, Google Cloud Functions uses gVisor microVMs [76]. gVisor [9] is a kernel-based virtualization tool used to securely sandbox containers. These containers are the function instances that run user code, taking only one invocation at a time [75]. MicroVMs allow to strongly isolate real resources between tenants; however, there is no information about how many containers can be packed in the same microVM or how the service ensures each of them has the resources configured for the function.

Approach. There is no information available about the internals of the service that enables us to make any detailed evaluation of its scheduling policy. Documentation points to a push-based approach [75], where a controlling component manages invocations and scale.

Scaling. Following the previous category, we sketch the existence of a controller component in the system that collects system information and decides when to scale in or out. The reasons behind scaling decisions are listed in the documentation [78] and include the usual running time (short functions scale more), the cold start time, the rate limits of the service, function error rates, and the load of the servers at the time.

Resources. Users configure memory for their functions. The service offers 5 possible sizes from 128 MiB to 2 GiB and assigns CPU therefrom [77]. Note that this relation is not proportional: for instance, 256 MiB functions are given 400 MHz of CPU, while 2 GiB, 2.4 GHz. The documentation states this numbers as approximations and not guaranteed resources. Thus, we expand this information with a simple exploratory work. Inspecting system information (`/proc`), we see that all containers run in VMs with 2 GiB of memory and 2 CPUs at 2.3 or 2.7 GHz. This happens irrespective of the function configuration, which tells us that all microVMs are equally sized. Again, it is unknown if different containers are packed in the same VM.

Parallelism. There is no limit on invocation concurrency for functions called with HTTP requests [78]. Event-triggered invocations are limited to 1000 concurrent executions per function (not increasable). Advanced configuration also allows users to limit the number of concurrent instances. Since each instance only allows one invocation at a time, parallelism is also bound by these limits. Thus, maximum parallelism is unbound for HTTP-triggered functions.

Rate limits. Google Cloud Functions sets a per region limit of 100M function invocations per 100 seconds [78]. Additionally, the CPU usage is limited by other rates. These quotas are fairly generous for the majority of applications.

3.2.5 Architecture of IBM Cloud Functions

IBM Cloud Functions is a cloud-managed Apache OpenWhisk [4] deployment, an open-source FaaS started by IBM and donated to the Apache Software Foundation. Its design is expounded in both documentations [90]. Figure 3.4 overviews the platform, with four main components: the Controller acts as a load balancer and manages instance resources; the Invoker machines are VMs that run several containers (the function instances, usually Docker); a Kafka deployment communicates them at scale; and a database (CouchDB) stores function information, request data (payload and results), and logs.

Function deployment

In IBM Cloud, functions are called “Actions” and deployed individually to the service. Actions must be contained in a namespace, which belongs to a resource group, and may be organized in packages. Action definitions (code and configuration) are registered in the database through the Controller, that exposes an HTTP API. Like Actions, the user defines Triggers and Rules. Triggers identify

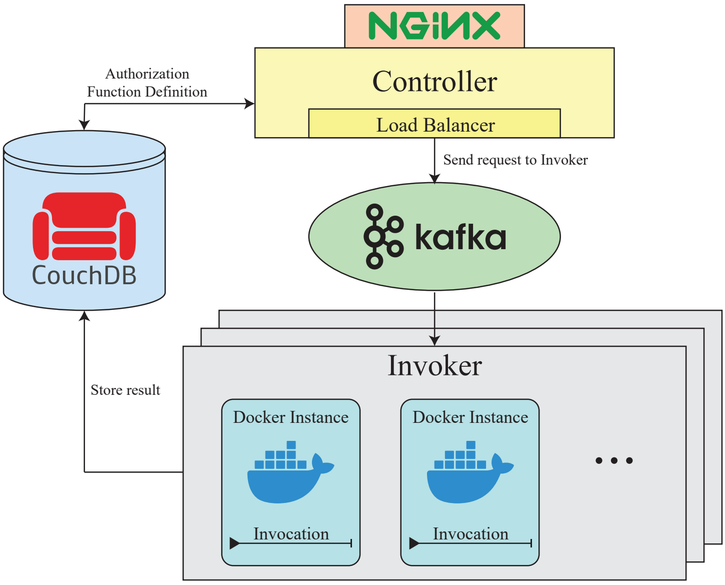


FIGURE 3.4: IBM Cloud Functions (OpenWhisk) architecture.

event sources to monitor, while Rules are event filters to map Triggers to Actions. Actions can always be invoked directly with HTTP requests.

Resources and scale

Technology. Function instances are containers that are run on a cluster of Invoker machines, which are VMs. Each Invoker manages its local pool of containers, while the Controller is responsible for the pool of Invoker machines. Thus, IBM's FaaS platform has two levels of virtualization that we can analyze.

Approach. OpenWhisk follows a push-based scheduling approach [90]. The invocation control logic is split between two components. The Controller, upon a request, forwards it to a designated Invoker machine. The Invoker then creates or reuses an idle container (instance) to run it. This one-to-one communication is performed asynchronously through Kafka. The Controller acts as a load balancer while monitoring the state of all Invoker VMs. Thus, the Controller proactively pushes function invocations to the instances that run them.

Scaling. The scale control logic is also split. Each Invoker machine locally manages its containers. With a fixed set of memory assignable to containers on the machine and the functions' memory configuration, the Invoker responds to requests by creating containers with the right resources and informs the Controller of its usage levels. The Controller manages the general pool of Invokers and sends requests to them prioritizing the ones that already have warm, but idle, containers. If none is available, it chooses one Invoker with enough free resources to create a new instance. There is no information on when or how Invokers are created or removed, or if the set is fixed.

Resources. Users configure function memory, and each instance provides those resources to each invocation. The service does not ensure any CPU resources for a given memory, but it claims to scale resources proportionally. To collect more specific information, we empirically study the platform (more details in Section 3.7.2). Inspecting system information (`/proc`) we see that all explored machines (Invokers) run 4-core CPUs and 16 GiB of RAM. However, in our experiments, a single Invoker seems to dedicate only up to 8 GiB for container hosting. If resources scale proportionally, this CPU-memory relation tells us that we could ensure a full CPU core with 2 GiB functions.

Parallelism. The service has a limit of 1000 executing or queued concurrent invocations per namespace—increasable under request [91]. Each instance only takes one invocation at a time, meaning that the maximum parallelism of the platform is the same as this imposed limit. In fact, OpenWhisk offers a configuration parameter to manage per-instance concurrency [7], with which a single instance could take several invocations at the same time (unavailable on the IBM Cloud). While this increases concurrency, it does not improve parallelism.

Rate limits. No more than 5000 invocations can be submitted per namespace per minute—also increasable [91]. It does not directly affect parallelism, since the concurrency limit is smaller. However, for applications that run many small tasks, it can be easy to reach. Examples are parallel computations with dynamic load balancing and consecutive batches of tiny tasks.

3.2.6 Architecture summary

Table 3.1 summarizes all traits collected for the four FaaS platforms. They are all obtained directly from their documentation and official publications as of October 2020. The exceptions are Google's scheduling approach and scaling, which are not clearly described; and the guaranteed resources at IBM, which we empirically assess in next sections. Let us review these traits next:

Technology. Each provider uses a different virtualization technology. AWS and Google use several virtualization levels and include microVMs. This allows finer resource management with small start-up times and increased security. IBM also

Table 3.1: Traits of FaaS platforms (as of October 2020). See Section 3.2.1 for trait descriptions.

	Technology	Approach	Scaling
AWS	microVM	push-based	Control plane
Azure	VM	pull-based	Scale Controller
GCP	microVM	*push-based*	*Controller*
IBM	VM + Container	push-based	Controller

	Resources	Parallelism	Rate limit
AWS	1792 MiB = 1 CPU	(ext.) 1000	3000 + 500/min
Azure	1 CPU[2]	200	unbound
GCP	2 GiB = 2.4 GHz	unbound	100M/100 s
IBM	2 GiB = 1 CPU	(ext.) 1000	5000/min

has several virtualization levels, but does not use microVMs. Consequently, packs of containers run on each VM, requiring a different approach to security. Azure only has one level of virtualization, simplifying resource management at the cost of elasticity. In sum, the schema of virtualization technologies is really important for the architecture, as it influences several factors that must be considered for scheduling and managing the service, e.g., security and the time it takes to start an instance.

Approach. Only Azure clearly uses a pull-based approach to scheduling work. The other providers build push-based architectures that create instances more eagerly. This benefits parallelism, as they are faster to create instances. From the table, the scheduling approach seems tightly related to the virtualization technologies used. Azure manages a single VMs level and takes a conservative approach to scale. Meanwhile, the others use lighter technologies and spawn instances with more liberty.

Scaling. There is always a controlling component that manages scale in the system. In push-based platforms, scale and invocation distribution logics are dealt by the same control component. In the pull-based, the controller manages scale based on the state of the system but does not deal with invocations.

Resources. Instances usually have fixed resources, based on function configuration. Most providers let users configure function memory and scale other resources, like

[2] In Azure, a single invocation enjoys a full CPU only with instance concurrency limited to one (see Section 3.2.3).

CPU, proportionally. Azure does not allow configuring resources but monitors usage during execution to adjust billing [26]. Even with their different configuration options, all providers offer at least 1 vCPU with around 2 GiB of memory. They allow users to ensure certain amounts of resources.

Parallelism. The achievable parallelism is quite good for AWS, GCP and IBM, with generous limits on concurrency. Azure, however, has restricted parallelism due to its scheduling approach, strict limits, and system tuning.

Rate limit. Invocation rate limits do not generally restrict parallelism in any platform. 5000 invocations per minute at IBM is the most restrictive; but it can be increased under request.

This section provides a summary of the four platforms and several aspects that heavily affect the parallelism they offer. Combined with some reasoning, we can start to shape our expectations for the different services. However, none of them guarantee these properties through an SLA. For example, the instance resources described in the documentation should be taken as approximations and the maximum parallelism as just an upper limit. For this reason, we empirically assess these properties in the next sections.

3.3 Experiment methodology

We explore the different architectures by empirically evaluating a real parallel workload. For this, we design an experiment to show how multiple simultaneous function invocations are distributed across instances. This validates the performance of parallel tasks on each FaaS platform.

The general methodology consists in running many concurrent requests to a function while gathering execution information. We use this information to draw an execution *chart*. In particular, we plot parallelism clearly by depicting the invocations on a timeline that identifies function instances.

This section starts by setting up a set of questions that motivate the experiment. Followed by a description of the test function and its different configurations, including a big scale setup. We discuss several considerations regarding the execution of this evaluation and define a common notation for the experiment parameters. Then, we present a set of metrics that characterize each platform for parallel computations. The section ends with the description of the plot resulting from the experiment.

3.3.1 Questions for discussion

The benchmark is designed with the following questions in mind, which define our goals for evaluation. We analyze them on a per-platform basis through Sections 3.4 to 3.7.

Q1 *Does the service scale function instances elastically to fit parallel tasks?* Related to the technology and scheduling approach of each platform, this question validates if the design is useful to reach parallelism in practice. In essence, do concurrent invocations actually get different instances? Coincidentally, we also identify the maximum parallelism achieved in practice, in contrast to the upper bound described before.

Q2 *Does the service ensure instance resources so that there is no interference across function invocations?* This question validates the actual resources gotten on each platform and if there are any issues, such as resource interference, when running a parallel workload. The objective is to verify the information about instance resources from the documentation (Section 3.2).

Q3 *What can we deduce from the scheduling of the system and its general performance?* This last question embraces general information that can be learned from the experiments. Including: invocation latency and how it changes with scale; possible performance issues; tendencies in cold starts; insights on internal policies and tuning for resource management and scheduling; and any other useful information.

3.3.2 Function definition

The questions above determine the information that we need to collect in our experiments. Next, we detail the definition of the function that will run our benchmark to collect that data on the different FaaS platforms. The function has two jobs: gathering information and performing work. We obtain as much execution information as possible for each platform, which means different code and resulting plots. Nonetheless, there are three basic items that we require: (1) client-side execution times for each invocation, (2) intra-function execution times (actual invocation duration), and (3) function instance identification. Client-side times can be acquired irrespective of the platform. However, the other two items may be obtained differently on each service. Function instance identifiers are never exposed by the services, and we use different techniques to obtain them (detailed

Table 3.2: Function configuration for the different platforms.

	Memory (MiB)	CPU	Region
AWS	256, 2048	1/7, 8/7 CPU	`us-east-1`
Azure	1536	1 CPU	France Central
GCP	256, 2048	0.4, 2.4 GHz	`us-central1`
IBM	256, 2048	1/8, 1 CPU	Washington DC

on their respective sections). We complement the data by inspecting `/proc` when available, since it can offer valuable information about the virtualization level and system configuration. Extended discussion on how to obtain execution information can be found in the literature [122, 177].

As for workload, we experiment with two kinds of tasks: a simple sleep and a compute-intensive job. The sleep is a baseline to explore the scheduling pattern of the service. We use a 1-second sleep, which is enough to plot a comprehensive timeline, while longer tasks could complicate the information due to concurrency. The compute task is intended to mimic a real embarrassingly parallel workload and reveal issues with resource availability and interference. For easy reasoning, this task has a clearly defined time duration. In particular, we run a Monte Carlo simulation where an invocation performs x iterations to approximate π. x is configured and evaluated to represent a consistent amount of time, close to 1 second.

In detail, the function does the following: (1) get the current time, (2) identify invocation and instance, (3) perform the workload, (4) get the current time, and (5) return the collected data. We obtain the initial time right from the start to represent when user code starts to run in the cloud. We checked that the overhead of the second step is consistent across invocations and not significant against the actual workload under test.

The invocations are run with a Python script that performs synchronous HTTP requests concurrently with the `asyncio` module. We use the `httpx` client with the authentication methods required by each platform. For AWS, we use the `aiobotocore` client: a simple wrapper for signed HTTP calls. The information collected is complemented with client-side data and appended to a file, that is later used to draw the execution plot.

3.3.3 Function configuration

Table 3.2 summarizes the function configuration parameters for each platform. The default timeouts on all platforms are enough for our one-second functions

TABLE 3.3: Compute-intensive task on each platform.

	Runtime	Iterations	2048 MiB	256 MiB
AWS	Python	5M	1.1 s	7.7 s
Azure	$C^\sharp$	20M	1.2 s (1.5 GiB)[4]	
GCP	Python	5M	1.3 s	3.5 s[5]
IBM	Python	5M	1.3 s	1.3 s[6]

(5 min on Azure and 1 on the others). We test two memory configurations to assess performance and resource management for different function sizes. One (*big* – 2048 MiB) intends to reach a full CPU on all platforms; the other (*small* – 256 MiB) is small enough to reveal the scheduling of the system.[3] In Table 3.2, we include the presumed CPU for each platform and memory configuration; refer to Section 3.2 for details on memory and CPU mapping. Regions are chosen based on what they offer (availability zones, better network, more services, etc.) to ensure best function performance and parallelism. Different regions may affect request latency, but not the service parallelism we analyze. The function is written in Python for all platforms but Azure ($C^\sharp$), whose support for the language was in preview during the experiments. This does not affect the benchmark since we execute 1 second of computation on all platforms either way. To that end, the compute-bound task performs 5M iterations on Python and 20M on $C^\sharp$. See Table 3.3 for a complete relation of task duration on each platform and configuration. While different languages may affect cold start time, configuration is consistent for each execution and the parallelism in the plots is unaffected. We take this into consideration when comparing across platforms. All functions are triggered by HTTP requests and have logs and monitoring services active.

3.3.4 On a bigger scale

We want to confirm our conclusions by assessing large scale executions of the benchmark. Our detailed plot (Section 3.3.7) becomes too noisy for analysis when targeting such configurations. For this reason, we complement our results with an extra execution of 1000 invocations that uses a simplified plot. This plot includes the function execution time bars in a timeline together with a curve representing the number of function instances running at each instant, showing the evolution

[3] This does not apply for Azure since resources cannot be configured.

[4] Since Azure does not allow resource configuration, we only show one time.

[5] Results from cold starts. Warm containers are slower. See Section 3.6.

[6] We discuss IBM's equal performance for both configurations in Section 3.7.

of the experiment concurrency. In addition, we add a complementary histogram of the invocation execution time that helps identify resource interference between invocations.

With that many invocations, synchronous HTTP requests are inconvenient for parallel executions, so we opt for asynchronous invocations instead. This difference may result in different strategies for the platforms to scale resources and we will keep this in mind when analyzing these executions. In any case, the results are in line with the tendencies observed in the more detailed experiments, which tells us that the invocation method may not affect parallel performance.

This experiment runs a CPU-intensive task. Specifically, each task computes several matrix multiplication calculations that last around a minute in total. Function memory is fixed to 1024 MiB for each FaaS service. This implies that the portion of CPU assigned to a function varies between cloud providers. The scale is 1000 invocations of this task. This means that the same workload performed on a single core would take approximately 16 hours. In this case, the use of asynchronous triggers requires the result to be sent to the object storage available in the cloud provider (e.g., S3 for AWS Lambda) and retrieved from the client after completion. Note that the times displayed in the plot only represent the function execution time and not the overhead produced in uploading the result to object storage.

3.3.5 Experiment execution

The benchmark was run during May 2020 from a single client machine (a laptop with a 4-core hyper-threaded CPU @2.6 GHz) invoking functions to the different platforms. The consequent invocation latency reflects only in the time between the client timestamp and the function start, and thus does not influence the display of parallelism in the plots.

Executions were run during different days and hours. All configurations were tested several times, and all showed similar results. The complex nature of the plots (detailing a single execution to show its work distribution) makes it difficult to show all the data in an aggregated format that is readable and informative. Therefore, we selected some of the executions to give the reader the general idea of the behavior of each platform.

The executions may find arbitrary numbers of warm and cold instances as they are executed in succession. This is because warm starts depend on the platform and its particular policies for recycling instances. Consequently, it is not possible to ensure a consistent number of warm instances across executions. However, we can collect this information afterwards and compare it with the number of instances available in the previous execution. Since we consider it important for evaluating parallelism, we include data on the number of warm starts experienced on each execution. For example, when running an execution first with 10 invocations and then with 50, if the platform creates 10 instances for the first run, the second one is expected to usually find 10 instances warm.

To account for all the different configurations and system state, we use a simple notation system throughout the evaluation to describe the complete setup of each experiment. The notation is: $I/W/T/M$. Where I is the number of invocations in that experiment, W is the expected number of warm instances staying from a previous execution, T is the workload type for the function (S–sleep or C–compute), and M is the memory size for the functions (s–*small* or b–*big*, as introduced above). For example, the notation $200/50/C/b$ indicates an execution with 200 invocations, expecting 50 warm instances, and performed the compute-intensive task on *big* (2048 MiB) functions.

3.3.6 Metrics

To summarize the results of our benchmark, we establish the following metrics that characterize the capabilities of the different FaaS platforms to host parallel computations:

Cold start. Instance creation overhead is a direct result of the virtualization technology and the scheduling approach. Other benchmarks [127, 177] show that the cold start depends on the function runtime configuration and analyze it in detail. We do not consider our values for cross-platform comparison due to different latencies to each cloud. Hence, we only point out general tendencies and its effects to the system in its behavior.

Completion time. This is a good indicator of the achieved parallelism, and specially of the simultaneity of invocations. With this metric we quantify approximately how long it takes to run 200 *big* compute tasks on each platform. Each task individually takes one second. Hence a perfect system would run any number of this task within that second. However, platforms add overhead to the execution, such as invocation delay.

Parallel degree. We define the *parallel degree of a platform* in an experiment as the maximum number of instances used at the same time throughout the experiment. We also include the percentage that this represents out of the total number of invocations. We account this for the same setup as the previous metric, so 100% parallelism means the use of 200 instances at the same time.

Failed requests. These are a hassle for parallelism, as they become stragglers, need retrying, and heavily impact total computation time. With synchronous invocations, like our case, the platform delegates retries to the caller, making the process slower and increasing complexity for the user.

3.3.7 Plot description

The information gathered by all invocations in an experiment is represented in a Gantt-like plot showing the execution period of each function invocation in a global timeline of the run. Using different colors, the plot shows on which function instance each invocation has run. This allows to see the real parallelism achieved and spot concurrency problems (like per-instance concurrency or invocation throttling).

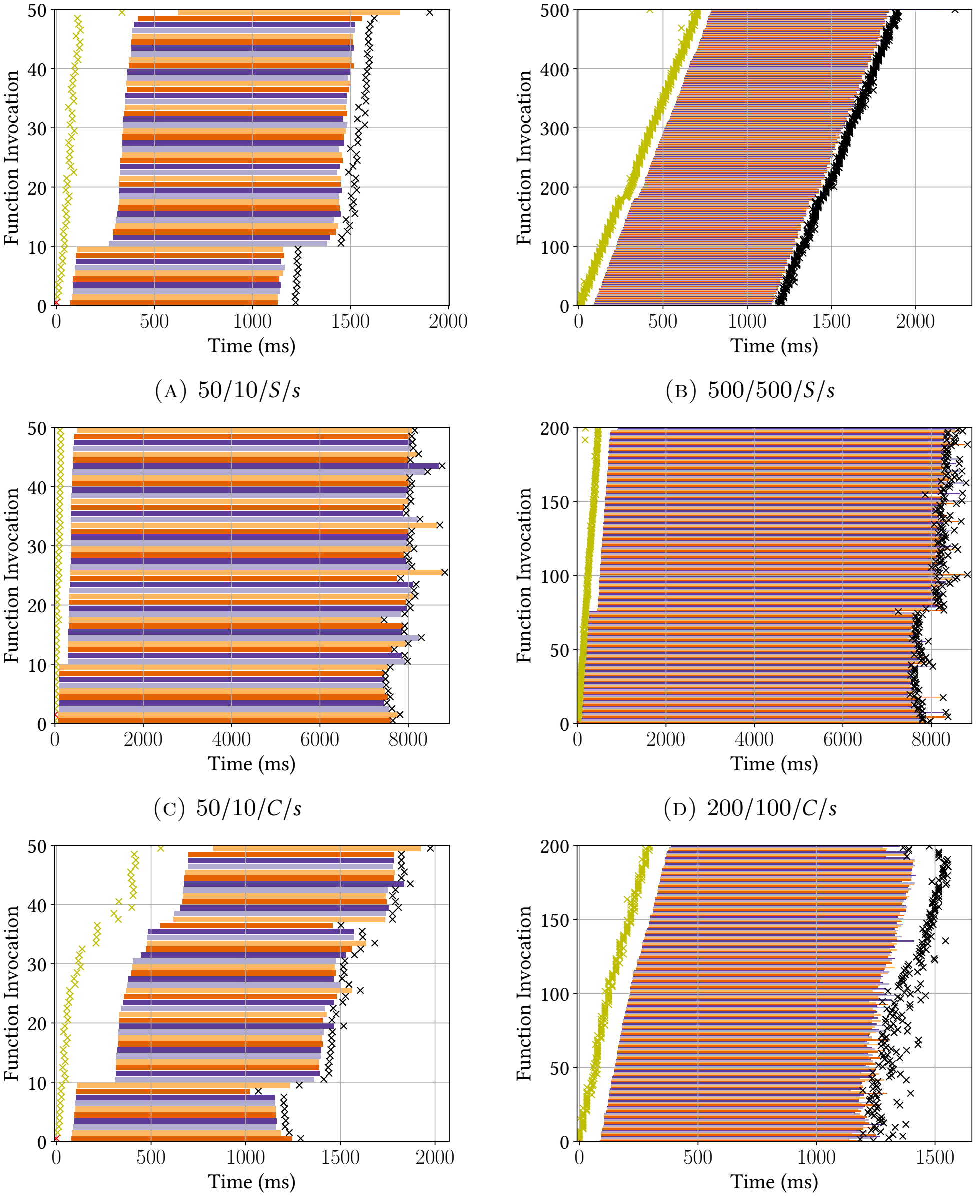
In the plots, the horizontal axis is the timeline. Our time zero is the minimum timestamp playing in the experiment: the first client invocation (red `X`). All other times are deltas to this one. The vertical axis stacks the function invocations. Each invocation is drawn as a horizontal bar indicating its timespan, i.e. the time it has been running in the cloud. The yellow `X`s indicate the client-side invocation timestamp and the black ones, the request return. Bar colors differentiate the instance where each invocation has been run. Although colors are limited to four, since the plot groups invocations by instance, instances sharing a color are always separated by instances with other colors, making the distinction clear.

3.4 Experiment on Amazon Web Services

We deploy and update our function with the AWS CLI. The invocation ID is obtained through the function context object. The instance ID is the random identifier present at `/proc/self/cgroup`, starting with `sandbox-root` [177].

3.4.1 Results

Experiments with sleeping functions We start with the *small* (256 MiB) functions and the sleeping task. A first run with $10/0/S/s$ shows how the system creates a different container for each invocation, allowing full parallelism. A subsequent execution with $50/10/S/s$ results in Figure 3.5a. Note that the cold start increases invocation latency by $\approx$ 200 ms. Still, the service achieves full parallelism. Figure 3.5b shows $500/500/S/s$; the service still creates different containers for each invocation.



(A) 50/10/S/s

(B) 500/500/S/s

(C) 50/10/C/s

(D) 200/100/C/s

(E) 50/10/C/b

(F) 200/200/C/b

FIGURE 3.5: Experiment on AWS.

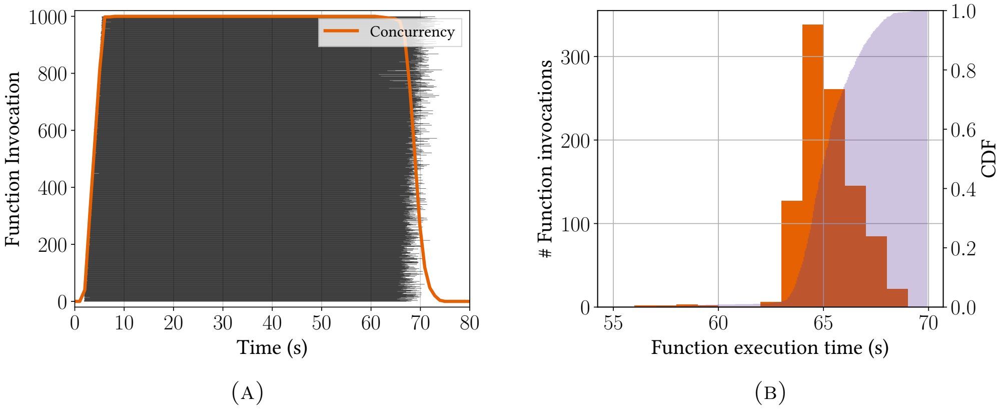


FIGURE 3.6: Large-scale experiment on AWS.

Experiments with computing functions Still with *small* functions, we move to the compute-intensive task. Running a single invocation, the computation takes 7.7 seconds average with this configuration. Figures 3.5c and 3.5d show subsequent invocations of this experiment with different parallelism. The variance of execution time is within one second.

With the *big* functions, which have a full CPU, execution time for the individual run reduces to 1.1 seconds average. Figures 3.5e and 3.5f show the experiment with different parallelism. Execution time is never far from the individual execution with a bit more variance than with the *small* configuration.

On a bigger scale Figure 3.6 shows the results of executing the larger configuration with 1000 parallel requests. Full parallelism is fulfilled even for big scale executions on AWS Lambda. The histogram shows that all invocations do not vary much from around 65 s run time, confirming resource homogeneity.

3.4.2 Discussion

Q1 All experiments show good parallelism, scaling rapidly to the number of requests. The overhead is small, and all invocations run in different instances at the same time. In particular, the experiment with 500 concurrent requests shows that the server can keep dealing invocations at the pace the client is able to create. Our larger configuration confirms that AWS Lambda scales to thousands with asynchronous invocations [98].

Q2 CPU resources scale with memory as documented [10]. Function performance is constant with little variance (i.e., there is no interference). This suggests that provisioning and isolation are strict, not only for memory but also for other resources. We can clearly see this from the compute-intensive experiments. Our task takes ≈ 1.1 s with the full CPU (*big* functions) and ≈ 7.7 s with the *small* functions. Since a full CPU is reached at 1792 MiB, our 256 MiB functions are 7 times smaller and should have 1/7 of CPU. Accordingly, our *small* functions take 7 times more than the *big* ones.

Q3 The experiments also reveal these conclusions: (i) Scheduling allows generous resource allocation in burst. Containers are immediately created when none are available. (ii) Instances are set up for processing quite fast, probably a result of using a microVM technology. (iii) Even with cold starts, invocation latency is usually below 300 ms, including client-cloud latency.

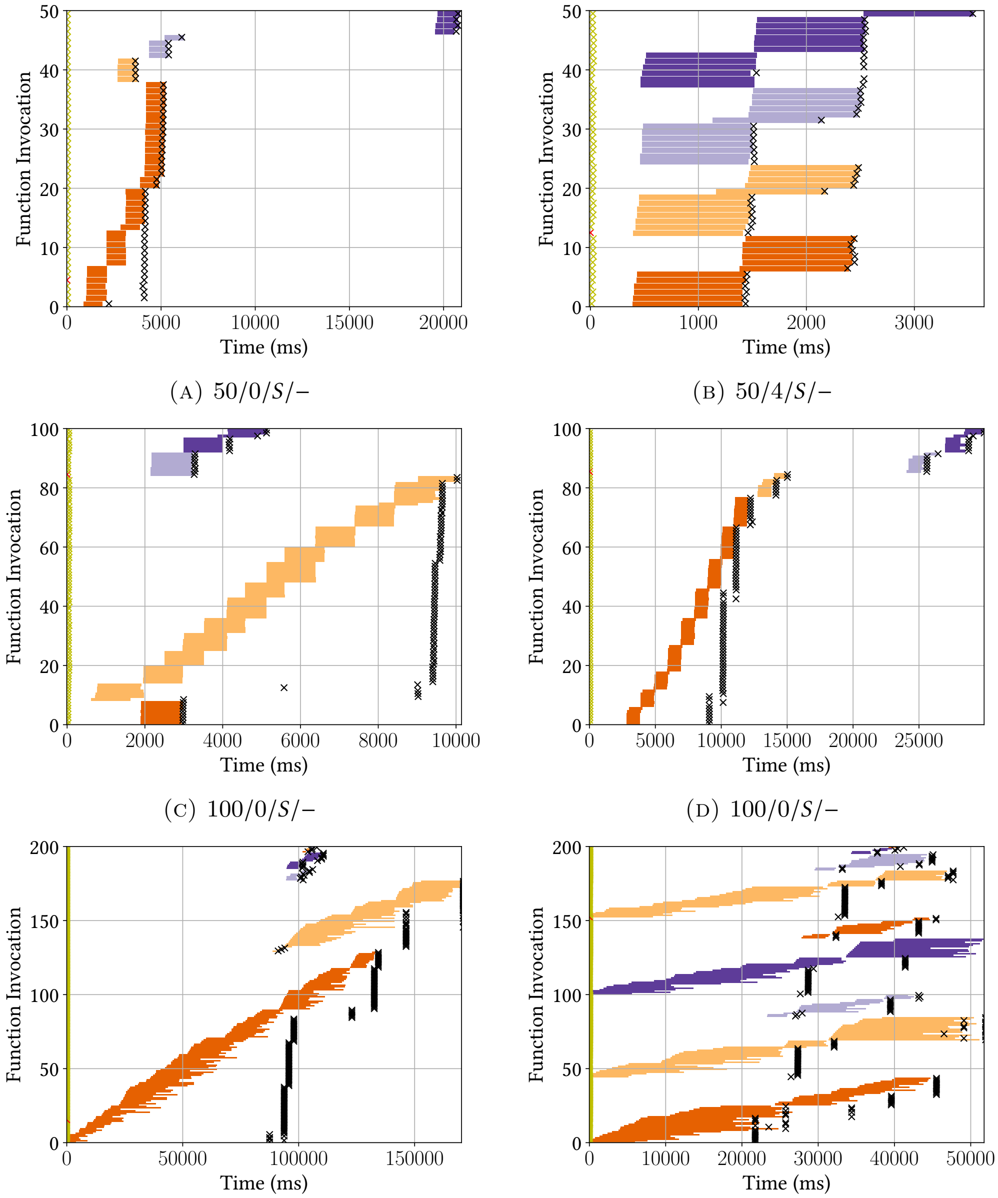
3.5 Experiment on Microsoft Azure

The development and deployment of Function Apps is managed with the Visual Studio Code extensions, as recommended in the documentation [23]. The invocation ID is obtained through the function context object available as an optional function parameter (inherited from WebJobs). For the instance ID, an environment variable (“`WEBSITE_INSTANCE_ID`”) is present from Azure WebJobs and identifies a function instance [29]. We also use Live Metrics, an Azure service that shows real time detailed information for a Function App, such as the number of active servers (instances), or CPU and memory usage, among others.

3.5.1 Results

Experiments with sleeping functions We start with the default configuration and the sleeping task. A first execution with 50 parallel requests results in Figure 3.7a, which shows a cold start. With this run, the service ends with 4 instances. A subsequent execution of the same experiment results in Figure 3.7b. In this case, the 4 hosts were already running, and start processing invocations right away. Figures 3.7c and 3.7d show the same experiment with 100 parallel requests, both without previously running instances. They demonstrate that two executions with the same parameters can be scaled differently in this platform.

Experiments with computing functions Now, we switch to the compute-intensive tasks. Running 50 or 100 parallel requests do not get more than a single



(A) 50/0/S/−

(B) 50/4/S/−

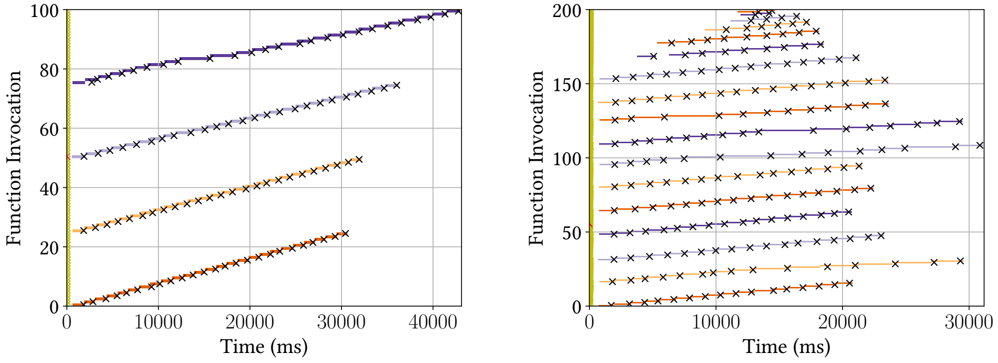
(C) 100/0/S/−

(D) 100/0/S/−

(E) 200/1/C/−

(F) 200/4/C/−

FIGURE 3.7: Experiment on Azure.



(A) 100/4/C/–

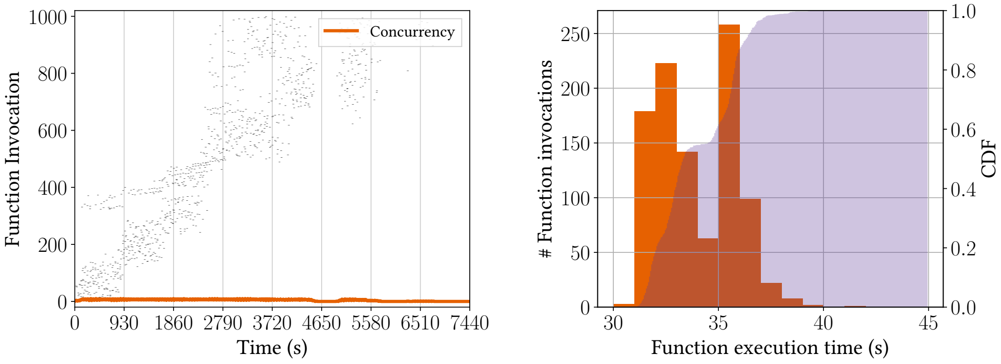
(B) 200/13/C/–

FIGURE 3.8: Experiment on Azure with limited concurrency.

instance, thus we increase the workload. Repeating the same experiment (50 and 100 parallel requests) several times in quick succession does not alter results. We then run 200 parallel requests, which results in Figure 3.7e. The system finally creates new instances (up to 7 as confirmed with Live Metrics). Right away, we run the experiment again, which we plot in Figure 3.7f. We see that requests only run on 4 instances at first, but then scale out to 9. In this case, Live Metrics tells us that the service created up to 12 servers, but some of them did not get any work.

Limiting function invocation concurrency per instance Since the default configuration is a bad fit for compute-intensive tasks, we run the experiments limiting per-instance invocation concurrency as explained in Section 3.2.3.

Due to the CPU-intensive nature of our tasks, our experiment benefits from limiting concurrency to 1 to avoid resource interference. Now invocations take the expected time ($\approx$ 1.2 s). In the previous executions, resource sharing was extending execution time by 40x. Like before, with larger experiments the system does not create more than 4 instances (Figure 3.8a) until reaching 200 concurrent requests. For instance, Figure 3.8b shows an execution where 13 instances were already up and ends with 18 servers processing invocations. As a note, this last experiment runs 200 tasks (embarrassingly parallel), each of them with an expected duration of 1.2 seconds. Such computation should take 1.2 seconds plus some system overhead (all tasks are parallel). However, the whole experiment takes more than 30 seconds with a maximum parallelism of 18.



(A)

(B)

FIGURE 3.9: Large-scale experiment on Azure.

Revisited Due to the poor parallelism experienced, we decide to revisit this experiment in March 2021. The configuration is the same but for the region of deployment. Since the experiments on the other platforms were performed on US regions, we move to "Central US". This way, we discard the datacenter from causing these problems and avoid peak hours on that region in case heavy traffic of other users may have affected performance. However, we find the same behavior experienced months before. Indeed, the low parallelism seems related to the Scale Controller component and its policies for spawning new instances and not to the load in a specific datacenter.

On a bigger scale Figure 3.9 shows the results of executing the larger configuration with 1000 parallel requests. In line with the previous runs, concurrency is very limited with just a few instances, affecting the total execution time. The histogram shows fairly consistent run times, meaning that, when limiting per instance concurrency, the resources for each invocation are well ensured. Azure, always having a full vCPU regardless of configuration, has faster execution times than the other platforms (in this experiment the others have less than a vCPU).

3.5.2 Discussion

Q1 Azure Functions is not designed for high parallelism or heavy computation. Our experiments clearly show that the service is reluctant to scale and function invocations are queued on a few instances. Also, instances take invocations at irregular intervals, even when processing other invocations. In general, but most noticeable with computing tasks, the service does not create instances until there is high load, meaning that, in some cases, 100 requests end up being handled by the same instance. Changing configuration to limit instance concurrency confirms that the system needs considerable load to spin up new instances. In particular, only a parallelism of 18 is achieved when running 200 concurrent invocations. We should note that the service does not target this kind of applications, and that their approach is resource-efficient for I/O tasks.

Q2 More than one invocation is assigned to each instance concurrently, producing the stairs-like shape in the plots. This happens for both sleeping and computing tasks, which unlinks its cause from the resource usage of a task. The consequence is an important interference that, although sleeping functions obviously do not notice, it heavily affects computing tasks. Invocations that should take 1.2 seconds span out to minutes with 200 concurrent requests (Figure 3.7f). We find a solution for this issue in limiting per-instance concurrency. Although we still do not reach the desired parallelism for the job, execution time is much better and consistent with this limit.

We also see that responses to the client are throttled when there is high concurrency in an instance, perceived on client times (black `X`s). On less busy instances, responses are almost immediate (Figure 3.7b). This hints to more interferences.

Q3 In the cases that include cold starts, host creations are at least a second apart, in line with the documentation [28]. However, we also see that the delay in host creation can be significant and function requests are assigned to new instances even before they can process them, resulting in important delays. For example, in Figure 3.7a most of the invocations are resolved in the first 6 s by 3 fast-spawning hosts, but some of them were assigned to a fourth instance that took almost 20 s to start, delaying invocations that could have run earlier on the other hosts.

Azure Functions is generally conservative with resources. For example, we do not see much scale until reaching 200 parallel requests, and it is restricted by the one “instance per second” limit. This prudent scheduling configuration is what mainly differentiates Azure from other providers. While others create new instances quite eagerly, Azure tends to pack as many invocations as possible to reduce resource consumption. The approach works really well for the I/O-bound tasks the service primarily targets, since it makes better use of resources, reduces costs, and facilitates management.

3.6 Experiment on Google Cloud Platform

We deploy and update our function with the GCP CLI. The invocation ID is obtained from one of the request headers in the function. It is also available for the client in the HTTP response. Differently from other providers, Google erases all information that could identify a container for the instance ID. To check if the container is the same, we use global code that generates an identifier during a cold start. This is reliable since the Python file is only loaded once per container.

3.6.1 Results

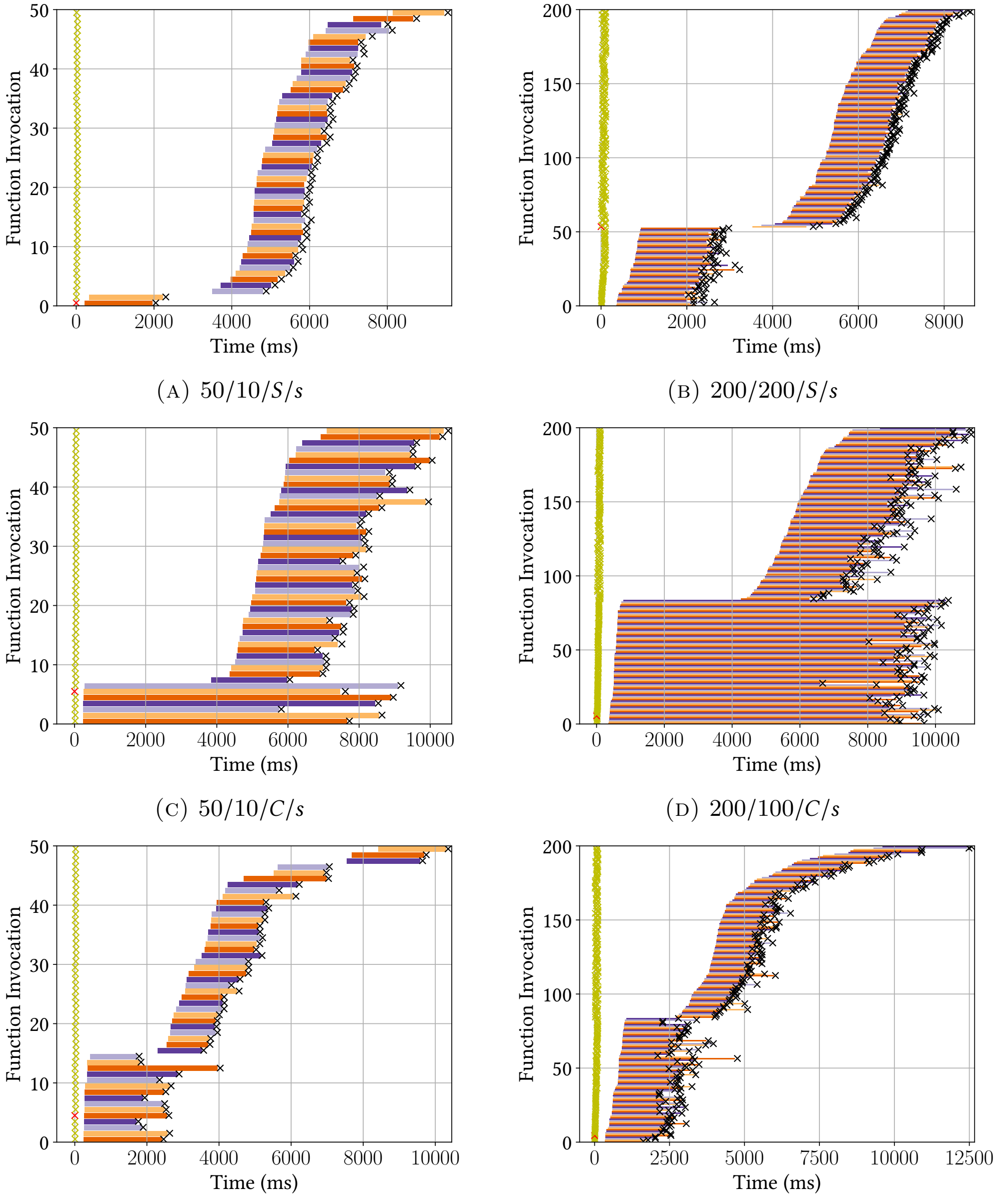
Experiments with sleeping functions With the *small* functions (256 MiB) and the sleeping task, an execution with $50/10/S/s$ results in Figure 3.10a. Each invocation runs on a different container. Note how the run only keeps 2 instances warm from a previous execution of 10. Figure 3.10b shows $200/200/S/s$. It is the second consecutive execution with this configuration, so we expect all instances warm; however, most hit a cold start. Still, the service runs each request on a different container.

Experiments with computing functions Still with *small* functions, we test the compute-intensive tasks. We start running the function individually and assess that the computation takes 5.5 s with this configuration. Figures 3.10c and 3.10d show invocations of this experiment with different parallelism, where we clearly see the performance difference between cold and warm containers. On cold invocations, the computation takes 3.5 s, while warm executions take up to 10 s. Also, warm containers are recycled very quickly. For instance, the 200-requests execution, run right after a 100 one, only finds 84 warm containers.

With 2 GiB functions (*big* configuration), the maximum memory configurable for GCP, function time for the individual execution reduces to 1.3 s. Figures 3.10e and 3.10f show subsequent invocations of this experiment with different parallelism. Like previously, the system keeps full parallelism. However, execution time still varies significantly from 1.3 to 4 s.

These are the best scenarios experienced. However, the system seems to throttle *big* functions, queueing some invocations and even rejecting them. Figure 3.11 shows samples of such cases, experienced after performing less than 1000 requests.

On a bigger scale Figure 3.12 depicts the results when running the larger configuration with 1000 asynchronous invocations. We see that, despite requesting 1000 invocations at once, only 550 functions run in parallel at first, and then



(A) 50/10/S/s

(B) 200/200/S/s

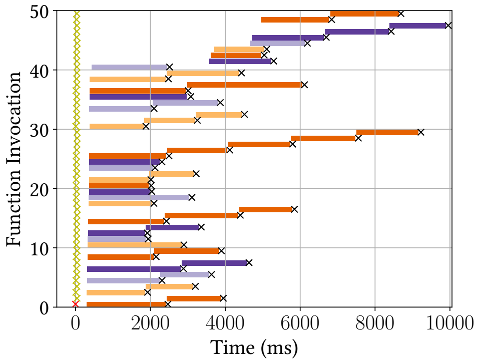
(C) 50/10/C/s

(D) 200/100/C/s

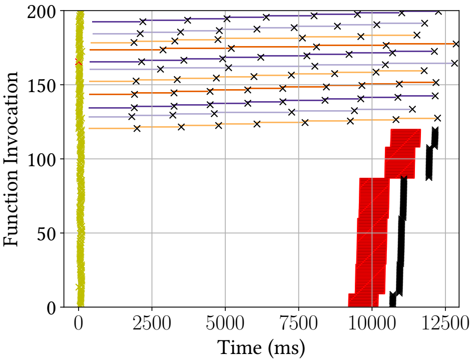
(E) 50/20/C/b

(F) 200/100/C/b

FIGURE 3.10: Experiment on GCP.



(A) 50/50/C/b



(B) 200/100/C/b

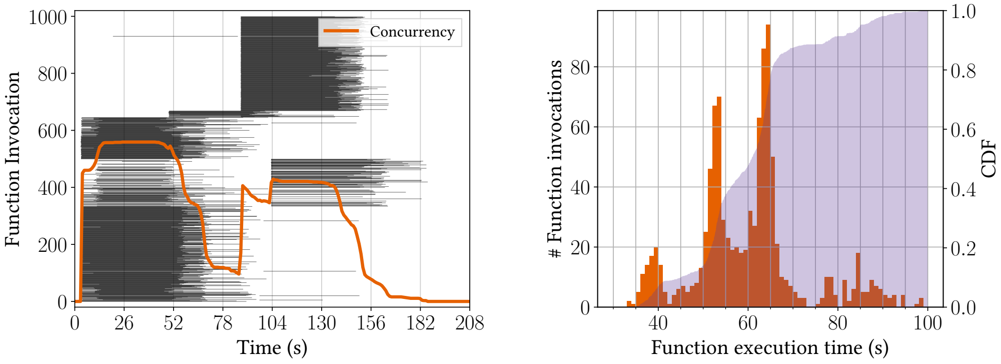
FIGURE 3.11: Server errors on GCP. The red bars are rejected requests.

another batch of 450 functions are run later. With the help of the histogram, we also notice a wide variety of function execution times. This behavior seems to confirm the differences between cold and warm invocations seen before, but also evinces further interferences in resources and/or heterogeneity of resources.

3.6.2 Discussion

Q1 Mostly, all invocations get a new instance, which allows good parallelism. However, the scheduling looks more complicated than in other platforms and imposes several rate limits. For instance, functions with more memory are less elastic. We experienced a lot of throttling with 2 GiB functions and even failed requests. Given the size of our experiments, this suggests a more restrictive rate limit than stated in the documentation [78]. While this does not affect functions at small scale, it is an issue for large-scale embarrassingly parallel tasks. Also, the service removes idle containers very quickly and subsequent runs of the experiment do not all find warm containers, and there are always cold starts. This can be an important issue for latency-sensitive applications, and also hinders parallelism. As an example, although invocations run on different instances, not all of them are running in parallel, simultaneously. E.g., from 200 requests less than 100 run in parallel and the big scale experiment only found a concurrency of 550.

Q2 With the information gathered from the environment, we see that all invocations run on a 2 GiB microVM. This is different from AWS, where each microVM is configured with its memory corresponding to the function configuration. The



(A)

(B)

FIGURE 3.12: Large-scale experiment on GCP.

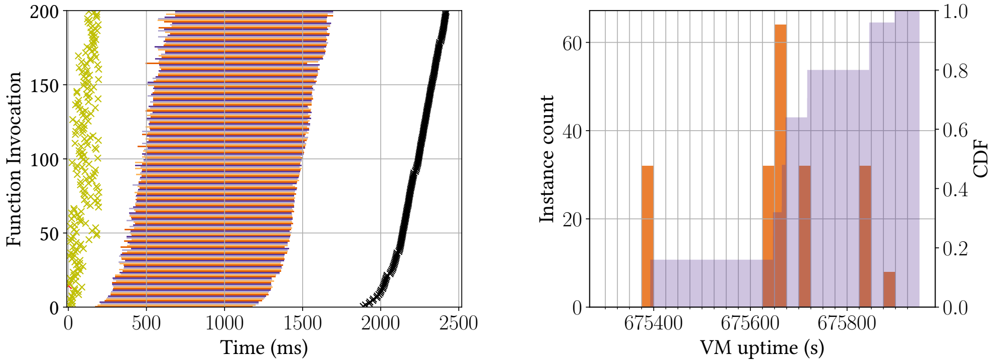
microVMs also have 2 vCPUs, which in most instances run at 2.7 GHz, and some at 2.3 GHz. Since all functions run on equally sized microVMs, the different CPU limits in the documentation [77] are probably imposed through CPU slices.

However, in experiments with the compute task, execution time is not consistent across invocations, suggesting that the limit is not well ensured. For instance, the 256 MiB functions complete in 2 and up to 10 s. Even with 2 GiB functions (corresponding to a full microVM), performance is inconsistent, ranging from 1.3 to 4 s. The most surprising finding is that there seems to be a significant performance difference between warm and cold invocations, being cold ones much faster.

Q3 We can add the following conclusions: (i) Scheduling is based on several parameters (e.g., function size, invocation rate, function run time, etc.), and it affects scalability. (ii) Cold starts usually induce a delay around 3 seconds, but it increases with parallelism and memory size.

3.7 Experiment on IBM Cloud

We deploy and update our function with the IBM Cloud CLI on the default package in a simple namespace, by directly uploading the source code. The invocation ID is at the environment variable "`__OW_ACTIVATION_ID`". The most reliable way to identify a container is through the randomly generated identifier present at `/proc/self/cgroup`; Docker writes the container name there [58]. We obtain the system uptime to identify the VM where each container runs. Even collected from



(A) Plot without the VM inference

(B) Uptime distribution

Figure 3.13: Guessing the VM on IBM for an execution with 200 requests.

a container, the uptime corresponds to the container host, which is the Invoker VM. Although not fully reliable, it can help us guess container co-residency.

3.7.1 Results

Guessing the VM from the system uptime The plot for a 200-requests execution would look like Figure 3.13a. Each invocation is running on a different container, but some of them could be on the same VM. By getting the system uptime, we can display container co-residency.[7] We represent the system uptime gotten at each function instance in Figure 3.13b. If the uptime gotten by different invocations is similar, they are likely co-residents of the same VM. Since invocations are not exactly simultaneous (they do not read the uptime at the same instant), never two of them will get the exact same uptime. However, since the whole experiment lasts 3 seconds, two co-resident invocations will get an uptime different by at most 3 seconds (usually in the same second since it is collected near function start). The CDF gives a very precise view. Each step in the curve is all the invocations that got a similar uptime, and thus co-residents. With the information from the histogram, we can count how many invocations run on each VM. If two VM uptimes are too close, the accompanying histogram may pack invocations from different machines in the same bar, but we can still distinguish them with the CDF. Since we know that all invocations run concurrently, this gives us the VM maximum concurrency. For instance, we see that most histogram bars

[7] A detailed description of a similar method for machine identification was previously introduced by Lloyd et al. [122].

count 32. One reaches 64, but we see in the CDF that it comprises two steps, thus being in fact two VMs. This means that it is very likely that each machine holds a maximum of 32 containers in this experiment.

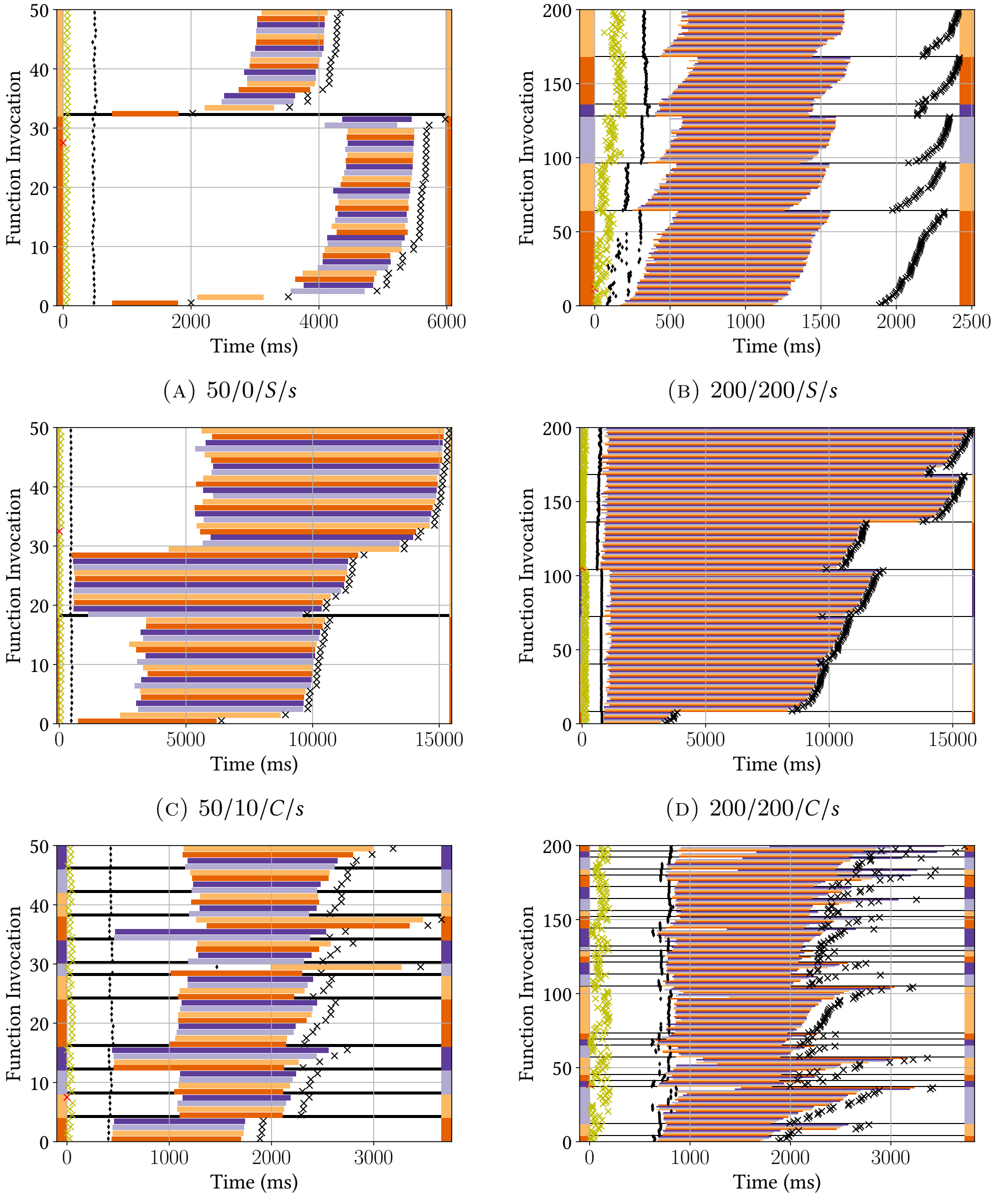
We merge this data into our plot to build Figure 3.14b. The color blocks at the sides indicate the guessed VM based on the system uptime. There is also a black line that separates VMs for clarity. Additionally, the service collects the time the invocation has been waiting in the system. We plot it as black diamonds to indicate when the system received the request.

Experiments with sleeping functions With *small* functions and the sleeping task, we first run a cold execution with 10 parallel requests. A subsequent execution with 50 requests results in Figure 3.14a. Figure 3.14b shows 200/200/S/s.

Figure 3.15a shows a cold start for 500 parallel invocations. In this case, two side blocks are twice as big as the others. However, in the CDF (Figure 3.15b) it is clear that each step is in fact of 32 containers. This case uses more VMs than the previous, and it is easier to find several machines with very similar uptime. This experiment also shows an interesting behavior of cold starts in OpenWhisk. Each VM has one invocation that runs almost as in a warm start, while the others take some extra seconds. This corresponds to the fact that OpenWhisk starts an empty container on each Invoker machine before receiving any request. It also validates that the bigger side blocks are in fact two VMs since they have two of these early invocations.

Experiments with computing functions We now switch to the compute-intensive task, still with *small* functions. An individual execution assesses that the computation takes 1.3 s. Figures 3.14c and 3.14d show subsequent invocations with different parallelism. We see clearly that execution time is affected and increases as the VMs fill up with containers. Our 1.3-second tasks take from 8 to 15 s on machines full with 32 containers.

With 2 GiB functions, which should have a full CPU as per our calculations, an individual execution still takes 1.3 s. Figures 3.14e and 3.14f show subsequent invocations with different parallelism. In this case, function run time is more consistent and maintains around 1.3 s. However, some executions span for up to an additional second, which hints us to other resource interferences. Here, the 200 execution requires more VMs than any previous experiment, leading to a similar situation than with the previous 500 execution (Figure 3.15a). In its uptime distribution (omitted), we identify 50 steps, which proves that each VM holds four 2 GiB containers.



(A) 50/0/S/s

(B) 200/200/S/s

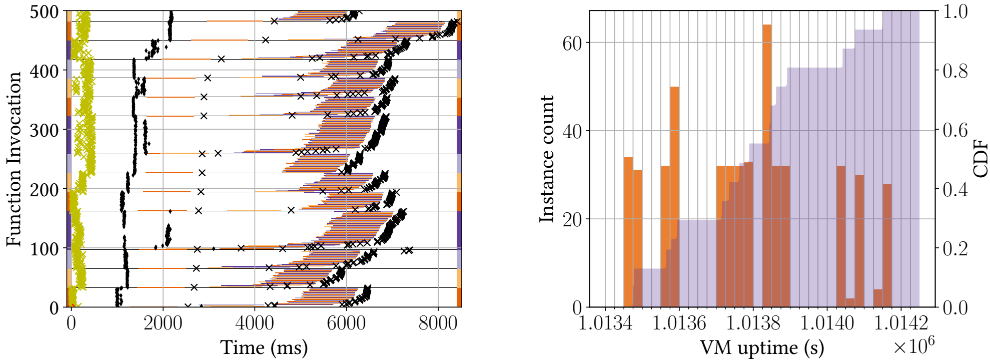
(C) 50/10/C/s

(D) 200/200/C/s

(E) 50/10/C/b

(F) 200/200/C/b

FIGURE 3.14: Experiment on IBM.



(A) Parallelism

(B) Uptime distribution

FIGURE 3.15: 500/0/S/s on IBM.

On a bigger scale The plot for the configuration with 1000 invocations appears in Figure 3.16, showing full parallelism from the start. However, several invocations take significantly longer to finish computation, doubling total completion time. The histogram shows this wide distribution of function run time. Resource heterogeneity and the interferences we perceived in previous experiments are possible causes of this variability.

3.7.2 Discussion

Q1 Generally, IBM Cloud Functions shows compelling parallelism with all new invocations starting a new container if there is none immediately available. This allows high-level parallelism as invocations come and enables full parallelism in all our experiments. This behavior presents a good fit for parallel tasks. Nonetheless, we have seen two unusual exceptions were an invocation got delayed in the system and reused a container.

Q2 We can infer function resource management and VM distribution from the experiments. Gathering system information, each machine presents 4 CPUs and 16 GiB of RAM. However, only 8 GiB are assigned to functions on each machine. We deduce this by seeing that a single VM only allocated 32 instances of 256 MiB, or 4 of 2048 MiB.

The compute tasks show that CPU is not strictly limited by the system, but the amount of memory given to each container will determine how much interference with others there will be, and thus how much CPU can be guaranteed to each one.

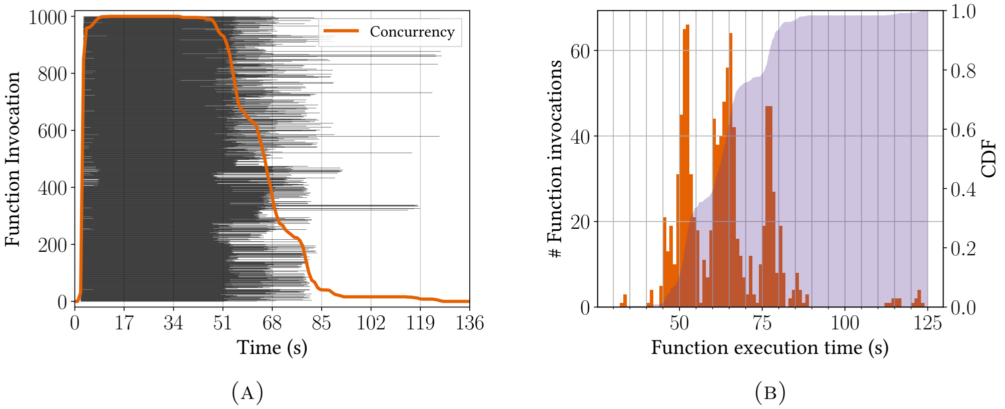


FIGURE 3.16: Large-scale experiment on IBM.

This resolves that each container with 2 GiB of memory will get a full CPU but could use up to four if the remaining of the VM is not used. 256 MiB containers will get at least 0.125 CPU in a congested machine but could also get all 4 CPUs in a free machine. It is a generous policy where the provider gives users more resources than requested.

We have seen this resource interference clearly in our experiments. With *small* functions, an individual invocation takes the same as in a 2 GiB function: 1.3 s. However, with parallel requests, the functions run in groups of 32 per VM and get 0.125 of CPU each, which means a time increase of 8×: 10.4 s. We see this behavior in the plots, although with considerable variance (10–15 s). In contrast, the invocations that run on less crowded machines run much faster (see Figure 3.14d)

Q3 The experiments also sketch that: (i) Scheduling is straightforward: if upon request arrival there are no containers idle, a new one is created. (ii) Cold starts can be as low as 1 or 2 s, but they grow with parallelism. (iii) We see that each VM provides a container pre-warmed. Although it can be helpful for certain applications, it is not that important for parallel workloads. (iv) The non-strict resource assignment is a good advantage, but the user should be conscious of it to avoid unexpected behavior.

TABLE 3.4: Summary of experiment results. Parallelism metrics from Section 3.3.6.

	AWS	**Azure**	**GCP**	**IBM**
Cold start (≈)	300 ms	2–20 s	2–6 s	1–4 s
Completion time	1.5 s	31 s	12.5 s	3.5 s
Parallel degree	200 100%	18 11%	< 100 < 50%	200 100%
Failures	None	None	Rejects	None

3.8 Experiment summary

Table 3.4 summarizes the metrics defined in Section 3.3.6 as perceived in Sections 3.4 to 3.7. We discuss them next for each platform:

AWS Lambda Currently, cold starts tend to stay around 300 ms [127]. Our experiments (see Section 3.4) match this tendency consistently, without substantial changes with increased concurrency. AWS Lambda completes the 200 requests in just 1.5 seconds (Figure 3.5f), which is the fastest with just half a second of overhead. This is possible because all invocations run on different instances and instantiation is quick, hence the parallel degree of 200 (100%). We did not experience any failure.

Azure Functions Instances generally start in 2–6 s [127]. However, we find much larger delays (Section 3.5), sometimes over 20 s. This could be explained by increased delay in finding resources or the scale controller delaying instantiation and not directly by the overhead of creating an instance. The 200 requests experiment is completed in about 31 s (Figure 3.8b). This is precisely because the service only used a maximum of 18 instances, which is only an 11% of the total invocations. However, none of the invocations failed or were rejected.

Google Cloud Functions Other benchmarks [127] place GCP's cold starts around 3 seconds. Our experiments (Section 3.6) show a similar trend: *small* instances starting in4 s and *big* ones in 2 s. However, they experience increasing delay with parallelism; up to 8 s (Figure 3.10e). The completion time for the 200 requests is at 12.5 s (Figure 3.10f). Although all invocations run in different instances, the service did not keep all of them warm, and only 84 are available from the start. The presence of cold starts delays some invocations and expands completion time. Incidentally, the number of instances used at the same time never reaches 100, leaving parallelism below 50%. Additionally, many requests are throttled or rejected with the *big* setup, as shown in Figure 3.11.

IBM Cloud Functions We typically see cold starts ranging from 1 to 2 s, and the experiments (Section 3.7) indicate it increases with scale, reaching up to 5 s with 500 requests. The 200-request experiment finishes in 3.5 s. All invocations run on different instances, achieving the maximum parallelism of 200 (100%), which leaves all overhead to instance creation delay. All invocations completed without failures.

3.9 Do FaaS platforms fit parallel computation?

With the analysis of the architectures in Section 3.2, the empirical study in Sections 3.3 to 3.7, and the metrics summary in Section 3.8, we can finally answer the main question proposed in this chapter: *Do FaaS platforms fit parallel computation?*. We do so through the discussion of our main conclusions next.

Not all FaaS platforms follow the same architecture, which has high impact on parallel performance. Two aspects directly influence their support for parallel computation:

Virtualization technologies. They establish how secure and isolated are function instances and how much it takes to start them. As discussed in Section 3.2, Table 3.1 shows a relation between the technology and the general architecture design, both impacting invocation latency. Table 3.4 reveals that platforms with lighter technologies generally provide better cold starts. AWS Lambda shows the best latency with its Firecracker microVMs.

Scheduling approach. It defines resource management and how invocations traverse the system. We identified two approaches in Section 3.2. The push-based approach is generous with resources since it can rush decisions and immediately spin up instances when none is available. AWS and IBM clearly show this on Figures 3.5 and 3.14. It improves parallelism, but to be efficient for the provider, resources need to be managed at fine granularity and instances spawn very quickly. The pull-based approach utilizes resources more efficiently, packing more invocations on the same instances. Usefully, it can enhance management for the provider, and reduce costs for the users. A downside is that its reactive elasticity is slower to adapt to current demand and is very dependent on its tunning. Azure is fairly restrictive in that way, as experienced in Section 3.5.

Azure Functions stands out from the other platforms when dealing with parallelism. Its behavior is very different due to its particular scheduling (how invocations are sent to instances) and resource management (how instances are created and removed). These characteristics, described in Section 3.2.3 and visualized in Section 3.5, explain the poor elasticity experienced by Kuhlenkamp et al. [113], and the limited request throughput assessed by Maissen et al. [127], among other works [116, 177]. The service is tuned for efficiency in cost and resource management. It packs invocations on a few instances to maximize resource utilization and reduce costs for the users and management for the provider. This configuration makes sense, since the service is built atop Azure WebJobs, focused on web applications, and it is great for short I/O-bound tasks where the high per-instance concurrency is a big ally. However, it does not work well for parallel, compute-intensive tasks (see, e.g., Figure 3.7f), since scaling is degraded in favor of instance concurrency. Even when limiting instance concurrency to enhance compute-bound applications, the service prefers queueing invocations to a few instances before starting new ones, incurring in significant delays (Figure 3.8b).

Performance for parallel computation changes considerably between platforms, since none was, at least initially, designed for this kind of applications. AWS and IBM's services are able to provide full parallelism for parallel workloads, as demonstrated by PyWren [98] and IBM-PyWren [149]. Our experiments show in detail how each invocation is dealt by a different instance and invocation latency is kept low, enabling all tasks to run in parallel. Google's platform also shows similar scaling behavior in our detailed tests. However, as discussed earlier (Section 3.8), we start to see failed invocations with relatively small parallelism (the aforementioned papers run thousands of parallel functions). Finally, we already discussed above how Azure Functions is not prepared for these tasks (Table 3.4), and it would struggle to support them.

In sum, **FaaS is not inherently good for parallel computation** and performance strongly depends on the platform design and configuration by the provider. Consequently, users must be aware of the parallel capabilities of the platform they choose in order to understand how their applications will behave.

Our conclusions help explain several benchmarking works in the literature [49, 113, 116, 127, 177]. Indeed, they already point to the good performance of AWS and IBM or the sometimes-strange behavior in GCP. And most importantly, the difference in performance for Azure was already sketched in the literature [112]. However, in this dissertation we analyzed the different platforms from the perspective of parallelism and took a deep look into the different architecture designs, which adds new information and helps to understand the causes of these behaviors.

3.10 Chapter summary

This chapter analyzes the architectures of four major FaaS platforms: AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. Our research focused on the capabilities and limitations the services offer for highly parallel computation. The design of the platforms revealed two important traits influencing their performance: *virtualization technology* and *scheduling approach*. We further explored them with detailed experiments to plot parallel executions and show task distribution in the platform. The experiments evinced that the different approaches to architecture heavily affect how parallelism is achieved on a FaaS platform. AWS, IBM Cloud, and GCP run different function instances for each function invocation, while Azure packs invocations in a few instances. In consequence, parallelism is thwarted on the latter (only 18% of invocations run in parallel) and parallel computation suffer big overhead (a 1 s computation takes 31 s). AWS and IBM always achieve good parallelism (100%). However, although GCP's approach is also prone to parallelism, our experiments show conflicting performance. The appearance of failed invocations produces stragglers in the computation and increases complexity for the user, who must manage the errors.

Chapter 4

Serverless Stateful Computation

DESPITE THE BENEFITS OF serverless computing, applications that require fine-grained support for mutable state and coordination are notoriously hard to build. In this chapter, we aim at bridging this gap. We present CRUCIAL, a system to program highly parallel stateful serverless applications. CRUCIAL retains the simplicity of serverless computing and offers a model similar to concurrent programming but at the scale of a datacenter. A distributed shared memory layer answers the needs for fine-grained state management and coordination.

The results of this chapter have been published in a conference paper and an article [34, 35].

4.1 Introduction

Current practices show that serverless computing works well for applications that require a small amount of storage and memory due to the operational limits set by the providers (see, e.g., AWS Lambda [14]). However, there are more limitations. While cloud functions can initiate outgoing network connections, they cannot directly communicate with each other, and have little bandwidth compared to a regular virtual machine [43, 177]. This is because the model was originally designed to execute event-driven functions in response to user actions or changes in the storage tier (e.g., uploading a file to Amazon S3 [21]). Despite these constraints, serverless computing applies to many areas. Some recent works show that this paradigm allows to process big data [98, 141, 149], encode videos [65], and perform linear algebra [157] and Monte Carlo simulations [93].

4.1.1 Scope and challenges

All these pioneering works prove that serverless computing can escape its initial area of usage and expand to traditional computing applications. However, programming some of these tasks still faces fundamental challenges. Although the list is too long to recount here, convincing cases of these ill-suited applications are distributed stateful computations such as machine learning (ML) algorithms. Just an imperative implementation of k-means [121] raises several issues: first, the need to efficiently handle a globally-shared state at fine granularity (the cluster centroids); second, the problem to globally coordinate the cloud functions, so that the algorithm can correctly proceed to the next iteration; and finally, the prerogative that the shared state survives system failures.

No serverless system currently addresses all these issues effectively. First, due to the impossibility of function-to-function communication, the prevalent practice for sharing state across functions is to use remote storage. For instance, serverless frameworks, such as PyWren and NumPyWren [157], use highly scalable object storage services to transfer state between cloud functions. Since object storage is too slow to share short-lived intermediate state in serverless applications [109], some recent works use faster storage solutions. This has been the path taken by Locus [141], which proposes to combine fast, in-memory storage instances with slow storage to scale shuffling operations in MapReduce. However, with all the shared state transiting through storage, one of the major limitations of current serverless systems is the lack of support to *handle mutable state at a fine granularity* (e.g., to efficiently aggregate small granules of updates). Such a concern has been recognized in various works [43, 99], but this type of fast, enriched storage layer for serverless computing is not available today in the cloud, leaving fine-grained state sharing as an open issue.

Similarly, FaaS orchestration services (such as AWS Step Functions [22] or OpenWhisk Composer [5]) offer limited capabilities to coordinate cloud functions [66, 99]. They have no abstraction to signal a function when a condition is fulfilled, or for multiple functions to synchronize, e.g., in order to guarantee data consistency, or to ensure joint progress to the next stage of computation. Of course, such fine-grained coordination should be also low-latency to not significantly slow down the application. Existing stand-alone notification services, such as AWS SNS [39] and AWS SQS [69], add significant latency, sometimes hundreds of milliseconds. This lack of efficient coordination mechanisms means that each serverless framework needs to develop its own solutions. For instance, PyWren enforces the coordination of the map and reduce phases through object storage, while ExCamera has built a notification system using a long-running VM-based rendezvous server. As of today, there is no general way to let multiple functions coordinate via abstractions hand-crafted by users, so that *fine-grained coordination* can be truly achieved.

4.1.2 Contributions

To overcome the aforementioned issues, we propose Crucial, a framework for the development of stateful distributed applications in serverless environments. The base abstraction of Crucial is the *cloud thread* which maps a thread to the invocation of a cloud function. Cloud threads manipulate global shared state stored in the *distributed shared objects* (DSO) layer. To ease data sharing between cloud functions, DSO provides out-of-the-box strong consistency guarantees. The layer also implements fine-grained coordination, such as collectives, to harmonize the functions. Objects stored in DSO can be either ephemeral or persistent, in which case they are passivated on durable storage. DSO is implemented with the help of state machine replication and executes atop an efficient disaggregated in-memory data store. Cloud threads can run atop any standard Function-as-a-Service platform.

The programming model of Crucial is quite simple, offering conventional multi-threaded abstractions to the programmer. With the help of a few annotations and constructs, single-machine multi-threaded stateful programs can execute as cloud functions. In particular, since the global state is manipulated as remote shared objects, the interface for mutable state management becomes virtually unlimited, only constrained by the expressiveness of the programming language (Java in our case).

Our evaluation shows that Crucial can scale traditional parallel jobs, such as Monte Carlo simulations, to hundreds of workers using basic code abstractions. For applications that require fine-grained updates, like ML tasks, our system can rival, and even outperform, Apache Spark running on a dedicated cluster. We also establish that an application ported to serverless with Crucial achieves similar performance to a multi-threaded solution running on a dedicated high-end server.

This chapter describes the following novel contributions:

- We provide the first concrete evidence that stateful applications with needs for fine-grained data sharing and coordination can be efficiently built using stateless cloud functions and a disaggregated shared objects layer.
- We design Crucial, a system for the development and execution of stateful serverless applications. Its simple interface offers fine-grained semantics for both mutable state and coordination. Crucial is open source and freely available online [170].
- We show that Crucial is a convenient tool to write serverless-native applications, or port legacy ones. In particular, we describe a methodology to port traditional single-machine applications to serverless.
- Crucial is suited for many applications such as traditional parallel computations, machine learning algorithms, and complex concurrency tasks. The results show comparable or superior performance to traditional computing and state of the art solutions with very little coding involvement. This is achieved at comparable cost but with the added value of no system management, provided by serverless technologies.

4.2 Background

This section introduces serverless computing and the challenges addressed in this chapter. We first contextualize this programming model with a description of AWS Lambda, although other platforms are equally well-suited for this purpose (e.g., Google Cloud Functions, Azure Functions or Apache OpenWhisk). Further, we focus on the dilemma of storing and sharing data across functions, then provide a high-level overview of the solution proposed in Crucial.

4.2.1 FaaS computing: value under restraint

AWS Lambda is a cloud service designed to run user-supplied functions, called *cloud functions*, in response to events (e.g., file uploads, message arrivals, etc.), or explicit API calls (via HTTP requests). A cloud function can be written in different target languages.[1] Before being usable, the code of the function and its dependencies are uploaded to the FaaS platform. Once deployed, the function is managed by AWS Lambda, that executes it on demand and at scale. Functions are stateless, that is, they do not keep a trace of execution from one invocation to another.

AWS Lambda, as other FaaS computing platforms, gives the advantages of rapid provisioning, high elasticity, and just-right cost: containers used to deploy a function can be launched within a few hundreds of milliseconds; they can quickly scale to match demand; and the service charges at sub-second granularity the duration of their execution. All these properties make possible to run various workloads in the cloud with minimal overhead [98, 141, 149, 157].

However, due to their lightweight nature, cloud functions are also subject to stringent resource restrictions. For instance, AWS Lambda [14] imposes a 15-minute limit per function invocation and caps memory usage to a few GiBs. Similar limits are applied by other FaaS providers. In addition, while a user can execute functions concurrently, direct communication is impossible between them. As a consequence, the linear scalability in function execution is in practice only achievable for embarrassingly parallel tasks [83, 99].

Function invocations can also fail for different reasons (e.g., the function raises an exception, times out or runs out of memory). When an error occurs, AWS Lambda may automatically retry the invocation [13]. However, this requires the programmer to carefully consider such a behavior when designing the application, e.g., by ensuring call idempotence.

4.2.2 The dilemma of shared data

Cloud functions are not addressable in the FaaS platform. This means that they may initiate a connection with a remote node (e.g., to fetch a web page), but they cannot listen for incoming requests. FaaS platforms currently do not support cross-functions communication. Another restriction is that cloud functions are *stateless*, that is they do not keep trace of a call from one invocation to another. These properties greatly simplify scheduling and scalability for the platform. However, they require the programmer to rely on external services for the application state [99, 180].

So far, the prevalent choice for storing data has been to rely on a disaggregated object storage such as Amazon S3. Typically, object stores have high access latency (>10 ms) and deliver either limited or costly I/O performance [99,

[1] AWS Lambda supports many languages directly (e.g., Java, Python), and any other by providing a custom runtime.

180]. Consequently, most serverless frameworks, like PyWren [98], only allow coarse-grained operations on shared data. To alleviate this problem, some recent works [135, 141, 157] use their own in-memory storage instances. Although these systems offer low latency, they do not provide durability, nor convenient abstractions to synchronize cloud functions.

Another recurring problem is the need to ship data to code [83]. Existing serverless frameworks access data using storage services that either offer a CRUD interface or provide a limited set of data types. As a consequence, data is repeatedly transported back and forth between the cloud functions and the storage layer. This negatively impacts performance (especially for large objects) and restrains concurrency on shared data.

4.2.3 An overview of Crucial

Crucial offers a simplified view of FaaS computing where cloud functions are seen as a set of cloud threads that communicate through shared state. To achieve this, the framework organizes mutable shared data in a layer of distributed shared objects (DSO). Cloud functions remotely call the methods of the objects to read or update them at fine granularity.

The DSO layer is implemented within a low-latency in-memory data store and deployed jointly with the serverless application. It delivers sub-millisecond latency —like other in-memory systems such as Redis (see Table 4.2)— and achieves even better throughput for complex, CPU-bound, concurrent operations (see Figure 4.2). Both properties, low latency and high throughput, make it an excellent substrate for mutable shared state and coordination. Crucial also permits data to persist after the computation, ensuring their durability through replication and passivation to stable storage.

Although the idea of distributed objects is not novel, to the best of our knowledge, it has never been applied to serverless computing. Such an approach simplifies the programming of stateful applications atop serverless architectures and further closes the gap between cloud and conventional computing. The next sections describe the programming model of Crucial and its internals.

4.3 Using Crucial

This section details the programming interface of Crucial and illustrates it with several applications. We also present a methodology to port a conventional single-machine application to serverless with the help of our framework.

Table 4.1: Programming abstractions

Abstraction	**Description**
`CloudThread`	Cloud functions are invoked like threads.
`ServerlessES`	A simple executor service for task groups and distributed parallel *for*s.
Shared objects	Linearizable (wait-free) distributed objects (e.g., `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicByteArray`, `List`, `Map`).
Coordination objects	Shared objects for thread coordination primitives (e.g., `Future`, `Semaphore`, `CyclicBarrier`).
`@Shared`	User-defined shared objects. Their methods run on the DSO servers, allowing fine-grained updates (e.g., `.add()`, `.update()`, `.merge()`).
Data persistence	Long-lived shared objects are replicated. Persistence may be activated with `@Shared(persistence=true)`.

4.3.1 Programming model

The programming model of Crucial is object-based and can be integrated with any object-oriented programming language. As Java is the language supported in our implementation, the following description considers its jargon.

Overall, a Crucial program is strongly similar to a regular multi-threaded, object-oriented Java one, besides some additional annotations and constructs. Table 4.1 summarizes the key abstractions available to the programmer that are detailed hereafter.

Cloud threads A `CloudThread` is the smallest unit of computation in Crucial. Semantically, this class is similar to a `Thread` in conventional concurrent computing. To write an application, each task is defined as a `Runnable` and passed to a `CloudThread` that executes it. The `CloudThread` class hides from the programmer the execution details of accessing the underlying FaaS platform. This enables access transparency to remote resources [53, 68].

Serverless executor service The `ServerlessES` class may be used to execute both `Runnable` and `Callable` instances in the cloud. This class implements the `ExecutorService` interface, allowing the submission of individual tasks and fork-join parallel constructs (`invokeAll`). The full expressiveness of the original JDK interface is retained. In addition, this executor also includes a distributed parallel *for* to run n iterations of a loop across m workers. To use this feature, the user specifies the in-loop code (through a functional interface), the boundaries for the iteration index, and the number of workers m.

State handling Crucial includes a library of base shared objects to support mutable shared data across cloud threads. The library consists of common objects such as integers, counters, maps, lists and arrays. These objects are *wait-free* and *linearizable* [124]. This means that each method invocation terminates after a finite number of steps (despite concurrent accesses), and that concurrent method invocations behave as if they were executed by a single thread. Crucial also gives programmers the ability to craft their own custom shared objects by decorating a field declaration with the `@Shared` annotation. Annotated objects become globally accessible by any thread. Crucial refers to an object with a key crafted from the field's name of the encompassing object. The programmer can override this definition by explicitly writing `@Shared(key=k)`. Our framework supports distributed references, permitting a reference to cross the boundaries of a cloud thread. This feature helps to preserve the simplicity of multi-threaded programming in Crucial.

Data Persistence Shared objects in Crucial can be either *ephemeral* or *persistent*. By default, shared objects are ephemeral and only exist during the application lifetime. Once the application finishes, they are discarded. Ephemeral objects can be lost, e.g., in the event of a server failure in the DSO layer, since the cost of making them fault-tolerant outweighs the benefits of their short-term availability [109]. Nonetheless, it is also possible to make them persistent with the annotation `@Shared(persistent=true)`. Persistent objects outlive the application lifetime and are only removed from storage by an explicit call.

Coordination Current serverless frameworks support only uncoordinated embarrassingly parallel operations, or bulk synchronous parallelism (BSP) [83, 99]. To provide fine-grained coordination of cloud threads, Crucial offers several primitives such as cyclic barriers and semaphores. These coordination primitives

Listing 4.1: Monte Carlo simulation to approximate π.

```
 1 public class PiEstimator implements Runnable {
 2   private final static long ITERATIONS = 100_000_000;
 3   private Random rand = new Random();
 4   @Shared(key="counter")
 5   AtomicLong counter = new AtomicLong(0);
 6
 7   public void run() {
 8     long count = 0;
 9     double x, y;
10     for (long i = 0L; i < ITERATIONS; i++) {
11       x = rand.nextDouble();
12       y = rand.nextDouble();
13       if (x * x + y * y <= 1.0) count++;
14     }
15     counter.addAndGet(count);
16   }
17 }
18
19 List<Thread> threads = new ArrayList<>(N_THREADS);
20 for (int i = 0; i < N_THREADS; i++) {
21   threads.add(new CloudThread(new PiEstimator()));
22 }
23 threads.forEach(Thread::start);
24 threads.forEach(Thread::join);
25 double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);
```

are semantically equivalent to those in the standard `java.util.concurrent` library. They allow a coherent and flexible model of concurrency for cloud functions that is non-existent as of today.

4.3.2 Sample applications

Listing 4.1 presents an application implemented with Crucial. This simple program is a multi-threaded Monte Carlo simulation that approximates the value of π. It draws a large number of random points and computes how many of them fall in the circle enclosed by the unit square. The ratio of points falling in the circle converges with the number of trials toward $\pi/4$ (line 25).

The application first defines a regular `Runnable` class that carries the estimation of π (lines 1 to 17). To parallelize its execution, lines 23 and 24 run a fork-join pattern using a set of `CloudThread` instances. The shared state of the application is a `counter` object (line 5). This counter maintains the total number

Listing 4.2: Using the `ServerlessES` to perform a Monte Carlo simulation.

```java
ExecutorService se = new ServerlessES();
List<Callable> tasks = IntStream.range(0, N_THREADS)
      .mapToObj(i -> Executors.callable(new PiEstimator()))
      .collect(Collectors.toList());
se.invokeAll(tasks);
```

Listing 4.3: Mandelbrot set computation in a distributed parallel *for*.

```java
1  public class Mandelbrot implements Serializable {
2    @Shared(key = "mandelbrotImage")
3    private MandelbrotImage image = new MandelbrotImage();
4
5    private static int[] computeRow(
6        int row, int width, int height, int maxIters
7    ) {...}
8
9    private void compute() {
10     image.init(COLS, ROWS);
11     ServerlessES se = new ServerlessES();
12     se.invokeIterativeTask(
13         row -> image.setRowColor(
14             row, computeRow(row, COLS, ROWS, MAX_INTERNAL_ITERS)
15         ),
16         N_TASKS, 0, ROWS
17     );
18     se.shutdown();
19   }
20 }
```

of points falling into the circle, which serves to approximate π. It is updated by the threads concurrently using the `addAndGet` method (line 15).

The previous fork-join pattern can also be executed conveniently with the `ServerlessES`. In this case, we simply replace lines 19 to 24 in Listing 4.1 with the content of Listing 4.2.

A second application is shown in Listing 4.3. This program outputs an image approximating the Mandelbrot set (a subset of $\mathbb{C}$) with a gradient of colors. The output image is stored in a custom Crucial shared object (line 3). To create the image, the application computes the color of each pixel (line 5). The color indicates when the pixel escaped from the Mandelbrot set (after a bounded number of iterations). The rows of the image are processed in parallel, using the invokeIterativeTask method of the ServerlessES class. As seen at line 12, this method takes as input a functional interface (IterativeTask) and three integers. The interface defines the function to apply on the index of the *for* loop. The integers define respectively the number of tasks among which to distribute the iterations, and the boundaries of these iterations (fromInclusive, toExclusive).

This second example illustrates the expressiveness and convenience of our framework. In particular, as in multi-threaded programming, Crucial allows to define concurrent tasks with lambda expressions and pass them shared variables defined in the encompassing class.

4.3.3 Portage to serverless

The previous sections detail the programming interface of Crucial and illustrate it with base applications. In this section, we turn our attention to the problem of porting existing applications to serverless. We first explain the benefits an application may have from a port to serverless and a methodology to achieve it. Further, we present the limitations of this methodology and how the programmer can overcome them. Section 4.6.4 evaluates the successful application of this methodology to port Smile [118], a state-of-the-art machine learning library.

Benefits & Target applications Crucial can be used not only to program serverless-native applications, but also to port existing single-machine applications to serverless. Successfully porting an application comes with several incentives: namely the ability to (i) access on-demand computing resources; (ii) scale these resources dynamically; and (iii) benefit from a fine-grained pricing for their usage. To match the programming model of Crucial, Java applications that can benefit from a portage should be multi-threaded. Moreover, as with other parallel programming frameworks (e.g., MPI [161] or MapReduce [55]), they should be inherently parallel.

Methodology Crucial allows to port an existing Java multi-threaded application to serverless with low effort. To this end, the following steps should be taken: **(1)** Replace the ExecutorService or Thread instances with their Crucial counterparts, as listed in Table 4.1. **(2)** Make Serializable each immutable object passed between cloud threads. **(3)** Substitute the concurrent mutable objects shared by threads with the equivalent ones provided by the DSO layer. For example, an instance of java.util.atomic.AtomicBoolean is replaced with org.crucial.dso.AtomicBoolean. **(4)** Regarding coordination primitives, transform them into distributed objects. As an example, a cyclic barrier can be replaced with `org.crucial.dso.CyclicBarrier`, an implementation based internally on a monitor. Alternatively, `org.crucial.dso.ScalableCyclicBarrier` implements the collective described in [85]. **(5)** If the `synchronized` keyword is used, some rewriting is necessary. Recall that this keyword is specific to the Java language and allows to use any (non-primitive) object as a monitor [86]. Crucial does not support the `synchronized` keyword out of the box since it would require modifying the JVM. Two solutions are offered: (a) create a monitor object in DSO and use it where appropriate; or (b) create a method for the object used as a monitor that contains all the code in the `synchronized{..}` block. Then, this object is annotated as `@Shared` in the application, and the method called where appropriate. The first solution is simple, but it might not be the most efficient since it requires to move data back and forth between the cloud threads that use the monitor. The second solution needs rewriting part of the original application. However, it is more in line with the object-oriented approach in Crucial, where an operation updating a shared object is accessible through a (linearizable) method, and it may perform better.

Limitations & Solutions The above methodology works for most applications, yet it has limitations. First, some threading features are not available in the framework —e.g., signaling a cloud thread. Second, Crucial does not natively support arrays (e.g., `T[] tab`). Indeed, recall that the Java language offers native methods to manipulate such data types. For instance, calling `tab[i]=x` assigns the value (or reference) x to `tab[i]`. Transforming a native call is not possible with just annotations.[2] The solution to these two problems is to rewrite the application appropriately, as in the case of `synchronized`.

Another issue is related to data locality. Typically, a multi-threaded application initializes shared data in the main thread and then makes it accessible to other threads for computation. Porting such a programming pattern to FaaS implies that data is initialized at the machine starting up the application, then serialized to be accessible elsewhere; this is very inefficient. Instead, a better approach is to pass a distributed reference that is lazily de-referenced by the thread. To illustrate this point, consider Listing 4.4 which counts the number of occurrences of the word "`serverless`" in a document. The application first constructs a reference to the document (line 2). Then, the document is split into chunks. For each chunk, the number of occurrences of the word is counted by a cloud thread (line 8). The results are then aggregated in the shared counter "`wordcount`". Reading the document in full at line 2 and serializing it to construct the chunks

[2] It is however possible with bytecode manipulation tools (e.g., [41]).

Listing 4.4: Parallel word count.

```java
1  public class WordCount {
2    private Document document = new Document(LOCATION);
3    private String word = "serverless";
4
5    private void compute() {
6      AtomicLong count = new AtomicLong("wordcount");
7      ServerlessES se = new ServerlessES();
8      se.invokeIterativeTask(
9        i -> count.addAndGet(countWords(word, document.split(i))),
10       N_TASKS, 0, N_TASKS
11     );
12   }
13 }
```

is inefficient. Instead, the application should send a distributed reference to the cloud threads at line 8. Then, upon calling `split`, the chunks are created on each thread by fetching the content from remote storage.

4.4 System design

Figure 4.1 presents the overall architecture of Crucial. In what follows, we detail the components and describe the lifecycle of an application in our system.

Crucial encompasses three main components (from left to right in Figure 4.1): (I) the client application; (II) the FaaS computing layer that runs the cloud threads; and (III) the DSO layer that stores the shared objects. A client application differs from a regular JVM process in two aspects: threads are executed as cloud functions, and they access shared data using the DSO layer. Moreover, Crucial applications may also rely on external cloud services, such as object storage to fetch input data (not modeled in Figure 4.1).

4.4.1 The distributed shared objects layer

Each object in the DSO layer is uniquely identified by a reference. Fine-grained updates to the shared state are implemented as methods of these objects. Given an object of type T, the reference to this object is (T, k), where k is either the name of the annotated object field or the value of the parameter *key* in the annotation `@Shared(key=k)`. When a cloud thread accesses an object, it uses its reference to invoke remotely the appropriate method.

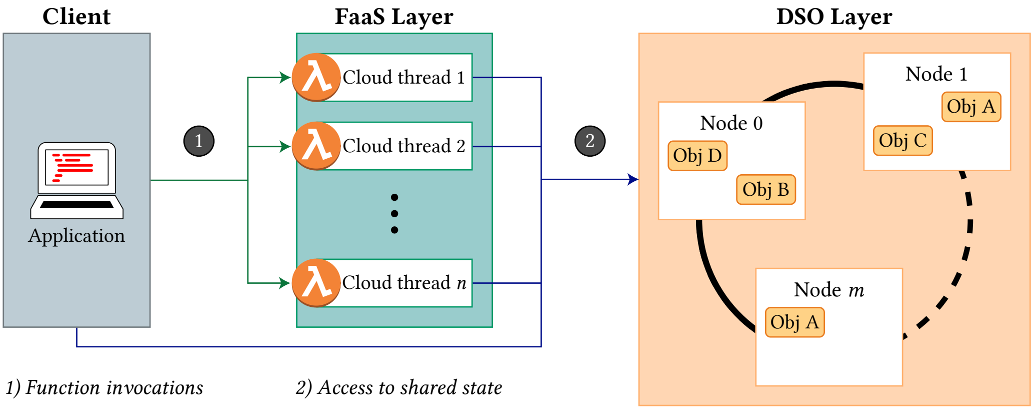


FIGURE 4.1: CRUCIAL's overall architecture.

CRUCIAL constructs the DSO layer using consistent hashing [103], similarly to Cassandra [114]. Each storage node knows the full storage layer membership and thus the mapping from data to node. The location of a shared object o is determined by hashing the reference (T, k) of o. This offers the following usual benefits: (i) no broadcast is necessary to locate an object; (ii) disjoint-access parallelism [94] can be exploited; and (iii) service interruption is minimal in the event of server addition and removal. The latter property is useful for persistent objects, as detailed next.

Persistence One interesting aspect of CRUCIAL is that it can ensure durability of the shared state. This property is appealing, for instance, to support the different phases of a machine learning workflow (training and inference). Objects marked as persistent are replicated rf (replication factor) times in the DSO layer. They reside in memory to ensure sub-millisecond read/write latency and can be passivated to stable storage using standard mechanisms (marshalling). When a cloud thread accesses a shared object, it contacts one of the server nodes. The operation is then forwarded to the actual replicas storing the object. Each replica executes the incoming call, and one of them sends the result back to the caller. Notice that for ephemeral (non-persistent) objects, rf is 1.

Consistency CRUCIAL provides linearizable objects and programmers can reason about interleaving as in the shared-memory case. This greatly simplifies the writing of stateful serverless applications. For persistent objects, consistency across replicas is maintained with the help of state machine replication (SMR) [153]. To handle membership changes, the DSO layer relies on a variation of virtual synchrony [48]. Virtual synchrony provides a totally-ordered set of views to the server nodes. In a given view, for some object x, the operations accessing x are sent using total order multicast. The replicas of x deliver these operations in a total order and apply them on their local copy of x according to this order. A distinct replica (primary) is in charge of sending back the result to the caller. When a membership change occurs, the nodes re-balance data according to the new view.

4.4.2 Fast aggregates through remote procedure call

As indicated in Section 4.2, stateful applications aggregate and combine small granules of data (e.g., the training phase of a ML algorithm). Unfortunately, cloud functions are not network-addressable and run separate from data. As a consequence, these applications are routinely left with no other choice but to "ship data to code". This is known as one of the biggest downsides of FaaS platforms [83], and we explore it further in Chapter 5.

To illustrate this point, consider an `AllReduce` operation where N cloud functions need to aggregate their results by applying some commutative and associative operator f (e.g., a sum). To achieve this, each function first writes its local result in the storage layer. Then, the functions await that their peers do the same, fetch the N results, and apply f sequentially. This algorithm is expensive and entails a communication cost of N^2 messages with the storage layer.

Crucial fully resolves this anti-pattern with minimal efforts from the programmer. Complex computations are implemented as object methods in DSO and called by the cloud functions where appropriate. Going back to the above example, each function simply calls $f(r)$ on the shared object, where r is its local result. This is for instance the case at line 9 in Listing 4.4 with the method `counter.addAndGet`. With this approach, communication complexity is reduced to $O(N)$ messages with the storage layer.

We exploit this key feature of Crucial in our serverless implementation of several ML algorithms (e.g., k-means, linear regression, random forest). Its performance benefits are detailed in Section 4.6.2.

4.4.3 Lifecycle of an application

The lifecycle of a Crucial application is similar to that of a standard multithreaded Java one. Every time a `CloudThread` is started, a Java thread (i.e., an instance of `java.lang.Thread`) is spawned on the client. This thread pushes the `Runnable` code attached to the `CloudThread` to a generic function in the FaaS platform. Then, it waits for the result of the computation before it returns.

Accesses to some shared object of type `T` at cloud threads (or at the client) are mediated by a proxy. This proxy is instantiated when a call to "`new T()`" occurs, and either the newly created object of type `T` belongs to Crucial's library, or it is tagged `@Shared`. As an example, consider the counter used in Listing 4.1. When an instance of `PiEstimator` is spawned, the field `counter` is created. The "`new`" statement is intercepted and a local proxy for the counter is instantiated to mediate calls to the remote object hosted in the DSO layer. If this object does not exist in the DSO layer, it is instantiated using the constructor defined at line 5. From thereon, any call to `addAndGet` (line 15) is pushed to the DSO layer. These calls are delivered in total order to the object replicas where they are applied before sending back a response value to the caller.

The Java thread remains blocked until the cloud function terminates. Such a behavior gives cloud threads the appearance of conventional threads minimizing code changes and allowing the use of the `join()` method at the client to establish synchronization points (e.g., fork/join pattern). It must be noted, however, that as cloud functions cannot be canceled or paused, the analogy is not complete. If any failure occurs in a remote cloud function, the error is propagated back to the client application for further processing.

The case of the `ServerlessES` builds on the same idea as `CloudThread`. A standard Java thread pool is used internally to manage the execution of all tasks. In the case of a callable task, the result is accessible to the caller in a `Future` object.

4.4.4 Fault tolerance

Fault tolerance in Crucial is based on the disaggregation of the compute and storage layers. On one hand, writes to DSO can be made durable with the help of data replication. In such a case, Crucial tolerates the joint failure of up to $rf-1$ servers.[3] On the other hand, Crucial offers the same fault-tolerance semantics in the compute layer as the underlying FaaS platform. In AWS Lambda, this means that any failed cloud thread can be re-started and re-executed with the exact same input. Thanks to the cloud thread abstraction, Crucial allows full control over the retry system. For instance, the user may configure how many retries are allowed and/or the time between them. If retries are permitted, the programmer should ensure that the re-execution is sound (e.g., it is idempotent). Fortunately, atomic writes in the DSO layer make this task easy to achieve. Considering the k-means example depicted in Listing 4.5 (or any other iterative algorithm), it simply

[3] Coordination objects (see Table 4.1) are not replicated. This is not an important issue due to their ephemeral nature.

consists in sharing an iteration counter (line 6). When a thread fails and re-starts, it fetches the iteration counter and continues its execution from thereon.

4.5 Implementation

The implementation of Crucial is open source and available online [170]. It consists of around 10K SLOC, including scripts to deploy and run Crucial applications in the cloud. The DSO layer is written atop the Infinispan in-memory data grid [129] as a partial rewrite of the Creson project [167].

A Crucial application is written in Java and uses Apache Maven to compile and manage its dependencies. It employs the abstractions listed in Table 4.1 and has access to scripts that automate its deployment and execution in the cloud.

To run cloud threads, our prototype implementation relies on AWS Lambda. Lambda functions are deployed with the help of a Maven plugin [115] and invoked via the AWS Java SDK. To control the replay mechanism, calls to Lambda are synchronous. The adherence of Crucial to Lambda is limited and the framework can execute atop a different FaaS platform with a few changes. In Chapter 2, we discuss this platform dependency.

The `ServerlessES` implements the base `ExecutorService` interface. It accepts `Callable` objects and task collections. The invocation of a `Callable` returns a (local) `Future` object. This future is completed once a response from AWS Lambda is received. For `Runnable` tasks, the response is empty unless an error occurs. In that case, the system interprets it and throws an exception at the client machine, referencing the cause.

To create a distributed parallel *for*, the `ServerlessES` provides a convenient method (as illustrated at line 12 in Listing 4.3). This method accepts an `IterativeTask` functional interface similar to `java.util.function.Consumer` but limited to iteration indexes (i.e., the input parameter must be an integer). Internally, the iterative task creates a collection of `Callable` objects. In the current prototype, the scheduling is static and based on the number of workers and tasks given in parameter.

When an AWS Lambda function is invoked, it receives a user-defined `Runnable` (or `Callable`) object. The object and its content are marshalled and shipped to the remote machine, where they are re-created. Initialization parameters can be given to the constructor. As pointed out in Section 4.3.1, a distributed reference is sent in lieu of a shared object.

Proxies for the shared objects are waved into the code of the client application using AspectJ [106]. In the case of user-defined objects, the aspects are applied to the annotated fields (see Section 4.3.1). Such objects must be serializable, and they should contain an empty constructor (similarly to a JavaBean). The `jar` archive containing the definition of the objects is uploaded to the DSO servers where it is dynamically loaded.

Coordination objects (e.g., barriers, semaphores, futures) follow the structure of their Java counterparts. Some of them rely internally on Java monitors. When a client performs a call to a remote object, it remains blocked until the request responds. The server processes the operation with a designated thread. During the method invocation, that thread may suspend itself through a `wait` call on the object until another thread awakes it.

State machine replication (SMR) is implemented using Infinispan's interceptor API. This API enables the execution of custom code during the processing of a data store operation. It follows the visitor pattern as commonly found in storage systems. Infinispan relies on JGroups [81] for total order multicast. The current implementation uses Skeen's algorithm [38].

In our prototype, the deployment of the storage layer is explicitly managed (like, e.g., AWS ElastiCache). Automatic provisioning of storage resources for serverless computing remains an open issue [43, 99], with just a couple works appearing very recently in this area [109, 141].

4.6 Evaluation

Goal and scope The core objective of this evaluation is to understand the benefits of Crucial to program applications for serverless. To this end, we distinguish two types of applications: serverless-native and ported applications. Serverless-native applications are those written from scratch for a FaaS infrastructure. Ported applications are the ones that were initially single-machine applications and were later modified to execute atop FaaS. For both types of applications, our evaluation campaign aims at providing answers to the following questions:

- *How easy is it to program with Crucial?* In addressing this question, we specifically focus on the following applications: machine learning, data analytics and coordination tasks. These applications are parallel and stateful, that is they contain parallel components that need to update a shared state and coordinate to make progress.

- *Do applications programmed with Crucial benefit from the capabilities of serverless (e.g., scalability and on-demand pricing)?*

- *How efficient is an application programmed with Crucial?* For serverless-native applications, we compare Crucial to PyWren, a state-of-the-art solution for serverless programming. We also make a comparison with Apache Spark, the de facto standard approach to program stateful cluster-based programs. For ported applications, we compare Crucial to a scale-up approach, using a high-end server.

- *How costly is Crucial with respect to other solutions?* Here we are interested both in the programming effort to code a serverless application and its monetary cost when running atop a FaaS platform. Again, answers are provided for both serverless-native and ported applications.

Experimental setup All the experiments are conducted in Amazon Web Services (AWS), within a Virtual Private Cloud (VPC) located in the `us-east-1` region. Unless otherwise specified, we use `r5.2xlarge` EC2 instances for the DSO layer and 3 GiB AWS Lambda functions. Experiments with concurrency over 300 cloud threads are run outside the VPC due to service limitations.

The code of the experiments presented in this section is available online [170].

Outline We first evaluate the runtime of Crucial with a series of micro-benchmarks (Section 4.6.1). Then, we focus on fine-grained updates to shared mutable data (Section 4.6.2) and fine-grained coordination (Section 4.6.3). Further, we explore porting a existing library to serverless (Section 4.6.4). Finally, we analyze the usability of our framework when writing (or porting) applications (Section 4.6.5).

4.6.1 Micro-benchmarks

As depicted in Figure 4.1, the runtime of Crucial consists of two components: a Function-as-a-Service (FaaS) platform and the DSO layer. In this section, we evaluate the performance of this runtime across several micro-benchmarks. We first measure the latency and throughput of DSO, then we turn our attention to evaluate the parallelism offered by the underlying FaaS platform (AWS Lambda).

Latency Table 4.2 compares the latency to access a 1 KiB object sequentially in Crucial (DSO), Redis, Infinispan, and S3. We chose Redis because it is a popular key-value store available on almost all cloud platforms, and it has been extensively used as storage substrate in prior serverless systems [98, 109, 141]. Each function performs 30K operations and we report the average access latency.

TABLE 4.2: Average latency comparison – 1 KiB payload

	PUT	**GET**
S3	34,868 μs	23,072 μs
Redis	232 μs	229 μs
Infinispan	228 μs	207 μs
CRUCIAL	231 μs	229 μs
CRUCIAL ($rf = 2$)	512 μs	505 μs

In Table 4.2, CRUCIAL exhibits a performance similar to other in-memory systems. In particular, it is an order of magnitude faster than S3. This table also depicts the effect of object replication. When data is replicated, SMR adds an extra round-trip, doubling the latency perceived at a client. The number of replicas does not affect this behavior, as shown in the next experiment.

Throughput We measure the throughput of CRUCIAL and compare it against Redis. For an accurate picture, replication is enabled in both systems to capture their performance under scenarios of high data availability and durability.

In this experiment, 200 cloud threads access 800 shared objects during 30 s. The objects are chosen at random. Each object stores an integer offering basic arithmetic operations. We consider simple and complex operations. The simple operation is a multiplication. The complex one is the sequential execution of 10K multiplications. In Redis, these operations require several commands which run as Lua scripts for both consistency and performance.

To replicate data, Redis uses a master-based mechanism. By default, replication is *asynchronous*, so the master does not wait for a command to be processed by the replicas. Consequently, clients can observe stale data. In our experiment, to minimize inconsistencies and offer guarantees closer to CRUCIAL, functions issue a `WAIT` command after each write [144]. This command flushes the pending updates to the replicas before it returns.

We compare the average throughput of the two systems when the replication factor (rf) of a datum varies as follows: ($rf = 1$) Both CRUCIAL and Redis (2 shards with no replicas) are deployed over a 2-node cluster; ($rf = 2$) In the same 2-node cluster, Redis now uses one master and one replica; ($rf = 3$) We add a third node to the cluster and Redis employs one master and two replicas. In Figure 4.2, "Redis `WAIT` r" indicates that r is the number of synchronously replicated copies of shared objects.

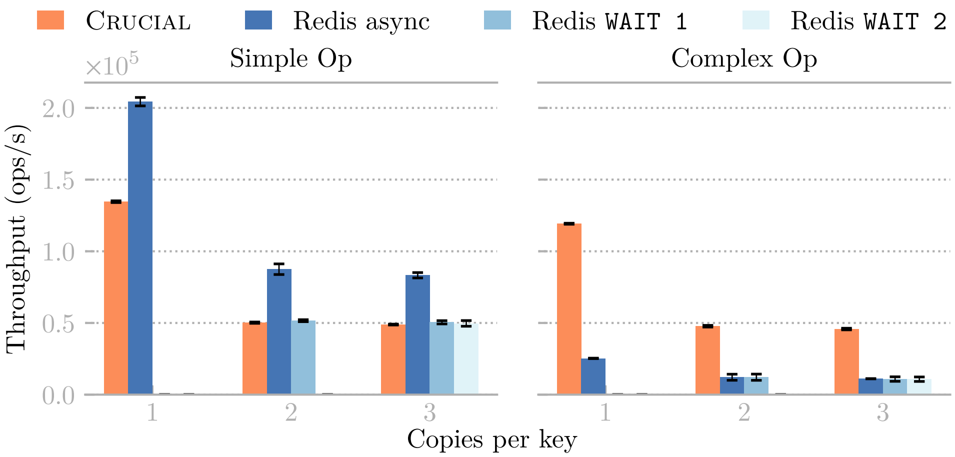


FIGURE 4.2: Operations per second performed on CRUCIAL and Redis (with and without replication). Cloud threads access uniformly at random 800 different keys/objects.

The experimental results reported in Figure 4.2 show that CRUCIAL is not sensitive to the complexity of operations. Redis is 50% faster for simple operations because its implementation is optimized and written in C. However, for complex operations, CRUCIAL is almost five times better than Redis. Again, implementation-specific details are responsible for this behavior: while Redis is single-threaded, and thus concurrent calls to the Lua scripts run sequentially, CRUCIAL benefits from disjoint-access parallelism [94]. When objects are replicated, the comparison is similar. In particular, Figure 4.2 shows that CRUCIAL and Redis have close performance when Redis operates in synchronous mode.

This experiment also verifies that the performance of CRUCIAL is not sensitive to the number of replicas. Indeed, the throughput in Figure 4.2 is roughly equivalent for all values of $rf \geq 2$. This comes from the fact that CRUCIAL requires a single RTT to propagate an operation to the replicas.

Parallelism We first evaluate our framework with the Monte Carlo simulation presented in Listing 4.1. This algorithm is embarrassingly parallel, relying on a single shared object (a counter). The simulation runs with 1 to 800 cloud threads, and we track the total number of points computed per second. The results, presented in Figure 4.3a, show that our system scales linearly and that it exhibits a 512× speedup with 800 threads.

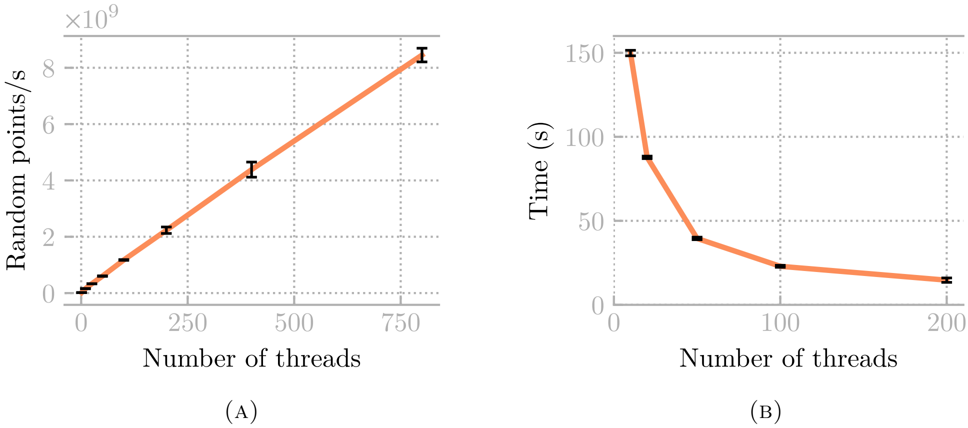


FIGURE 4.3: (A) Scalability of a Monte Carlo simulation to approximate π. CRUCIAL reaches 8.4 billion random points per second with 800 threads. (B) Scalability of a Mandelbrot computation with CRUCIAL.

We further evaluate the parallelism of CRUCIAL with the code in Listing 4.3. This second experiment computes a 30K×30K projection of the Mandelbrot set, with (at most) 1000 iterations per pixel. As shown in Figure 4.3b, the completion time decreases from 150 s with 10 threads to 14.5 s with 200 threads: a speedup factor of 10.2× over the 10-thread execution. This super-linear speedup is due to the skew in the coarse-grained row partitioning of the image. It also underlines a key benefit of CRUCIAL. If this task is run on a cluster, the cluster is billed for the entire job duration, even if some of its resources are idle. Running atop serverless resources, this implementation ensures instead that row-dependent tasks are billed for their exact duration.

Takeaways The distributed shared objects (DSO) layer of CRUCIAL is on par with existing in-memory data stores in terms of latency and throughput. For complex operations, it significantly outperforms Redis due to data access parallelism. CRUCIAL scales linearly to hundreds of cloud threads. Applications written with the framework benefit from the serverless provisioning and billing model to match irregularities in parallel tasks.

4.6.2 Fine-grained state management

This section shows that CRUCIAL is efficient for parallel applications that access shared state at fine granularity. We detail the implementation of two machine

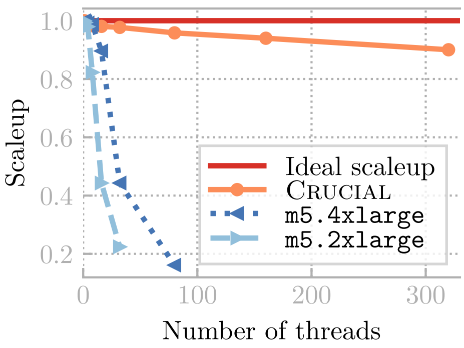


Figure 4.4: Scalability of the k-means clustering algorithm with Crucial versus single-machine multi-threading.

learning algorithms in the framework. These algorithms are evaluated against a single-machine solution, as well as two state-of-the-art frameworks for cluster computing (Apache Spark) and FaaS-based computation (PyWren).

A serverless k-means

Listing 4.5 details a k-means clustering algorithm written with Crucial. This program computes k clusters from a set of points across a fixed number of iterations, or until some convergence criterion is met (line 21). The algorithm is iterative, with recurring synchronization points (line 19), and it uses a small mutable shared state. Listing 4.5 relies on shared objects for the convergence criterion (line 4), the centroids (line 8), and a synchronization object to coordinate the iterations (line 2). At each iteration, the algorithm needs to update both the centroids and the criterion. The corresponding method calls (lines 14, 17 and 18) are executed remotely in DSO.

Section 4.6.2 compares the scalability of Crucial against two EC2 instances: `m5.2xlarge` and `m5.4xlarge`, with 8 and 16 vCPUs respectively. In this experiment, the input increases proportionally to the number of threads. We measure the *scale-up* computed with respect to that fact: *scale-up* $= T_1/T_n$, where T_1 is the execution time of Listing 4.5 with one thread, and T_n when using n threads.[4] Accordingly, *scale-up* $= 1$ means a perfect linear scale-up, i.e., the increase in the number of threads keeps up with the increase in the workload size (top line in

[4] In Section 4.6.2, threads are AWS Lambda functions for Crucial, and standard Java threads for the EC2 instances.

Listing 4.5: *k*-means implementation with Crucial.

```java
public class KMeans implements Runnable {
  private CyclicBarrier barrier = new CyclicBarrier();
  @Shared(key = "delta")
  private GlobalDelta globalDelta = new GlobalDelta();
  @Shared(key = "iterations")
  private AtomicInteger globalIterCount = new AtomicInteger();
  // Wraps a list of @Shared centroids
  private GlobalCentroids centroids = new GlobalCentroids();

  public void run() {
    loadDatasetFragment();
    int iterCount = globalIterCount.intValue();
    do {
      correctCentroids = globalCentroids.getCorrectCoord();
      resetLocalStructures();
      localDelta = computeClusters();
      globalDelta.update(localDelta);
      centroids.update(localCentroids, localSizes);
      barrier.await();
      globalIterCount.compareAndSet(iterCount, iterCount++);
    } while (iterCount < maxIterations && !endCondition());
  }
}
```

Section 4.6.2). The scale-up is sub-linear when *scale-up* < 1. As expected, the single-machine solution quickly degrades when the number of threads exceeds the number of cores. The solution using Crucial is within 10% of the optimum. For instance, with 160 threads, the scale-up factor is approximately 0.94. This lowers to 0.9 for 320 threads due to the overhead of creating the cloud threads.

Comparison with Spark

Apache Spark [182] is a state-of-the-art solution for distributed computation in a cluster. As such, it is extensively used to scale many kinds of applications in the cloud. One of them is machine learning (ML) training, as enabled by Spark's MLlib [131] library. Most ML algorithms are iterative and share a modest amount of state that requires per-iteration updates. Consequently, they are a perfect fit to assess the efficiency of fine-grained updates in Crucial against a state-of-the-art solution. This is the case of logistic regression and *k*-means clustering, which we use in this section to compare Crucial and Spark.

Setup. For this comparison, we provide equivalent CPU resources to all competitors. In detail, Crucial experiments are run with 80 concurrent AWS Lambda functions and one storage node. Each AWS Lambda function has 1792 MiB and 2048 MiB of memory for logistic regression and k-means, respectively. These values are chosen to have the optimal performance at the lowest cost (see Section 4.6.5).[5] The DSO layer runs on a `r5.2xlarge` EC2 instance. Spark experiments are run in Amazon EMR with 1 master node and 10 `m5.2xlarge` worker nodes (*Core nodes* in EMR terminology), each having 8 vCPUs. Spark executors are configured to utilize the maximum resources possible on each node of the cluster. To improve the fairness of our comparison, the time spent in loading the dataset from S3 and parsing it is not considered for both solutions. For Spark, the time to provision the cluster is not counted. Regarding Crucial, FaaS cold starts are also excluded from measurements due to a global barrier before starting the computation.

Dataset. The input is a 100 GiB dataset generated with spark-perf [54] that contains 55.5M elements. For logistic regression, each element is labeled and contains 100 numeric features. For k-means, each element corresponds to a 100-dimensional point. The dataset has been split into 80 equal-size partitions to ensure that all partitions are small enough to fit into the function memory. Each partition has been stored as an independent file in Amazon S3.

Logistic regression. We evaluate a Crucial implementation of logistic regression against its counterpart available in Spark's MLlib [131]: particularly the class `LogisticRegressionWithSGD`. A key difference between the two implementations is the management of the shared state. Each iteration, Spark broadcasts the current weight coefficients, computes, and finally aggregates the sub-gradients in a MapReduce phase. In Crucial, the weight coefficients are shared objects. Each iteration, a cloud thread retrieves the current weights, computes the sub-gradients, updates the shared objects, and synchronizes with the other threads. Once all the partial results are uploaded to the DSO layer, the weights are recomputed, and the threads proceed to the next iteration.

In Figures 4.5a and 4.5b, we measure the running time of 100 iterations of the algorithm and the logistic loss after each iteration. Results show that the iterative phase is 18% faster in Crucial (62.3 s) than with Spark (75.9 s), and thus the algorithm converges faster. This gain is explained by the fact that Crucial aggregates and combines the sub-gradients in the storage layer. On the contrary,

[5] Starting with a configuration of 1792 MiB, an AWS Lambda function has the equivalent to 1 full vCPU (`https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html`). Also, with this assigned memory, the function uses a full Elastic Network Interface (ENI) in the VPC.

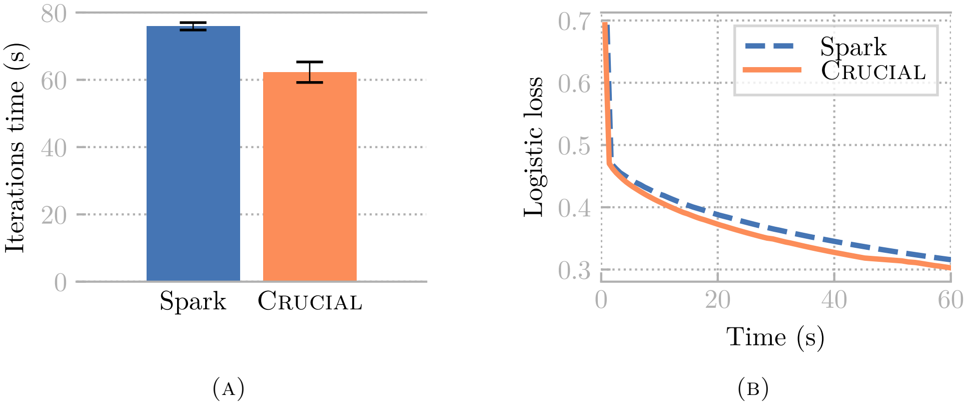


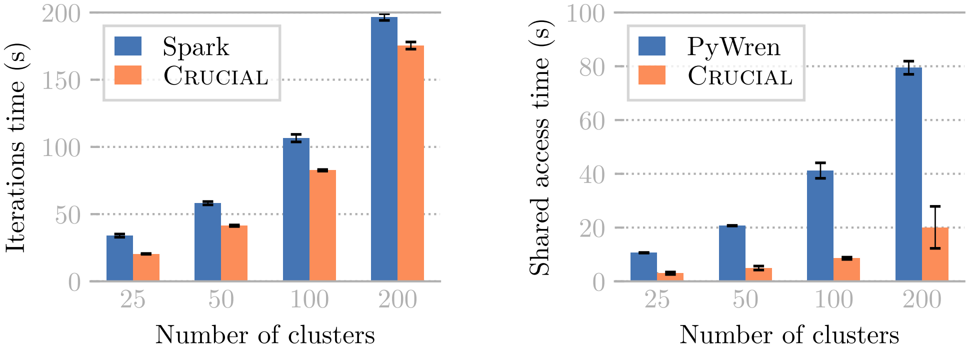
Figure 4.5: Comparison of Crucial and the state-of-the-art. (a) Average logistic regression iterative phase completion time (100 iterations). (b) Logistic regression performance.

each iteration in Spark requires a reduce phase that is costly both in terms of communication and synchronization.

k-means. We compare the k-means implementation described above to the one in MLlib. For both systems, the centroids are initially at random positions and the input data is evenly distributed among tasks. Figure 4.6a shows the completion time of 10 iterations of the clustering algorithm. In this figure, we consider different values of k to assess the effectiveness of our solution when the size of the shared state varies. With $k = 25$, Crucial completes the 10 iterations 40% faster (20.4 s) than Spark (34 s). The time gap is less noticeable with more clusters because the time spent synchronizing functions is less representative. In other words, the iteration time becomes increasingly dominated by computation. As in the logistic regression experiment, Crucial benefits from computing centroids in the DSO layer, while Spark requires an expensive reduce phase at each iteration.

Comparison with PyWren

We close this section by comparing Crucial to a serverless-native state-of-the-art solution. To date, the most evaluated framework to program stateful serverless applications is PyWren [98]. Its primitives, such as `call_async` and `map` are comparable to Crucial's cloud thread and serverless executor abstractions. Our evaluation employs Lithops [119], a recent and improved version of PyWren. PyWren is a MapReduce framework. Thus, it does not natively provide advanced



(A)

(B)

FIGURE 4.6: Comparison of CRUCIAL and the state-of-the-art. (A) Average k-means iterative phase completion time (10 iterations) with varying number of clusters. (B) Average k-means shared state access time.

features for state sharing and synchronization. Therefore, following the recommendations by Jonas et al. [98], we use Redis for this task.

Setup. We employ the same application, dataset, and configuration as in the previous experiment. The two frameworks use AWS Lambda for execution. A single `r5.2xlarge` EC2 instance runs DSO for CRUCIAL, or Redis for PyWren.

k-means. Implementing k-means above PyWren requires to store the shared state in Redis, that is the centroids and the convergence criterion. Following Jonas et al. [98], we use a Lua script to achieve this. At the end of each iteration, every function updates (atomically) the shared state by calling the script. This approach is the best solution in terms of performance. In particular, it is more efficient than using distributed locking due to the large number of commands needed for the updates. To synchronize across iterations, we use the Redis barrier covered in Section 4.6.3.

The CRUCIAL and PyWren k-means applications are written in different languages (Java and Python, respectively). Consequently, the time spent in computation for the two applications is dissimilar. For that reason, and contrary to the comparison against Spark, Figure 4.6b does not report the completion time. Instead, this figure depicts the average time spent in accessing the shared state during the k-means execution for both CRUCIAL and PyWren. This corresponds to the time spent inside the loop in Listing 4.5 (excluding line 16).

In Figure 4.6b, we observe that the solution combining PyWren and Redis is always slower than Crucial. This comes from the fact that Crucial allows efficient fine-grained updates to the shared state. Such results are in line with the throughput evaluation presented in Section 4.6.1.

Takeaways The distributed shared objects (DSO) layer of Crucial offers abstractions to program stateful serverless applications. DSO is not only convenient but, as our evaluation confirms, efficient. For two common machine learning tasks, Crucial is up to 40% faster than Spark, a state-of-the-art cluster-based approach, at comparable resource usage. It is also faster than a solution using jointly PyWren, a well-known serverless framework, and the Redis data store.

4.6.3 Fine-grained coordination

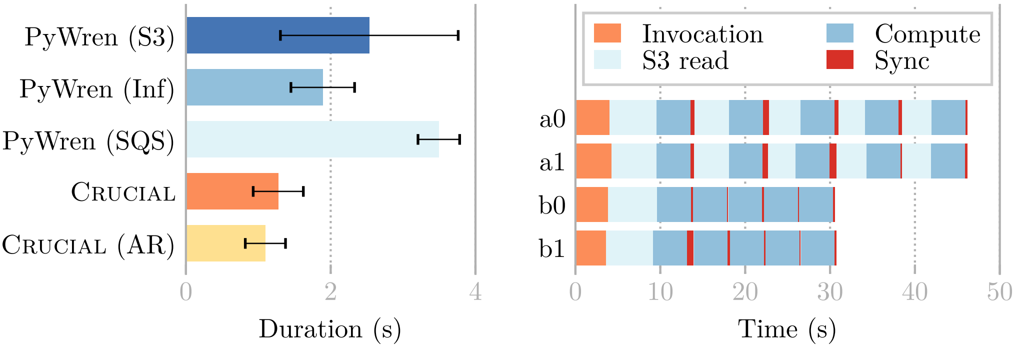
This section analyzes the capabilities of Crucial to coordinate cloud functions. We evaluate the coordination primitives available in the framework and compare them to state-of-the-art solutions. We then demonstrate the use of Crucial to solve complex coordination tasks by considering a traditional concurrent programming problem.

Synchronizing a map phase

Many algorithms require coordination at various stages. In MapReduce [55], this happens between the map and reduce phases, and it is known as shuffle. Shuffling ensures that the reduce phase starts when all the appropriate data was output in the preceding map phase. Shuffling the map output is a costly operation in MapReduce, even if the reduce phase is short. For that reason, when data is small and the reduction operation simple, it is better to skip the reduce phase and instead aggregate the map output directly in the storage layer [56]. Crucial allows to easily implement this approach.

In what follows, we compare different techniques to synchronize cloud functions at the end of a map. Namely, we compare (i) the original solution in PyWren, based on polling S3; (ii) the same mechanism but using the Infinispan in-memory key-value data store; (iii) the use of Amazon SQS, as proposed in some recent works (e.g., Flint [107]); and (iv) two techniques based on the `Future` object available in Crucial. The first solution outputs a future object per function, then runs the reduce phase. The second aggregates all the results directly in the DSO layer (AR).

We compare the above five techniques by running back-to-back the Monte Carlo simulation in Listing 4.1. The experiment employs 100 cloud functions, each



(a)

(b)

Figure 4.7: (a) Synchronizing a map phase in MapReduce with PyWren, Amazon SQS and Crucial. (b) Performance breakdown of an iterative task using either multiple stages (a0/a1), or a single stage with a Crucial barrier (b0/b1).

doing 100M iterations. During a run, we measure the time spent in synchronizing the functions. On average, this accounts for 23% of the total time.

Figure 4.7a presents the results of our comparison. Using Amazon S3 proves to be slow, and it exhibits high variability —some experiments being far slower than others. This is explained by the combination of high access latency, eventual consistency, and the polling-based mechanism. The results improve with Infinispan, but being still based on polling, the approach induces a noticeable overhead. Using Amazon SQS is the slowest approach of all. It needs a polling mechanism that actively reads messages from the remote queue. The solution based on `Future` objects allows to immediately respond when the results are available. This reduces the number of connections necessary to fetch the result and thus translates into faster synchronization. When the map output is directly aggregated in DSO, Crucial achieves even better performance, being twice as fast as the polling approach atop S3.

Coordination primitives

Cloud functions need to coordinate when executing parallel tasks. This section evaluates some of the coordination primitives available in Crucial to this end.

For starters, we study the performance of a barrier when executing an iterative task. In Figure 4.7b, we depict a breakdown of the time spent in the phases of

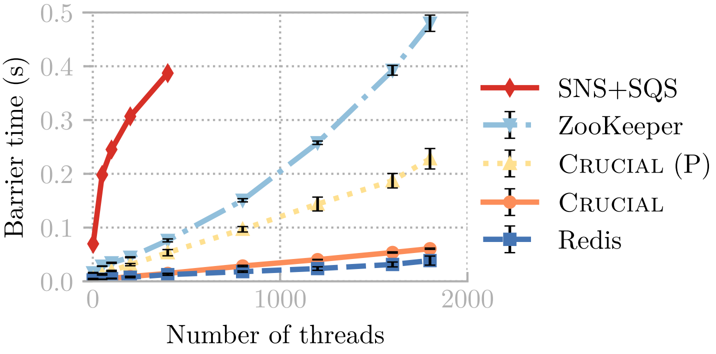


Figure 4.8: Average time threads spend waiting on a barrier.

each iteration (Invocation, S3 read, Compute, and Sync). The results are reported for 2 cloud functions out of 10 —the other functions behave similarly.

The breakdown in Figure 4.7b considers two approaches. The first launches a new stage of functions (a0 and a1) at each iteration that do not use the barrier primitive. The second launches a single stage of functions (b0 and b1) that run all the iterations and use the barrier primitive to synchronize. In the first case, data must be fetched from storage at each iteration, while in the second approach it is only fetched once. Overall, Figure 4.7b shows that this latter mechanism is clearly faster. In particular, the total time spent in coordinating the functions is lower when the barrier is used (Sync).

Figure 4.8 draws a comparison between two different barrier objects available in Crucial and several state-of-the-art solutions. More precisely, the figure reports the performance of the following approaches: (i) a pure cloud-based barrier, which combines Amazon SNS and SQS services to notify the functions; (ii) a ZooKeeper cyclic barrier based on the official double barrier [8] in a 3-node cluster; (iii) a non-resilient barrier using the Redis `BLPOP` command ("blocking left pop") on a single server; (iv) the default cyclic barrier available in Crucial, with a single server instance; and (v) a resilient, poll-based (P) barrier implementing the algorithm in [85] on a 3-node cluster with replication.

To draw this comparison, we measure the time needed to exit 1000 barriers back-to-back for each approach. An experiment is run 10 times. Figure 4.8 reports the average time to cross a single barrier for a varying number of cloud functions.

The results in Figure 4.8 evidence that the single server solutions, namely Crucial and Redis, are the fastest approaches. With 1800 threads, these barriers are passed after waiting 68 ms on average. The fault-tolerant barriers (Crucial

(P) and ZooKeeper) create more contention, incurring a performance penalty when the level of parallelism increases. With the same number of threads, passing the poll-based barrier of Crucial takes 287 ms on average. ZooKeeper requires twice that time. The solution using Amazon SNS and SQS is an order of magnitude slower than the rest.

It is worth noting the difference between the programming complexity of each barrier. Both barriers implemented in Crucial take around 30 lines of basic Java code. The solution using Redis has the same length, but it requires a proper management of the connections to the data store as well as the manual creation/deletion of shared keys. ZooKeeper substantially increases code complexity, as programmers need to deal with a file-system-like interface and carefully set watches, requiring around 90 lines of code. Finally, the SNS and SQS approach is the most involved technique of all, necessitating 150 lines of code and the use of two complex cloud service APIs.

A concurrency problem

Thanks to its coordination capabilities, Crucial can be used to solve complex concurrency problems. To demonstrate this feature, we consider the Santa Claus problem [171]. This problem is a concurrent programming exercise in the vein of the dining philosophers, where processes need to coordinate in order to make progress. Common solutions employ semaphores and barriers, while others, actors [36].

Problem. The Santa Claus problem involves three sets of *entities*: Santa Claus, nine reindeer and a group of elves. The elves work at the workshop until they encounter an issue that needs Santa's attention. The reindeer are on vacation until Christmas eve, when they gather at the stable. Santa Claus sleeps and can only be awakened by either a group of three elves to solve a workshop issue, or by the reindeer to go delivering presents. In the first case, Santa solves the issues, and the elves go back to work. In the second, Santa and the reindeer execute the delivery. The reindeer have priority if the two situations above occur concurrently.

Solution. Let us now explain the design of a common solution to this problem [36]. Each entity (Santa, elves, and reindeer) is a thread. They communicate using two types of coordination primitives: *groups* and *gates*. Elves and reindeer try to join a group when they encounter a problem or Christmas is coming, respectively. When a group is full —either including three elves or nine reindeer—, the entities enter a room and notify Santa. A room has two gate objects: one for entering and one for exiting. Gates act like barriers, and all the entities in the group wait

TABLE 4.3: Santa Claus problem's completion time (in seconds) on a single machine and using CRUCIAL.

	Threads	**Threads + DSO**	**Crucial**
p50	20.15	20.91	21.97
p99	21.09	22.03	22.66
Overhead	–	3.8%	9.0%

for Santa to open the gate. When Santa is notified, he looks whether a group is full (either of reindeer or elves, prioritizing reindeer). He then opens the gate and solves the workshop issues or goes delivering presents. This last operation is repeated until enough deliveries, or *epochs*, have occurred.

We implemented the above solution in three flavors. The first one uses plain old Java objects (POJOs), where groups and gates are monitors and the entities are threads. Our second variation is a refinement of this base approach, where the coordination objects are stored in the DSO layer. The conversion is straightforward using the API presented in Section 4.3. In particular, the code of the objects used in the POJO solution is unchanged. Only adding the `@Shared` annotation is required. The last refinement consists in using CRUCIAL's cloud threads instead of the Java ones.

Evaluation. We consider an instance of the problem with 10 elves, 9 reindeer and 15 *deliveries* (epochs of the problem). Table 4.3 presents the completion time for each of the above solutions.

The results in Table 4.3 show that CRUCIAL is efficient in solving the Santa Claus problem, being at most 9% slower than a single-machine solution. In detail, storing the group and gate objects in CRUCIAL induces an overhead of around 4% on the completion time. When cloud threads are used instead of Java ones, a small extra time is further needed —less than a second. This penalty comes from the necessary remote calls to the FaaS platform to start computation.

Takeaways The fine-grained coordination capabilities of CRUCIAL permit cloud functions to coordinate efficiently. The coordination primitives available in the framework fit iterative tasks well and perform better than state-of-the-art solutions at large scale while being simpler to use. This allows CRUCIAL to solve complex concurrency problems efficiently.

4.6.4 Smile library

The previous section presented the portage to serverless of a solution to the Santa Claus problem. In what follows, we further push this logic by considering a complex single-machine program. In detail, we report on the portage to serverless of the random forest classification algorithm available in the Smile library. Smile [118] is a multi-threaded library for machine learning, similar to Weka [88]. It is widely employed to mine datasets with Java and contains around 165K SLOC. In what follows, we first describe the steps that were taken to conduct the portage using Crucial. Then, we present performance results against the vanilla version of the library.

Portage. Porting `smile.classification.RandomForest` consists in adapting the random forest classification algorithm [40] with the help of our framework. In the training phase, this algorithm takes as input a structured file (commonly, `.csv` or `.arff`) which contains the dataset description. It outputs a random forest, i.e., a set of decision trees. During the classification phase, the forest is used to predict the class of the input items. Each decision tree is calculated by a training task (`Callable`). The tasks are run in parallel on a multi-core machine during the training phase. At the same time, the algorithm also extracts the out-of-bag (OOB) precision, that is the forest's error rate induced by the training dataset.

To perform the portage, we take the following three steps. First, a proxy is added to stream input files from a remote object store (e.g., Amazon S3). This proxy lazily extracts the content of the file, and it is passed to each training task at the time of its creation. Second, the training tasks are instantiated in the FaaS platform. With Crucial, this transformation simply requires calling a `ServerlessES` object in lieu of the Java `ExecutorService`. Third, the shared-memory matrix that holds the OOB precision is replaced with a DSO object. This step requires to change the initial programming pattern of the library. Indeed, in the original application, the `RandomForest` class creates a matrix using the metadata available in the input file (e.g., the number of features). If this pattern is kept, the application must load the input file to kick off the parallel computation, which is clearly inefficient. In the portage, we instead use a barrier to synchronize the concurrent tasks. The first task to enter the barrier is in charge of creating the matrix in the DSO layer.[6]

For performance reasons, Smile uses Java arrays (mono or multi-dimensional) and not object-oriented constructs (such as `ArrayList`). As pointed out previously in Section 4.3.3, it is not possible to build proxies for such objects in Java without changing the bytecode generated during compilation. Thus, the portage

[6] This pattern is reminiscent of a Phaser object in Java.

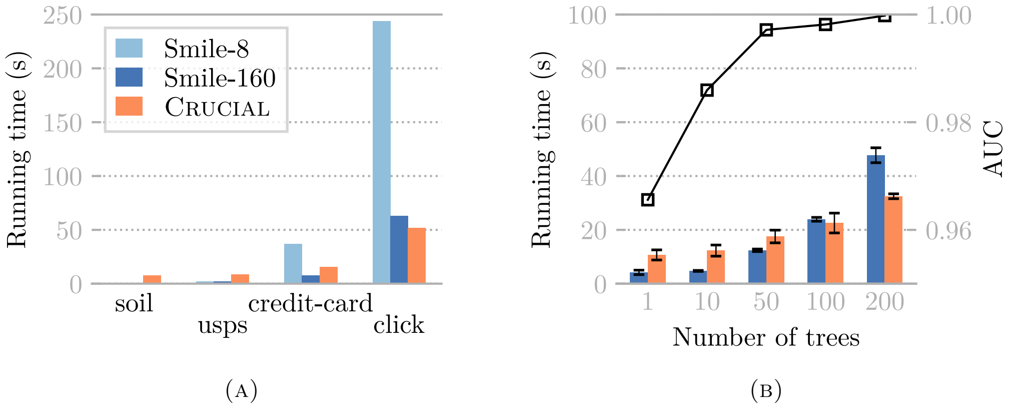


FIGURE 4.9: Smile portage. (A) Performance per dataset using 50 trees. (B) Varying the number of trees for the credit-card dataset [140].

necessitates to transform these arrays into high-level objects. These objects are then replaced with their CRUCIAL counterparts.

Overall, the portage modifies 378 SLOC in the Smile library (version 1.5.3). This is less than 4% of the original code base to run the random forest algorithm.

Evaluation. In Figure 4.9, we compare the vanilla version of Smile to our portage to CRUCIAL. To this end, we use 4 datasets: (*soil*) is built using features extracted from satellite observations to categorize soils [72]; (*usps*) contains normalized handwritten digits scanned from envelopes by the U.S. Postal Service [130]; (*credit-card*) is a set of both valid and fraudulent credit card transactions [140]; (*click*) is a 1% balanced subset of the KDD 2012 challenge (Task 2) [92].

We report the performance of each solution during the learning phase. As previously, CRUCIAL is executed atop AWS Lambda. The DSO layer runs with $rf = 2$ in a 3-node (4 vCPU, 16 GiB of RAM) Kubernetes cluster. For the vanilla version of Smile, we use two different setups: an hyperthreaded quad-core Intel i7-8550U laptop with 16 GiB of memory (tagged Smile-8 in Figure 4.9), and a quad-Intel CLX 6230 hyperthreaded 80-core server with 740 GiB of memory (tagged Smile-160 in Figure 4.9).[7]

As expected for small datasets (*soil* and *usps*), the cost of invocation outweighs the benefits of running over the serverless infrastructure. For the two large datasets, Figure 4.9a shows that the CRUCIAL portage is up to 5× faster.

[7] The JVM executes with additional flags (+XX:+UseNUMA -XX:+UseG1GC) to leverage the underlying hardware architecture.

Interestingly, for the last dataset the performance is 20% faster than with the high-end server.

In Figure 4.9b, we scale the number of trees in the random forest, from a single tree to 200. The second y-axis of this figure indicates the area under the curve (AUC) that captures the algorithm's accuracy. This value is the average obtained after running a 10-fold cross-validation with the training dataset. In Figure 4.9b, we observe that the time to compute the random forest triples from around 10 to 30 s. Scaling the number of trees helps improving classification. With 200 trees, the AUC of the computed random forest is 0.9998. This result is in line with prior reported measures [140] and it indicates a strong accuracy of the classifier. Figure 4.9b indicates that training a 200-trees forest takes around 30 s with Crucial. This computation is around 50% slower with the 160-threads server. It takes 20 minutes on the laptop test machine (not shown in Figure 4.9b).

Takeaways Overall, the above results show that the portage is efficient, bringing elasticity and on-demand capabilities to a traditional monolithic multi-threaded library. We focused on the random forest classification algorithm in Smile, but other algorithms in this library can be ported to FaaS with the help of Crucial.

4.6.5 Usability of Crucial

This section evaluates how Crucial simplifies the writing of stateful serverless applications and their deployment and management in the cloud.

Data availability

Our first experiment assesses that Crucial indeed offers high availability to data persisted in the DSO layer. To this end, the membership of DSO is changed during the execution of the serverless k-means. Figure 4.10 shows a 6-minute run during which inferences are executed with the model trained with k-means in Section 4.6.2. The model is stored in a cluster of 3 nodes with $rf = 2$. The inferences are performed using 100 cloud threads. Each inference executes a read of all the objects in the model, i.e., the 200 centroids.

During the experiment, at 120 s and 240 s, we crash and add, respectively, a storage node to the DSO layer. Figure 4.10 shows that our system is elastic and resilient to such changes. Indeed, modifications to the membership of the DSO layer affect performance but never block the system. The (abrupt) removal of a node lowers performance by 30%. The initial throughput of the system (490 inferences per second) is restored 20 s after a new storage node is added.

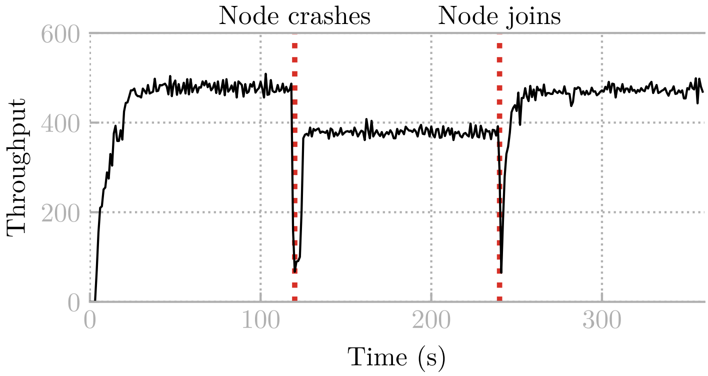


FIGURE 4.10: Inferences per second performed on a k-means model for 6 minutes. Up to 100 concurrent FaaS functions connecting to the shared model on up to 3 DSO nodes with $rf = 2$. Note the FaaS cold start at the beginning.

Notice that handling catastrophic (or cascading) events is possible by running DSO across several availability zones, or even datacenters. In such cases, SMR can be tailored to accommodate with the increased latency between data replicas [134]. Evaluating these geo-distributed scenarios is, however, beyond the scope of this work.

Programming simplicity

Each of the applications used in the evaluation is initially a single-machine program. Table 4.4 lists the modifications that were necessary to move each program to serverless with CRUCIAL. The differences between the single-machine, parallel code and its serverless counterpart are small. In the case of Smile, as mentioned earlier, they mainly come from the use of low-level non-OOP constructs in the library (e.g., Java arrays). For the other programs, e.g., the logistic regression algorithm detailed in Section 4.6.2, the changes account for less than 3%.

Starting from a conventional OOP program, CRUCIAL requires only a handful of changes to port it to FaaS. We believe that this smooth transitioning can help everyday programmers to start harvesting the benefits of serverless computing.

Cost comparison

Although one may argue that the programming simplicity of serverless computing justifies its higher cost [98], running an application serverless should not significantly exceed the cost of running it with other cloud appliances (e.g., VMs).

TABLE 4.4: Lines of code changed in each application to move it to FaaS with CRUCIAL.

Application	Total lines	Changed lines	
Monte Carlo	44	2	(4.5%)
Mandelbrot	88	3	(3.4%)
Logistic regression	430	10	(2.3%)
k-means	329	8	(2.4%)
Santa Claus problem	255	15	(5.9%)
Random Forest [8]	9882	378	(3.8%)

Table 4.5 offers a cost comparison between Spark and CRUCIAL based on the experiments in Section 4.6.2. The first two columns list the time and cost of the entire experiments, including the time spent in loading and parsing input data, but not the resource provisioning time. The last column lists the cost that can be attributed to the iterative phase of each algorithm. To compare fairly the two approaches, we only consider the pricing for on-demand instances.

With the current pricing policy of AWS [14], the cost per second of CRUCIAL is always higher than Spark: 0.25 and 0.28 cents per second for 1792 MiB and 2048 MiB function memory, respectively, against 0.15 cents per second. Thus, when computation dominates the running time, as in k-means clustering with $k = 200$, the cost of using CRUCIAL is logically higher. This difference disappears when CRUCIAL is substantially faster than Spark (e.g., $k = 25$).

To give a complete picture of this cost comparison, there are two additional remarks to make here. First, the solution provided with CRUCIAL using 80 concurrent AWS Lambda functions employs a larger aggregated bandwidth from S3 than the solution with Spark. This reduces the cost difference between the two approaches. Second, as pointed in Section 4.6.1, CRUCIAL users only need to pay for the execution time of each function, and not the time the cluster remains active. This includes bootstrapping the cluster as well as the necessary trial-and-error processes found, for instance, in machine learning training or hyper-parameter tuning [176].[9]

Takeaways The programming model of CRUCIAL allows to easily write conventional object-oriented applications for serverless. Starting from a single-machine code, the changes are minimal. In particular, the distributed shared objects (DSO)

[8] Transitive closure of the dependencies of `smile.classification.RandomForest` in Smile.

[9] Provisioning the 11-machine EMR cluster takes 2 min (not billed) and bootstrapping requires an extra 4 min. A DSO node starts in 30 s.

TABLE 4.5: Monetary costs of the experiments

		Total time	**Total cost**	**Iterations cost**
Logistic regression	Spark	192 s	$0.282	$0.111
	CRUCIAL	122 s	$0.302	$0.154
***k*-means** ($k = 25$)	Spark	168 s	$0.246	$0.050
	CRUCIAL	87 s	$0.244	$0.057
***k*-means** ($k = 200$)	Spark	330 s	$0.484	$0.288
	CRUCIAL	234 s	$0.657	$0.492

layer offers the exact same semantics for state sharing and coordination as a conventional multi-threaded library (e.g., `java.util.concurrent`). Being serverless, applications written with CRUCIAL are scalable. Moreover, they execute at a comparable cost to cluster-based solutions without high upfront investment.

4.7 Chapter summary

This chapter presents CRUCIAL, a system to program highly concurrent stateful serverless applications. CRUCIAL can be used to construct demanding serverless programs that require fine-grained support for mutable shared state and coordination. We show how to use it to implement applications such as traditional data parallel computations, iterative algorithms, and coordination tasks.

CRUCIAL is built with an efficient disaggregated in-memory data store and a Function-as-a-Service (FaaS) platform. Unlike other works, such as Faasm [160] and Cloudburst [164], CRUCIAL can run on any standard FaaS platform, simply requiring the existence of a Java runtime.

Our evaluation shows that, for two common machine learning (ML) algorithms, CRUCIAL achieves superior or comparable performance to Apache Spark. CRUCIAL is also a good fit for function coordination, outperforming ZooKeeper in this task. In particular, it can solve efficiently complex coordination problems despite the inherent costs of its disaggregated design. For data sharing across cloud functions, CRUCIAL compares favorably against storage alternatives such as Redis.

Our framework allows to move traditional single-machine, multi-threaded Java programs to serverless. We use it to port Smile, a state-of-the-art multi-threaded

ML library. The portage achieves performance comparable to the one of a dedicated high-end server, while providing elasticity and on-demand capabilities to the application.

Crucial offers conventional multi-threaded abstractions to the programmer. In our evaluation, less than 6% of the application code bases differ from standard solutions using plain old Java objects. We believe that this simplicity can help to broaden the horizon of serverless computing to unexplored domains.

Chapter 5

Serverless Ephemeral Computational Storage

Data-shipping is a well-known problem in serverless computing. Some works optimize data transfers in the disaggregated architecture and others modify the FaaS platform with caching capabilities. The former does not lessen data-shipping problems. The latter reduces far data transfers but interferes with FaaS properties on elasticity and low latency. This is especially relevant with large data that either complicates function scheduling or easily overflows the cache.

This chapter explores an alternative approach where FaaS remains unchanged, and we push the appropriate (lightweight) computations to the storage tier. For this, we design Glider, the first serverless system for in-storage ephemeral stateful computation. We prototype this approach and evaluate its benefits. Glider effectively eliminates part of the intermediate data in serverless data processing, reduces network complexity, and diminishes data transfer.

A paper with the results of this chapter is in process of publication.

5.1 Introduction

On one hand, serverless computing is constantly gaining traction for data analytics, machine learning, and other distributed computations due to the ability of serverless functions services (FaaS) to provide compute units at large scale, low latency, and fine granularity. This model enables cloud providers and their users to efficiently model applications to reduce operational costs (users manage less resource complexity), monetary expenses (no idle resources are billed), and environmental impact (cloud resources are better utilized).

On the other hand, the stateless, ephemeral nature of serverless functions forces the distributed computations atop them into the traditional two-tiered model with “separation of concerns” between the compute and storage clusters. While the model presents advantages on scalability, it resorts into heavy data transit between components. This is known as a *data-shipping architecture*.

5.1.1 Scope and challenges

In data processing workloads, a data-shipping architecture generates huge traffic of intermediate data, which puts big pressure on the network and commonly becomes a bottleneck in commodity hardware. Additionally, temporary data requires a versatile storage solution to effectively handle its variability [166]. This problem is especially present in serverless computing [83, 141, 150]. Serverless workers are stateless and short-lived, so, contrary to traditional computation systems like Spark or Flink, they cannot keep data in memory throughout different computation stages. Additionally, their memory and computation power are limited, which means that jobs may need further partitioning and extra stages with a resulting increase in intermediate data. To top it off, network connections for serverless functions have limited bandwidth [108, 177].

A solution to serverless data-shipping is still unavailable on cloud platforms. Research projects and tools to orchestrate functions [64, 65, 98] try to alleviate the problem by simplifying programming and data dependencies. However, they do not solve its fundamental issues, i.e., far network transfers of large data. Some works explore locality in functions [3, 160] and different forms of caching data within the FaaS platform [136, 146, 164, 186]. Unfortunately, they require modification of the FaaS platform and are limited to small amounts of data. Furthermore, completely merging storage and computation in a single system has been studied before with results that discourage such an approach [46, 148].

Past research on active storage studies data-shipping in cluster computing [80, 132, 148]. The key idea is to perform computation close to the data, but instead of adding storage to the computation cluster, ship computation to the storage tier. In particular, these systems offload certain operations (data-bound tasks) to small workers deployed directly in the storage system with the objective to minimize the amount of data transferred between tiers. They exhibit impressive data movement reduction and the corresponding benefits on performance and cost.

This concept has not been applied to serverless computing, nor with temporary data, which presents several novel challenges. First, a joint management of ephemeral data and computation in the same system requires to correctly isolate them to avoid resource contention, but still allow their synergy in an elastic system. Second, the restrictions in serverless computing make the model heavily benefit from stateful computation in the storage (such as aggregations or data shuffling). This is not possible with the stateless interceptors we find in existing active storage. Third, serverless workers have modest resources and data processing workloads handle large data. Thus, the storage must provide an effective interface to access data and compute elements that does not overload the workers.

5.1.2 Contributions

In this chapter, we explore a solution to serverless data-shipping with a novel cloud storage service that features *serverless in-storage ephemeral stateful computation*. To fully exploit the advantages of serverless computing, we keep the FaaS platform unchanged and follow a disaggregated model where we offload the appropriate (data-bound) computations to a storage system designed for temporary data.

With this idea, we design Glider, a storage system that integrates ephemeral stateful computation with ephemeral data and cooperates with existing serverless functions to perform big data processing jobs. To achieve it, Glider faces the above challenges of in-storage computation with three principles: storage spaces, storage actions, and a streaming I/O interface. Storage spaces encapsulate resources and allow to equally manage storage and compute resources elastically. Storage actions are a storage element integrated in the storage namespace and allow to perform stateful computation near the data. The streaming I/O is a key interface element to allow serverless workers and storage actions to process a flow of data through a chain of chunks and be efficient with their resources.

We evaluate Glider through several applications. The experiments show how our solution effectively reduces the amount of temporary data that travels through the network, it decreases the number of necessary storage accesses, and lowers overall storage utilization. Consequently, Glider significantly speeds up the performance of data processing applications.

5.2 Background and motivation

In this section, we analyze the temporary data that is generated during data processing workloads and the peculiarities that arise when using serverless services to perform these jobs. Then, we introduce the problem of data-shipping in the serverless computing model. Finally, we provide an overview of our solution with Glider, and review the general challenges that it encounters.

5.2.1 Serverless and temporary data

A large and important part of the data that is transferred in analytics workloads is temporary. For temporary data, we refer to the intra-job data that is created, handled, or consumed during processing and, thus, is not relevant after the job completes. In other words, temporary data is short-lived and easy to regenerate. This excludes the original input and final output application data.

Supporting temporary data requires a dedicated solution that correctly handles its peculiarities. For instance, the variability in size and access patterns that presents ephemeral data need a versatile system that embraces these irregular residents. Some data may require highly performant storage (e.g., in-memory) while others are best stored in cost-efficient hardware (e.g., HDD or SSD). We can find this kind of specialized storage systems in the context of cluster computing [166].

In serverless computing, temporary data is especially present. Traditional cluster computing may communicate several workers to directly pass intermediate data during execution or keep information in their memory space for future computation stages. Unfortunately, this is not possible with serverless functions. Their transient nature and inability to communicate directly denies them to cache data through stages or deliver it to the next function directly. Instead, functions are forced to ship data through far storage.

Additionally, the common choice to relay data in these applications is a cloud-managed object store service, which is slow for some types of temporary data [99, 166]. More performant storage solutions are not available in the serverless cloud. In fact, achieving serverless properties for high-performance storage (in-memory) is not trivial. One recurrent problem in this endeavor is fine-grained elasticity of in-memory stateful elements, an open research topic with some recent work tackling its challenges [105, 109, 164]. Nonetheless, there are more important challenges in serverless data processing that cannot be solved only with faster storage. The fundamental problem is the constant movement of temporary data between serverless functions and the storage service, which is a by-product of a data-shipping architecture.

5.2.2 Data-shipping in serverless

Serverless data processing applications produce large amounts of data that must be transferred back and forth between compute and storage resources. We refer to this pattern as a data-shipping architecture, and it is a well-known problem in the literature due to the pressure it exerts on the network. The peculiarities of serverless workers (e.g., limited memory and compute power, short lifespan, reduced bandwidth, etc.) increase the amount of data transfers needed and their cost [83], resulting in sub-par performance for applications. Here we shortly discuss different approaches to this problem. For an extended review of these topics, we refer the reader to Section 2.3.

In a first instance, several works [98, 141, 150] optimize these data dependencies with high-performance storage systems or intelligent data partitioning and distribution with the objective to achieve the best cost to performance tradeoff. Nonetheless, they fail to reduce the amount of data transfers and thus suffer the performance and monetary costs of shipping lots of data.

In light of this, other projects decide to fully combine a FaaS platform with a storage solution. We find several instances of this idea, from functions sharing memory [3, 160] to FaaS platforms implemented on top of cache stores [136, 146, 164, 186]. Unfortunately, this model does not work well for data processing applications. We have already seen in the literature that compute and storage tiers must be disaggregated in order to correctly scale to distinct demands and isolate their performance to avoid contention [46, 148].

A traditional solution to data-shipping is to ship code to data instead. That is, to offload computation into the storage system with the objective to minimize the amount of data transferred between clusters. Active storage research [80, 132, 148] exhibits impressive reduction of data movement in traditional cluster applications and the corresponding benefits on performance and cost. However, these solutions are not prepared to support serverless computing. As we have seen, serverless data processing is more intensive on temporary data and requires new considerations. We explore them in Section 5.2.4.

5.2.3 An overview of Glider

In this chapter, we present Glider, a novel service model for ephemeral computational storage. Building on past research and ideas (see the different approaches to data-shipping in Section 2.3 and Figure 2.1), Glider explores a new approach to serverless data processing. In essence, the goal is to let data “glide” through the computation pipeline, instead of jumping back and forth between services. A diagram of Glider’s approach appears in Figure 5.1. It affirms that compute and

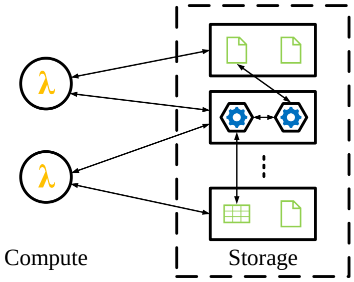


Figure 5.1: Glider's approach to data-shipping.

storage must be disaggregated so that they are individually managed to correctly support their particular demands. To wit, serverless functions must be able to spawn and expire without depending on stateful components. Previous research on active storage [46, 148] has shown us that we cannot fully attach data and computation. Therefore, in order to reduce the amount of data movements, we must ship code to storage instead. In other words, we should offload the appropriate computations to the storage tier. We have also seen that temporary data requires a specialized solution to handle its peculiarities. In this sense, *Glider proposes to add ephemeral computation into an ephemeral storage solution.*

5.2.4 Requirements and challenges

To integrate compute and storage within the same system and face the data-shipping problem in serverless, we study the following three requirements.

Synergize compute and storage Running computation within the storage tier is tricky. This is especially relevant when the storage system is deployed as a multi-tenant service. If the compute operations grow uncontrolled on storage resources, they can create contention and highly impact performance for basic storage operations and other users [46, 148]. Offering a closed library of storage operations may enable such control but pushing meaningful operations to storage requires allowing users to define them in arbitrary code. Consequently, arbitrary operations must be correctly isolated to avoid system performance degradation and still exploit data locality to smooth out the problems of data-shipping. A serverless system requires special caution in this matter, since the fine-grained elasticity expected from it restricts how resources can be managed.

Stateful computation Typically, in-storage computation [132, 148] intercepts data accesses and executes operations in the data path. These ephemeral computations exist only during the operation, and they are anonymous and stateless. While certainly convenient, they are also limited in usability. For instance, multiple data accesses trigger duplicate computations and, to perform an aggregation, the application must move all data twice. This happens frequently in serverless computing due to its transient workers without direct communication. Using computation on storage to aggregate data and other complex patterns heavily reduces the amount of network transfers in serverless data processing workloads (we prove this in Section 5.7). However, this is not possible with just in-line stateless processors and there is no existing system that offers ephemeral stateful computational elements in storage.

Large intermediate data Intermediate data in data processing workloads can be very large, such as when shuffling in MapReduce jobs. These data can be very hard to handle correctly. For instance, traditional in-memory caches are not prepared to support large amounts of data. For this reason, there exist specialized systems to support temporary data efficiently [166]. In serverless computing, this is especially relevant. On one hand, the limited resources of serverless workers prohibits them to load big files entirely. On the other hand, their often-limited network burthens them with long transmission times. In a similar way, in-storage ephemeral computation must also handle large data effectively. These ephemeral compute elements should not keep large data in their own space to remain lightweight and easy to manage.

Other considerations We note that, for temporary data, usual storage features such as durability and fault-tolerance are not a priority [166]. Durability is mostly irrelevant for short-lived intermediate data. In the case of fault-tolerance, although generally useful, it is often implemented at the level of the compute framework (e.g., a function orchestrator in serverless), hinders management at the storage system, and harms its performance.

There are previous projects in the literature that propose serverless ephemeral storage systems. Pocket [109] builds automatic scaling and multi-tenant management into an ephemeral storage solution. It provides it with the capacity to adapt its size dynamically and precisely for serverless data processing jobs as they come and go. Its evaluation shows huge improvements for storing intermediate data against existing storage solutions in the cloud. Jiffy [105] builds upon Pocket and transforms the system into a scalable remote memory for serverless functions to use as a shared space that grows and shrinks on demand. Glider is orthogonal to these projects and it may benefit from their contributions to make it elastic, multi-tenant, and dynamic. However, they do not address the problems derived from data-shipping. Therefore, in this work we focus our description on the novel challenges presented before.

5.3 Glider

We present Glider, a novel serverless service model for ephemeral computational storage. In essence, Glider is a storage system dedicated to large temporary data that also runs ephemeral computation within it. The main goal is to exploit the synergy of both elements coexisting in the same system to counteract the issues presented by the data-shipping architecture adopted in serverless data processing workloads. Fundamentally, Glider is built as a companion to serverless computing (FaaS) services, and it allows intermediate data to "glide" through the different computation stages, instead of constantly jumping back and forth between compute and storage services.

Against the effects of data-shipping, the ephemeral computational storage idea of Glider allows to (i) reduce the amount of data transferred (bytes through network), (ii) decrease the number of data transfers (connections to storage), and (iii) lower storage utilization (data stored).

Glider must face the challenges introduced above for integrating computational capacity to a storage system. To wit, compute and storage resources must be correctly isolated, ephemeral computation must be stateful, and it also must be prepared to handle large data easily. To resolve these challenges, Glider follows these three principles: (i) compute and storage resources join the system as isolated *storage spaces*; (ii) computation is encapsulated in *storage actions*, arbitrary stateful objects; and (iii) compute and storage elements share a common *streamed I/O interface.*

Storage spaces In Glider, storage spaces allow to manage compute and storage resources indistinctly within the system. Elementally, users manage storage namespaces, a logical structure of storage elements (e.g., files or directories). Namespaces and the data they contain are private to their owners. The capacity of each namespace is dynamic and adapts to demand through the joining and leaving of storage spaces. A storage space is a set of isolated resources that contributes a certain amount and type of storage capacity to a determinate namespace. Some storage spaces contribute data capacity, while others contribute compute capacity.

This brings two important benefits. First, they allow to easily integrate computation resources into the storage system. This enables a close synergy between compute and data elements while keeping both isolated to avoid performance interferences. Second, they build one of the fundamental pillars in serverless services: fine-grained resource management. Storage spaces may join and leave the system very quickly (since they only hold ephemeral elements), allowing it to match application demand and achieve better resource utilization.

Storage actions The computational elements that Glider incorporates into the storage system are called *storage actions*. Actions are integrated into the storage namespace at the same level of other elements such as files or directories. In this sense, they are handled as any other storage element in the system. In particular, actions have a name or identifier, and applications may use it to directly read or write into the action, or to organize them in the namespace. Furthermore, actions are automatically managed and distributed by the system in the same way as other elements. This heavily simplifies its usage for developers and management for the service. The logic of an action is arbitrary code provided by the users. Action definitions are deployed into the system beforehand, and applications instantiate them as needed in a storage namespace. The action code is triggered when accessed for the typical read and write operations, where it can process data in and out as desired, access other storage elements, or even utilize remote services. Thanks to being part of the storage namespace, actions are stateful and can be directly addressed multiple times. In consequence, an operation on an action may depend on the results of a previous one, which makes them great for aggregating or caching data, among many other uses. Like any other element in Glider, actions are still ephemeral, as they represent intermediate data processing stages, and should not hold long term data.

Streamed I/O Data processing workloads handle huge amounts of data, and the temporary data that is generated during a job can typically grow even bigger. Transferring and processing so much data takes a lot of time and space, thus being able to stream it in small chunks is essential. This is especially true for serverless computing. First, serverless workers are typically small and cannot hold many data at the same time. The ability to simply process the information through I/O streams allows to process it with fewer and smaller workers, which greatly reduces cost. Second, serverless workers often have limited network bandwidth and may invest much time downloading and uploading data. With I/O streams, workers process and transfer data in parallel, speeding up execution time. Therefore, Glider defines a stream-based I/O interface for all the storage elements, including actions. In the case of actions, this adds extra benefits. When a worker sends the result of its execution to an action through a stream as it generates it, the action starts its own light processing as it receives it. Generally, this opens the possibility to parallelize stages in the data processing pipeline in a simple and straightforward way to improve the overall performance. This kind of I/O streaming has been adopted recently in the cloud by S3 Object Lambda [17] with a similar objective. Object Lambda, however, only intercepts accesses to objects in the storage, its functions are stateless and anonymous, and they are completely decoupled from the storage system in a different layer.

5.3.1 What code should we ship?

When offloading computation to storage, we face an important question: which tasks are "appropriate" to offload? From a utility point of view, our objective is that these tasks reduce the amount of data that travels between tiers. Besides, we may argue that these tasks should be lightweight and transient to facilitate their management within the storage system. With this in mind, we clearly identify two types of computation: compute-bound and data-bound tasks. Compute-bound tasks truly shine in the dedicated compute tier, where they can be freely scaled, and would be heavily hurt from being attached to storage. Examples of these tasks are those that perform data crunching and heavy mathematical computations. In contrast, data-bound tasks are the ones that benefit most from near data computation and have the potential to reduce the amount of data transfers in an application. These are data management tasks. Therefore, it makes sense to offload data-bound computations to storage, while we should never do so with compute-bound tasks. In the case of data-bound tasks, traditional active storage only supports stateless tasks such as data filtering, transformation, or simple queries. We have seen that serverless data processing benefits from stateful computation too. Examples of potential stateful data-bound tasks are aggregations, data shuffling, indexing, or interactive queries.

5.3.2 Using storage actions

Here we provide a general overview of how a developer interacts with a Glider service. Section 5.5 presents a more detailed explanation of using our concrete implementation of the model.

The Glider model is, in essence, that of a common cloud-managed storage service. Users manage their storage namespace, where they can add or remove elements following a certain structure. Applications only interact with storage elements, which have typical access operations for reading, writing, or removing them (e.g., CRUD). Storage spaces are managed by the Glider service, and never by users. However, a cloud vendor must put some limitations to using storage spaces (such as resource limits or timeouts) in order to efficiently manage resources. To this end, a service may allow users to configure some parameters (e.g., size, compute power, timeout) to adjust the behavior of the system when managing the resources for a specific storage namespace.

To use storage actions, programmers should first code its logic by following an interface. The interface defines a set of functions or object methods that the developer may implement as desired. Each of these code elements will run for different operations performed on the action. The main operations are reading and writing, for which the application utilizes I/O streams.

Action definitions (the code) must be deployed into the service. This procedure is similar to deploying a function in a FaaS platform. Users upload a package and register each action so that they can reference them later for instantiation. The service may also allow certain configuration parameters for that action.

Actions are instantiated as any other storage element. Developers create them by providing an identifier within the storage namespace and a reference to the action definition they wish to instantiate. Likewise, applications can get references to existing actions through their identifier. Action deletion works similarly and may choose to wait for on-going operations to finish or abort them. Data operations on actions are identical to other storage elements. The application obtains an I/O stream that it may use to read/write data from/to the action.

5.4 System design

The core principles of Glider based on storage spaces, storage actions, and a streamed I/O interface can be put into practice in several ways. Here we present our concrete proposal, which we use to evaluate our claims.

We create a design for Glider by extending a multi-tiered, high-performance ephemeral storage architecture: the NodeKernel [166]. We choose NodeKernel because it is extensible and specialized for ephemeral data in data processing workloads. But more importantly, its design can easily be expanded for managing multi-tenant deployments and allows to grow and handle resources elastically at fine granularity [109]. This makes it a great substrate for a serverless service.

With Glider, we integrate into NodeKernel the concepts of storage spaces for computation, actions, and their streamed I/O interface. This adds multiple desireable benefits for serverless data processing applications. Near-data computation reduces the problems derived from the data-shipping architecture taken by the serverless model. The statefulness of actions allows to redistribute computation for more efficient pipelines. And the I/O interface allows to process large data with modest sized actions and parallelize tasks.

5.4.1 NodeKernel in brief

Glider builds upon the NodeKernel architecture, a state-of-the-art storage architecture specialized for temporary data in traditional data processing workloads. Here we summarize the key components of the base architecture before detailing the additions that Glider contributes. For a complete description, we refer the reader to its original publication [166].

Storage semantics The higher-level storage semantics in the architecture are provided as data "nodes" and organized in a hierarchical namespace. Nodes are defined as specialized data types implementing a common interface with basic operations to handle data (e.g., `read`, `append`, `size`, etc.) or structure (e.g., `getPath`, `addChild`, etc.). Each node can hold data of arbitrary size. The general organization is managed by a shared storage kernel, which is responsible for allocating storage resources for nodes, handling the hierarchical namespace, and implementing data access operations. Applications connect to the kernel to create, look up, remove, and attach or fetch data to/from nodes. To identify nodes within the storage hierarchy, they are given path names similarly to a file system.

Data nodes are extensible and can provide different specialized data access operations with multiple implementations of the common node interface. Likewise, node types may restrict the way they can be stacked. For instance, a certain node type may only allow a closed set of types as its children (e.g., a table can only hold key-value pairs). As a basis, the NodeKernel architecture defines five custom node types that semantically represent data (`File` and `KeyValue`), containers (`Directory` and `Table`), or specialized structures (`Bag`).

System architecture The NodeKernel architecture manages data in a set of metadata and storage servers. Internally within the system, data is handled in blocks. A block is a fixed sequence of bytes residing in a storage server. The nodes in the storage hierarchy present their data as a byte stream that abstracts a sequence of blocks. The metadata servers administer the hierarchical namespace and the fleet of blocks. Metadata servers may distribute their work by partitioning the namespaces, allowing to scale the system if needed. The storage servers allocate storage blocks and register them on a metadata server. In essence, the metadata servers maintain a list of free and used blocks (and their mapping to storage servers) and assign them to nodes as needed. This way, data is distributed across the cluster automatically. To perform data operations, clients first contact a metadata server, and it replies with the location of the storage block or blocks affected by the operation. The client then uses this information to contact the appropriate storage server(s) and perform the operation. Other operations that only modify the namespace are directly executed at the metadata server.

To accommodate for different types and sizes of data, the NodeKernel architecture supports a tiered storage design. Each storage server implements a type of storage (DRAM, NVMe, HDD, etc.) that it uses to allocate its blocks. Storage servers are deployed as logical entities encapsulating a set of resources, which allows them to exploit the different hardware in the same physical or virtual machine but manage them separately. When a storage server joins the system, it is registered into one, and only one, storage class. Storage classes allow to group storage servers and create relations between them. Typically, a storage class would represent a concrete technology, so that we may have a preferred DRAM tier that falls back to a NVMe tier when full. Nonetheless, multiple classes may contain the same technology and an application may freely choose the class it wants its nodes stored in, and the order classes are filled up. This freedom is key for temporary data that may have disperse requirements.

5.4.2 Versatility through storage spaces

Glider introduces the concept of storage spaces into Crail. We add this concept into the metadata servers to correctly manage storage and compute resources within a storage namespace. We leverage storage servers to create storage spaces. In this sense, each storage server joins the system as a storage space. We run storage servers within containerized resources to provide the necessary isolation and elasticity to bring storage spaces to life and harbor compute and storage elements in harmony within the same system. In particular, the distinction between compute and storage is achieved thanks to a new dedicated storage class: the active class. With this, compute and storage enjoy their isolated resources to avoid contention, performance interferences, and degradation. But at the same time, compute and storage are managed equally within the ephemeral storage system for improved synergy between the elements.

The isolation and transient model of storage servers has already been exploited as a substrate for a multi-tenant storage solution and to elastically scale storage resources on demand in a serverless flavor [105, 109]. These previous works pronounce good results. It allows to isolate different namespaces, from different owners, by each having an independent stack of storage classes with its own pool of storage servers. Moreover, fine-grained elasticity is also possible with storage servers. First, they can be of any size and adapt the storage capacity as needed. Second, they are easy to start and join the system, since the block management schema simply accepts the new capacity and does not require any data movement. Finally, removing storage servers is not a problem either. Data stored in the system is temporary and expires when the application does not require it anymore. Thus, storage servers can be removed without drawbacks as demand disappears.

Resource management through storage spaces is an interesting topic. Our design focuses on how to interconnect storage elements and does not put any constraint on storage space size and management. We believe these are implementation details to be tuned by each service or cloud vendor. Applications have different requirements for ephemeral storage and computation, which need variable capacity. On the other hand, a commercial service should put limits to resource occupancy to effectively manage the pool of resources. The service should thus determine the granularity of storage spaces and the mechanisms to scale them through evaluating a tradeoff between versatility and resource utilization. Some decisions include configurable capacity, CPU time limitation, or timeouts for storage spaces. This may be tricky for stateful elements. However, the temporary nature of the data and computation stored in Glider allows this kind of light resource management and simplified fault tolerance model, like we find in FaaS offerings. Some recent research has evaluated this matter [105, 109].

5.4.3 Actions within the storage namespace

To incorporate storage actions, Glider defines two new components to the system: a new node type and a new storage server type. Additionally, it requires custom management logic in the metadata servers to support the new elements, and the necessary client tools and abstractions to create and operate them.

The action node type From a high-level, logical view, actions are a new node type in the storage semantics. As such, they become another storage element and can be organized by the storage kernel in the hierarchical namespace. Figure 5.2 shows a view of Glider's storage semantics and this integration. Action nodes implement the common node interface so that they can be created, removed, or accessed in an equivalent way. Data access operations on actions are equivalent to other storage elements. Applications obtain I/O streams to send data to them or retrieve it. The overall process, however, is very different. Action nodes do not simply store and then return the data they are fed with, but they house an in-memory object that can process the data with arbitrary stateful logic. Each

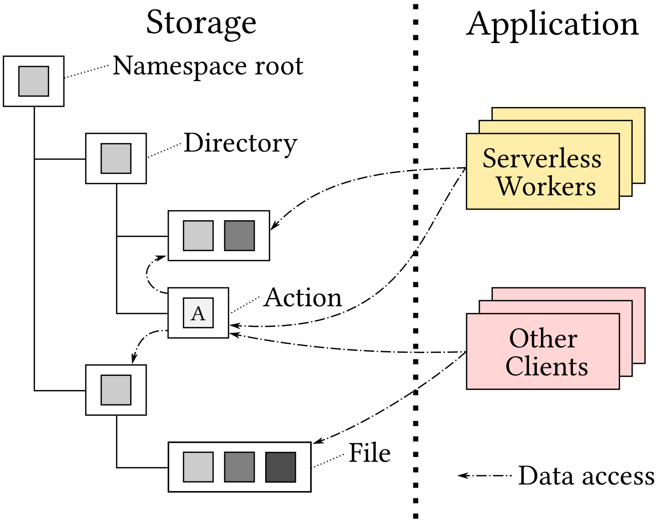


FIGURE 5.2: Glider's storage semantics for the application interface.

operation on an action node triggers the execution of one of the object methods. For example, a write operation on an action could compute an aggregation on the data that is being written and keep it for subsequent operations; or a read operation could generate random data in an infinite stream. We detail how developers may freely code the internal action objects in Section 5.5.2, and how the system executes them in Section 5.4.4.

Storage blocks for actions For other node types, the metadata server assigns storage blocks to them in a chain as their attached data grows. Actions, in contrast, represent computational entities and not pieces of data. As such, they operate on all their data in the same place and cannot be split into multiple storage blocks. Therefore, actions are allocated in a single block, where they will reside. Then, the metadata server only needs to provide one block reference to client applications for any operations they request on an action. Thus, reducing the number of necessary requests to the metadata servers.

The active storage server type Glider adds a new type of storage server into the architecture: the active storage server. Figure 5.3 draws this new storage servers into Glider's data management architecture. Like other storage servers, active servers are encapsulations of storage space. They register themselves with the metadata servers and contribute blocks for a namespace. There are two key

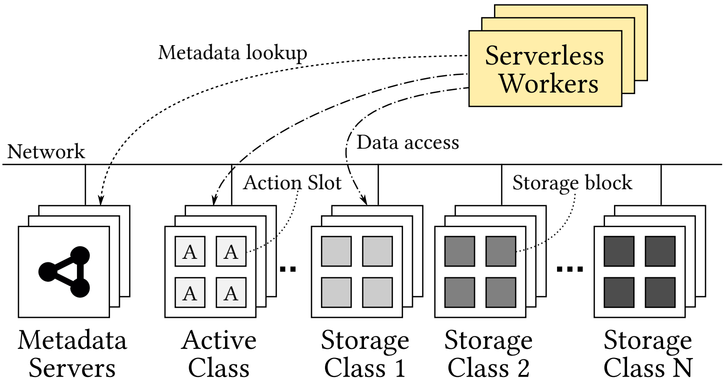


FIGURE 5.3: Glider's storage data management architecture.

differences that set active servers apart from the others: (i) they are grouped into the active storage class, and (ii) their storage blocks are, in fact, action slots. The dedicated storage class allows the storage kernel to allocate action nodes only on active servers. Action slots facilitate managing the size of actions in terms of resources. Similar to other types of storage servers that encapsulate a space for data (a range of bytes in memory or disk), active servers encapsulate a dedicated set of resources (memory and compute power) for running actions. Thus, the size of an active server and the number of slots it registers determine the capacity and performance of its actions. Since the performance requirements of actions depends on the application, it is up to the developer to configure these properties; similar to configuring function memory size in a FaaS platform.

Distributing actions Action distribution across storage servers is also an interesting topic. Like in the case of resource management before, this topic opens lines of research on its own. Glider does not determine a concrete mechanism to solve this, since applications may have very different requirements for this subject. A service may put effort in co-locating actions and other storage elements that have dependencies to exploit locality. However, others may invest in connecting storage spaces with a high-performance network that allows the different elements to perform at full capacity even without co-location [102]. We may find extensive work in the literature about sophisticated scheduling mechanisms for storage and compute elements with dependencies, e.g., gang scheduling [97], or affinity languages [151], to name some.

5.4.4 The streamed I/O and execution model

Accessing actions Actions in Glider are storage elements at the level of, e.g., files. This is very different from other computational storage solutions [17, 148] whose computations only intercept accesses to other elements. From an outside perspective, actions also contain data and can be accessed through a streamed I/O interface. Each stream opened to an action, however, triggers the execution of one of its methods on the server. The method handles the other end of the stream, so that client and action can pass data in a flow of chained data pieces.

Like with any other storage node type, clients must first contact a metadata server to obtain the location of the action. After that, clients communicate directly with the corresponding storage server to perform any operation on it. Since actions only occupy a single block (an action slot), each client only needs to contact the metadata server once. This is different to other node types that scatter their data in multiple storage blocks across storage servers.

The action I/O streams work similarly to reading or writing data in the other types. Long operations are split into multiple chunks of data, and each chunk is sent in a remote procedure call (RPC). Write RPCs contain a sequence of bytes, and read RPCs request one. Glider allows to perform these calls asynchronously to better utilize the network or to perform other tasks concurrently.

The main difference with other storage servers is in that the active storage server feeds the data into the action object, instead of simply copying it to store. The goal is that action methods process an I/O stream that is composed of several RPCs. To achieve this, actions run in execution threads decoupled form the network workers and communicate with them through task queues. When a request reaches the server, a network worker identifies its type and destination and queues it as a data task for a specific action. Each I/O stream has its own task queue that collects the multiple data tasks performed on it. Action threads consume task queues by running the appropriate action method. While the code of the action method consumes or populates its stream, internally, data tasks are processed and completed. When a data task has been processed, the network thread is signaled to complete the RPC. The client decides when to finalize the operation by closing the stream on its side. This sends a final RPC that closes the stream at the server side and ends the method execution.

Actions and concurrency As remote storage elements, multiple clients may access actions concurrently. Since actions are stateful elements, concurrent execution of their logic may result it unexpected behavior due to uncontrolled modification of the action state. Therefore, we need to define a concurrency model for actions that allows programmers to easily reason about their execution.

Glider executes each action as if it were run by a single thread. This means that, at any time, there is only one method being run on each action. Multiple actions may freely execute concurrently. This effectively eliminates unexpected action state modifications. Furthermore, it facilitates action development by freeing programmers from managing complex concurrency issues. To achieve this, action threads obtain exclusive control of an action object while running one of its methods.

There are special cases where an application may benefit from multiple I/O operations running concurrently on the same action. For instance, if we want multiple clients to read data from an action at the same time, the default concurrency model would serialize the operations from each client, paralyzing the read from one of them until the other one completes. To allow multiple clients to access actions concurrently, Glider supports action interleaving. Interleaving can be configured when creating the action. The concept is applied similarly to Orleans [37]. When activated, the execution of an action method may yield control while it is waiting for more I/O tasks. In such event, another action method may take control. Execution is still guaranteed to be single threaded, but methods may take turns in execution before their completion.

5.5 Using Glider

Glider allows users to code, create, and access actions through simple abstractions. The interface has two goals for developing actions: 1) they are managed like other storage elements, and 2) they are coded and deployed like functions in FaaS. This section defines the application interface to manage actions, details the process of developing actions, and illustrates it with an example.

5.5.1 Application interface

Glider's application interface extends the one in Apache Crail. Creating and handling the basic storage elements is equivalent for both systems. Glider creates new interface components to manage storage actions. Table 5.1 summarizes the most relevant elements. The top-level object in the interface is the `StoreClient`, that connects to a concrete namespace in the storage system (Namespace Client in Table 5.1). Its methods allow applications to manage data nodes in the storage hierarchy with `create`, `lookup`, and `delete`. The identifier for nodes is their path in the namespace, similar to a file system. When creating a new node, a parameter allows to specify a preferred storage class. Action nodes are always

Table 5.1: Application programming interface of Glider.

Object	**Methods**
Namespace Client	**create**<T extends Node>(*path*, *sc*) → future(*T*) creates a new data node of type *T* in the *sc* storage class
	lookup<T extends Node>(*path*) → future(*T*) finds an existing node at the given *path* as a *T* node type
	delete(*path*) → future(*boolean*) deletes an existing node at the given *path*
Action Node	**create**<T extends Action>(*il*) → future(*boolean*) creates an action object of type *T* in this node
	delete() → future(*boolean*) deletes the action object in this node
	getInputStream() → *InputStream* obtains a stream to read from this action
	getOutputStream() → *OutputStream* obtains a stream to write to this action
Action Object	abstract **onRead**(*outputStream*) → void
	abstract **onWrite**(*inputStream*) → void
	abstract **onCreate**() → void
	abstract **onDelete**() → void each method defines the action logic for each operation

stored in the special active class. Applications receive proxy references to nodes to further interact with them. For instance, the File and Directory nodes inherited from Crail provide primitives to read and write data or enumerate their content, respectively.

The action node proxy data type implements four basic primitives. The `create` method allows to instantiate an action object into the node, defining the action logic. This method requires a concrete action type definition, coded as explained in Section 5.5.2. An optional parameter (`il`) toggles interleaving for that action. The `delete` method removes the action object within the node. This allows to run action finalization logic, if necessary, but also to change the action definition on an existing node, or simply recreate it to clear its state. The other two primitives allow to obtain I/O streams to transfer data with the action.

It should be noted that all remote operations are non-blocking and asynchronous. Their execution is handled client-side through future objects. This common pattern integrates with modern software interfaces and allows to efficiently handle call results or failures. This allows the construction of buffered I/O streams that use this model to pipeline RPCs, keep a data operation always in flight, and not block the application on network access. Another implementation of direct streams gives the user full control of operations.

5.5.2 Developing actions

To define the logic of actions, developers specialize the `Action` data type (Action Object in Table 5.1). This interface defines four methods: `onCreate`, `onDelete`, `onRead`, and `onWrite`. `onCreate` and `onDelete` run when the action node instantiates or removes the action object, following the analogous method calls to the action node proxy. `onRead` and `onWrite` run for each I/O stream that connects to the action through the dedicated action node proxy methods. All of them are optional, with empty default implementations. `onCreate` and `onDelete` do not have parameters and may be used to initialize and finalize the action object. `onRead` and `onWrite` receive one parameter representing the corresponding I/O stream. For the read operation, the `onRead` method receives a writeable stream that it should populate with data as desired. For the write operation, the `onWrite` method receives a readable stream from which it can consume the data that is being written to the action. Additionally, applications may use the object fields as desired to keep a modest action state (e.g., a counter, a small key-value table, a custom data type, or references to other storage nodes). Conveniently, action objects dispose of a namespace client, by default, to access other storage nodes.

Programmers should make their action definitions available to Glider before creating actions in the storage system. To this end, programmers upload a package containing their definitions, which is then made available to active storage servers. Each action definition is registered with a name. Applications may use this name when instantiating action objects into storage nodes, as detailed in Section 5.5.1. This process resembles function deployment in FaaS platforms.

5.5.3 Application example

Aggregations are one of the main use cases for storage actions. They exploit the statefulness of actions to receive data from multiple workers in a single computational element and merge the result with less network transfers. Moreover, thanks to the streamed I/O interface, applications do not need to store the results from each worker in full. Storing only the merged data allows actions to use few resources and keep storage utilization low.

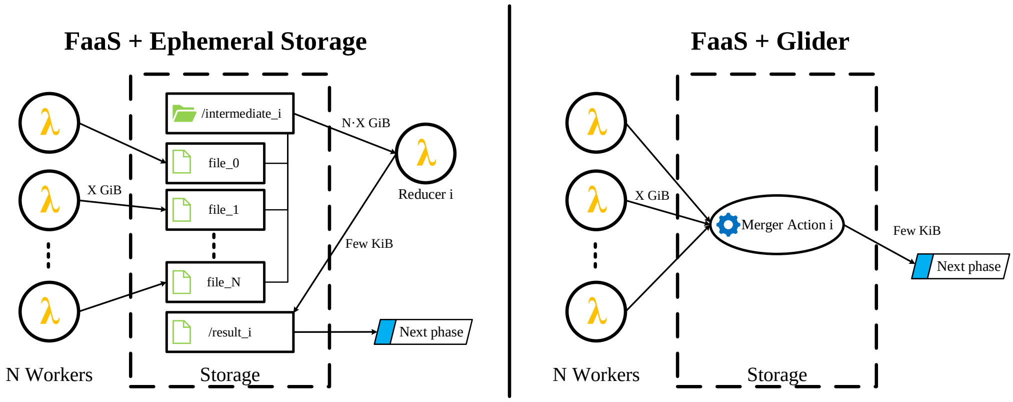


FIGURE 5.4: Diagram of a data processing aggregation with and without actions. For one reducer out of a group of them.

We illustrate these cases with an example implementation of an action performing a reduction in a word count job. Figure 5.4 shows a general diagram of this workload on Glider (right) against a solution with FaaS and traditional storage (left). In the diagram, we only model one reducer, but this is extensible to a group of them. Workers generate a list of key-value pairs representing the counting of words for their part of an input text. The reducer then combines this information into a single dictionary with the aggregated counting, which may be used in a future computation phase. In the base solution on the left, workers write their result in full as storage files. Each worker writes a file for each reducer. The reducer then reads these files, aggregates data, and writes a new file for the next phase. With Glider (on the right), this lightweight aggregation is performed by a storage action. Workers write their key-value pairs directly to actions. The action merges the keys as they arrive, so that it only stores the aggregated data. A next computation stage may read this result directly from the action.

The definition of the merger action is shown in Listing 5.1 with simplified Java code. This action contains an object field, a dictionary, to save the aggregated data. The creation method initializes this field. The `onWrite` method processes the text lines that workers write into the action, combining the keys appropriately into the local dictionary. Note that our application interface allows to wrap the input stream with specialized readers, such as to obtain a stream of lines. This code will process text lines until the stream reaches an end, meaning that the client has finalized the operation. The `onRead` method allows to read the aggregation result from the action. For that, it serializes the local dictionary into the output stream. Note that closing the stream finishes the operation and notifies the client.

Listing 5.1: Action definition to perform an aggregation.

```
 1 public class MergeAction extends Action {
 2   private Map<Integer, Long> result;
 3 
 4   public void onCreate() {
 5     result = new HashMap<>();
 6   }
 7 
 8   public void onWrite(InputStream input) {
 9     Stream<String> lines = input.lines();
10     lines.forEach(line -> {
11       result.merge(
12           Integer.parseInt(line.split(",")[0]),
13           Long.parseLong(line.split(",")[1]),
14           (val, acc) -> val + acc);
15     });
16   }
17 
18   public void onRead(OutputStream output) {
19     output.writeObject(result);
20     output.close();
21   }
22 }
```

This action benefits from interleaving to enable several workers to write to the same action concurrently. In this case, a write operation waiting for more text in line 10 may yield control to another write operation that has text available. This effectively optimizes network utilization.

We fully evaluate this example in Section 5.7. Here we can already see some of the benefits of storage actions. First, actions allow to eliminate the reducer on the FaaS side, which required to transfer all intermediate data again between tiers. Second, thanks to the streamed I/O, the merge action processes data as it receives it, reducing resource utilization from several GiB in multiple files to just a few KiB to store the aggregated dictionary.

5.6 System implementation

Glider is implemented in about 3K SLOC in the Java language. The code base is available online [71]. Glider extends Apache Crail's base implementation of the NodeKernel architecture. It inherits Crail's metadata plane, server management, basic node types, etc. On top, Glider integrates storage actions and all their features to resolve the unique challenges in confronting serverless data-shipping with in-storage ephemeral stateful computation. We implement the new action node type and the new active server type, we modify the metadata servers to support the new storage elements, and we provide new client abstractions to access them. Here we describe the implementation details of these novel features.

The metadata servers in Glider include the new structures to manage action nodes. Action nodes are composable in the storage hierarchy and implement the same management operations as other nodes. Block management for action nodes is modified so that they are allocated a single block. Metadata servers also implement the new specialized storage class for actions. This active class remains logically separated from the others. The metadata servers check and enforce that action nodes are placed in this class, while other nodes are allocated in the other classes as normal. Action nodes are distributed across the active storage servers that joined the class. In our prototype, we uniformly distribute actions in the system with a round-robin mechanism. As discussed in Section 5.3, action distribution and resource management are implementation decisions to be taken by the service provider.

The active server is implemented as a new type of storage server. It is based on the DRAM-backed storage server in Crail and uses TCP-IP for RPC requests. This choice is necessary so that serverless functions can interact with actions. Instead of the byte buffer storage logic of the DRAM server, the active server implements an action manager that handles the creation, execution, and deletion of action objects. The action manager allocates slots for actions depending on configuration and available resources, and it registers them on the metadata servers.

The active server employs a pool of network threads that respond to client requests. When an application requests the creation of an action, a thread uses the manager to instantiate the corresponding action object into memory and perform the necessary initialization. Upon a delete request, the object is discarded after finalizing any in-progress operations and running its finalization method. Data access methods are executed by a separated pool of action threads. This allows to decouple the execution of action methods from individual RPCs, since a single read or write operation may be composed of several requests (enabling the streamed I/O interface). Each read or write operation is assigned an id and a sequence number that are used to create a task queue. Action threads consume these queues to execute action methods. The single-thread-like execution of action methods is achieved with locks. Action threads take the action lock while running one of its methods. For actions with interleaving, the lock is released when a method is waiting for more data in its queue.

5.7 Evaluation

Goals and scope Our objective with this evaluation is to demonstrate and understand the benefits of Glider to execute serverless data processing workloads. To this end, we use different applications to quantify the following key indicators: (i) the amount of data transferred between compute and storage systems, (ii) the number of data transfers between the systems, and (iii) the storage utilization with intermediate data. Since Glider is built on top of Apache Crail, all its features, as well as the ones presented by Pocket for automatic scaling also apply to our system. However, since these properties are orthogonal to our goals, we do not evaluate them; we instead focus on Glider's new contributions to face data-shipping, i.e., reducing data transfer and intermediate data. Our baseline for comparison is the most common approach used for serverless computation as state-of-the-art, i.e., stateless serverless workers generate intermediate data that write and read from remote storage throughout different computation stages. We represent this model as described by PyWren [98] and the AWS Lambda MapReduce reference architecture [20] and as applied by Pocket [109].

In particular, we aim for answers to the following questions:

- How much data can Glider cut from network transmissions between compute and storage clusters?
- Is it desirable to push certain stateful computation stages into storage actions to eliminate a phase of network transfers?
- How much intermediate data can we avoid storing thanks to processing it with storage actions?
- What are the overall performance effects of these changes in the computation model?
- What is the overhead of accessing storage actions against accessing other storage elements?
- Do storage actions scale to the resources available in their storage space?

Setup We run our controlled experiments on a cluster with servers driving two Intel® Xeon® CPU E5-2690 @ 2.90GHz (16 physical cores) and 96 GiB of memory. The network link is a 100 Gbps Mellanox Technologies ConnectX-5 MT27800. We simulate serverless workers by running them on the same cluster. We run storage spaces on the same machines but give them limited resources. E.g., DRAM servers use a single core, while active servers may use up to 16. The storage system is configured by default with 1 MiB block size which is also used as chunk size for I/O streams to achieve buffered data transfers. All experiments require a single metadata server.

Table 5.2: Data ingested by workers, execution time, and data processing throughput for the pipeline benchmark. Processing 10 GiB with 10 workers.

	Ingested	**Time (s)**	**Throughput**
Data-shipping	10 GiB	28.866	2.98 Gbps
Glider	25.7 MiB	10.813	7.94 Gbps
Glider (RDMA)	25.7 MiB	9.182	9.36 Gbps

5.7.1 Benefits

Glider allows to offload computation to the storage tier. By doing so, it causes important impact on data transferred between workers and storage and on the size of the intermediate data that is stored. We explore the benefits of this approach on application performance for different patterns common in serverless computation.

Impact of actions on data movement We study this effect in a data processing pipeline. The pipeline represents a typical application situation where the distributed workers need to ingest data from storage. However, before computing the main operation, data needs to be parsed, arranged, or pre-processed.

In this situation, Glider's opportunity for improvement consists in offloading the filter pre-processing to in-storage actions to minimize the amount of data that transfers to the workers. Workers directly read from these actions that, in turn, read the original files that are distributed in the system. With this configuration, the communication between workers and storage is reduced to only the interesting lines, while actions benefit from near data execution within the storage system to achieve faster access.

We consider an example were text files need to be filtered on a text-based, per line condition, before counting the words on the resulting set. We run both approaches with around 10 GiB of data (from the Wikipedia backup files [179]) and 10 workers (approximately 1 GiB each). In this experiment, we use one active storage server, and one DRAM storage server (for files).

Table 5.2 summarizes the execution results. By using actions, data transfer between workers and storage is reduced by 99.75%. This heavily boosts the performance and cost of applications in real scenarios where workers are far from storage. In our base setup, workers and actions are in the same network as the storage servers. Therefore, the overhead of reading from a worker or an action is the same. However, we still experience time reduction by using actions. This behavior comes mainly from the streamed I/O interface of actions. In particular, it allows parallelism: actions filter data at the same time that the workers count words. We also demonstrate a potential benefit of having actions directly within the storage system. The storage servers may be connected with a high-performance network with RDMA. Actions can exploit this technology, which is unavailable for serverless workers, and further improve execution time.

Impact of actions on intermediate data The stateful computation of actions allows to push simple data management stages to them and reduce the number of connections to storage. Combined with the streamed I/O interface, some temporary data files are no longer needed to be stored in the system. To evaluate this impact, we consider a situation where data generated by a set of workers needs to be aggregated by a reduce operation. In the baseline approach, the reduction must be performed by another worker. This supposes that the intermediate data must be stored in full and then read back by the reduce worker.

In this case, Glider's opportunity for improvement consists in replacing the reduce worker with an action. This change, thanks to the stateful capabilities of actions, eliminates a stage in the compute tier and the intermediate data that it generates. The action will receive concurrently (with interleaving) the data from the first-stage workers and perform the aggregation at the same time.

We illustrate this situation with a synthetic example. The workers generate random numeric pairs (key, value) that they emit as strings. The reducer produces an aggregated dictionary summing the values for each key. In particular, the generated keys are 1024 distinct integers, and the values comprise the full range of a Java Long. Each worker generates 50M pairs, which translates into just over 1 GiB of data when sent through network.

Figure 5.5 presents the results for different numbers of workers. Glider reduces execution time by 27% with 5 workers and by 18% with 10 (see Figure 5.5 left). This is due to the combination of two factors: (1) Glider transfers half the data than the baseline approach (see Figure 5.5 right). (2) Glider allows pipelined processing thanks to I/O streams. That is, the action starts to aggregate data as soon as it starts receiving it while the workers generate it. This optimizes network transit for an overall faster computation. All in all, this demonstrates that pushing stages to stateful storage actions is desirable, as it reduces network transit without harming application execution time.

This experiment also evidences another advantage of Glider: storage utilization. While the baseline needs to store all generated data in storage (about 11 GiB with 10 workers), the action streams its input as it is generated, only storing the resulting dictionary (≈24 KiB), which is relevant for the next stage. In this case, this supposes a reduction in storage utilization of ≈ 99.8%.

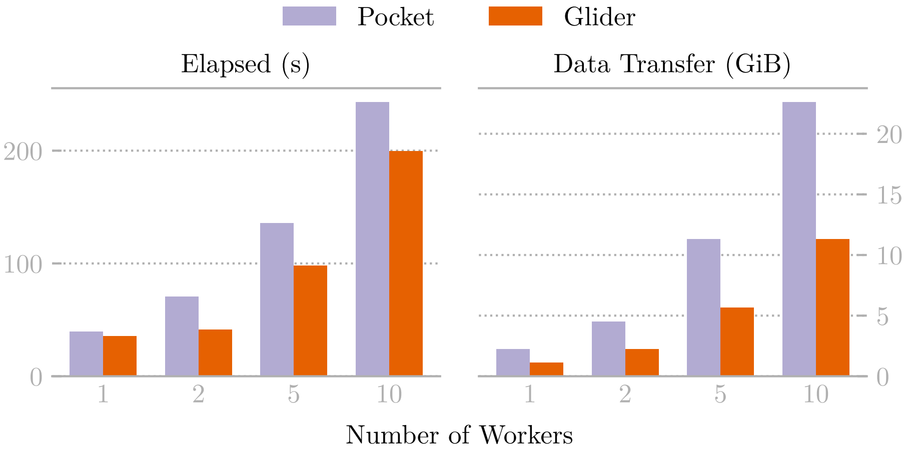


FIGURE 5.5: Reduce operation with Glider against a data-shipping model. Left shows total time elapsed. Right shows data transferred between application workers and storage.

Impact of actions on performance In these two experiments we have experienced an overall performance improvement. Several factors contribute to this matter. First, the reduction in data transfer significantly affects the total run time of these executions. Although the previous experiments run on a fast network, actual serverless functions have more limited bandwidth, which benefits even more from this matter. Another contributor to performance is the elimination of computation stages (and their intermediate data) thanks to offloading them to storage. Besides the reduction in storage utilization, it also lowers data movement. Indeed, it removes the need to transfer the full data back to the compute tier. Lastly, an important performance booster is the ability to stream data between workers and actions. This type of pattern is not possible between serverless functions, but it allows actions to work in parallel with workers to speed up application performance.

5.7.2 Micro-benchmarks

Action bandwidth The objective of this benchmark is to assess the bandwidth to an action in comparison to a base file element. The extra logic necessary to run arbitrary code when accessing an action suggests that a small penalty should be expected.

The experiment consists in using a direct stream (not buffered) to write and read 10 GiB to/from each data type for varying sizes of operation. The direct

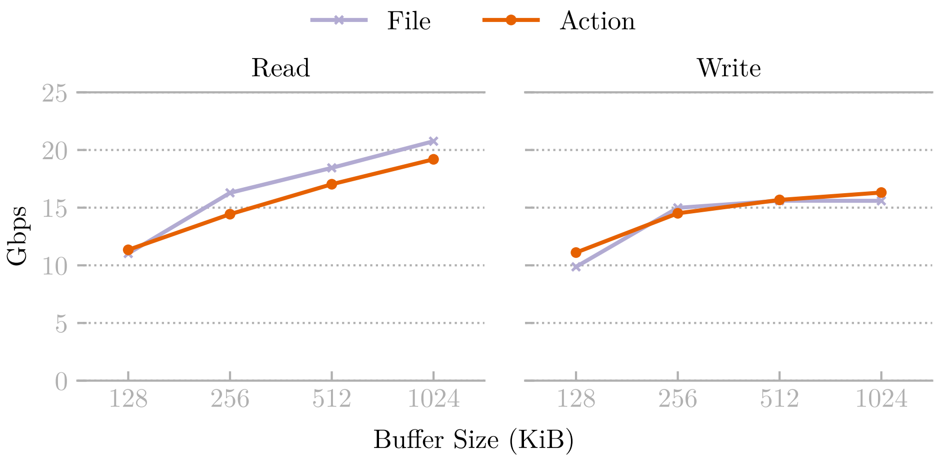


FIGURE 5.6: Average access bandwidth to files and actions for different operation sizes.

stream allows us to take full control of operations and maximize network utilization. To this end, asynchronous operations are done in batches so that there are always data transfers in flight without collapsing the network with too many requests. We adjust the batch size to achieve best performance on each configuration. Note that operations with more than 1 MiB surpass block size and would be split into smaller operations.

Figure 5.6 shows the average results of this experiment. Actions do not add overhead with respect to files for most applications. Read operations achieve at most 11% less bandwidth, while writes can reach up to 12% higher bandwidth. This allows the performance improvement we see in the rest of this evaluation. Note that actions may show slower to users if their logic (user-provided) creates additional overhead.

Action scale We evaluate the capacity of actions to leverage the full CPU and network resources in their storage space. We use the same setup of the previous experiment with 1 MiB operations and replicate it up to 8 parallel actions. Each action still transfers 10 GiB and is accessed by a dedicated client. The active storage space runs 8 network threads to enable this parallelism and bandwidth is computed globally for the aggregated result.

Figure 5.7 shows the results of this benchmark. Running parallel actions improves bandwidth but plateaus around 45 Gbps (which we identified as the limit for TCP operations in the cluster). Similar results are drawn from the same

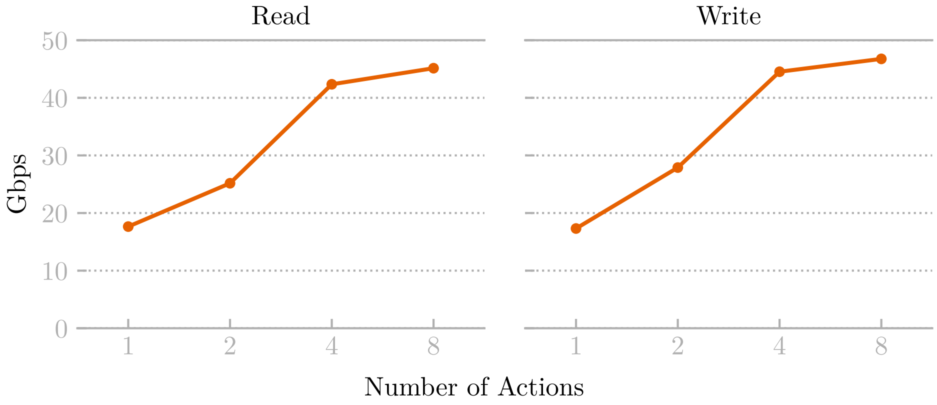


FIGURE 5.7: Average access bandwidth for different numbers of concurrent actions.

experiment assessing files. We conclude that actions scale to the resources of their storage space.

5.7.3 Extended application

In this section, we extend the evaluation of Glider's performance and benefits with a complete serverless computing application. In particular, we implement and study a distributed sort of data. Shuffle operations in MapReduce generate a lot of intermediate data and, consequently, large data transfers. In a serverless setting, this has proved to be especially difficult [83]. Sorting is a severe example of this, because the intermediate data generated contains the full input dataset [141, 150]. Since functions are stateless, each stage needs to read and write everything, and with the resource limitations of functions, this process can become slow and expensive. Even more, this kind of processing results arduous from the perspective of the cloud vendor. The large resources they require are difficult to manage, may hinder other users, and derive into avoidable ecological impact.

Our baseline is an implementation following the PyWren model as taken by Pocket [20, 98, 109]. To perform a sort, a set of workers compute in two phases (map (P1) and reduce (P2)). The input dataset, the intermediate data, and the resulting sorted data are saved in the storage system as files (in DRAM servers). The first stage reads the input dataset and, using a sorting key, distributes the text between the reducers. Each worker generates a new file for each reducer (intermediate data). In the second stage, reducers read back these files, sort their

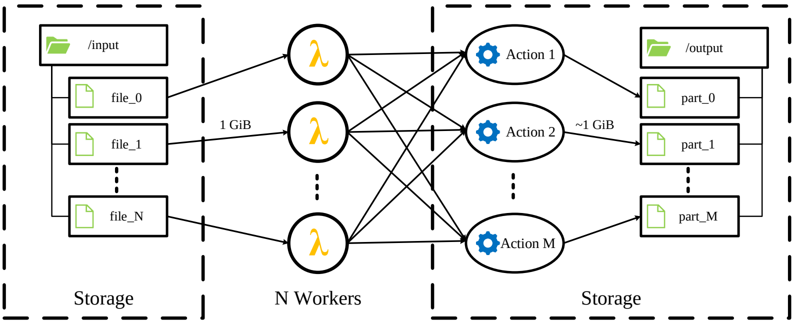


FIGURE 5.8: Diagram of a sort job with Glider.

content, and write the result again. This solution requires a structure similar to the one in Figure 5.4.

Glider improves this situation by pushing the shuffle/reduce operation to storage. We illustrate this mechanism with the diagram in Figure 5.8. This presents three clear advantages. First, an entire stage of workers is no longer needed, which reduces the number of storage accesses. Second, with less storage accesses, less data must be transferred in and out of the storage system. And third, thanks to actions and the streamed interface, part of the sorting can be done in parallel to the first stage, without waiting for the shuffle to finish. In detail, first-stage workers do not write into plain files but send the classified data to the appropriate actions. Note that the process of writing is unchanged since actions and files share the same streamed interface. As actions receive the data through a stream, they parse and keep it in memory (P1). When all workers have finished, the actions start sorting their part of data and write the result as new files (P2).

For the comparison, we use a randomly generated dataset with 1 GiB partitions. We evaluate this application for different number of workers, each of them reading a full partition (i.e., 16 workers sort 16 GiB in total). We use the same number of workers and actions for both phases and employ a DRAM storage server with enough space for storing the data three times and an active server with space for the 16 actions.

Figure 5.9 presents the results of this experiment. The solution using actions is always faster than the model with data-shipping. In particular, Glider achieves a 49.8% time reduction against the baseline with 16 workers. The data-shipping approach keeps the map phase time (P1) constant, but the reduction (P2) is slow due to the intensive reading and parsing of intermediate data from the far storage. On the contrary, Glider is slower during the first phase (P1) because it includes

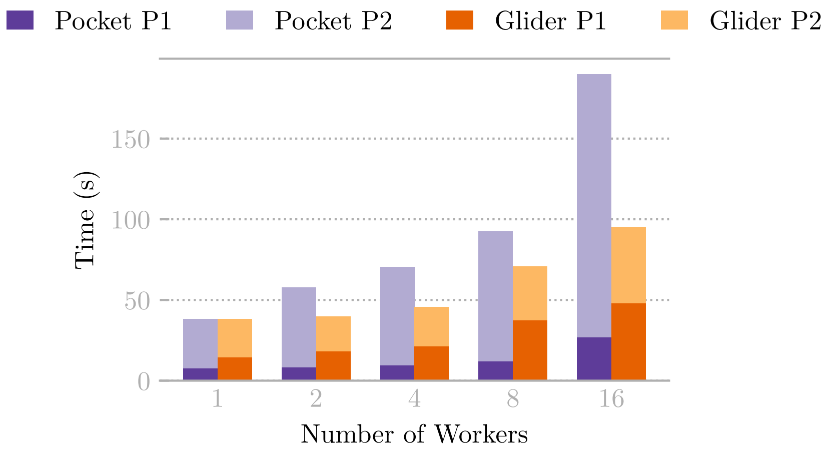


FIGURE 5.9: Sort execution time for a serverless architecture with Glider against a data-shipping approach.

the actions parsing their data. However, the second phase (P2) is up to 71% faster since actions avoid the extra data transfer from storage and already have the parsed data in memory.

Before finishing, let us recount and compare the amount of data movement in each approach. The baseline implementation fully reads and writes the entire dataset to storage twice; accounting for a data transfer of four times the size of the data. Glider only reads and writes once, since it only employs one stage of workers outside storage. In the second phase, actions do not read the data (which is streamed to them by the workers) and write the result from within the storage cluster. Therefore, data transfer is limited to twice the dataset size, for a 50% reduction in data movement.

5.8 Chapter summary

This chapter presents Glider, a novel cloud storage service model aimed to mitigate the data-shipping problem of serverless computation by reducing the amount of data transfers. We argue that FaaS should remain unchanged to not hinder its advantages, so Glider follows a disaggregated approach and defines a new storage service designed to collaborate with existing serverless services. The key contribution of Glider is serverless in-storage ephemeral stateful computation. We achieve this by following three principles: (i) storage spaces synergize compute and storage elements within an ephemeral store; (ii) storage actions encapsulate stateful computation; and (iii) a common streamed I/O interface facilitates handling large data.

We design and prototype Glider on top of NodeKernel and Apache Crail, a multi-tiered storage tailored for temporary data. Glider achieves an efficient integration of ephemeral computation into the storage system thanks to storage spaces. Combined with storage actions, this allows data processing applications to effectively operate on their temporary data. Storage actions go beyond active storage literature in enabling stateful computation. This is possible because actions are integrated as elements of the storage system itself, and it opens the door to more computation offloading and further data transfer reduction. Importantly, actions implement a streamed I/O interface that allows to process large intermediate data with few resources, which translates into heavily reduced storage utilization.

The evaluation of Glider shows clear benefits from pushing computation near storage, helping to reduce the issues of data-shipping present in serverless computing. In particular, we show important reduction in data transfers between compute and storage tiers, reducing worker data ingestion by 99.75%. Furthermore, Glider allows to eliminate stages from the computation, which decreases the amount of intermediate data and storage space utilization. In combination, this improves application performance effectively. For instance, a 16 GiB distributed sort executed with Glider reduces its total execution time by 49.8%.

In sum, this chapter reveals that stateful serverless computational entities are desirable to enhance the programmability and performance of cloud applications. At the same time, they are useful to minimize data transfers in the cloud and reduce resource consumption. This is of interest to both, cloud platforms and users, since it comes with associated benefits in cost, either monetary or environmental.

Chapter 6

Conclusions and Future Work

A lot of distributed applications are stateful. They either need to share mutable data, coordinate several tasks, or simply aggregate the results of multiple workers. Current serverless computing is strictly stateless and all these applications do not have an optimal solution in this new model. Nonetheless, stateful applications still want to benefit from the rapid elasticity, large scale, and just-right billing that serverless offers.

In this thesis, we allowed many new applications to run on serverless technologies. First, we presented the first empirical evaluation of parallelism in serverless computing. Second, we enabled stateless serverless workers to efficiently share their mutable application state at fine granularity and easily coordinate their simultaneous execution. We developed the first framework to code serverless stateful applications with a very simple and well-known interface. Third, we effectively reduced the repercussions of the data-shipping architecture that is enforced to run serverless data processing workloads. Our solution is the first exploration of serverless ephemeral computational storage.

In this section we review all our contributions by recalling our key research questions and presenting the most relevant results. Then, we present some open research lines.

6.1 Overview of contributions

With the objective to *enhance the management of state for serverless computing*, we have put our efforts towards three research questions. These questions define the major challenges tackled in this document and we next review our contributions facing them:

Question I: *What are the benefits and restrictions that serverless computing architectures provide to parallel computing?*

In Chapter 3, we first analyze the architectures of the four major cloud-managed FaaS platforms. It is the first detailed study of platform architecture design that explores the traits that derive in performance differences between services for different applications. We extended this information empirically with a novel benchmark that presents the most detailed evaluation of serverless functions parallelism in the literature.

Our study of FaaS architecture has revealed important differences between platforms that affect their performance. In particular, we have detected two parts of a platform that are highly relevant for parallel applications. First, the virtualization technology used to isolate function resources directly determines the quickness of the service to respond to invocations and the degree of interference between invocations. This overall affects the ability of the platform to provide resources simultaneously for function invocations, which is core for parallelism. The community is striving for lighter virtualization technologies such as microVMs to improve these qualities. Second, the function scheduling approach used to assign resources to invocations determines how resources are managed within the system and the general path of invocations traversing the platform. Again, this affects the quickness of the platform to put invocations in execution, which is essential for simultaneity. We identified two approaches in the studied platforms. A push-based approach provides resources eagerly for faster function spawning, which improves parallelism. A pull-based approach is conservative with resources (although configurable), which makes it more efficient in terms of cost, but it is slower to adapt its scale and heavily hinders parallel applications.

Results The experiments prove that these differences in architecture affect how parallel applications perform in them. We have seen how AWS Lambda, IBM Cloud Functions, and Google Cloud Functions spawn new resource environments for each invocation, allowing functions to run simultaneously and provide good parallelism. Azure Functions, however, packs invocations in very few resources, heavily preventing simultaneity. Experimentation also evinces further details on scheduling approaches, since different configurations on the services make some of them achieve higher parallelism, while others are limited to lower simultaneous executions. In sum, our study has proved that *FaaS is not inherently good for parallel computation.* Performance for these applications varies significantly across platforms, and the users should choose their services very carefully.

After showing that FaaS platforms can be a good substrate for traditional parallel applications with the correct configuration, our next contributions tackle the most important limitations of these platforms for such applications. We start with their need for tasks to share state and coordinate their execution:

Question II: *Can we efficiently use serverless computing for applications with mutable shared state and complex coordination requirements?*

In Chapter 4, we create a novel framework to program highly concurrent stateful serverless applications. Our system enables cloud developers to construct serverless applications that require support for mutable shared state and coordination. We combine a FaaS platform with an efficient disaggregated in-memory data store that allows functions to share state and coordinate at fine granularity. Unlike other works, we advocate against modifying the FaaS runtime to provide a solution that is readily usable in the current cloud. Moreover, this approach ensures that applications still benefit from the elasticity of serverless computing.

With our framework, users can code their serverless applications like traditional multi-threaded applications. FaaS functions are abstracted under a *cloud thread* construct that allows to code and run serverless functions as traditional threads and orchestrate them like so. In order to share program state between these threads, the programmer uses traditional OOP objects to encapsulate it and a few abstractions transfer the objects to the in-memory data store following a remote object pattern. Shared objects are made available to all application threads (functions) and their accesses are linearizable by default to simplify programming. This strong consistency of objects makes them an excellent tool to coordinate threads with well-known primitives or custom logic.

Results We show how to use our system to implement applications such as traditional data parallel computations, iterative algorithms, and coordination tasks. Furthermore, we evaluate these applications atop our system against state-of-the-art solutions. We prove that our serverless implementation of two common ML algorithms achieves superior or comparable performance to Apache Spark. For function coordination, we determine that our system can outperform ZooKeeper in this task; and for sharing mutable data, it compares favorably against available in-memory stores. Importantly, our framework allows to develop all these applications for the serverless model with very little involvement. Indeed, it only requires changing less than 6% of the code bases of standard single-machine implementations, even for porting a state-of-the-art multi-threaded ML library.

Besides application state and coordination, the problem of serverless computing with data is also extensive with huge transfers of intermediate data between computation stages. Hence, next we undertake the data-shipping problem in serverless:

Question III: *Can we improve the data-shipping problem in serverless computing without hindering the advantages of serverless functions?*

In Chapter 5, we design a novel service for serverless ephemeral computational storage. This novel architecture allows to perform computation directly where the data is stored and reduce the number of expensive transfers. Our design describes a system prepared to cooperate with FaaS, but keeps it separate to fully enjoy its benefits. Intentionally, our solution extends an auto-scalable ephemeral storage system, and it inherits these properties from it, making it adequate for fine-grained storage elasticity.

Our contributions are in the incorporation of compute capacity to the storage system in the form of actions. Actions are user-provided pieces of code that can be instantiated into the storage system. Each instance is managed as any other storage element (e.g., a file) for automatic scale and distribution, and provides a standard interface for reading and writing it. These operations on an action trigger the execution of its code, which can operate on a streamed input or output of data, and even access other elements in the storage, including other actions. Beyond past research on active storage, our design allows actions to be stateful, extending the range of applications that can exploit them.

Results We implemented our solution and evaluated it against the currently available approaches that suffer from data-shipping. Our experiments show up to 99% data transfer reduction between workers and storage, the elimination of computation stages with their associated storage accesses and data transfers, and a lower storage utilization by avoiding storing gigabytes of data. As an example, a distributed sort job achieves a 50% execution time reduction thanks to a combination of these qualities.

6.2 Future research directions

Supporting stateful applications on the current offer of serverless computing is an arduous task. Throughout the thesis, we have identified many challenges in this task and provided solutions to several of the problems presented. However, the challenges are complex and can be approached with many more research questions; either more generally, or to more specific matters. Moreover, there are entire research fields we left out of scope, such as security, resource management, scheduling, or sophisticated fault tolerance mechanisms. That aside, the contributions presented in this document also give way to new research opportunities. In this section, we discuss some of the topics and ideas that emerge from our work and that deserve consideration for future investigation.

New technologies for FaaS platforms As more applications take interest in the flashy benefits of serverless computing, these services will need to consider improving themselves to adapt to new requirements. Our study has identified two aspects in FaaS platforms that could be key to cover new applications: virtualization technologies and function scheduling mechanisms. We see future research exploring more towards improving these properties and supporting more applications. In fact, we have already seen work in this direction [2, 9] from both academy and industry and we believe we will see much more in the next years. Another open research line is the creation of a new serverless service similar to FaaS but tailored for distributed parallel applications. This presents similar challenges in virtualization, resource management, and scheduling to guarantee execution simultaneity.

New programming abstractions We presented the *cloud thread* abstraction to facilitate the execution and orchestration of serverless applications and a remote object interface to access shared data. Abstractions of this style improve the development experience and help users avoid common mistakes in distributed settings. We believe there is still new abstractions to explore that are best suited to other types of applications. Potential abstractions can adopt traditional models like other concurrency patterns, or devise novel primitives and constructs, e.g., to define compute and data affinities to schedule computation near data. In fact, we already find open research in tools and frameworks in this line [45, 120]. Abstractions do not stop with serverless computing and can also be applied to cloud development in general, to make the work of developers easier and more productive. We recently start to see novel research in this direction [47].

Transparency Serverless computing supposes great benefits in automatically managing and scaling computation in the cloud. However, it is difficult to migrate applications between platforms due to incompatible APIs when uploading or calling the functions. This requires users or frameworks to develop specific tools to simplify the deployment and execution of their applications and produces the so-called *vendor lock-in.* Transparency in the development of cloud applications has been discussed in recent research [68]. The goal of this research line is to abstract the peculiarities of each cloud service to enable a simplified programming model that can transparently run traditional cluster applications on the cloud with minimum modifications. The cloud vendor lock-in has also triggered interest in multi-platform and multi-region tools to transparently utilize services from different cloud providers [147]. This effort opens new research objectives to orchestrate applications in this novel setting.

Serverless data analytics Executing data analytics applications in a serverless flavor is still open for research in different points of view. We studied the issues that the data-shipping model generates in these jobs and designed a storage solution to counteract them. Nonetheless, other aspects like worker orchestration, pipeline optimization, distributed data structures, and other execution decisions are out the scope of our work. Serverless still needs a complete programming framework that facilitates developers to code their applications and then execute them automatically in the most efficient way. In a traditional cluster setup, we may use Apache Spark [182] or Ray [135] to achieve these goals. Recently, we have seen auto-scaling implemented into these systems [11, 79] to simulate a serverless experience [123]. However, future research is open to build a new data analytics framework that exploits serverless properties by design.

Bibliography

[1] A. Acharya, M. Uysal, and J. Saltz. “Active Disks: Programming Model, Algorithms and Evaluation”. In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VIII. San Jose, California, USA: Association for Computing Machinery, 1998, pp. 81–91. ISBN: 1581131070. DOI: 10.1145/291069.291026.

[2] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. “Firecracker: Lightweight Virtualization for Serverless Applications”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7.

[3] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. 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, pp. 923–935. ISBN: 978-1-931971-44-7.

[4] Apache. *Apache OpenWhisk*. https://openwhisk.apache.org/.

[5] Apache. *Apache OpenWhisk Composer*. https://github.com/apache/openwhisk-composer. 2017.

[6] Apache. *HBase - Coprocessor Introduction*. https://blogs.apache.org/hbase/entry/coprocessor_introduction. 2012.

[7] Apache. *OpenWhisk Concurrency*. https://github.com/apache/openwhisk/blob/master/docs/concurrency.md.

[8] Apache. *ZooKeeper barrier recipe.* https://zookeeper.apache.org/doc/current/recipes.html#sc_recipes_eventHandles. 2019.

[9] gVisor Authors. *What is gVisor?* https://gvisor.dev/docs/. 2020.

[10] AWS. *Configuring Functions in the AWS Lambda Console.* https://docs.aws.amazon.com/lambda/latest/dg/configuration-console.html. 2020.

[11] AWS. *EMR Serverless.* https://aws.amazon.com/emr/serverless/. 2021.

[12] AWS. *Fargate.* https://aws.amazon.com/fargate/. 2017.

[13] AWS. *Invoke - AWS Lambda.* https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html. 2015.

[14] AWS. *Lambda.* https://docs.aws.amazon.com/lambda. 2014.

[15] AWS. *Lambda function scaling.* https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html. 2020.

[16] AWS. *Lambda limits.* https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html. 2020.

[17] AWS. *S3 Object Lambda.* https://aws.amazon.com/s3/features/object-lambda/. 2021.

[18] AWS. *S3 Select.* https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html. 2017.

[19] AWS. *Security Overview of AWS Lambda.* https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf. 2019.

[20] AWS. *Serverless Reference Architecture: MapReduce.* https://github.com/awslabs/lambda-refarch-mapreduce. 2017.

[21] AWS. *Simple Storage Service.* https://aws.amazon.com/s3. 2008.

[22] AWS. *Step Functions.* https://aws.amazon.com/step-functions. 2016.

[23] Azure. *Create a function in Azure using Visual Studio Code.* https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-function-vs-code. 2020.

[24] Azure. *Durable Entities.* https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities. 2019.

[25] Azure. *Durable Functions.* https://functions.azure.com. 2016.

[26] Azure. *Estimating Consumption plan costs.* `https://docs.microsoft.com/en-us/azure/azure-functions/functions-consumption-costs`. 2020.

[27] Azure. *Functions HTTP output bindings.* `https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook-output`. 2020.

[28] Azure. *Functions scale and hosting.* `https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale`. 2020.

[29] *Azure runtime environment.* `https://github.com/projectkudu/kudu/wiki/Azure-runtime-environment`. 2019.

[30] D. Barcelona-Pons. "State Support for Serverless Cloud Services". In: *6th URV Doctoral Workshop in Computer Science and Mathematics.* Ed. by C. Julià and A. Valls. Tarragona, Spain: Publicacions URV, Apr. 2020, pp. 9–12. ISBN: 978-84-8424-865-1.

[31] D. Barcelona-Pons and P. García-López. "Benchmarking parallelism in FaaS platforms". In: *Future Generation Computer Systems* 124 (Nov. 2021), pp. 268–284. ISSN: 0167-739X. DOI: `10.1016/j.future.2021.06.005`.

[32] D. Barcelona-Pons, P. García-López, Á. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas. "FaaS Orchestration of Parallel Workloads". In: *Proceedings of the 5th International Workshop on Serverless Computing.* WOSC '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 25–30. ISBN: 978-1-4503-7038-7. DOI: `10.1145/3366623.3368137`.

[33] D. Barcelona-Pons, Á. Ruiz-Ollobarren, D. Arroyo-Pinto, and P. García-López. "Studying the feasibility of serverless actors". In: *Proceedings of the European Symposium on Serverless Computing and Applications, ESSCA@UCC 2018.* Ed. by J. Spillner. Vol. 2330. CEUR Workshop Proceedings. Zurich, Switzerland: CEUR-WS.org, 2018, pp. 25–29.

[34] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. 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, Dec. 2019, pp. 41–54. ISBN: 978-1-4503-7009-7. DOI: `10.1145/3361525.3361535`.

[35] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López. "Stateful Serverless Computing with Crucial". In: *ACM Trans. Softw. Eng. Methodol.* 31.3 (Mar. 2022). ISSN: 1049-331X. DOI: 10.1145/3490386.

[36] M. Ben-Ari. "How to Solve the Santa Claus Problem". In: *Concurrency: Practice and Experience* 10 (2001). DOI: 10.1002/(SICI)1096-9128(199805)10:63.0.CO;2-2.

[37] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Tech. rep. MSR-TR-2014-41. Mar. 2014.

[38] K. P. Birman and T. A. Joseph. "Reliable Communication in the Presence of Failures". In: *ACM Transactions on Computers Systems* 5.1 (Jan. 1987), pp. 47–76. ISSN: 0734-2071. DOI: 10.1145/7351.7478.

[39] S. Blum. *Amazon SNS vs PubNub: Differences for Pub/Sub.* https://www.pubnub.com/blog/2014-08-21-amazon-sns-pubnub-differences-pubsub/. 2014.

[40] L. Breiman. "Random Forests". In: *Mach. Learn.* 45.1 (Oct. 2001), pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324.

[41] E. Bruneton, R. Lenglet, and T. Coupaye. "ASM: A code manipulation tool to implement adaptable systems". In: *Adaptable and extensible component systems.* 2002.

[42] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. 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, pp. 13–24. ISBN: 9781450369732. DOI: 10.1145/3357223.3362711.

[43] J. Carreira, P. Fonseca, A. Tumanov, A. M. Zhang, and R. Katz. "A Case for Serverless Machine Learning". In: *Workshop on Systems for ML and Open Source Software at NeurIPS.* 2018.

[44] G. Casale, M. Artac, W. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, et al. "RADON: rational decomposition and orchestration for serverless computing". In: *SICS Softw.-Intensive Cyber Phys. Syst.* 35.1 (2020), pp. 77–87. DOI: 10.1007/s00450-019-00413-w.

[45] R. Chatley and T. Allerton. "Nimbus: Improving the Developer Experience for Serverless Applications". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (2020), pp. 85–88.

[46] C. Chen, Y. Chen, and P. C. Roth. "DOSAS: Mitigating the Resource Contention in Active Storage Systems". In: *2012 IEEE International Conference on Cluster Computing*. 2012, pp. 164–172. DOI: `10.1109/CLUSTER.2012.66`.

[47] A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano. "New Directions in Cloud Programming". In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[48] G. V. Chockler, I. Keidar, and R. Vitenberg. "Group Communication Specifications: A Comprehensive Study". In: *ACM Comput. Surv.* 33.4 (2001), pp. 427–469.

[49] CloudButton. *Serverless Benchmark*. `https://cloudbutton.github.io/benchmarks/`. 2020.

[50] Cloudflare. *Durable Objects*. `https://developers.cloudflare.com/workers/learning/using-durable-objects`. 2019.

[51] Cloudflare. *What Is Function-as-a-Service?* `https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/`. 2020.

[52] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler. *FMI: Fast and Cheap Message Passing for Serverless Functions*. 2022.

[53] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011.

[54] Databricks. *spark-perf*. `https://github.com/databricks/spark-perf`. 2014.

[55] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: `10.1145/1327452.1327492`.

[56] D. J. DeWitt and M. Stonebraker. *MapReduce: A major step backwards*. DatabaseColumn Blog. `http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html`. 2008.

[57] T. Distler, C. Bahn, A. Bessani, F. Fischer, and F. Junqueira. “Extensible Distributed Coordination”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 10:1–10:16. ISBN: 978-1-4503-3238-5. DOI: `10.1145/2741948.2741954`.

[58] Docker. *Runtime metrics*. `https://docs.docker.com/config/containers/runmetrics/`. 2020.

[59] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. “Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 467–481. ISBN: 9781450371025. DOI: `10.1145/3373376.3378512`.

[60] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. “NICA: An Infrastructure for Inline Acceleration of Network Applications”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 345–362. ISBN: 978-1-939133-03-8.

[61] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. “Azure Accelerated Networking: SmartNICs in the Public Cloud”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66. ISBN: 978-1-939133-01-4.

[62] *Fission*. `https://fission.io/`. 2016.

[63] B. Fitzpatrick. “Distributed Caching with Memcached”. In: *Linux J.* 2004.124 (Aug. 2004), p. 5. ISSN: 1075-3583.

[64] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. 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.

[65] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. 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)*. 2017.

[66] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Á. Ruiz Ollobarren, and D. Arroyo Pinto. "Comparison of FaaS Orchestration Systems". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 148–153. DOI: 10.1109/UCC-Companion.2018.00049.

[67] P. García-López, M. Sánchez-Artigas, S. Shillaker, P. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, and A. J. Ferrer. *ServerMix: Tradeoffs and Challenges of Serverless Data Analytics*. 2019. arXiv: 1907.11465 [cs.DC].

[68] P. García-López, A. Slominski, S. Shillaker, M. Behrendt, and B. Metzler. *Serverless End Game: Disaggregation enabling Transparency*. 2020. arXiv: 2006.01251 [cs.DC].

[69] S. L. Garfinkel. *An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS*. Tech. rep. TR-08-07. Harvard Computer Science Group, 2007.

[70] V. Giménez-Alventosa, G. Moltó, and M. Caballer. "A framework and a performance assessment for serverless MapReduce on AWS Lambda". In: *Future Generation Computer Systems* 97 (2019), pp. 259–274. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.02.057.

[71] *Glider - GitHub*. https://github.com/danielBCN/incubator-crail. 2022.

[72] M. Goldstein and S. Uchida. "A Comparative Evaluation of Unsupervised Anomaly Detection Algorithms for Multivariate Data". In: *PLOS ONE* 11.4 (Apr. 2016), pp. 1–31. DOI: 10.1371/journal.pone.0152173.

[73] Google. *Cloud Composer*. https://cloud.google.com/composer. 2018.

[74] Google. *Cloud Functions*. https://cloud.google.com/functions/. 2016.

[75] Google. *Cloud Functions Execution Environment*. https://cloud.google.com/functions/docs/concepts/exec. 2020.

[76] Google. *GKE Sandbox: Bring defense in depth to your pods*. https://cloud.google.com/blog/products/containers-kubernetes/gke-sandbox-bring-defense-in-depth-to-your-pods. 2019.

[77] Google. *Google Cloud Function Pricing*. https://cloud.google.com/functions/pricing. 2020.

[78] Google. *Google Cloud Function Quotas*. https://cloud.google.com/functions/quotas. 2020.

[79] Google. *Spark on Google Cloud.* https://cloud.google.com/solutions/spark. 2021.

[80] R. Gracia-Tinedo, M. Sánchez-Artigas, P. García-López, Y. Moatti, and F. Gluszak. "Lamda-Flow: Automatic Pushdown of Dataflow Operators Close to the Data". In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).* IEEE, 2019, pp. 112–121. DOI: 10.1109/CCGRID.2019.00022.

[81] R. Hat. *Reliable group communication with JGroups.* http://jgroups.org/manual/#TOA. 2015.

[82] Hazelcast. *Entry Processor.* https://docs.hazelcast.com/imdg/4.2/computing/entry-processor. 2021.

[83] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. "Serverless Computing: One Step Forward, Two Steps Back". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings.* 2019.

[84] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Serverless Computation with open-Lambda". In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing.* HotCloud'16. Denver, CO: USENIX Association, 2016, pp. 33–39.

[85] D. Hensgen, R. Finkel, and U. Manber. "Two Algorithms for Barrier Synchronization". In: *Int. J. Parallel Program.* 17.1 (Feb. 1988), pp. 1–17. ISSN: 0885-7458. DOI: 10.1007/BF01379320.

[86] C. A. R. Hoare. "Monitors: An Operating System Structuring Concept". In: *Commun. ACM* 17.10 (Oct. 1974), pp. 549–557. ISSN: 0001-0782. DOI: 10.1145/355620.361161.

[87] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. "SPIN: High-Performance Streaming Processing In the Network". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126970.

[88] G. Holmes, A. Donkin, and I. H. Witten. "WEKA: a machine learning workbench". In: *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference.* 1994, pp. 357–361.

[89] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: *USENIX Annual Technical Conference*. USENIX ATC. Boston, MA: USENIX Association, 2010.

[90] IBM. *How Cloud Functions works*. `https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-about`. 2020.

[91] IBM. *System details and limits*. `https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits`. 2020.

[92] T. Inc. *KDD Cup - 2012*. `https://www.openml.org/d/1220`. 2014.

[93] V. Ishakian, V. Muthusamy, and A. Slominski. “Serving Deep Learning Models in a Serverless Platform”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pp. 257–262. DOI: `10.1109/IC2E.2018.00052`.

[94] A. Israeli and L. Rappoport. “Disjoint-access-parallel Implementations of Strong Shared Memory Primitives”. In: *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC’94. 1994, pp. 151–160. DOI: `10.1145/197917.198079`.

[95] Z. Istvan, D. Sidler, and G. Alonso. “Active Pages 20 Years Later: Active Storage for the Cloud”. In: *IEEE Internet Computing* 22.4 (2018), pp. 6–14. DOI: `10.1109/MIC.2018.043051460`.

[96] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. “Formal Foundations of Serverless Computing”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: `10.1145/3360575`.

[97] K. R. Jayaram, V. Muthusamy, P. Dube, V. Ishakian, C. Wang, B. Herta, S. Boag, D. Arroyo, A. Tantawi, A. Verma, et al. “FfDL: A Flexible Multi-Tenant Deep Learning Platform”. In: *Proceedings of the 20th International Middleware Conference*. Middleware ’19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 82–95. ISBN: 9781450370097. DOI: `10.1145/3361525.3361538`.

[98] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. “Occupy the Cloud: Distributed Computing for the 99%”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC’17. 2017. DOI: `10.1145/3127479.3128601`.

[99] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019.

[100] S. Joyner, M. MacCoss, C. Delimitrou, and H. Weatherspoon. *Ripple: A Practical Declarative Programming Framework for Serverless Compute.* 2020. arXiv: `2001.00222 [cs.DC]`.

[101] B. Kalantari and A. Schiper. "14th International Conference Distributed Computing and Networking". In: ICDCN. Springer Berlin Heidelberg, 2013. Chap. Addressing the ZooKeeper Synchronization Inefficiency.

[102] A. Kalia, M. Kaminsky, and D. Andersen. "Datacenter RPCs can be General and Fast". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* Boston, MA: USENIX Association, Feb. 2019, pp. 1–16. ISBN: 978-1-931971-49-2.

[103] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". In: *29th Annual ACM Symposium on Theory of Computing.* STOC. 1997. DOI: `10.1145/258533.258660`.

[104] K. Keeton, D. A. Patterson, and J. M. Hellerstein. "A Case for Intelligent Disks (IDISKs)". In: *SIGMOD Rec.* 27.3 (Sept. 1998), pp. 42–52. ISSN: 0163-5808. DOI: `10.1145/290593.290602`.

[105] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. 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, pp. 697–713. ISBN: 9781450391627. DOI: `10.1145/3492321.3527539`.

[106] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. "An Overview of AspectJ". In: *15th European Conference on Object-Oriented Programming.* ECOOP. 2001.

[107] Y. Kim and J. Lin. "Serverless Data Analytics with Flint". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* Los Alamitos, CA, USA: IEEE Computer Society, July 2018, pp. 451–455. DOI: `10.1109/CLOUD.2018.00063`.

[108] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. 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.

[109] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. ISBN: 978-1-939133-08-3.

[110] *Knative*. `https://knative.dev/`. 2014.

[111] *Kubeless*. `https://kubeless.io/`. 2016.

[112] J. Kuhlenkamp, S. Werner, M. C. Borges, and D. Ernst. "All but One: FaaS Platform Elasticity Revisited". In: *SIGAPP Appl. Comput. Rev.* 20.3 (Sept. 2020), pp. 5–19. ISSN: 1559-6915. DOI: `10.1145/3429204.3429205`.

[113] J. Kuhlenkamp, S. Werner, M. C. Borges, D. Ernst, and D. Wenzel. "Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-Driven Design of Serverless Applications". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC '20. Brno, Czech Republic: Association for Computing Machinery, 2020, pp. 1576–1585. ISBN: 9781450368667. DOI: `10.1145/3341105.3373948`.

[114] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010).

[115] *lambda-maven-plugin*. `https://github.com/SeanRoy/lambda-maven-plugin`. 2019.

[116] H. Lee, K. Satyam, and G. Fox. "Evaluation of production serverless computing environments". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 442–450. DOI: `10.1109/CLOUD.2018.00062`.

[117] B. Li, K. Tan, L. (Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 1–14. ISBN: 9781450341936. DOI: `10.1145/2934872.2934897`.

[118] H. Li. *Smile*. `https://haifengl.github.io`. 2014.

[119] *Lithops - GitHub*. `https://github.com/lithops-cloud/lithops`. 2021.

[120] Y. Liu, B. Jiang, T. Guo, Z. Huang, W. Ma, X. Wang, and C. Zhou. *FuncPipe: A Pipelined Serverless Framework for Fast and Cost-efficient Training of Deep Learning Models.* 2022. DOI: 10.48550/ARXIV.2204.13561.

[121] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982), pp. 129–137. ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056489.

[122] 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. DOI: 10.1109/IC2E.2018.00039.

[123] B. Lorica, E. Liang, and I. Stoica. *The Ideal Foundation for a General Purpose Serverless Platform.* https://www.anyscale.com/blog/the-ideal-foundation-for-a-general-purpose-serverless-platform. 2020.

[124] N. A. Lynch. *Distributed Algorithms.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. ISBN: 1558603484.

[125] X. Ma and A. Reddy. "MVSS: an active storage architecture". In: *IEEE Transactions on Parallel and Distributed Systems* 14.10 (2003), pp. 993–1005. DOI: 10.1109/TPDS.2003.1239868.

[126] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. 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.

[127] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni. "FaaSdom: A Benchmark Suite for Serverless Computing". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems.* DEBS '20. Montreal, Quebec, Canada: Association for Computing Machinery, 2020, pp. 73–84. ISBN: 9781450380287. DOI: 10.1145/3401025.3401738.

[128] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. "My VM is Lighter (and Safer) than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles.* SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763.

[129] F. Marchioni and M. Surtani. *Infinispan Data Grid Platform.* Packt Publishing Ltd, 2012.

[130] O. Matan, H. S. Baird, J. Bromley, C. J. C. Burges, J. S. Denker, L. D. Jackel, Y. Le Cun, E. P. D. Pednault, W. D. Satterfield, C. E. Stenard, et al. "Reading Handwritten Digits: A ZIP Code Recognition System". In: *Computer* 25.7 (July 1992), pp. 59–63. ISSN: 0018-9162. DOI: `10.1109/2.144441`.

[131] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7.

[132] Y. Moatti, E. Rom, R. Gracia-Tinedo, D. Naor, D. Chen, J. Sampe, M. Sanchez-Artigas, P. García-López, F. Gluszak, E. Deschdt, et al. "Too Big to Eat: Boosting Analytics Data Ingestion from Object Stores with Scoop". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE).* IEEE, 2017, pp. 309–320. DOI: `10.1109/ICDE.2017.243`.

[133] S. K. Mohanty, G. Premsankar, and M. di Francesco. "An Evaluation of Open Source Serverless Computing Frameworks". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom).* 2018, pp. 115–120.

[134] I. Moraru, D. G. Andersen, and M. Kaminsky. "There is More Consensus in Egalitarian Parliaments". In: *ACM Symposium on Operating Systems Principles (SOSP).* 2013, pp. 358–372.

[135] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.* OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 978-1-931971-47-8.

[136] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, 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, pp. 228–244. ISBN: 9781450383349. DOI: `10.1145/3447786.3456239`.

[137] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 57–70. ISBN: 978-1-931971-44-7.

[138] *OpenFaaS*. `https://www.openfaas.com/`. 2016.

[139] M. Pawlik, K. Figiela, and M. Malawski. *Performance considerations on execution of large scale workflow applications on cloud functions*. 2019. arXiv: `1909.03555 [cs.DC]`.

[140] A. D. Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi. "Calibrating Probability with Undersampling for Unbalanced Classification". In: *2015 IEEE Symposium Series on Computational Intelligence*. 2015, pp. 159–166.

[141] Q. Pu, S. Venkataraman, and I. 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, 2019, pp. 193–206. ISBN: 978-1-931971-49-2.

[142] Redis. `https://redis.io/`. 2009.

[143] Redis. *EVAL*. `https://redis.io/commands/eval/`. 2022.

[144] Redis. *Replication*. `https://redis.io/topics/replication`. 2019.

[145] E. Riedel, G. A. Gibson, and C. Faloutsos. "Active Storage for Large-Scale Data Mining and Multimedia". In: *Proceedings of the 24rd International Conference on Very Large Data Bases*. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 62–73. ISBN: 1558605665.

[146] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini. "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21. Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 122–137. ISBN: 9781450386388. DOI: `10.1145/3472883.3486974`.

[147] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona. "Toward Multicloud Access Transparency in Serverless Computing". In: *IEEE Software* 38.01 (Jan. 2021), pp. 68–74. ISSN: 1937-4194. DOI: `10.1109/MS.2020.3029994`.

[148] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París. "Data-Driven Serverless Functions for Object Storage". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference.* Middleware '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 121–133. ISBN: 9781450347204. DOI: 10.1145/3135974.3135980.

[149] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López. "Serverless Data Analytics in the IBM Cloud". In: *Proceedings of the 19th International Middleware Conference Industry.* Middleware '18. Rennes, France: ACM, 2018, pp. 1–8. ISBN: 978-1-4503-6016-6. DOI: 10.1145/3284028.3284029.

[150] M. Sánchez-Artigas, G. T. Eizaguirre, G. Vernik, L. Stuart, and P. 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, pp. 31–37. ISBN: 9781450382014. DOI: 10.1145/3429357.3430522.

[151] B. Sang, P.-L. Roman, P. Eugster, H. Lu, S. Ravi, and G. Petri. "PLASMA: Programmable Elasticity for Stateful Cloud Computing Applications". In: *Proceedings of the Fifteenth European Conference on Computer Systems.* EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387553.

[152] J. Scheuner and P. Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review". In: *Journal of Systems and Software* (2020), p. 110708. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110708.

[153] F. B. Schneider. "Implementing fault-tolerant services using the state machine approach: a tutorial". In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167.

[154] *Serverless Framework.* https://www.serverless.com/. 2021.

[155] M. Shahrad, J. Balkind, and D. Wentzlaff. "Architectural Implications of Function-as-a-Service Computing". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1063–1075. ISBN: 9781450369381. DOI: 10.1145/3352460.3358296.

[156] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. *Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider.* 2020. arXiv: `2003.03423 [cs.DC]`.

[157] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman. "Serverless Linear Algebra". In: *Proceedings of the 11th ACM Symposium on Cloud Computing.* SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 281–295. ISBN: 978-1-4503-8137-6. DOI: `10.1145/3419111.3421287`.

[158] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Convergent and Commutative Replicated Data Types". In: *Bulletin of the European Association for Theoretical Computer Science (EATCS)* (June 2011).

[159] M. Shilkov. *From 0 to 1000 Instances: How Serverless Providers Scale Queue Processing.* `https://blog.binaris.com/from-0-to-1000-instances/`. retrieved 2020. 2018.

[160] S. Shillaker and P. Pietzuch. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, July 2020, pp. 419–433. ISBN: 978-1-939133-14-4.

[161] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core.* 2nd. (Revised). Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262692155.

[162] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. "Transactional storage for geo-replicated systems". In: *Symp. on Operating Systems Principles.* SOSP '11. Cascais, Portugal, 2011, pp. 385–400. ISBN: 978-1-4503-0977-6. DOI: `http://doi.acm.org/10.1145/2043556.2043592`.

[163] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro. "A Fault-Tolerance Shim for Serverless Computing". In: *Proceedings of the Fifteenth European Conference on Computer Systems.* EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: `10.1145/3342195.3387535`.

[164] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. "Cloudburst: Stateful Functions-as-a-Service". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097. DOI: `10.14778/3407790.3407836`.

[165] B. Strehl. *Serverless Benchmark 2.0*. https://medium.com/elbstack/serverless-benchmark-2-0-part-i-f23acb8e8a29. retrieved 2022. 2019.

[166] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler. "Unification of Temporary Storage in the NodeKernel Architecture". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 767–782. ISBN: 978-1-939133-03-8.

[167] P. Sutra, E. Rivière, C. Cotes, M. Sánchez-Artigas, P. García-López, E. Bernard, W. Burns, and G. Zamarreño. "CRESON: Callable and Replicated Shared Objects over NoSQL". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 115–128. DOI: 10.1109/ICDCS.2017.239.

[168] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka. "Sequoia: Enabling Quality-of-Service in Serverless Computing". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 311–327. ISBN: 9781450381376. DOI: 10.1145/3419111.3421306.

[169] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: ACM SIGOPS. Copper Mountain, CO, USA: ACM Press, Dec. 1995, pp. 172–182. DOI: 10.1145/224056.224070.

[170] *The Crucial Project - GitHub*. https://github.com/crucial-project. 2020.

[171] J. A. Trono. "A new exercise in concurrency". In: *SIGCSE Bulletin* 26.3 (1994), pp. 8–10. ISSN: 0097-8418. DOI: 10.1145/187387.187391.

[172] M. Uysal, A. Acharya, and J. Saltz. "Evaluation of active disks for decision support databases". In: *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*. 2000, pp. 337–348. DOI: 10.1109/HPCA.2000.824363.

[173] E. van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. L. Abad, and A. Iosup. "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". In: *IEEE Internet Computing* 23.6 (2019), pp. 7–18. DOI: 10.1109/MIC.2019.2952061.

[174] H. Vashishtha and E. Stroulia. “Enhancing Query Support in HBase Via An Extended Coprocessors Framework”. In: *Proceedings of the 4th European Conference on Towards a Service-Based Internet.* ServiceWave’11. Poznan, Poland: Springer-Verlag, 2011, pp. 75–87. ISBN: 9783642247545.

[175] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng. “InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache”. In: *18th USENIX Conference on File and Storage Technologies (FAST 20).* Santa Clara, CA: USENIX Association, Feb. 2020, pp. 267–281. ISBN: 978-1-939133-12-0.

[176] H. Wang, D. Niu, and B. Li. “Distributed Machine Learning with a Serverless Architecture”. In: *IEEE Conference on Computer Communications, INFOCOM 2019.* 2019.

[177] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. “Peeking Behind the Curtains of Serverless Platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18).* Boston, MA: USENIX Association, July 2018, pp. 133–146. ISBN: 978-1-939133-01-4.

[178] R. Wickremesinghe, J. S. Chase, and J. S. Vitter. “Distributed Computing with Load-Managed Active Storage”. In: *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing.* HPDC ’02. USA: IEEE Computer Society, 2002, p. 13. ISBN: 0769516866.

[179] Wikimedia. *Wikimedia downloads.* `https://dumps.wikimedia.org/`. 2022.

[180] C. Wu. *The State of Serverless Computing.* Presentation at QCon New York 2019. 2019.

[181] C. Wu, V. Sreekanti, and J. M. Hellerstein. “Autoscaling Tiered Cloud Storage in Anna”. In: *Proc. VLDB Endow.* 12.6 (Feb. 2019), pp. 624–638. ISSN: 2150-8097. DOI: `10.14778/3311880.3311881`.

[182] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).* San Jose, CA: USENIX, 2012, pp. 15–28. ISBN: 978-931971-92-8.

[183] L. Zeng, S. Chen, Q. Wei, and D. Feng. "SeDas: A self-destructing data system based on active storage framework". In: *2012 Digest APMRC*. 2012, pp. 1–8.

[184] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. "Fault-tolerant and transactional stateful serverless workflows". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1187–1204. ISBN: 9781939133199.

[185] J. Zhang, A. Wang, X. Ma, B. Carver, N. J. Newman, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, et al. *Sion: Elastic Serverless Cloud Storage*. 2022. DOI: `10.48550/ARXIV.2209.01496`.

[186] T. Zhang, D. Xie, F. Li, and R. 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, pp. 1–12. ISBN: 9781450369732. DOI: `10.1145/3357223.3362723`.

UNIVERSITAT
ROVIRA i VIRGILI