Data-centric Serverless Cloud Architecture

Dmitrii Ustiugov

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

The University of Edinburgh

2022

Abstract

Serverless has become a new dominant cloud architecture thanks to its high scalability and flexible, pay-as-you-go billing model. In serverless, developers compose their cloud services as a set of functions while providers take responsibility for scaling each function's resources according to traffic changes. Hence, the provider needs to timely spawn, or tear down, function instances (i.e., HTTP servers with user-provider handles), which cannot hold state across function invocations.

Performance of a modern serverless cloud is bound by *data movement*. Serverless architecture separates compute resources and data management to allow function instances to run on any node in a cloud datacenter. This flexibility comes at the cost of the necessity to move function initialization state across the entire datacenter when spawning new instances on demand. Furthermore, to facilitate scaling, cloud providers restrict the serverless programming model to stateless functions (which cannot hold or share state across different functions), which lack efficient support for cross-function communication.

This thesis consists of four following research contributions that pave the way for a *data-centric* serverless cloud architecture. First, we introduce STeLLAR, an open-source serverless benchmarking framework, which enables an accurate performance characterization of serverless deployments. Using STeLLAR, we study three leading serverless clouds and identify that all of them follow the same conceptual architecture that comprises three essential subsystems, namely the worker fleet, the scheduler, and the storage. Our analysis quantifies the aspect of the data movement problem that is related to moving state from the storage to workers when spawning function instances ("cold-start" delays). Also, we study two state-of-the-art production methods of cross-function communication that involve either the storage or the scheduler subsystems, if the data is transmitted as part of invocation HTTP requests (i.e., inline).

Second, we introduce vHive, an open-source ecosystem for serverless benchmarking and experimentation, with the goal of enabling researchers to study and innovate across the entire serverless stack. In contrast to the incomplete academic prototypes and proprietary infrastructure of the leading commercial clouds, vHive is representative of the leading clouds and comprises only fully open-source production-grade components, such as Kubernetes orchestrator and AWS Firecracker hypervisor technologies. To demonstrate vHive's utility, we analyze the cold-start delays, revealing that the high cold-start latency of function instances is attributable to frequent page faults as the function's state is brought from disk into guest memory one page at a time. Our analysis further reveals that serverless functions operate over stable working sets - even across function invocations.

Third, to reduce the cold-start delays of serverless functions, we introduce a novel snapshotting mechanism that records and prefetches their memory working sets. This mechanism, called REAP, is implemented in userspace and consists of two phases. During the first invocation of a function, all accessed memory pages are recorded and their contents are stored compactly as a part of the function snapshot. Starting from the second cold invocation, the contents of the recorded pages are retrieved from storage and installed in the guest memory before the new function instance starts to process the invocation, allowing to avoid the majority of page faults, hence significantly accelerating the function's cold starts.

Finally, to accelerate the cross-function data communication, we propose Expedited Data Transfers (XDT), an API-preserving high-performance data communication method for serverless. In production clouds, function transmit intermediate data to other functions either inline or through a third-party storage service. The former approach is restricted to small transfer sizes, the latter supports arbitrary transfers but suffers from performance and cost overheads. XDT enables direct function-to-function transfers in a way that is fully compatible with the existing autoscaling infrastructure. With XDT, a trusted component of the sender function buffers the payload in its memory and sends a secure reference to the receiver, which is picked by the load balancer and autoscaler based on the current load. Using the reference, the receiver instance pulls the transmitted data directly from sender's memory, obviating the need for intermediary storage.

Lay summary

Serverless has emerged as a new popular cloud computing paradigm that offers an efficient programming model, automatic scaling with load changes, and a flexible, pay-as-you-go billing model. In serverless, developers structure focus on writing the business logic of their application, structured as a workflow of connected pieces of logic called functions, while the providers manage cloud resources on demand. This labor division opens great opportunities for systems researchers who seek to innovate in serverless computing. Unfortunately, leading serverless providers, like AWS Lambda and Azure Functions, continue to rely on proprietary infrastructure, leaving academics to black-box research with production offerings or building their in-house prototypes of various serverless infrastructure components.

This thesis analyzes the state-of-the-art cloud designs and proposes a novel high-performance and resource-efficient cloud architecture. We start by introducing an open-source framework for end-to-end and in-depth performance analysis of commercial, even proprietary, clouds. This analysis identifies that a modern cloud architecture's performance is limited by ubiquitous data movement, which arises due to the separation of compute and data management. To address this problem, our work introduces a *data-centric* serverless cloud architecture, which we implement in three steps. First, we introduce vHive, an open-source framework for serverless experimentation, that enables cross-stack innovation in serverless systems. Second, using vHive, we introduce a record-and-prefetch technique that allows to reduce the time to spawn new instances of serverless functions on demand, obviating the need for keeping idle instances alive for a long time. Finally, we design a high-performance serverless-native communication fabric, prototyping it in vHive, that allows functions to communicate at a high speed without the need to pass data via a traditional storage service.

Acknowledgements

As much as my Ph.D. journey was twisty and full of unexpected moments, this important part of my life was enriched by the moments that I shared with my family, advisors, colleagues, and collaborators. Their support and care in the times of storms and daily routines was priceless. Most importantly, if I went back in time, I would not change any of my decisions. I completed my Ph.D. at EASE (Lab), and I am very grateful for being given a chance to do so.

First and foremost, I thank my parents, Iuliia and Sergei, and my sister, Alina, who were always there for me, always patient and comforting. Their continuous support were replenishing my physical and emotional energy required during my pursuit of the academic career. I would like to especially thank my fiancée, Nanqian, for her love and care. Our mutual trust and deep understanding of each other made every day of my life full of joy and gave my life a deeper meaning. I also thank my grandparents: Nadezhda Pimenova who inspired me to exercise and improve, Anatolii and Irina Ustiugov whose life and career were and are a great inspiration for what I do.

I am infinitely grateful to my advisors Ed and Boris, whose patient guidance drove me to the finish line of this journey and to the new beginning. I thank Boris for being my guiding star in academia, showing me that I am able and worthy, teaching me how to write great papers and position my work, as well as being an endless source of positive energy, self-expression, and creativity. I am grateful to Ed for being my inspiration for doing computer systems research, showing me how to do experimental science, believing in my abilities, and having my back in the darkest hour.

I am deeply grateful to my other supervisors who played a big role in my Ph.D. I thank Ana Klimovic for being my role model as a young-career advisor and researcher and teaching me how to build an inclusive and supportive environment among colleagues and advisees; Dionisios Pnevmatikatos for showing me that advisorship can be relevant, modest, and humble; Virendra Marathe for believing in my research approach and staying by my side at the tough time; Babak Falsafi for demonstrating me the "per aspera ad astra" approach.

The success of my Ph.D. work would not be possible without the fantastic students, interns, and collaborators for their limitless efforts and faith in vHive and other projects. I thank Marios Kogias for being my guide in the computer systems community and a very good friend who showed me the way through the academic job search avenues; Plamen Petrov for stretching the boundaries of what the state-of-the-art hardware and software can do and pulling out both offensive and constructive projects in single year; Theodor Amariucai for showing the commercial their place (at the moment); Shyam Jesalpura for the sleepless nights and enabling real autoscaling in serverless for data-intensive applications; Michal Baczun for excellence in disciplined coding and going beyond the limits and comfort zones for our team; Bora M. Alper for bringing continuous integration to a new level; Amory Hoste for opening the international borders to vHive; Yuchen Niu for demystifying serverless microarchitectural implications; Franky Romero for making vSwarm comprehensive and showing that collaboration with an 8-hour time zone difference can be effective; Rustem Feyzkhanov for being a true friend and collaborator who inspired me to pursue research in serverless and who contributed his knowledge and code to connect the industry and academia via creating the vSwarm initiative; Lei Yan for formulating an effective research approach as *out-of-order thinking, in-order execution*.

Next, I would like to acknowledge my lab(s) mates for the fruitful joint research work and limitless fun we had together. I thank Mark for being the only friend and colleague who remained by my side throughout my transitions across all three labs I was a member of, for opening the world of wine to me, and for the endless chats about work, life, and everything else; Artemiy Margaritov for showing how to be creative in research, for always being there in the hard times, and for a true friendship; Amna Shahab for being supportive and inspiring and for being the true leader of the leasure activities at the EASE Lab; Prof. Alex Daglis for the valuable support and care he gave me during the time of my cross-country, cross-university transition; Javier Picorel for his total knowledge of everything technical and for being my first role model in research; Mario Drumond for his demonstrating true friendship by being direct and his search efforts in the darkness of St. Sulpice; Nooshin for showing the soft power in a predator environment; Arash Pourhabibi for always bringing the delicious “thingies” to the lab; David Schall for always being there to help and showcasing that sleep and rest are totally optional; Stanko Novakovic for his guidance during my first baby steps in systems research; Mia Primorac for being my birthday buddy and the academic sister and all the cheerful moments; Sid Gupta for many memorable moments.

Many thanks go to the brilliant people and friends I met at the University of Edinburgh: Antonio Barbalace for showing me that great research can be done with passion and style; Michio Honda for the support he gave me when I started my faculty job search and being the role model of an advisor in experimental science; Luo Mai for demonstrating the way to do impactful and relevant research at the crossroads of Machine Learning and Systems; and many others.

Probably, the best moments of my Ph.D. journey were away from my office during the salsa/bachata classes and fiestas latinas. I would like to especially thank Paula Espinosa for sharing my passion for technology and creativity, for being my muse and guide in new creative pieces; Jaden Kimura for showing me new horizons of self-expression; Ola Ciszek for being fun and never complaining about my mistakes in salsa; Jasmine Ng for my makeup adventures and an excellent company in the dark Edinburgh winter evenings.

My Ph.D. life would not be complete without the football gang, I thank them for showcasing my value as a forward in football: Vasilis Gavrielatos for stressing the value of the work-life balance by showing excellence in both research and leisure; Antonis Katsarakis for holding our back in defense; Dimitar for encouraging my development as a football player; and many others including Cilian, Fazl, Plamen, Radu, Theo.

I would like to thank my good old friends who always hold my back and share any good or bad times of my life. I thank Kolya, Pasha Yudin, Pasha Zaichenkov, Rustem Feyzkhanov, and Senya Zaostrovnykh for being my best always-there friends, classmates, flat and even room mates; Paul Merinov, Sasha Ashikhmin, Vova Dmitriev for being my best friends and the New Year Pre-Party disciples with whom I share a lot of fun and the challenges of the MIPT times and afterwards; Yura Savchenko for being my best friend and flatmate during our undergrad at the MIPT and always; Masha Sokolova, Serega Sukhachev, Artem Murashov, Vanya Muratov, Sasha Polonov for being my best school friends and for all the great time and fund we had together from the school times till now; Vasya Oreshko, Robert Khasanov, Tanya, Kamil Garifullin, Serezha Shishlov, Vova Gnatyuk, Sasha Titov, and Sasha Butuzov for being excellent colleagues and mates; Masha Litovchenko for hours of inspirational discussions; Yang Ye for showing that friendship knows no distance and time zones; Anya Petrovich for offering me the unique perspective of art and ambition, for sharing the warm evenings in the cold times, and for being my guide in the world of books and fine arts; and others.

I would like to thank other amazing people who I met during Ph.D. years, including Arthur, Rorry, Rishabh, Yuchao, Hoss, for the countless hours we spent sharing ones of the most enjoyable moments of my life. I would like to particularly thank the people who were my muses in some periods of my Ph.D. time, whose support and care kept my heart warm even in the coldest days: Somya Arora for seeing me as an advisor and a leader for the first time; Regina Tsokur for sparking creativity in me and the every day calls that kept me above the water during the beginning of my Ph.D.

Thank you all, you make my life a truly fascinating story and a real adventure!

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the material used in this thesis has been published in the following papers:

- D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing Tail Latency in Serverless Clouds with STeLLAR," in *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021. [138]

- D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021. [143]

In addition to the works mentioned above, which form the backbone of this thesis, I also contributed to other relevant publications during my studies including:

- D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm Serverless Functions: Characterization and Optimization," *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*, 2022.

- D. Ustiugov, A. Daglis, J. Picorel, M. Sutherland, E. Bugnion, B. Falsafi, and D. Pnevmatikatos, "Design Guidelines for High-Performance SCM Hierarchies," in *Proceedings of the 4th International Symposium on Memory Systems (MEMSYS)*, 2018. [139]

- A. Margaritov, D. Ustiugov, B. Grot, and E. Bugnion, "Towards Virtual Address Translation via Learned Page Table Indexes." in *Workshop on ML for Systems at the 32nd Conference on Neural Information Processing Systems (NeurIPS)*, 2018. [109]

- A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched Address Translation," in *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*, 2019. [108]

- A. Margaritov, D. Ustiugov, A. Shahab, and B. Grot, "PTEMagnet: Fine-grained Physical Memory Reservation for Faster Page Walks in Public Clouds," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021. [110]

- M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, “The Mondrian Data Engine,” in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017. [62]

- M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. N. Pnevmatikatos, “Algorithm/Architecture Co-Design for Near-Memory Processing,” *ACM SIGOPS Operating Systems Review*, 2018. [63]

- D. Ustiugov, P. Petrov, M. R. S. Katebzadeh, and B. Grot, “Bankrupt Covert Channel: Turning Network Predictability into Vulnerability,” in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT), co-located with USENIX Security*, 2020. [140]

- A. Daglis, D. Ustiugov, S. Novakovic, E. Bugnion, B. Falsafi, and B. Grot, “SABRes: Atomic Object Reads for In-Memory Rack-Scale Computing,” in *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, 2016. [53]

- S. Novakovic, A. Daglis, D. Ustiugov, E. Bugnion, B. Falsafi, and B. Grot, “Mitigating Load Imbalance in Distributed Data Serving with Rack-scale Memory Pooling,” *ACM Transactions on Computer Systems (TOCS)*, 2019. [120]

(*Dmitrii Ustiugov)*

To my grandfather, Arkadii Ustiugov, who inspired me to pursue a career in science.

Contents

Chapter 1

Introduction

In today's data-driven economy, the ability to collect, store, and process data in real-time is a critical competitive advantage. As the demand for data and compute power continues to grow exponentially, the design of cloud storage and compute services must evolve to meet the requirements of modern applications.

Serverless computing has emerged as an increasingly popular cloud computing paradigm and architecture, which attracts cloud users with its simple programming model, fine-grain billing, and elastic resource scaling in response to application load changes [84, 44]. Serverless computing today is the fastest growing cloud service and deployment model of the past few years, increasing its Compound Annual Growth Rate (CAGR) from 12% in 2017 to 21% in 2018 [45, 111].

In the "serverless" computing paradigm, application code still executes on servers, however developers are freed from the burden of managing cloud server infrastructure. Developers write code for their applications as a collection of fine-grain, short-running tasks, called functions, which can be chained together and invoked based on events (e.g., user clicks). For each function invocation, cloud providers automatically allocate and launch a function instance, which consists of CPU and memory resources as well as a copy of the user code required to execute the function.

The key benefit of serverless computing is that cloud providers quickly and automatically scale the number of function instances up and down based on the load and users pay only for the amount of resources consumed over time. This "autoscaling" capability of serverless systems is key for offering programming simplicity and economical efficiency. However, to facilitate scaling, providers tend to restrict functions to be stateless, disallowing them to hold or share state across invocations of the same or different functions. The labor division between application developers and cloud providers opens opportunities for systems researchers to innovate in serverless computing.

Unfortunately, leading serverless providers, such as AWS Lambda and Azure Functions, rely on *proprietary* infrastructure. While existing in other computer systems domains, where the entire infrastructure is hidden from the cloud users, this gap is particularly large for serverless computing, preventing academics from discovering and solving relevant problems in realistic scenarios. Hence, it is paramount for computer systems researchers to devise a methodology and a toolchain for analyzing state-of-the-art systems to pinpoint fundamental problems in real-world systems, even without access to source code of particular commercial clouds. The researchers then need a framework to design and prototype full-stack solutions to the identified problems, assessing their efficiency in the context of a real serverless cloud.

Serverless cloud infrastructure takes complete responsibility for cloud resources management, which is beneficial for cloud users but raises a number of important challenges for cloud providers. The first challenge lies in enabling serverless cloud to react quickly to load changes, spawning new function instances on demand. This reaction time primarily depends on the time the infrastructure takes to spawn a new instance of a function in a cloud datacenter. The second challenge stems from the fact that providers restrict the serverless programming model to stateless functions, which cannot share data across one another. Many of serverless applications are data-intensive [66, 67], making serverless cloud support for fast cross-function communication an essential requirement to allow the applications to achieve their performance goals.

1.1 Thesis Contributions

To address the two design challenges mentioned above, this thesis takes a holistic approach to designing a serverless cloud architecture. We start by building STeLLAR, the methodology and the framework that enables performance analysis of production clouds. Using STeLLAR, we identify that performance of modern commercial serverless clouds is dominated by *data movement*. We identify two key bottlenecks that are inherent to the modern serverless architecture (and common for all three cloud providers we study) namely the cold-start delays that is the time to spawn a new instance in response to traffic changes and slow communication across serverless functions. To enable full-stack innovation in serverless systems, we introduce a framework, called vHive, that integrates cutting-edge technologies and components released by the leading serverless cloud providers. Using vHive, we build two solutions that address both of the aforementioned bottlenecks. First, to reduce the cold-start delays, we find that serverless functions operate over stable memory working sets - even across function invocations - that can be effectively recorded and prefetched. Second, to accelerate cross-function communication, we introduce a serverless-native data communication fabric combining high performance with autoscaling capabilities. Together these contributions lead to the following thesis statement:

Thesis Statement
Designing a serverless cloud as a data-centric system
allows combining high performance and resource efficiency.

1.1.1 Serverless Clouds Characterization with STeLLAR

We devise a methodology and an open-source framework called *Serverless Tail Latency Analyzer (STeLLAR)* for detailed tail latency analysis of production serverless platforms, each of which comprises many proprietary software components.[1] To enable performance analysis of cloud infrastructure, we find that the architecture of *any* serverless system has the *fundamental set of components* that define the overall system performance. These components include: a cluster scheduler that adjusts the number of instances of a function based on invocation traffic, the fleet of physical hosts that spawn and run these instances, and a storage service that holds function images and also stores the data transmitted across functions. Using STeLLAR, we analyze three leading serverless platforms (AWS Lambda, Azure Functions, and Google Cloud Functions) by evaluating the performance of each of the three fundamental components individually, with targeted stress-tests and load scenarios. The studies show that the response time – both the median and tail latencies – in the commercial clouds is heavily dominated by *data movement* induced by the two performance bottlenecks common for all three providers we study. The first bottleneck is attributable to moving the function initialization state to the target datacenter node to spawn a new instance of the function. The second bottleneck is related to the lack of efficient support for fast communication across functions, which are restricted by the providers to be stateless.

[1]This was a joint work with equal contributions from myself and an undergraduate student, Theodor Amariucai, who completed his Honours Project at the University of Edinburgh.

1.1.2 vHive Framework for Serverless Experimentation

The results obtained with STeLLAR show that the cold start latency is attributable to data movement because the delays are proportional to the size of the initialization state of the function. Unfortunately, further analysis at the per-component level is cumbersome because commercial cloud providers tend to keep their infrastructure proprietary. Thus, we build *vHive*, an *open-source framework for serverless experimentation* with all required datacenter- and host-level components for end-to-end serverless benchmarking. vHive reuses available open-source serverless components, bridging the gap between a production deployment and an agile research prototyping framework.

With vHive, we construct a prototype, similar to AWS Lambda, that uses Firecracker MicroVM snapshots to spawn function instances. We discover that in the state-of-the-art serverless system the high cold-start delays are due to frequent page faults as the function's state is brought from disk into guest memory one page at a time. Our analysis also reveals that a function accesses a stable working set of pages across different invocations.

1.1.3 Record-and-Prefetch Snapshots

With this insight, we build *Record-and-Prefetch (REAP)*, which reduce the cold-start delays by eliminating the majority of the page faults during launching a new function instance from a snapshot. REAP snapshots is a light-weight software mechanism for serverless hosts, which operates in two phases. In the first "record" phase that occurs only upon the first invocation of a function, the hypervisor records the addresses of the memory pages accessed during processing that invocation, capturing the content of these pages in a working set file enclosed in the function snapshot. Then, upon all further cold invocations of that function, the hypervisor proactively prefetches the entire set from disk into memory, slashing the cold-start delays by 3.7x, on average. We implement REAP entirely in userspace, with minimal changes (<200 LoC) to the hypervisor and no changes to the host operating system.

1.1.4 Expedited Data Transfers

The second key performance bottleneck that we identify with STeLLAR is cross-function data communication. Data movement can dominate the response time of short, sub-second functions, in a data-intensive application, such as video analytics pipeline and machine learning. The stateless and ephemeral nature of function instances prevents sharing function state across different functions that comprise application logic. Inter-function communication generally occurs when one function, the *producer*, invokes one or more *consumer* functions in the workflow passing inputs to them. Crucially, instances of the consumer functions are not known by the producer at invocation time because they are selected by the cloud provider's load balancer and autoscaler components. Hence, functions resort to communicating via an intermediate storage service, which requires the producer function to first store the data, then invoke the consumer, and subsequently have the consumer retrieve the data from storage. The indirection through a storage layer avoids payload size limitation in inline transfers, but also introduces significant latency overheads and adds the cost of using an external storage service.

To address this problem, we introduce Expedited Data Transfers (XDT), an API-preserving high-performance data communication method for serverless that enables *direct function-to-function transfers* in a manner fully compatible with the existing autoscaling infrastructure. With XDT, a trusted component of the sender function buffers the payload in its memory and sends a secure reference to the receiver, selected by the load balancer and autoscaler. Using this reference, the receiver pulls the transmitted data directly from the sender's memory. We prototype our system in vHive, showing that XDT delivers 1.8-12.3$\times$ higher transmission bandwidth compared to AWS S3. As a result, XDT accelerates three real-world serverless applications by 1.1-2.7$\times$, while avoiding the cost of storage.

1.2 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2** provides the background knowledge on serverless cloud architecture, its key underlying technologies and mechanisms, and an overview of serverless workload characteristics.
- **Chapter 3** introduces the *Serverless Tail Latency AnalyzeR (STeLLAR)* framework and the characterization results obtained from analyzing leading commercial clouds.
- **Chapter 4** presents the *vHive* framework and demonstrates its effectiveness by analyzing the root causes of the lengthy cold-start delays in a state-of-the-art serverless system.

- **Chapter 5** describes the *Record-and-Prefetch (REAP)* snapshots technique that we introduce to reduce the cold-start delays when launching a new instance of a serverless function.
- **Chapter 6** introduces *Expedited Data Transfers (XDT)*, a high-speed API-preserving communication method that integrates seamlessly with the existing autoscaling infrastructure that underpins cloud deployments.
- **Chapter 7** concludes the thesis, summarizing the key contributions and outlining the directions for future work.

Chapter 2

Background and Related Work

In this chapter, we discuss the necessary the key concepts and the state-of-the-art in serverless cloud architecture. We start by providing an overview of a serverless cloud, its workload characteristics, and serverless autoscaling infrastructure. Then, we describe the state-of-the-art approaches and system challenges for designing a fast and resource-efficient serverless cloud.

2.1 Serverless Computing Overview

Serverless has emerged as an important cloud computing paradigm in which cloud service developers organize their application logic as a series of *stateless* functions and offload the infrastructure management and resources (auto)scaling entirely to the cloud providers. This programming model is often referred to as *Function-as-a-Service* (FaaS). To implement a serverless application, developers combine multiple functions in the form of HTTP servers with corresponding user-defined handlers. Within an application, functions are connected in a *workflow*, in which they invoke and communicate with each other.

In addition to the lower time-to-market advantage, serverless clouds offer the flexible, pay-as-you-go pricing model, where the provider charges the application developers based on the cloud resources consumed by their functions (e.g., allocated memory and CPU time). Note that the developers are not charged for the delays and extra resource usage imposed by the provider infrastructure. Hence, low infrastructure-related delays and the "autoscaling" ability of the infrastructure to dynamically adjust the cloud resources allocated to the functions is essential for the cloud to be economically feasible.

To adjust the resources allocated to functions, cloud providers scale the number of *instances* of these functions on demand, based on the invocation traffic in front of each of the active instances of the function. The autoscaling requirement leads to the following design decisions fundamental to serverless functions and their instances. First, to facilitate scaling, functions are *stateless* (i.e., they cannot hold or share data) and their instances are identical so that any of these instances can process any invocation of the function it implements. Second, function instances are *ephemeral* and can be created and torn down at any point in time, based on the decisions the cloud infrastructure makes in response to load changes. As a result, this stateless and ephemeral nature of functions prevents direct communication across functions because any of the functions may have an arbitrary number of active instances at any moment.

The efficiency of the autoscaling infrastructure defines how fast a serverless cloud can react to load changes. Since serverless cloud users pay only for useful processing, the providers try to minimize the number of idle function instances, by tearing them down when not in use after a short period of time, to improve overall cloud resources utilization. As overprovisioning of the number of function instances is economically impractical, the time the infrastructure takes to launch a new instance of a function becomes a critical concern to provide high and predictable performance to serverless applications.

More than 30% of serverless applications [66, 67] are data-intensive, transmitting more than 1MB across functions in an application workflow. However, cross-function communication is endemic due to the stateless nature of serverless functions. This often places inter-function communication on the critical path, performance-wise, necessitating a low-latency high-bandwidth communication substrate to provide high performance for data-intensive applications. The quest for high-performance inter-function communication is complicated by the autoscaling aspect of serverless, since a producer instance in a workflow might not know the consumer until the latter is invoked by the producer. It is only at the invocation point that the cloud infrastructure picks either an existing instance of the consumer function or spawns a new one based on observed load. Thus, any optimizations to the inter-function communication substrate must respect the cloud provider's autoscaling policies.

2.2 Serverless Workload Characteristics

A recent study of Azure Functions in production shows that serverless functions are short-running, invoked infrequently, and function invocations are difficult to predict [133]. Specifically, the Azure study shows that half of the functions complete within 1 second while >90% of functions have runtime below 10 seconds. Another finding is that functions tend to have small memory footprints: >90% of functions allocate less than 300MB of virtual memory. Lastly, 90% of functions are invoked less frequently than once per minute, albeit >96% functions are invoked at least once per week. This application characteristics are corroborated by other studies [55, 56, 66].

Given these characteristics of functions, the providers seek to aggressively co-locate thousands of function instances that share physical hosts to increase utilization of the provider's server fleet [19]. For example, a stated goal for AWS Lambda is deploying 4-8 *thousand* instances on a single host [19, 21, 36].

This high degree of co-location brings several challenges. First, serverless functions run untrusted code provided by untrusted cloud service developers that introduces a challenge for security. Second, serverless platforms aim to be general-purpose, supporting functions written in different programming languages for a standard Linux environment. As a result, most serverless providers use virtualization sandboxes that either run a full-blown guest OS [2, 10, 19, 35, 122] or emulate a Linux environment by intercepting and handling a sandboxed application's system calls in the hypervisor [75].

Another challenge for serverless deployments is that idle function instances occupy server memory. To avoid wasting memory capacity, most serverless providers tend to limit the lifetime of function instances to 8-20 minutes after the last invocation due to the sporadic nature of invocations, deallocating instances after a period of inactivity and starting new instances on demand. Hence, the first invocation after a period of inactivity results in a start-up latency that is commonly referred to as the serverless function *cold-start* delay. In the last few years, high cold-start latencies have become one of the central problems in serverless computing and one of the key metrics for evaluating serverless providers [134, 137].

2.3 Serverless Autoscaling Infrastructure

In serverless, the cloud provider is responsible for dynamically scaling the number of instances of each function based on instantaneous load changes. We describe the operation of a serverless autoscaling infrastructure (Fig. 2.1) using the Knative [11] terminology, since it resembles production clouds, mostly closely AWS [19].

The autoscaling infrastructure of serverless needs to satisfy two goals. First, new instances can be requested at any point in time in response to a change in the traffic

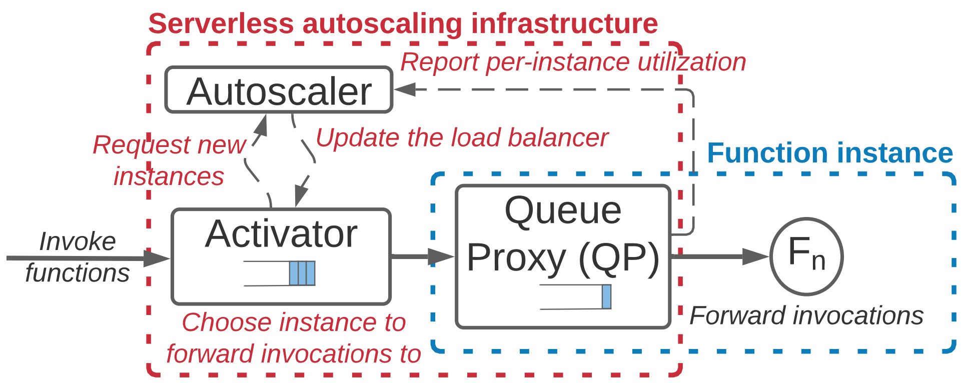


Figure 2.1: Operation of serverless autoscaling infrastructure.

while idle instances should be recycled to free up the cluster resources. Second, the load should be balanced across all active instances of a function at any point in time.

To monitor the load of the currently active instances, cloud infrastructure must timely collect utilization metrics from each of the active function instances and timely react with new instance placement decisions. To implement this, serverless infrastructure depends on two main components. Each function invocation traverses a provider-managed *queue-proxy* component, which is in charge of forwarding incoming requests to the function instance with which the queue proxy is co-located. Queue proxy also collects and reports utilization metrics of that instance to the *autoscaler* control plane component. The *autoscaler* is the knowledge base about the current load in front of each of the active instances.

To balance the load among all active instances of a function, serverless clouds employ a *load balancer* whose job is to steer requests to one of the instances. Every function invocation must traverse the load balancer. Referred to as *activator* in Knative it is a load-aware L7 load balancer. The activator periodically receives updates from the autoscaler, regarding the active instances and their current load. If there is an incoming request for a function and there are no active instances available or all of them are busy, the activator needs to request new instances of that function from the autoscaler. The autoscaler makes a placement decision and spawns a new function instance while the activator buffers the pending function invocation request. Once the new instances are up, the activator steers the buffered invocations to their corresponding function instances via their corresponding queue proxies.

This triplet of components, namely the queue proxy, the autoscaler, and the load balancer, work together to enable autoscaling of serverless functions. The rest of the serverless system is designed around this triplet to deliver seamless scalability to serverless application developers and efficient resource usage to cloud providers.

2.4 Challenges for Serverless Cloud Architecture

Next, we discuss the challenges that are inherent to the autoscaling infrastructure of serverless clouds, namely the cold-start delays of spawning new instances of functions upon demand and data communication across serverless functions.

2.4.1 Function Cold Starts

The delays related to launching new instances of functions depend on the underlying technologies used for isolating the user-provided code of serverless functions. [64, 19]. In this section, we describe the key state-of-the-art technologies for function code isolation, which the providers tailor to the fine-grain, stateless nature of serverless functions (§2.2).

2.4.1.1 Hypervisor Specialization

Leading serverless vendors, including Amazon Lambda, Azure Functions, Google Cloud Functions, and Alibaba Cloud, choose virtual machines (VMs) as their sandbox technology in order to deliver security and isolation in a multi-tenant environment. Although historically virtualization is known to come with significant overheads [128], recent works in hypervisor specialization, including Firecracker [19] and Cloud Hypervisor [2], show that virtual machines can offer competitive performance as compared to native execution (e.g., Docker containers), even for the cold-start delays.

Firecracker is a recently introduced hypervisor with a minimal emulation layer, supporting just a single *virtio* network device type and a single block device type, and relying on the host OS for scheduling and resource management [19]. This lightweight design allows Firecracker to slash VM boot time to 125ms and reduces the hypervisor memory footprint to 3MB [19, 21]. However, we measure that booting a Firecracker VM within production-grade frameworks, such as Containerd [50] or OpenNebula [122], takes 700-1300ms since their booting process is more complex, e.g., it includes mounting an additional virtual block device that contains a containerized function image [4, 131]. Finally, the process inside the VM, which receives the function invocation in the form of an RPC, takes up to several seconds to bootstrap before it is able to invoke the user-provided function, which may have its own initialization phase [64]. Together these delays – which arise on the critical path of function invocation – significantly degrade the end-to-end execution time of a function.

2.4.1.2 Virtual Machine Snapshots

To reduce cold-start delays, researchers have proposed a number of VM *snapshotting* techniques [5, 64, 75]. Snapshotting captures the current state of a VM, including the state of the virtual machine monitor (VMM) and the guest-physical memory contents, and store it as files on disk. Using snapshots, the host orchestrator (e.g., Containerd [50]) can capture the state of a function instance that has been fully booted and is ready to receive and execute a function invocation. When a request for a function without a running instance but with an existing snapshot arrives, the orchestrator can quickly create a new function instance from the corresponding snapshot. Once loading finishes, this instance is ready to process the incoming request, thus eliminating the high cold-boot latency.

Snapshots are attractive because they require no main memory during the periods of a function's inactivity and reduce cold-start delays. The snapshots of function instances can be stored in local storage (e.g., SSD) or in a remote storage (e.g., disaggregated storage service).

The state-of-the-art academic work on function snapshotting, Catalyzer [64], showed that snapshot-based restoration in the context of gVisor [75] virtualization technology can be performed in 10s-100s of milliseconds. To achieve such a short start-up time, Catalyzer minimizes the amount of processing on the critical path of loading a VM from a snapshot. First, Catalyzer stores the minimum amount of snapshot state that is necessary to resume VM execution de-serialized to facilitate VM loading. After that, Catalyzer maps the plain guest-physical memory file as a file-backed virtual memory region and resumes VM execution. Crucially, the guest-physical memory of the VM is not populated with memory contents, which reside on disk, when the user code of the function starts running. As a result, each access to a yet-untouched page raises a page fault. These page faults occur on the critical path of function execution and may significantly increase the runtime cost of a function loaded from a snapshot.

2.4.2 Communication Methods in Serverless

Any meaningful serverless application must combine multiple stateless functions that communicate with each other. The resulting inter-function communication can significantly affect the performance of the entire serverless application. We next discuss the various inter-function communication mechanisms, including solutions deployed in production in today's clouds and proposals from the academic community.

2.4.2.1 Production Methods

Serverless clouds, e.g., AWS Lambda or Google Cloud Functions, provide two methods of data transfers: either inline as part of the invocation HTTP request, or via a third party storage service, i.e. AWS S3 and Google Cloud Storage.

Inline transfers do not require external storage and can thus provide low latency, but pose limitations on the amount of data that can be transferred between function instances. Because invocation requests travel via the serverless autoscaling infrastructure, providers tend to restrict the maximum aggregate size of inlined payloads to reduce the pressure on the load balancer services, which are shared across the datacenter. For instance, in AWS Lambda, inline transfers are limited to 6MB and 256KB per HTTP request/response for synchronous and asynchronous invocation [26], respectively.

In contrast to inline transfers, *through-storage* transfers can be used for arbitrarily large objects but require the use of an external storage service. For example, in AWS, a function A, which needs to pass a large object as an input to function B would save the object in an S3 bucket and pass the corresponding S3 key to B to then retrieve the object. Problematically, the use of a storage service adds latency in the critical path of communication and incurs additional financial costs to the developer due to the use of a storage service.

2.4.2.2 Research Proposals

To overcome the limitations of inter-function communication methods used in production clouds, recent works have considered two alternative strategies. The first strategy focuses on improving the performance of data transfers through the use of tiered storage. For example, Locus [125] uses different storage tiers for specific purposes, namely Redis for shuffling and S3 for cold storage. Pocket [90] and SONIC [103] employ a similar idea and develop a control-plane solution to multiplex different storage services based on inferred application needs. Faa$T[129], Cloudburst [136], and OFC [116] propose using key-value stores as a cache for ephemeral data transfers.

While being an improvement over production offerings, approaches that rely on a storage layer for data transfers fundamentally increase system complexity for the provider, impose additional coordination overheads at the application level, and incur latency overheads for storing and retrieving the data.

The second strategy advocated in research papers is to implement direct communication between function instances by exposing IP endpoints to functions. The unmediated direct IP communication in serverless introduces several problems. First, exposing IP addresses of the instances to untrusted user code is a security concern. Indeed, allowing a malicious user to infer serverless cloud network topology can be of help in denial-of-service or side channel attacks. Secondly, communication using static IP addresses prevents the autoscaler from scaling individual functions independently and places the burden of load balancing on the user. Doing so is fundamentally at odds with the autoscaling principle of serverless computing, and is thus highly unattractive for practical usage.

2.5 Summary

To summarize, serverless has emerged as a popular programming paradigm and cloud architecture. In serverless, application developers compose an application as a workflow that comprises several stateless functions, leaving the actual cloud resource management to the cloud providers. The providers deploy autoscaling infrastructure that monitors the load on each function in a workflow, spawning new instances of each function (or tearing down the idle instances) whenever necessary, charging the application developers only for the resources their functions consume. To make serverless computing economically practical while delivering sufficiently high performance, the providers strive to make autoscaling fast and resource-efficient. It is essential to characterize the delays imposed by the complex distributed serverless infrastructure, featuring a deep software stack. Then, the providers need to minimize the delays imposed by the infrastructure, in particular, the ones imposed by launching new function instances and cross-function data communication.

Chapter 3

Serverless Cloud Characterization

3.1 Introduction

Online services have stringent performance demands, with even slight response-time hiccups adversely impacting revenue [57, 49]. Hence, providing not only a low average response time but also a steady tail latency is crucial for cloud providers' commercial success [57]. The question we ask in this chapter is what level of performance predictability do industry-leading serverless providers offer? Answering this question requires a benchmarking tool for serverless deployments that can precisely measure latency across a span of load levels, serverless deployment scenarios, and cloud providers.

While several serverless benchmarking tools exist, we find that they all come with significant drawbacks. Prior works have characterized the throughput, latency, and application characteristics of several serverless applications in different serverless clouds; however, these works lack comprehensive tail latency analysis [152, 51, 77, 104, 143, 132]. These works also do not study the underlying factors that are responsible for the long tail effects, the one exception being function cold starts, which have been shown to contribute significantly to end-to-end latency in a serverless setting [19, 41, 143].

In this work, we introduce STeLLAR[1], an open-source provider-agnostic benchmarking framework for serverless systems' performance analysis, both end-to-end and per-component. To the best of our knowledge, our framework is the first to address the lack of a toolchain for tail-latency analysis in serverless computing. STeLLAR features a provider-agnostic design that is highly configurable, allowing users to model various aspects of load scenarios and serverless applications (e.g., image size, execution time),

[1]STeLLAR stands for Serverless Tail-Latency Analyzer. The source code is available at `https://github.com/ease-lab/STeLLAR`.

and to quantify their implications on the tail latency. Beyond end-to-end benchmarking, the framework supports user-code instrumentation, allowing the accurate measurement of latency contributions from different cloud infrastructure components (e.g., storage accesses within a cross-function data transfer) with minimal instrumentation effort.

Using STeLLAR, we study the serverless offerings of three leading cloud providers, namely AWS Lambda, Google Cloud Functions, and Azure Functions. We configure STeLLAR to pinpoint the inherent causes of latency variability inside cloud infrastructure components, including function instances, storage, and the cluster scheduler. With STeLLAR, we also assess the delays induced by data communication and bursty traffic and their impact on the tail latency.

Our analysis reveals that storage accesses and bursty function invocations are the key factors that cause latency variability in today's serverless systems. Storage accesses include the retrieval of function images during the function instance start-up as well as inter-function data communication that happens via a storage service. Bursty traffic stresses the serverless infrastructure by necessitating rapidly scaling up the number of function instances, thus causing a significant increase in both median and tail latency. We also find that the scheduling policy, specifically whether multiple invocations may queue at a single function instance, can significantly increase request completion time by up to two orders of magnitude, particularly for functions with long execution times. Our analysis also reveals factors that, somewhat surprisingly, contribute little to latency variability; one such factor is the choice of language runtime.

3.2 Motivation

3.2.1 A Lifecycle of a Function Invocation

We describe the serverless infrastructure organization (Fig. 3.1), summarizing available information about the leading serverless provider, AWS Lambda [19], and the state-of-the-art open-source research framework for serverless experimentation, vHive [143]. First, a function invocation, e.g., triggered by an external source like a click, arrives as an RPC or HTTP request at one of the servers of a scale-out *front-end* fleet that authenticates this request and its origin ①. The request is then forwarded to the *load balancer* that routes invocations to physical hosts, called *Workers*, that have instances

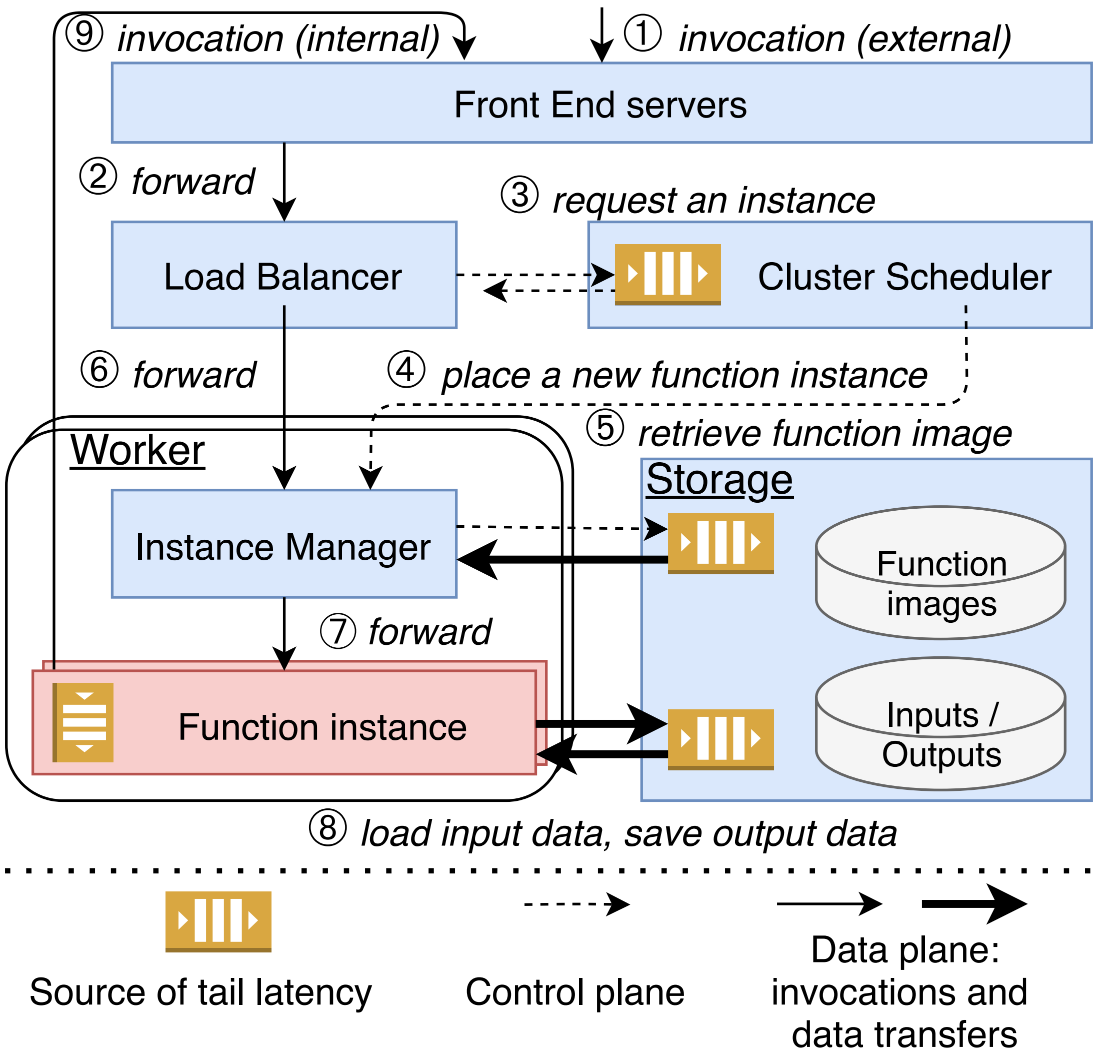


Figure 3.1: Serverless infrastructure overview.

of the function currently running ②.[2] Request routing is based on the load in front of currently active instances.

If all function instances are busy upon an invocation arrival, the load balancer buffers the invocation while asking the *cluster scheduler* to spawn a new instance for the function ③. The scheduler keeps track of the entire cluster resource utilization, which informs decisions regarding function instance placement. Once the scheduler chooses a Worker to run a new function instance, it contacts the Worker's *instance manager* asking to launch a new instance ④.[3] The instance manager retrieves the necessary function state, e.g., a Docker image or an archive with sources, from a storage service and starts the instance ⑤. Note that the instance manager also acts as a part of the invocation data plane, terminating connections to the load balancer and the function instances on the Worker.

When a new instance of the function is ready, the instance manager informs the load balancer, which can then steer the invocation to the instance manager ⑥, which relays it to the instance ⑦. The function then performs language runtime initialization, after which the user-provided code of the function might retrieve the function invocation's inputs, e.g., the output produced by the corresponding caller function ⑧. Finally, the function processes the invocation. During processing, a function may call other functions with inputs that can be transferred inline inside the callee's invocation RPC arguments or by saving the input data in storage, which is required for larger payloads. These *internal* function invocations also need to traverse the front-end and/or the load balancer ⑨, effectively repeating the whole aforementioned procedure.

3.2.2 Sources of Tail Latency in Serverless Clouds

Serverless infrastructures aim to deliver a high quality of service to the majority of cloud application users. Specifically, similarly to conventional clouds, serverless infrastructures strive to minimize tail latency. Providing tail latency guarantees is a hard task for serverless clouds while delivering continuous scaling of function instances and given the necessity of running these instances on any node in a serverless cluster.

Fig. 3.1 shows the components that could potentially become sources of tail latency. These are: the function instances themselves, the storage services used by both the instance manager and the user code, and the cluster scheduler. For each of these

[2]The load-balancer component is referred to as Worker Manager in AWS Lambda [19] and Activator in Knative [11].

[3]The instance manager is referred to as MicroVM Manager in AWS Lambda [19].

components, we identify low-level application characteristics and scenarios that could hurt tail latency.

First, spawning new function instances induces a significant delay, often referred to as a cold start delay in the literature [143, 64, 133]. The key factors that contribute to cold start delays include the language runtime (chosen by the application developer), the provider's sandbox technology, and the size of a function image. These factors impact the cold start delays not only directly but also indirectly by interacting with various infrastructure components, like storage services and network switches. For example, interpreted languages, like Python, are known to have higher startup delays compared to compiled languages, like Go. Also, providers use different sandboxes for function instances; e.g., MicroVMs in AWS Lambda [19] and Google Cloud Functions [75], whereas Azure Functions run in containers atop of regular VMs [100]. Finally, large function images can take non-negligible amounts of time when retrieved from storage. Function images are usually resident in low-cost storage that is not optimized for low latency access since the majority of functions are invoked once per hour or less [133].

Second, the functions that transfer large payloads experience delays induced by the involved infrastructure components. Functions can perform data transfers by embedding their payloads inside the invocation RPC, albeit sizes are restricted to 256KB-10MB [26, 74]. For transferring larger payloads, functions have to resort to storage services suffering from the tail-adverse effects of cost-optimized services, similarly to functions that have large function images. Hence, functions that sporadically transfer large amounts of data – both inline and via storage – may encounter significant tail latency bloat.

The cluster scheduler is another important source of tail latency as prior work shows that function invocation traffic can be bursty [146]. Serverless schedulers attempt to right-size the number of active function instances by quickly reacting to changes in the invocation traffic. Prior work showed that most functions are short-running [133], which has the effect of placing a high load on the cluster scheduler which must cope with a flurry of scheduling decisions at small time intervals. One important aspect of the scheduler is the choice of the policy that deals with whether to steer multiple requests in a burst to an existing warm instance or spawn new instances. The trade-off is between inducing queuing at a warm instance or incurring long cold-start delays to avoid queuing.

Take-away: a comprehensive measurement framework for tail latency analysis must stress all the relevant infrastructure components and technologies to pinpoint, assess and compare all tail latency contributors for a given provider.

3.3 STeLLAR Design

This work introduces STeLLAR, an open-source, provider-agnostic framework for serverless cloud benchmarking that enables systems researchers and practitioners to conduct comprehensive performance analysis. STeLLAR is highly configurable, allowing users to model specific load scenarios and selectively stressing distinct cloud infrastructure components. For instance, as we show in this work, STeLLAR not only helps assess the implications of the language runtime on the function startup time, like prior work [86, 51, 121, 20], but can selectively stress the cluster scheduler by steering frequent bursts of invocations to a set of functions. Also, STeLLAR can stress the storage layer by invoking functions with large image sizes and by configuring data transfer sizes across chains of functions. We envision STeLLAR's users to be able to configure the framework to model other scenarios and stress tests. STeLLAR uses a robust performance measurement methodology (see §3.4 for details), tailored for tail latency analysis, supporting both end-to-end and localized in-depth performance studies.

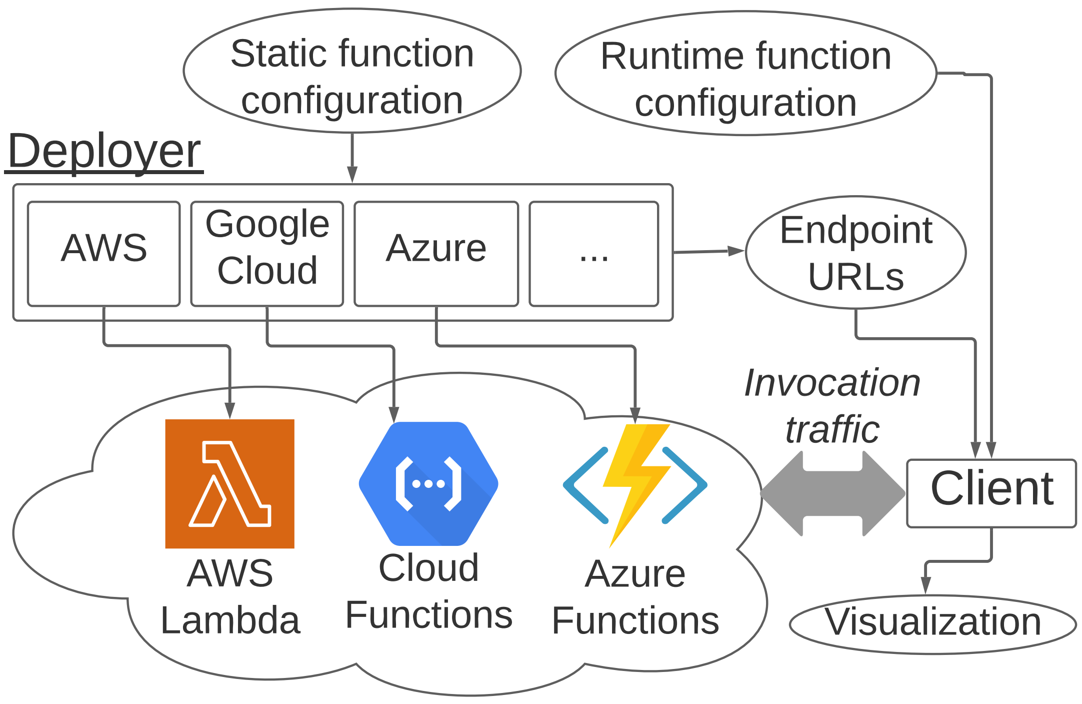


Figure 3.2: STeLLAR architecture overview.

STeLLAR architecture, shown in Fig. 3.2, comprises of two main components, namely *deployer* and *client*. The deployer features a set of provider-specific plugins, each of which are responsible for deploying functions in the corresponding provider's cloud, using a programmatic interface offered by the providers. The deployer's logic is configured via a file with a *static function configuration* that abstracts away the details and terminology of various providers. Using this configuration file, STeLLAR users can define a wide range of static function parameters for each of the deployed functions, as follows:

- Function deployment methods, namely with a ZIP archive and with a Docker container;
- Maximum memory size of the deployed function's instances;
- A number of identical replicas of the function. This is particularly useful to accelerate cold-start latency measurements by invoking the replicas in parallel instead of invoking a single function once in a long keep-alive period, each time waiting for the providers to tear down an idle instance of the function.

Provided with the static configuration file, the deployer automatically configures and deploys the requested set of functions, producing a file that contains a set of *endpoint URLs*, each of which corresponds to a single function and is assigned by the appropriate provider. The static function configuration file may define a function handler's code, a maximum memory size of a function instance, the effective image size, as well the provider the target availability zone. For some providers, the deployer supports several deployment methods, namely ZIP archive-based deployment, which is common to all providers, and a more recent Docker-based deployment option available in some serverless clouds, e.g., AWS Lambda [31]. The users can configure the effective function image size by instructing the deployer to add a random-content file to the corresponding ZIP archive or Docker image.

After configuring and uploading a set of functions using the deployer, STeLLAR can drive the load to these functions, measuring their response time and visualizing the measurements with a set of plotting utilities. The client is provider-agnostic, generating function invocation traffic as HTTP requests to the endpoints that are defined in the file produced by the deployer component. The clients invokes functions from the file with the endpoints' URLs in a round-robin fashion, calling them one after another according to the specified IAT. To send a request to a deployed function, the client spawns a goroutine that sends an HTTP request to the function's URL, blocking till the function's response arrives. For each of the requests, its goroutine measures the latency between the time when the request was issued and the time when the corresponding response was received. The measurements are then aggregated in a single file for further data analysis and visualization.

The client supports further customization with a *runtime configuration file* where users may specify a number of runtime parameters:

- The file may contain an arbitrary mix of deployed functions (every function replica is treated as a separate function in this file);
- Inter-arrival time (IAT) distribution of the function invocation traffic, with a fixed, stochastic, or bursty distribution;
- Function execution time;
- Chain length that is the number of functions in a *chain*, where each preceding function invokes its following function while transferring a payload of a configurable size.
- The type of data transfers if the chain is longer than a single function, namely inline transfers and transfers via a storage service (AWS S3, Google Storage, and Azure Blob Storage are supported).

STeLLAR lets the users specify the IAT distribution (e.g., round-robin across all deployed functions) along with the number of requests issued in each step, i.e., the burst size, which is essential to evaluate cloud infrastructure efficiency in the presence of bursty traffic. Also, STeLLAR users can specify the execution time of a function with a busy-spin loop of a configurable duration. Other parameters define the data transfer behavior across chains of functions, where each function calls the next function in the chain and waits for its response before returning. The users can specify the transport for data transfers (inline arguments inside the invocation HTTP requests or a cloud storage service) and the length of function chains.

STeLLAR supports intra-function instrumentation, by adding calls into Go's native Time module. For example, to capture the data transfer delays, we add a timestamp in the producer function before saving a payload to a storage service (e.g., AWS S3) and another timestamp in the consumer function after retrieving the payload from the storage. The functions' code concatenates these timestamps and passes them to

STeLLAR's client as a string. The overhead of this instrumentation is sub-microsecond, as Go relies on Linux' `clock_gettime()` whose overhead has been measured to be within 30ns [78, 48].

Finally, STeLLAR can visualize latency measurements as a cumulative distribution function or as latency percentiles as a function of one of the serverless-function parameters (e.g., the payload size in a data transfer, function image size). The ability to collect both the end-to-end time and the internal timestamps, e.g., for measuring the data transfer time, allows users to cross-validate their measurements.

3.4 Methodology

We use STeLLAR to characterize three leading serverless cloud offerings, namely AWS Lambda, Google Cloud Functions, and Azure Functions. In the rest of this section, we discuss different aspects of performance, approaches to function deployment, the configuration of STeLLAR, and the metrics that we focus on.

Factor Analysis Vectors

To assess the impact of each of the identified tail-latency sources, we conduct studies along the following four vectors. First, we evaluate the response time of warm and cold functions under a non-bursty load (i.e., allowing no more than a single outstanding request to each function).[4] Second, we assess the cold function delays that appear on the physical node that hosts these functions, varying the language runtime and the functions' image size. Third, we study the data communication delays for chained functions, where one function transmits a payload of varied size to the second function; we consider two data transfer methods: inline (i.e., inside the HTTP request) and via a cloud storage service. Lastly, we investigate the behavior of serverless clouds in the presence of bursty function invocations, varying the number of requests in a burst (further referred to as the burst size) and their inter-arrival time.

Function Deployment Configuration

We deploy the functions in datacenters located near the western coast of the USA in close proximity to the CloudLab Utah datacenter, where STeLLAR runs. The functions

[4]Here, we call a function *warm* if it has at least one instance online and idle upon a request's arrival, otherwise we refer to the function as a *cold* function.

are configured with the maximum memory sizes, which are 1.5GB for Azure and 2GB for AWS and Google for a single CPU core per function instance [26, 74, 34]. These high-memory configurations are not subject to CPU throttling applied by the providers to low-memory instances.

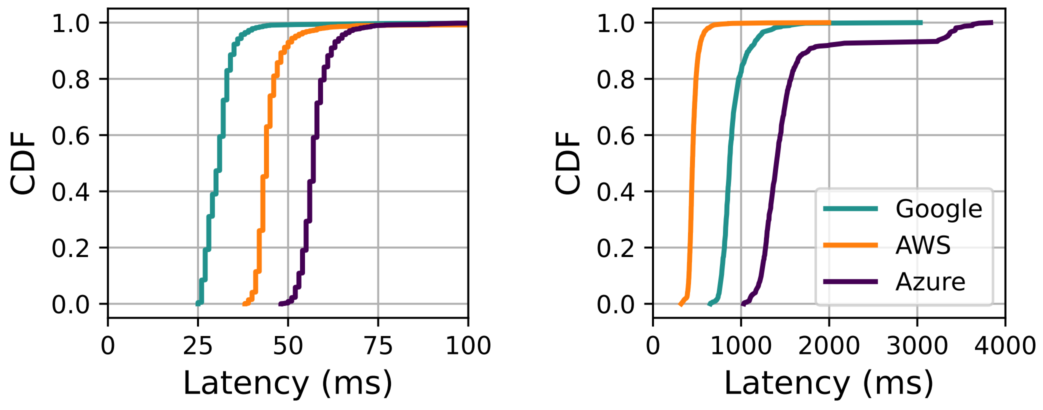
Unless specified otherwise, we deploy all functions using the ZIP-based deployment method, which is supported by all studied providers. We deploy Python 3 functions for all experiments except the function image size (§3.5.2.2) and the data transfer (§3.5.3) experiments. In those experiments, we use Golang functions to minimize the image size and increase the accuracy of the internal timestamp measurements required in the data transfer experiment. To measure the data transfer time in a chain of two functions, the first function records an initial timestamp before storing it in the transferred payload. The payload is then sent using the second function's invocation request or cloud storage. Finally, the second function is invoked, after which it loads the data from the request or storage and then records a second timestamp. STeLLAR's client computes the effective transfer time by subtracting these timestamps. All functions return immediately, unless specified otherwise.

STeLLAR Configuration

We run the STeLLAR client on an xl170 node in CloudLab Utah datacenter which features a 10-core Intel Broadwell CPU with 64GB DRAM and a 25Gb NIC [65]. The propagation delays between the STeLLAR client deployment and the AWS, Google, and Azure datacenters in the US West region, as measured by the Linux `ping` utility, are 26, 14, and 32ms, respectively.

In all experiments, unless stated otherwise, functions return immediately with no computational phase. To study warm function invocations, the client invokes each function with a 3-second inter-arrival time (IAT), further referred to as *short* IAT, that statistically ensures that at least one function instance stays alive. To evaluate cold function invocations, the client invokes each function with a 15-min IAT, further referred to as *long* IAT, which was chosen so that the providers shut down idle instances with a likelihood of over 50%.[5] We configure the STeLLAR client to invoke each function either with a single request or by issuing a burst of requests simultaneously. A serverless request completes in >20ms, as observed by the client, which means that requests in the same burst create negligible client-side queuing. For each evaluated configuration,

[5]We found that AWS Lambda always shuts down idle function instances after 10 minutes of inactivity, which allowed us to speed up experiments on AWS by issuing requests with a long IAT of 10 minutes.



(a) Short IAT

(b) Long IAT

Figure 3.3: Latency distributions for functions invoked with short and long inter-arrival times.

STeLLAR collects 3000 latency samples (each request in a burst is one measurement).

In all experiments, to speed up the measurements, we deploy a set of identical independent functions that the client invokes in a round-robin fashion, ensuring no client-side contention. For example, to benchmark cold functions, we deploy over 100 functions, each of which is invoked with a fixed IAT.

Latency and Bandwidth Metrics

We compare the studied cloud providers using several metrics that include the median response time, the $99^{\text{-th}}$ percentile (further referred to as the tail latency), and the *tail-to-median ratio* (TMR) that we define as the $99^{\text{-th}}$ percentile normalized to the median. Both median and tail latencies are reported as observed by the client, i.e., the latencies include the propagation delays between the client deployment and the target cloud datacenters. The TMR metric acts as a measure of predictability which allows the comparison of response time predictability between different providers. We consider a TMR above 10 potentially problematic from a performance predictability perspective. In the data-communication experiments, we estimate the effective data transmission bandwidth as the payload size divided by the median latency of the transfer.

3.5 Results

3.5.1 Warm Function Invocations

We start by evaluating the response time of functions with warm instances by issuing invocations with a short inter-arrival time (IAT). For this study, at most one invocation to an instance is outstanding at any given time. Fig. 3.3a shows cumulative distribution functions (CDFs) of the response times as observed by the STeLLAR client.

We note that propagation delays to and from the datacenter (§3.4) constitute a significant fraction of the latency for warm invocations. Since the propagation delays to the different providers' datacenters differ, we subtract them to focus on the intra-datacenter latencies. Note that this is the only place in the paper where we do so; the rest of the results (including Fig. 3.3a) include the propagation delays.

With propagation delays subtracted, we find that Google has the lowest median latency (17ms), followed by AWS (18ms), and Azure (25ms). In line with having the lowest median latency, Google also displays a lower tail latency (47ms) compared to the other two providers (74ms for AWS, 75ms for Azure). Despite having the highest median latency, Azure yields the lowest TMR of all (1.3), compared to 1.5 for Google and 1.7 for AWS.

Observation 1. *Invocations of warm functions impose low delays and variability, with a median latency of $\leq$25ms and TMRs $<$2.*

3.5.2 Cold Function Invocations

We perform a factor analysis of cold-start delays, starting from latencies corresponding to baseline invocations followed by an investigation of the effect of function image size, language runtime and function deployment method.

3.5.2.1 Baseline Cold Invocation Latency

We study the response time of cold functions by issuing invocations with the long IAT. As shown in Fig. 3.3b, the median and tail latencies of cold invocations are, respectively, 10-28$\times$ and 9-49$\times$ higher than their warm-start counterparts. We observe that the lowest median latency (448ms) and the lowest latency variability (TMR of 1.5) are delivered by AWS. Google ranks in the middle and displays a median latency of 870ms and a TMR of 1.8. Finally, Azure shows the highest median latency of 1401ms with the highest variability (TMR of 2.6).

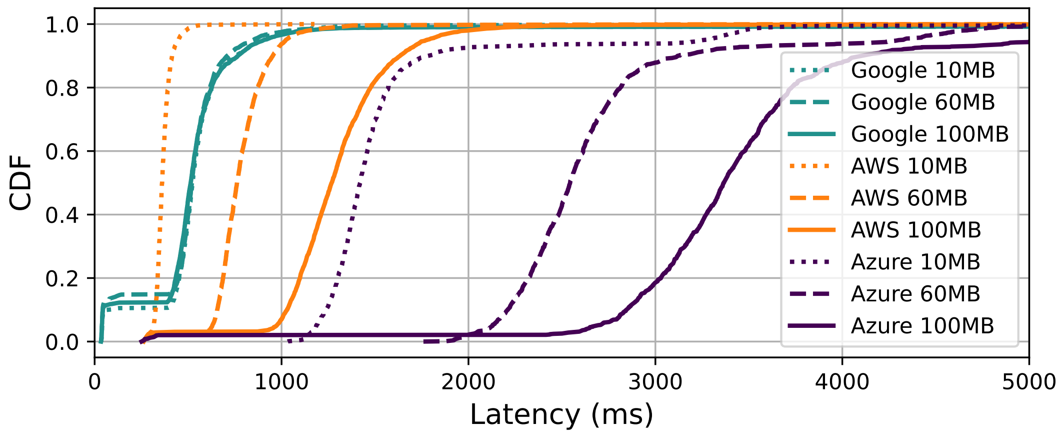


Figure 3.4: Cold-start latency as a function of the extra random-content file added to the function and provider.

3.5.2.2 Impact of Function Image Size

In this experiment, we assess the impact of the image size on the median and tail response times, by adding an extra random-content file to each image. We only consider ZIP-based images as these are supported by all three of our studied cloud providers.

Results are shown in Fig. 3.4. We observe that Google is not sensitive to the image size, with near-identical CDFs for images with added 10MB and 100MB files. It is possible that Google allows retrieving images at much higher network and/or storage bandwidth than its competitors, while masking the image retrieval with other cold-start related operations.

In contrast, AWS show considerable sensitivity to the image size: increasing the added file's size from 10MB to 100MB results in 3.5× and 4.2× increase of the median and the tail latency, respectively. Azure exhibits even higher sensitivity to the image size, with the median and tail latencies increasing by 2.4× and 1.6× when adding 100MB to the image, compared to the response time of the functions with added 10MB. For functions with larger images, the tail latency may reach 2155ms in AWS and 5723ms in Azure. However, for both AWS and Azure, the usage of large function images has a moderate impact on latency variability, with a TMR of 1.7. In contrast to AWS and Azure, Google shows considerably lower sensitivity to varying the image size. Also, for large image sizes, e.g., with a 100MB file, Azure exhibits lower median latencies (approx. 510ms) than its competitors albeit showing higher latency variability, with a TMR of 3.6.

Observation 2. *Invocations of cold functions may impose large delays of up to several*

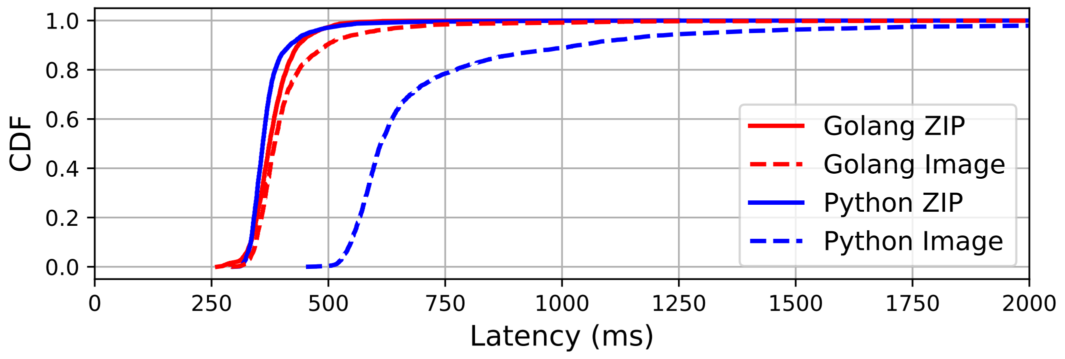


Figure 3.5: Cold-start latency distributions in AWS for different language runtimes and deployment methods.

seconds to the median response time, particularly for functions that have large images. However, variability of cold-starts is moderate, with TMR of <3.6.

3.5.2.3 Impacts of Deployment Method and Language Runtime

We next study the implications of different deployment methods and language runtimes. Deployment methods refer to how a developer packages and deploys their functions, which also affects the way in which serverless infrastructures store and load a function image when an instance is cold booted. We study the two deployment methods that are in common use today: (1) ZIP archive, and (2) container-based image. With respect to language runtime, our intent is not to evaluate all possible options, but rather to focus on two fundamental classes of runtimes: compiled and interpreted. To that end, we study functions written in Python 3 (interpreted) and Golang 1.13 (compiled) deployed via ZIP and container-based images. This study is performed exclusively using AWS Lambda because, at the time of this paper's submission, Google Cloud Functions did not support container-based deployment and Azure Functions did not support Golang.

The results of the study are shown in Fig. 3.5, from which we draw three observations. First, for ZIP-based deployment, both Golang and Python's CDFs nearly overlap, showing median and tail latencies of 360ms and 570ms, respectively. This result demonstrates that the choice of a language runtime has negligible implications for cold-start delays. The result is surprising in that it contradicts academic works that showed that the choice of a runtime impacts cold-start latency [64, 152, 20]. However, these academic works did not study production clouds, instead focusing on in-house or open-source systems. Thus, we hypothesize that academic systems may lack important optimization employed by production systems, such as having a pool of warm generic function instances [19]. We call attention to this issue as one that requires further study.

Our second observation is that the latency CDFs of Python and Golang runtimes with container-based deployment differ significantly. Python functions show considerably higher median and tail latency of 612ms and 2882ms, respectively, whereas for Golang, the latency CDF of container-based deployment is close to that of ZIP-based one. One possible explanation of this phenomenon is that a Golang program is compiled as a static binary, suggesting that both ZIP and container image comprise the same binaries that are likely to be stored in the same storage service. Meanwhile, for Python, a container-based deployment shows higher median and tail latencies, compared to the corresponding ZIP deployment. We attribute this behavior to the fact that Python imports modules dynamically, requiring on-demand accesses to multiple distinct files in the function image. When combined with a container-based deployment method, we hypothesize that this results in multiple accesses to the function image storage, since containers support splintering and on-demand loading of image chunks [107]. The additional accesses to the image store would explain the high cold start time and latency variability for Python container-based deployments.

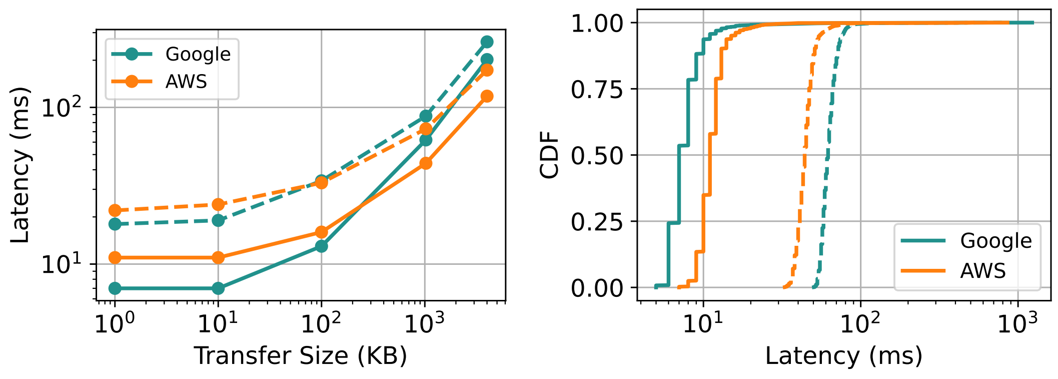
Third, we characterize the latency variability induced by the language runtime and the deployment method selection. We observe that Golang ZIP and Python ZIP-based functions show similar TMRs of 1.5 and 1.7, indicating little impact on the tail latency. Golang container-based deployment has slightly higher variability with a TMR of 2.4. In contrast to Python ZIP deployment, Python container-based functions exhibit much higher latency variability with a TMR of 4.7.

Observation 3. *The choice of the language runtime has low impact on the cold function invocation delay with a <15ms difference between Golang and Python functions' median response time with ZIP deployment. In contrast, the deployment method strongly impacts cold-start delays for functions written in an interpreted language such as Python; compared to ZIP, container-based deployment significantly increases both median and tail latencies by 1.7× and 8.0×, respectively.*

3.5.3 Data Transfer Delays

To study the impact of the data-transfer delays on the overall response time tail latency, we deploy a producer-consumer chain of two functions in AWS and Google.[6] The producer function invokes the consumer with an accompanying payload of a specified

[6]We plan to extend this study to Azure Functions by the time of publication.



(a) Median (solid) and tail (dashed) payload transfer latencies

(b) Latencies for 10KB (solid) and 1MB (dashed) payload transfers

Figure 3.6: Inline data transfer latency as a function of payload size. Note that both axes in (a) and the X-axis in (b) are logarithmic.

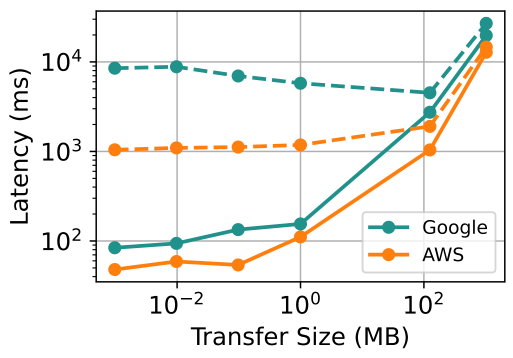
size. The payload is transmitted in one of two ways: inline or via a storage service. We report the latency from the start of the payload transmission, including the consumer function invocation time, to the point when the payload is retrieved by the consumer function.

3.5.3.1 Inline Transfers

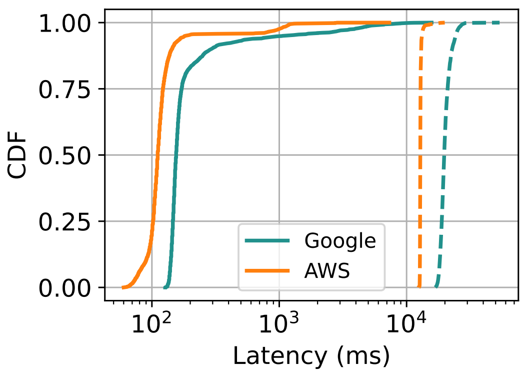
First, we investigate the transmission time of transferring the payloads inline. Note that both providers restrict the maximum HTTP request sizes, and hence the inline payload size, to 6 and 10MB in AWS [26] and Google [74], respectively. For larger data transfers, application developers must resort to storage-based transfers, discussed below.

Fig. 3.6a shows the median and the tail transmission latency of inline transfers. Google delivers the lowest median latency for small payloads (1-10KB), completing the transfer 1.6$\times$ faster than AWS. E.g., a 1KB transfer completes in 7ms in Google vs 11ms in AWS.

In contrast to small-payload transfers, which complete relative quickly, transferring large payloads may take hundreds of milliseconds. For instance, AWS and Google complete a 4MB transfer with median latencies of 124ms and 202ms, and TMRs of 1.4 and 1.3, respectively. For functions that run for less than 10 seconds, which account for >70% of all functions as reported in Azure Functions' trace [133], such data transfer overheads might be prohibitively high.



(a) Median (solid) and tail (dashed) latencies



(b) Latencies for 1MB (solid) and 1GB (dashed) payload transfers

Figure 3.7: Storage-based data transfer latency as a function of payload size. Note that both axes in (a) and the X-axis in (b) are logarithmic.

Next, we compare data transfer time variability in AWS and Google (Fig. 3.6b). For both providers, the variability is low, with TMRs of 1.7 and 1.4, respectively. With such low TMRs, we find that inline transfers have a fairly low impact on tail latency compared to other sources of variability.

Finally, we study the effective bandwidth of inline data transfers, which we compute by dividing the payload size by the observed median transmission time. We find that AWS and Google functions deliver a relatively meager 264Mb/s and 152Mb/s of bandwidth, respectively. This bandwidth is significantly lower than the bandwidth of commodity datacenter network cards (e.g., 10-100Gb/s in non-virtualized AWS EC2 instances [22]).

3.5.3.2 Storage-based Transfers

First, we evaluate the latency and the effective-bandwidth characteristics of storage-based transfers in AWS and Google, sweeping the size of the transmitted payloads from 1KB to 1GB. Fig. 3.7a demonstrates that the lowest median latency is delivered by AWS. For instance, a 1MB payload transfer completes 1.4× faster in AWS than in Google (111ms in AWS vs 155ms in Google).

Second, we investigate the effective transmission bandwidth of storage-based transfers and compare it to the bandwidth that we measure for inline transfers (§3.5.3.1). We observe that storage-based transfers provide significantly larger effective bandwidth than the corresponding inline transfers. For example, 1MB transfers between functions in AWS and Google yield 72Mb/s and 48Mb/s, respectively. The achieved bandwidth is much higher for >100MB payloads, reaching up to 960Mb/s and 408Mb/s for AWS and Google, respectively. Despite the higher bandwidth achieved by large transferred sizes, it is still more than an order of magnitude lower than what a low-end commodity 10Gb NIC can offer.

Finally, we assess how storage transfers contribute to latency variability in serverless. Fig. 3.7b shows that storage-based transfer delays exhibit large tail latency. For instance, when transferring 1MB of data, the tail latency is 1177ms in AWS and 5781ms in Google, with corresponding TMRs of 10.6 and 37.3 in AWS and Google, respectively. In contrast, transferring 1MB inside function invocation requests, i.e., inline, yields much lower TMRs of 1.7 in AWS and 1.4 in Google (§3.5.3.1).

We speculate that the high variability of storage transfers is due to the fact that storage services, by design, optimize for cost rather than performance. With the lack of a fast and cheap communication alternative for large payload transfers, we identify storage as one of the key contributors to the overall response time and performance variability in serverless.

Observation 4. *Storage-based data transfers significantly contribute to both median and tail latencies. E.g., for a 1MB transfer in Google, these delays result in 155ms median and 5774ms tail latencies, yielding a high TMR of 37.3. In contrast, inline transfers are fast and predictable: e.g., for a 1MB transfer in Google, these delays result in 62ms median and 88ms tail latencies with a much lower TMR of 1.4.*

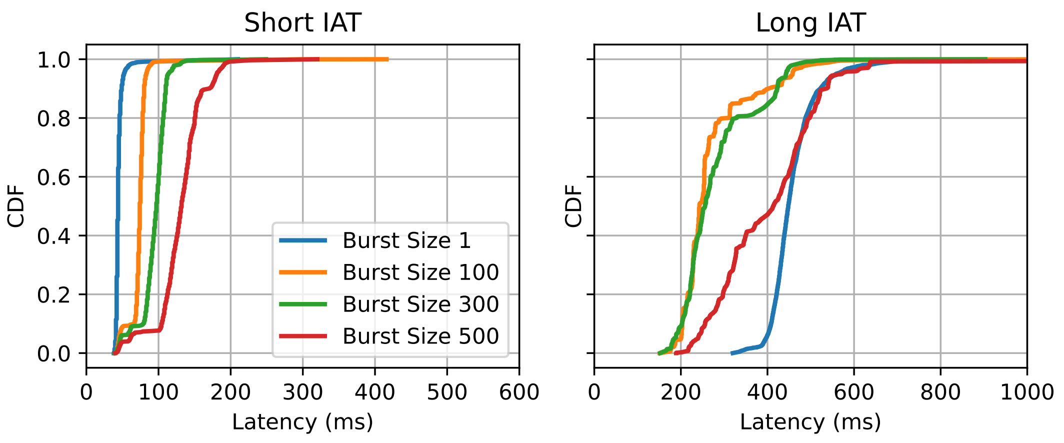
3.5.4 Bursty Invocations

We study the response time of functions in the presence of bursty invocations and assess the impact of the scheduling policy on request completion time.

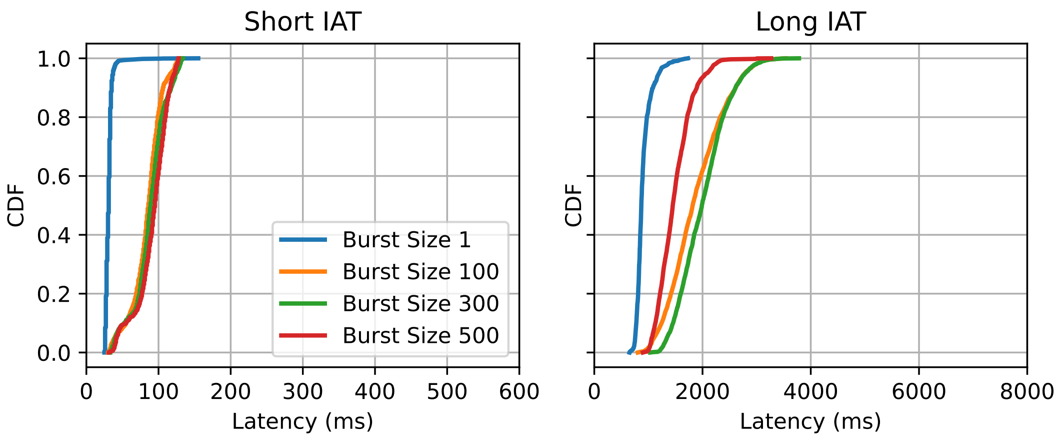
Fig. 3.8 shows the response time for requests arriving in bursts with short and long IATs, corresponding to (mostly) warm and (mostly) cold invocations. We observe that the burst size, i.e., the number of requests sent in a single burst, impacts both median and tail latency characteristics for all three providers. However, different providers exhibit different degrees of sensitivity to the burst size. Note that a burst size of 1 corresponds to a single invocation (Fig. 3.3).

3.5.4.1 Bursty Invocations with Short IAT

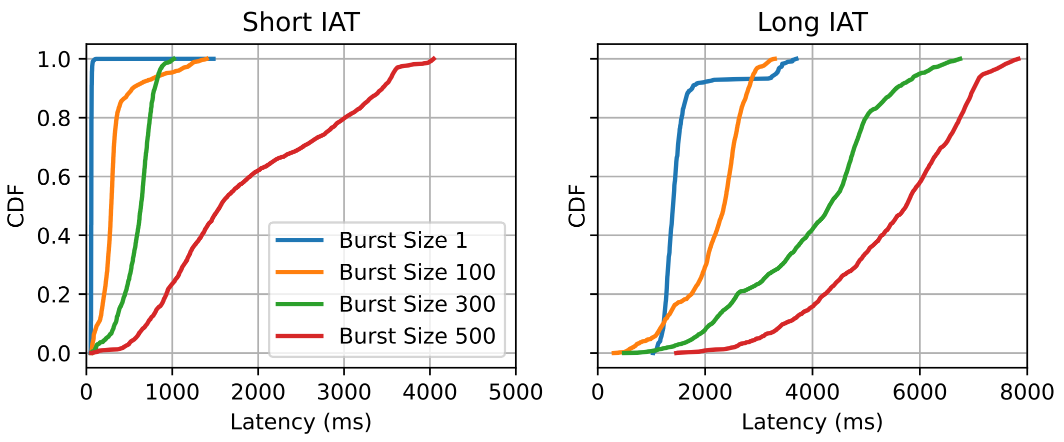
Fig. 3.8 (left subfigures) plot the latency CDFs of the three providers when bursts are issued with short IATs. We observe that all providers exhibit the similar behavior:



(a) AWS Lambda.



(b) Google Cloud Functions.



(c) Azure Functions.

Figure 3.8: Latency CDFs for short and long IATs for different burst sizes. Note that X-axes vary across graphs.

serving larger bursts leads to an increase in both median and tail latencies. Azure displays the highest sensitivity to the burst size: increasing the burst size from 1 to 500 leads to an increase in both median and tail latencies by 33.4× and 98.5×, respectively. Noticeably, Google shows the lowest sensitivity to increasing the burst size from 100 to 500, with the median latencies being within 15ms for different burst sizes (the tail latencies are within 50ms).

We next compare latency variability across the three providers using a burst size of 100 as a base for comparisons. We observe that Google shows the lowest variability, followed by AWS, and Azure with TMRs of 1.7, 6.2, and 7.9, respectively. Increasing the burst size from 100 to 500 results in lower variability for AWS and Azure, with TMRs of 4.4 and 3.9, but slightly goes up for Google, with a TMR of 1.9.

3.5.4.2 Bursty Invocations with Long IAT

Next, we compare invocations in the presence of bursts with long IAT. Results are plotted in the right-hand subfigures of Fig. 3.8. We find that different providers exhibit different behavior as burst size increases.

For AWS, increasing the burst size from 1 to 100 results in 1.8× and 1.3× decrease of the median and tail latencies, respectively. This latency reduction suggests that AWS optimizes retrieval of function images from storage, possibly by employing an in-memory storage-side caching. While increasing the burst size from 100 to 300 requests results in minimal changes, within 12%, in both median and tail latencies in AWS, we observe that when serving a burst of even 500 requests, both median and tail latencies continue to be lower than for an individual request.

Google's median and tail latencies in the presence of bursts are higher than in the presence of individual requests (i.e., burst size of 1). For instance, the median and tail latencies is 870ms and 1567ms for a burst size of 1 vs. 1818ms and 3095ms for a burst size of 100, respectively. Increasing the burst size to 300 results in further increase of both median and tail latencies Interestingly, increasing the burst size to 500 results in a reduction of both median and tail latencies. We hypothesize that this behavior might be attributed to the effects coming from the function image storage subsystem that might adjust aggressiveness of images caching based on load.

One can also see that AWS and Google functions' response time never drops to the range attributable to warm function invocations, i.e., 25-100ms (Fig. 3.3a). This suggests that these providers do not allow multiple requests to queue at an already-executing instance, and, instead, a dedicated instance services each and every request in

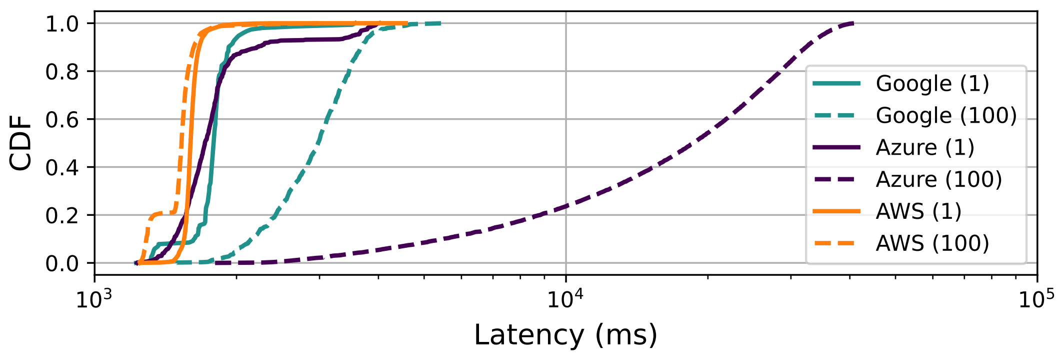


Figure 3.9: Latency CDFs for different request burst sizes, arriving with a long IAT, for the functions with 1-second long execution time.

a burst. This corroborates AWS documentation [24]. Azure exhibits a different behavior, as its CDF suggests that such queuing may occur, albeit limited to a very small fraction of requests (<5%).

Azure functions show that both median and tail latencies significantly increase when increasing the burst size. For instance, increasing the burst size from 1 to 500 increases the median and the tail latencies by 4.1× (i.e., by 4344ms) and 2.1× (i.e., by 4037ms).

Finally, we note that all three providers have low latency variability for bursts with long IAT. For the burst size of 100, AWS shows the highest variability with a TMR of 2.2. Meanwhile, Google and Azure enjoy lower TMRs of 1.7 and 1.4, respectively.

Observation 5. *For bursts arriving with a short IAT, two out of the three providers experience a moderate increase in the median latency by 3.1-3.3× and the tail latency by 4.2-8.4×, compared to to serving individual invocations. The third provider exhibits higher sensitivity with its median and tail latencies increasing by 33.5× and 98.5×, respectively. Meanwhile latency variability is moderate for all providers, with TMRs <7.9.*

Observation 6. *For bursts arriving with a long IAT, all providers show moderate latency variability with TMRs of 1.3-2.6. Despite that, the median and the tail latencies of two out of three providers increase by up to 740ms and 4344ms.*

3.5.4.3 Implications of Scheduling Policy

In this experiment, we study the scheduling policies that different providers apply in the presence of bursty invocations with a long IAT. In contrast to the previous study, where functions responded immediately (i.e., took no time other than to generate the response), here we deploy functions with an execution time of 1 second. Our goal is to understand whether providers allow concurrent requests to queue at an active instance to alleviate the lengthy cold-start delays. We chose the function execution time to be 1 second as it exceeds the median cold-start delays for all providers, as shown in §3.5.2.1. Moreover, the traces released by Microsoft Azure show that 50% of functions run for 1 second, on average [133]. Intuitively, with the function execution time being longer than the cold-start latency, a scheduling policy optimized exclusively for performance would cold-start a new instance for each request in a burst instead of allowing multiple requests to queue at an existing instance.

We perform the experiment with burst sizes of 1 and 100. Results are shown in Fig. 3.9. First, we observe that for non-bursty execution (i.e., a burst size of 1), CDFs for all providers are close to each other, as there is no potential for queuing. For bursty execution, the providers exhibit dramatically different behavior from each other. For instance, for burst sizes of 1 and 100, we observe that AWS demonstrates nearly identical latency CDFs with median and tail latencies of 1598ms and 1865ms, respectively. As both of these latencies are below 2 seconds, it is clear that all requests execute on separate instances, and no request waits for another request, which is in line with the observation we made in §3.5.4.2.

In contrast, Google delivers median and tail latencies of 2978ms and 4595ms, respectively, indicating that up to four requests may queue at one function instance. Meanwhile, Azure demonstrates median and tail latencies of 18637ms and 38545ms, respectively, showing that more than 30% of requests in a burst may be executed by the same instance.

While it is difficult to ascertain that either Google or Azure do, in fact, allow requests to queue at an active instance, the results certainly suggest that. Indeed, doing so would be a sensible policy, particularly for shorter functions, aimed at striking a balance between function execution time and resource utilization in terms of the number of active instances. Both policies (i.e., allowing queuing or not) have pros and cons, which points to a promising optimization space for future research.

Observation 7. *The choice of scheduling policy with respect to whether multiple invocations may queue at a given function instance has dramatic implications on request completion time and resource utilization (i.e., number of active instances). For functions with long execution times, a scheduling policy that allows queuing may increase both median and tail latency by up to two orders of magnitude.*

	AWS		Google		Azure	
Factor	MR	TR	MR	TR	MR	TR
Base warm (§3.5.1)	1	2	1	2	1	1
Base cold (§3.5.2.1)	10	15	28	50	25	64
Image size (§3.5.2.2)	29	49	17	60	59	100
Inline transfer (§3.5.3.1)	1	2	2	3	n/a	n/a
Storage transfer (§3.5.3.2)	3	27	5	187	n/a	n/a
Bursty warm (§3.5.4.1)	2	11	3	5	5	41
Bursty cold (§3.5.4.2)	6	12	59	100	41	58
Bursty long[7](§3.5.4.3)	12	16	64	102	309	619

Table 3.1: *Median to base median (MR)* and *tail to base median (TR)* metrics per studied tail-latency factor across providers. Cells with MR or TR $>$10 highlighted in red.

3.6 Discussion

In this section, we first recap our findings by focusing on key sources of execution time variability induced by the serverless infrastructure. We next discuss variability in actual function execution time by analyzing data from a publicly-available trace of serverless invocations in Microsoft Azure.

3.6.1 Variability due to Serverless Infrastructure

We summarize our findings in Table 3.1. For each of the factors that we study, we compute two metrics, namely *median to base median ratio (MR)* and *tail to base median ratio (TR)*, which normalize the median and tail delays as induced by the corresponding factor to median latency of an individual warm function invocation. This normalization is done separately for each provider, i.e., the reported median or tail latency for a given experiment with a particular provider is normalized to the median latency of a warm invocation on that provider. We consider an MR or TR above 10 to be potentially problematic as it implies a high degree of variability. Such cells are highlighted in red in Table 3.1.

We identify two trends that are common across the studied providers. First, we find storage to be a key source of long tail effects. Indeed, both cold function invocations, which require accessing the function image from storage, and storage-based data

[7]We subtract the 1s function execution time from the measured latencies to account only for infrastructure and queuing delays in order to compute the MR and TR metrics.

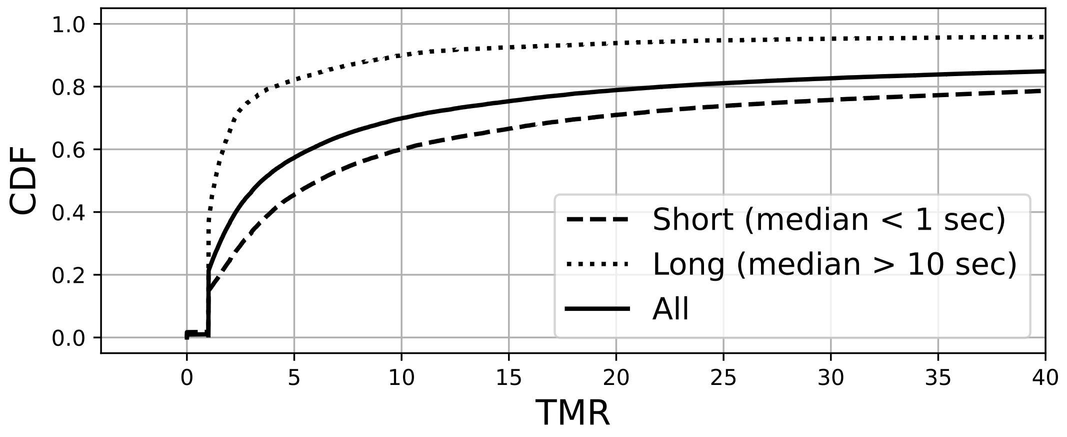


Figure 3.10: Tail-to-median ratio (TMR) CDFs for per-function execution times, as reported in Azure Function's trace [133].

transfers induce high MR (up to 59) and high TR (up to 187). To put these numbers in perspective, a hypothetical warm function with a median execution latency of 20ms would see its median latency skyrocket to 1.18s with MR of 59 and its tail to 3.74s with TR of 187.

The second trend we identify is that all studied providers exhibit high sensitivity to bursty traffic, particularly, when bursts arrive with a long IAT (rows "Bursty cold" and "Bursty long" in Table 3.1). While part of the reason for the resulting high latencies can be attributed to storage accesses for cold invocations, we note that the scheduling policy also seems to play a significant role. For functions with a long execution duration (1s, in our experiments), if requests to a function are allowed to queue at an active instance, we observe MR and TR of 309 and 619, respectively.

3.6.2 Variability in Function Execution Time

We ask the question of how the variability induced by serverless infrastructure compares to the variability in function execution time, i.e., the useful work performed by functions. Given the many options for the choice of implementation language, the numerous ways for breaking up a given functionality into one or more functions, the actual work performed by each function and other effects that determine function execution time, we do not attempt to characterize the execution-time variability on our own. Instead, we use a publicly-available trace from Azure Functions that captures the distribution of function execution times as a collection of percentiles [133], including a $99^{\text{-th}}$ percentile and a median, allowing us to compute the tail-to-mean ratio (TMR) for each function.

For each function, the trace captures the time between the function starting execution until it returns. Even though each function's reported execution time excludes cold-start delays, this measurement may still include some infrastructure delays, e.g., if that function invokes other functions or interacts with a storage service. Hence, the computed TMRs are the *upper bound* for the pure function execution time variability.

Fig. 3.10 shows the CDF of the TMRs for each of the functions in the trace. We find that 70% of all functions have a TMR less than 10, indicating moderate variability in function execution times. However, other functions exhibit significant variability, roughly in the same range is the variability induced by storage-based transfers which have a TMR of between 10.6 and 37.3. We observe that these conclusions generally stand for both short- and long-running functions captured in the trace; however, short functions exhibit higher variability in their execution time. Thus, only 60% of the functions that run for less than a second have a TMR of less than 10; meanwhile, 90% of the functions that run for more than ten seconds have a sub-10 TMR.

3.7 Related Work

Prior work includes a number of benchmarking frameworks and suites for end-to-end analysis of various serverless clouds. FaaSDom [104], SebS [51], and BeFaaS [77] introduce automated deployment and benchmarking platforms, along with a number of serverless applications as benchmarks, supporting many runtimes and providers. ServerlessBench [152] and FunctionBench [86, 87] present collections of microbenchmarks and real-world workloads for performance and cost analysis of various clouds [152, 86, 87]. In contrast, STeLLAR stresses the components of serverless clouds to pinpoint their implications on tail latency whereas the prior works focus on measuring performance of distinct applications or evaluate the efficiency of certain serverless test cases, e.g., invoking a chain of functions or concurrently launching function instances.

Another body of works study the performance of particular components of serverless systems. Wang et al. conducted one of the first comprehensive studies of production clouds [148], investigating a wide range of aspects, including cold start delays for different runtimes.While we analyze many more tail latency factors, we also find that some of their results in 2018 are now obsolete, e.g., in contrast to their findings, we show that the choice of runtime minimally affects the tail latency in AWS (§3.5.2.2). vHive is a framework for serverless experimentation and explores the cold-start delays of MicroVM snapshotting techniques [143]. Li et al. studies the throughput of the cluster infrastructure of open-source FaaS platforms in the presence of concurrent function invocations [97]. Hellerstein et al. analyzes the existing I/O bottlenecks in modern serverless systems [80]. FaaSProfiler conducts microarchitectural analysis of serverless hosts [132].

Other works investigate the efficiency of serverless systems for different classes workloads, namely ML training [83], latency-critical microservices [73], data-intensive applications[89, 90, 129], and confidential computations [98]. Eismann et al. categorizes open-source serverless applications according to their non-performance characteristics [67]. Shahrad et al. analyzes invocation frequency and execution time distributions of applications in Azure Functions and explores the design space of function instance keep-alive policies [133].

3.8 Conclusion

Measuring and analyzing tail latency and its sources is crucial when designing latency-critical cloud applications. To the best of our knowledge, STeLLAR is the first open-source provider-agnostic benchmarking framework that enables tail-latency analysis of serverless systems, allowing to study performance both end-to-end and per-component. By design, STeLLAR is highly configurable and can model various load scenarios and vary the characteristics of serverless applications, selectively stressing various components of serverless infrastructure. Using STeLLAR, we perform a comprehensive analysis of tail latency characteristics of three leading serverless clouds and show that storage accesses – in particular, moving function instance initialization state upon a cold start and cross-function data communication – and bursty traffic of function invocations are the largest contributors to latency variability in modern serverless systems. We also find that some of the important factors, like the choice of language runtime, have a minor impact on tail latency.

Chapter 4

vHive Framework for Serverless Systems Research

In this chapter, we introduce vHive, an open-source full-stack framework for experimenting with serverless systems, which integrates production-grade components from leading commercial clouds into a representative research platform. Using vHive, we analyze the root cause of the cold-start delays, which is the first performance bottleneck we found in modern serverless clouds (§3).

4.1 Introduction

Today most serverless providers rely on proprietary infrastructure, preventing innovation efforts in academic environment, where researchers lack access to the complex software stack of the real-world clouds. As the serverless premise implies that serverless cloud providers take complete responsibility for cloud infrastructure management, large parts of the leading clouds' stack are obscure to academic researchers. As a result, these researchers are limited to experiment with small-scale, often incomplete prototypes built in-house, diminishing the value of their research results in the context of real-world systems.

To facilitate deeper understanding and experimentation with serverless computing, we introduce *vHive*, an open-source framework for serverless experimentation, which enables systems researchers to innovate across the entire serverless stack.[1] Existing open-source systems and frameworks are ill-suited for researchers, being either incomplete, focusing only on one of the components, such as a hypervisor [75], or rely on

[1]The code is available at `https://github.com/ease-lab/vhive`.

insufficiently secure container isolation [32, 6, 7, 81, 94]. vHive integrates open-source production-grade components from the leading serverless providers, namely Amazon Firecracker [19], Containerd [50], Kubernetes [13], and Knative [11], that offer the latest virtualization, snapshotting, and cluster orchestration technologies along with a toolchain for functions deployment and benchmarking.

To illustrate vHive's utility, we study the cold-start delays, i.e., the breakdown of a a function instance bootstrapping time, analyzing the contributions by various components of the serverless software stack. Because of their short execution time, booting a function (i.e., *cold start*) has overwhelmingly high latency, and can easily dominate the total execution time. Moreover, customers are not billed for the time a function boots, which de-incentivizes the cloud vendor from booting each function from scratch on-demand. Customers also have an incentive to avoid cold starts because of their high impact on latency [119]. As a result, both cloud vendors and their customers prefer to keep function instances memory-resident (i.e., *warm*) [76, 113, 119]. However, keeping idle function instances alive wastefully occupies precious main memory, which accounts for 40% of a modern server's typical capital cost [19]. With serverless providers instantiating thousands of function on a single server [19, 21], the memory footprint of keeping all instances warm can reach into hundreds of GBs.

To avoid keeping thousands of functions warm while also eliding the high latency of cold-booting a function, the industry has embraced *snapshotting* as a promising solution. With this approach, once a function instance is fully booted, its complete state is captured and stored on disk. When a new invocation for that function arrives, the orchestrator can rapidly load a new function instance from the corresponding snapshot. Once loaded, the instance can immediately start processing the incoming invocation, thus eliminating the high latency of a cold boot.

Using vHive, we study the cold-start latency of functions from the FunctionBench suite [86, 87], their memory footprint, and their spatio-temporal locality characteristics when the functions run inside Firecracker MicroVMs [19] as part of the industry-standard Containerd infrastructure [4, 50]. We focus on a state-of-the-art baseline where the function is restored from a snapshot on a local SSD, thus achieving the lowest possible cold-start latency with existing snapshotting technology [5, 64].

Based on our analysis, we make three key observations. First, restoring from a snapshot yields a much smaller memory footprint (8-99MB) for a given function than cold-booting the function from scratch (148-256 MB) – a reduction of 61-96%. The reason for the greatly reduced footprint is that only the pages that are actually used by the function are loaded into memory. In contrast, when a function boots from scratch, both the guest OS and the function's user code engage functionality that is never used during serving a function invocation (e.g., function initialization).

Our second observation is that the execution time of a function restored from a snapshot is dominated by serving page faults in the host OS as pages are lazily mapped into the guest memory. The host OS serves these page faults one by one, bringing the pages from the backing file on disk. We find that these file accesses impose a particularly high overhead because the guest accesses lack spatial locality, rendering host OS' disk read-ahead prefetching ineffective. Altogether, we find that servicing page faults on the critical path of function execution accounts for 95% of actual function processing time, on average – a significant slowdown, compared to executing a function from memory (i.e., "warm").

Our last observation is that a given function accesses largely the same set of guest-physical memory pages across multiple invocations of the function. For the studied functions, 97%, on average of the memory pages are the same across invocations.

We summarize our contributions as following:

- We release vHive, an open-source framework for serverless experimentation, combining production-grade components from the leading serverless providers to enable innovation in serverless systems across their deep and distributed software stack.

- Using vHive, we demonstrate that the state-of-the-art approach of starting a function from a snapshot results in low memory utilization but high start-up latency due to lazy page faults and poor locality in SSD accesses. We further observe that the set of pages accessed by a function across invocations recurs.

4.2 vHive: an Open-Source Framework for Serverless Experimentation

To enable a deeper understanding of serverless computing platforms, this paper introduces *vHive*, an open-source framework for experimentation with serverless computing. As depicted in Fig. 4.1, vHive integrates production-grade components from the leading serverless providers, such as Amazon and Google.

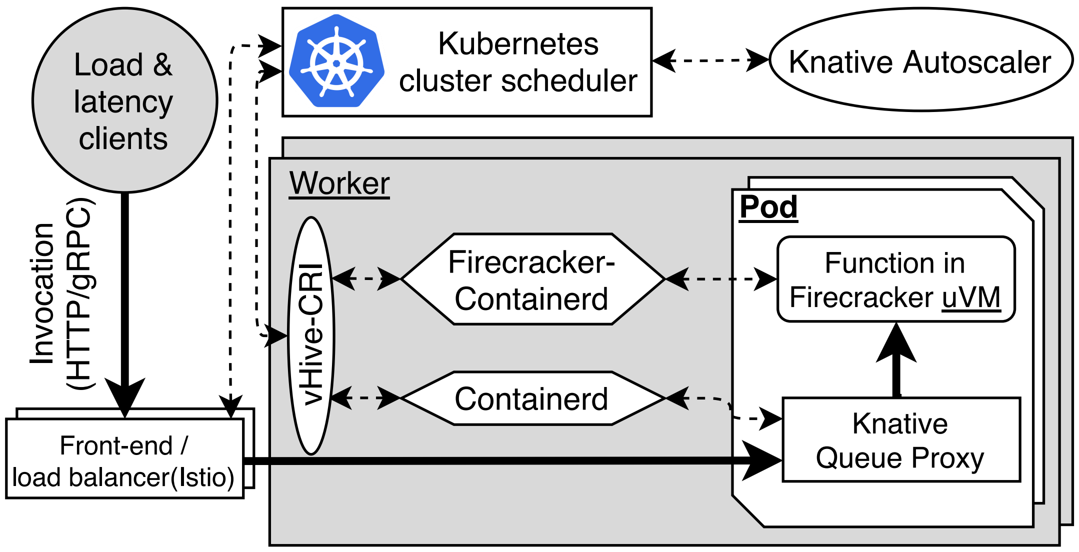


Figure 4.1: vHive architecture overview. Solid and dashed arrows show the data plane and the control plane, respectively.

4.2.1 Deploying and Programming with Functions in vHive

vHive adopts Knative [11], a serverless framework that runs on top of Kubernetes [13] and offers a programming and deployment model that is similar to AWS Lambda [33] and Azure functions [113]. To deploy an application in vHive, one can deploy application functions by supplying Knative with each function's Open Container Initiative (OCI) [141] image, e.g., a Docker image, along with a configuration file. This OCI image contains the function's handle code, which is executed by an HTTP or gRPC server upon an invocation. The configuration file contains the relevant environment variables and other parameters for function composition and function instances scaling. Using the configuration files, the application developers can compose their functions with any conventional "serverful" services with Kubernetes providing their URLs to the relevant functions. For example, functions that use large inputs or produce large outputs, like photos or videos, often have to save them in an object store or a database.

Upon a function's deployment, Knative provides a URL for triggering this function. Using these URLs, application developers can compose their functions, e.g., by specifying the URL of a callee function in the configuration file of the caller function. For each function, Knative configures the load-balancer service, sets up the network routes and dynamically scales the number of instances of the function in the system, according to changes in the function's invocation traffic.

4.2.2 vHive Infrastructure Components

Serverless infrastructure comprises of the front-end fleet of servers that expose the HTTP endpoints for function invocations, the worker fleet that executes the function code, and the cluster manager that is responsible for managing and scaling function instances across the workers [19, 54, 133]. These components are connected by an HTTP-level fabric, e.g., gRPC [8], that that enables management and resources monitoring [19].

A function invocation, in the form of an HTTP request or an RPC, first arrives at one of the front-end servers for request authentication and mapping to the corresponding function. In vHive, the Istio service [9] plays the roles of an HTTP endpoint and a load balancer for the deployed functions. If the function that received an invocation has at least one active instance, the front-end server simply routes the invocation request to an active instance for processing.

If there are no active function instances, the load balancer contacts the cluster manager to start a new instance of the function before the load balancer routes this invocation to a worker. vHive relies on Kubernetes cluster orchestrator to automate services deployment and management. Knative seamlessly extends Kubernetes, which was originally designed for conventional "serverful" services, to enable autoscaling of functions. For each function, Knative deploys an autoscaler service that monitors the invocation traffic to each function and makes decisions on scaling the number of functions instances in the cluster based on observed load.

At the autoscaler's decision, a chosen worker's control plane starts a new function instance as a pod, the scaling granule in Kubernetes, that contains a Knative Queue-Proxy (QP) and a MicroVM that runs the function code. The QP implements a software queue and a health monitor for the function instance, reporting the queue depth to the function's autoscaler, which is the basis for the scaling decisions. The function runs in a MicroVM to isolate the worker host from the untrusted developer-provided code. vHive follows the model of AWS Lambda, which deploys a single function inside a MicroVM that processes a single invocation at a time [19].

To implement the control plane, vHive introduces a vHive-CRI orchestrator service that integrates the two forks of Containerd – the stock version [50] and the Firecracker-specific version developed for MicroVMs [4] – for managing the lifecycle of containerized services (e.g., the QP) and MicroVMs. The vHive-CRI orchestrator receives Container-Runtime Interface (CRI) [47] requests from the Kubernetes control plane and processes them, making the appropriate calls to the corresponding Containerd services.

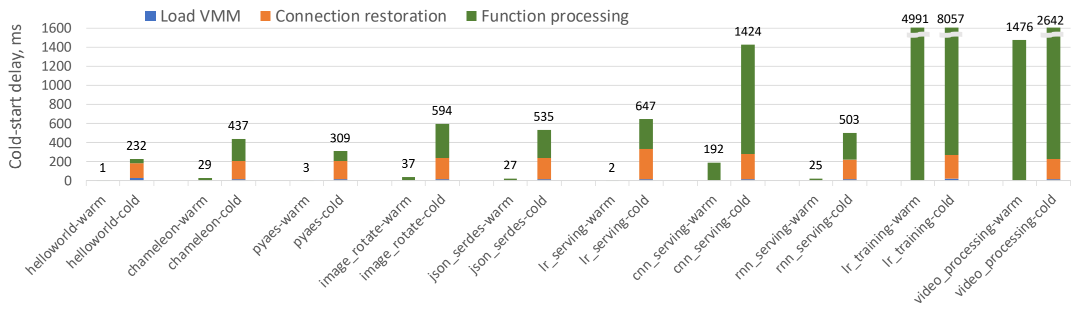


Figure 4.2: Cold-start latency breakdown for Firecracker's snapshot load mechanism, compared to the warm latency.

Once the load balancer, which received the function invocation, the QP, and the function instance inside a MicroVM establish the appropriate HTTP-level connections, the data plane of the function is ready to process function invocations. When the function instance finishes processing the invocation, it responds to the load balancer, which forwards the response back to the invoking client.

vHive enables systems researchers to experiment with serverless deployments that are representative of production serverless clouds. vHive allows easy analyzing of the performance of an arbitrary serverless setting by offering access to Containerd and Kubernetes logs with high-precision timestamps or by collecting custom metrics. vHive also includes the client software to evaluate the response time of the deployed serverless functions in different scenarios, varying the mix of functions and the load. Finally, vHive lets the users experiment with several modes for cold function invocations, including loading from a snapshot or booting a new VM from a root filesystem.

4.3 Serverless Latency and Memory Footprint Characterization

In this section, we use vHive to analyze latency characteristics and memory access patterns of serverless functions, deployed in Firecracker MicroVM instances with snapshot support [5].

4.3.1 Evaluation Methodology

Similarly to prior work [64], we focus on the evaluation of a single worker server. The existing distributed serverless stack contributes little, e.g., less than 30ms as shown by AWS [19], to the overall end-to-end latency, as compared to many hundreds of milliseconds of the worker-related latency that we demonstrate below. Prior work measured the cold-start delay as the time between starting to load a VM from a snapshot to the time the instance executes the first instruction of the user-provided code of the function [64]. As the metric for our cold-start delay measurements, we choose the latency that includes not only the critical path of the VM restoration but the entire cold function invocation latency on a single worker. The measurements capture the latency from the moment a worker receives a function invocation request to the moment when the worker is ready to send the function's response back to the load balancer. This latency includes both the control-plane delays (including interactions with Containerd and Firecracker hypervisor) and data-plane time that is gRPC request processing and actual function execution.

Our experiments aim to closely model the workloads as in a modern serverless environment. First, we adopt a number of functions, listed in Table 5.1, from a representative serverless benchmark suite called FunctionBench [86, 87]. Second, to simulate the low invocation frequency of serverless functions in production [133], the host OS' page cache is flushed before each invocation of a cold function.

To evaluate the cold-start start delay in a serverless platform similar to AWS Lambda, we augment the vHive-CRI orchestrator to act as a MicroManager in AWS [19]. In this implementation, the vHive-CRI orchestrator not only implements the control plane but also acts as a data plane software router that forwards incoming function invocations to the appropriate function instance and waits for its response over a persistent gRPC connection. Note that in this setting, the worker infrastructure does not include the Queue-Proxy containers so that the data plane resembles per-function gRPC connections. Without a loss of generality, this work assumes the fastest possible storage for the snapshots that is a local SSD, which yields the lowest possible cold-start latency compared to a local HDD or disaggregated storage. §5.3.1 provides further details of the platform as well as the host and the guest setup.

To study the memory access patterns of serverless functions, we trace the guest memory addresses that a function instance accesses between the point when the vHive-CRI orchestrator starts to load a VM from a snapshot and the moment when the orchestrator receives a response from the function. As Firecracker lazily populates the guest memory, first memory access to each page from the hypervisor or the guest raises a page fault in the host that can be traced. We use Linux `userfaultfd` feature [99] that allows a userspace process to inspect the addresses and serve the page faults on behalf of the host OS.

4.3.2 Quantifying Cold-Start Delays

We start by evaluating the cold-start latency of each function under study and compare it to the invocation latency of the warm function instance. Recall that a warm instance is memory-resident and does not experience any cold-start delay when invoked. To obtain a detailed cold-start latency breakdown, we instrument the vHive-CRI orchestrator and invoke each function 10 times. To model a cold invocation, we flush the host OS page caches after each measurement.

Figure 4.2 shows the latency for the cold and warm invocations for each function. As expected, when a function instance remains warm (i.e., stays in memory), the instance delivers a very low invocation latency. By contrast, we find that a cold invocation from a snapshot takes one to two orders of magnitude longer than a warm invocation, which indicates that even with state-of-the-art snapshotting, cold-start delays are a major pain point for functions.

To investigate the performance difference, we examine the end-to-end cold invocation latency breakdown. First, the vHive-CRI orchestrator spawns a new Firecracker process and restores the virtual machine monitor (VMM) state as well as the state of the emulated network and block devices – the *Load VMM* latency component. After that, the orchestrator resumes the loaded function instance's virtual CPUs and restores the persistent gRPC connection to the gRPC server inside the VM. We name this latency component as *Connection restoration*. These two latency components are universal across all functions as they are part of the serverless infrastructure. Finally, we measure the actual function invocation processing time, referred to as *Function processing*.

The per-function latency breakdown is also plotted in Figure 4.2. We observe that the first two universal components, namely *Load VMM* and *Connection restoration*, take 156-317 ms. Meanwhile, the actual function processing takes much longer (95% longer on average) for cold invocations as compared to warm invocations of the same functions, reaching into 100s of milliseconds even for functions like `helloworld` and `pyaes` that take only a few milliseconds to execute when warm.

The state-of-the-art snapshotting techniques rely on lazy paging to eliminate the population of guest memory from the critical path of VM restoration (§2.4.1). A consequence of this design is that each page touched by a function must be faulted in at the first access, resulting in thousands of page faults during a single invocation

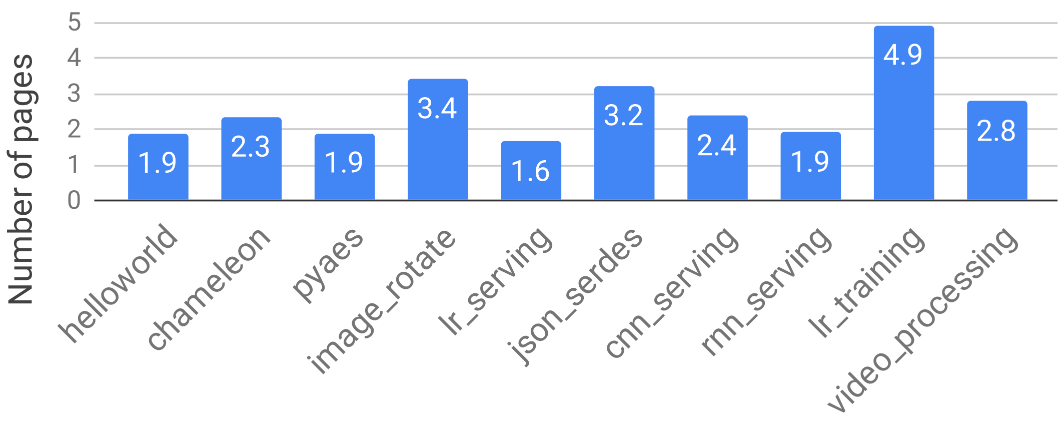


Figure 4.3: Guest memory pages contiguity.

of a function. Page faults are processed serially because the faulting thread is halted until the OS brings the memory page from disk and installs it into the virtual address space by setting up the memory mappings in the process page table. In this case, the performance of the guest significantly depends on the disk latency as the OS needs to bring the missing pages from the guest memory file.

We also study the contiguity of the faulted pages, with the results depicted in Fig. 4.3. We find that function instances tend to access pages that are located in non-adjacent locations inside the guest memory. This lack of spatial locality significantly increases disk access time, and thus page fault delays, because sparse accesses to disk cannot benefit from the host OS's run-ahead prefetching. Fig. 4.3 shows that the average length of the contiguous regions of the guest-physical memory is around 2-3 pages for all functions except `lr_training` that shows contiguity of up to 5 pages.

4.3.3 Function Memory Footprint & Working Set

The above analysis demonstrates the benefits of keeping functions warm, because cold function invocations significantly increase the end-to-end function invocation latency. In this subsection, we show that despite the advantages of warm functions, keeping many functions warm is wasteful in terms of memory capacity.

We first investigate the fraction of a function instance's memory footprint that is related to the actual function invocation. First, we measure the total footprint of a booted VM after the first function invocation is complete using the Linux `ps` command, since a VM appears as a regular process in the host OS. This footprint includes the hypervisor and the emulated layer overhead (around 3MB [19]), the memory pages

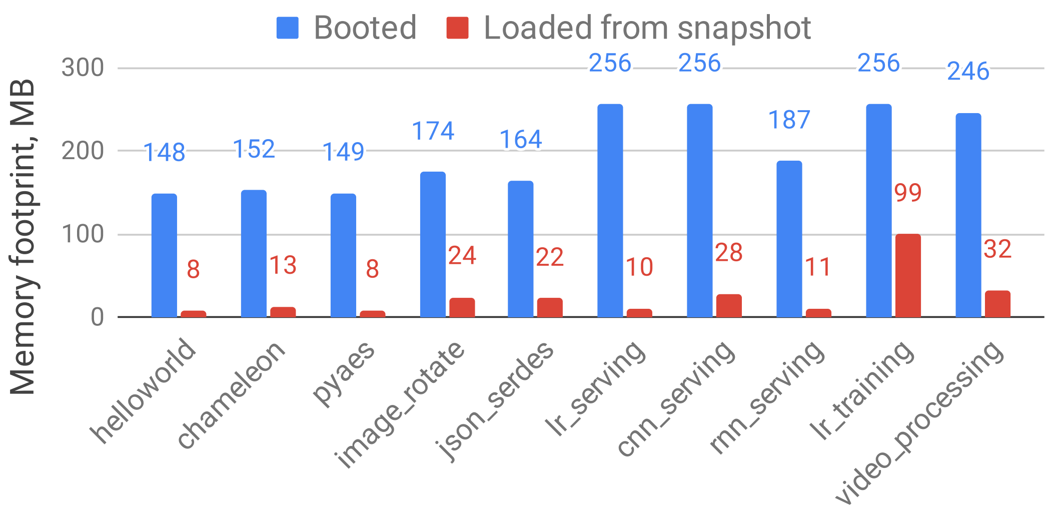


Figure 4.4: Memory footprint of function instances after one invocation.

that are accessed during the VM's boot process, function initialization, and the actual invocation processing. Second, for a VM that is loaded from a snapshot, we trace the pages, using Linux `userfaultfd` [99], that are accessed only during the invocation processing, i.e., from the moment the VM is loaded to the moment when the vHive-CRI orchestrator receives the response from the function. Unlike the first measurement, this footprint relates only to the invocation processing.

Figure 4.4 (the blue bars) shows the memory footprint of a single freshly-booted function instance. We observe that, for all functions, their memory footprint reaches 100-200MB. Thus, assuming that a serverless provider co-locates thousands of different functions instances on the same host and disallows memory sharing for security reasons (as is the case in practice [19]), the aggregate footprint of function instances will reach into hundreds of gigabytes.

Figure 4.4 also plots the footprint of the function instances loaded from a snapshot after the first invocation (red bars). We observe that, in this case, the functions' working sets span 8-99MB (24MB on average), which is 3-39%, and 9% on average of their memory footprint after booting. The reason why the memory footprint of a function booted from scratch is much higher than the one loaded from a snapshot is that starting an instance by booting requires many steps: booting a VM, starting up the Containerd's agents [131] as well as user-defined function initialization. This complex boot procedure engages much more logic (e.g., guest OS and userspace code) than just processing the actual function invocation, which naturally affects the former's memory footprint.

Despite the fact that, when loaded from a snapshot, the memory footprint of a function instance is relatively compact, the total memory footprint for thousands of such functions would still comprise tens of GBs. While potentially affordable memory-wise, we note that keeping this much state in memory is wasteful given the low invocation frequency of many functions (§2.2). Moreover, such a high memory commitment would preclude co-locating memory-intensive workloads on nodes running serverless jobs, thus limiting a cloud operator's ability to take advantage of idle resources. We thus conclude that while functions loaded from a snapshot present an opportunity in terms of their small memory footprint, by itself, they are not a solution to the memory problem.

4.3.4 Guest Memory Pages Reuse

After establishing that the working sets of serverless functions booted from a snapshot are compact, we study how the working set of a given function changes across invocations. Our hypothesis is that the stateless nature of serverless functions results in a stable working set across invocations.

User and guest kernel code pages account for a large fraction of functions' footprint. This code belongs either to the underlying infrastructure or the actual function implementation. Providers deploy additional control-plane services inside a function's sandbox and use general-purpose communication fabric (e.g., gRPC) to connect functions to the vHive-CRI orchestrator [19, 131]. The gRPC framework uses the standard TCP network protocol, similarly to AWS Lambda [19], that adds the guest OS's network stack to the instance footprint. Using the `helloworld` function, we estimate that this infrastructure overhead accounts for up to 8MB of a function's guest-memory footprint and is stable across function invocations.

We observe that functions naturally use the same set of memory pages while processing different inputs. For example, when rotating different images or evaluating different customer reviews, functions use the same calls to the same libraries and rely on the same functionality inside the guest kernel, e.g., the networking stack. Moreover, the functions engage the same functionality that is a part of the provider's infrastructure, e.g., the Containerd's agents inside a VM [131]. Finally, we observe that even when a function's code performs a dynamic allocation, the guest OS buddy allocator is likely to make the same or similar allocation decisions. These decisions are based on the state of its internal structures (i.e., lists of free memory regions), which is the same across invocations being loaded from the same VM snapshot. Hence, the lack of concurrency

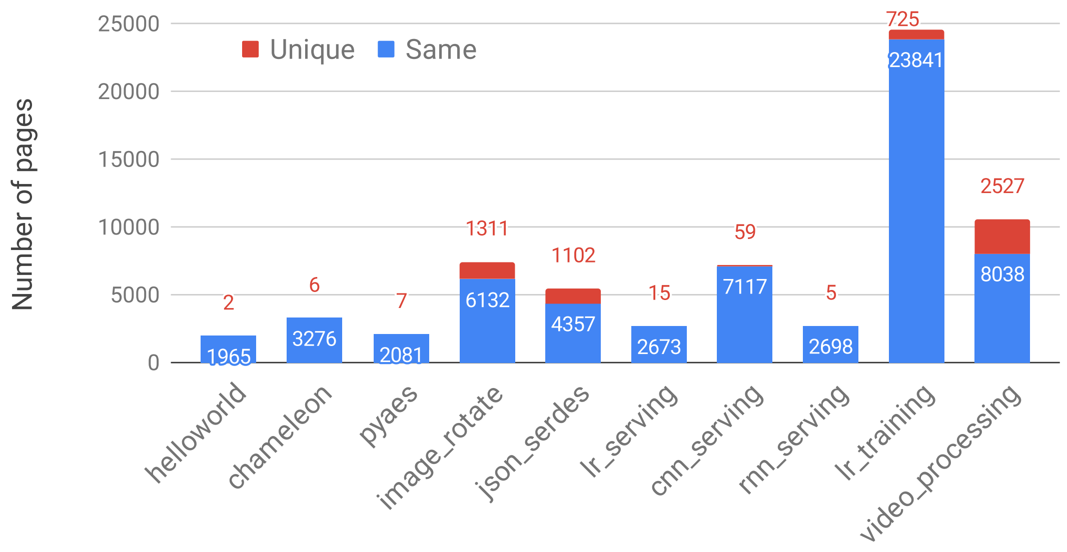


Figure 4.5: Number of pages that are unique or same across invocations with different inputs. The numbers above the bars correspond to uniquely accessed pages.

and non-determinism inside the user code of the functions that we study results in a similar guest physical memory layout.

We validate our hypothesis about the working sets by studying the guest memory pages that are accessed when a function is invoked with different inputs. Fig. 4.5 demonstrates that the majority of pages accessed by all studied functions are the same across invocations with different inputs. For 7 out of 10 functions, more than 97% of the memory pages are identical across invocations. For `image_rotate`, `json_serdes`, `lr_training`, and `video_processing`, reuse is lower because these functions have large inputs (photos, JSON files, training datasets, and videos, respectively) that are 1-10MB in size. Nonetheless, even for these three functions, over 76% of memory pages are the same across invocations.

4.3.5 Summary

We have shown that invocation latencies of cold functions may exceed those of warm functions by one to two orders of magnitude, even when using state-of-the-art VM snapshots for rapid restoration of cold functions. We found that the primary reason for these elevated latencies is that the existing snapshotting mechanisms populate the guest memory on-demand, thus incurring thousands of page faults during function invocation. These page faults are served one-by-one by reading non-contiguous pages from a snapshot file on disk. The resulting disk accesses have little contiguity and induce significant delays in processing of these page faults, thus slowing down VM restoration from a snapshot.

We have further shown that function instances restored from a snapshot have compact working sets of guest memory pages, spanning just 24MB, on average. Moreover, these working sets are stable across different invocations of the same function; indeed, the function instances access predominantly the same memory pages even when invoked with different inputs.

4.4 Related Work

Researchers release a number of benchmarks for serverless platforms. vHive adopts dockerized benchmarks from FunctionBench that provides a diverse set of Python benchmarks [86, 87]. ServerlessBench contains a number of multi-function benchmarks, focusing on function composition patterns [152]. Researchers and practitioners release a range of systems that combine the FaaS programming model and autoscaling [32, 6, 7, 81, 94]. Most of these platforms, however, rely on Docker or language sandboxes for isolating the untrusted user-provided function code that is often considered insufficiently secure in public cloud deployments [40, 133]. Kata Containers [10] and gVisor [75] provide virtualized runtimes that are CRI-compliant but do not provide a toolchain for functions deployment and end-to-end evaluation and do not support snapshotting. Compared to these systems, vHive is a open-source serverless experimentation platform – representative of the production serverless platforms, like AWS Lambda [33] – that uses latest virtualization, snapshotting, and cluster orchestration technologies combined with a framework for functions deployment and benchmarking.

4.5 Conclusion

Optimizing cold-start delays is key to improving serverless clients experience while maintaining serverless computing affordable. To understand the root cause of the long cold-start delays, we build vHive, an open-source full-stack framework for serverless experimentation, which integrates components from the leading serverless cloud providers in a single complete and representative platform. Our analysis identifies that the root cause for high cold-start delays is that the state-of-the-art solutions populate the guest memory on demand when restoring a function instance from a snapshot. This results in thousands of page faults, which must be served serially and significantly slow down a function invocation. We further find that functions exhibit a small working set of the guest memory pages that remains stable across different function invocations.

Chapter 5

Record-and-Prefetch Snapshots

In this chapter, we leverage the insights from the studies in the previous chapter (§4) and introduce Record-and-Prefetch (REAP) snapshots, a lightweight mechanism that reduces these delays.

5.1 Introduction

In the previous chapter (§4), we characterize a state-of-the-art snapshot-based serverless infrastructure available in the vHive framework, based on industry-leading Containerd orchestration framework and Firecracker hypervisor technologies. We find that the execution time of a function started from a snapshot is 95% higher, on average, than when the same function is memory-resident. We show that the high latency is attributable to frequent page faults as the function's state is brought from disk into guest memory one page at a time. Our analysis further reveals that functions access the same stable working set of pages across different invocations of the same function.

Leveraging the observations above, we introduce Record-and-Prefetch (REAP) – a light-weight software mechanism for serverless hosts that exploits recurrence in the memory working set of functions to reduce cold-start latency. Upon the first invocation of a function, REAP records a trace of guest-physical pages and stores the copies of these pages in a small working set file. On each subsequent invocation, REAP uses the recorded trace to proactively prefetch the entire function working set with a single disk read and eagerly installs it into the guest's memory space. REAP is implemented entirely in userspace, using the existing Linux user-level page fault handling mechanism [99]. Our evaluation shows that REAP eliminates 97% of the pages faults, on average, and reduces the cold-start latency of serverless functions by an average of 3.7$\times$.

We summarize our contributions as following:

- We present REAP, a record-and-prefetch mechanism that eagerly installs the set of pages used by a function from a pre-recorded trace. REAP speeds up function cold start time by 3.7×, on average, without introducing memory overheads or memory sharing across function instances.
- We implement REAP entirely in userspace with minimal changes to the Firecracker hypervisor and no modifications to the kernel. REAP is independent of the underlying serverless infrastructure and can be trivially integrated with other serverless frameworks and hypervisors, e.g., Kata Containers [10] and gVisor [75].

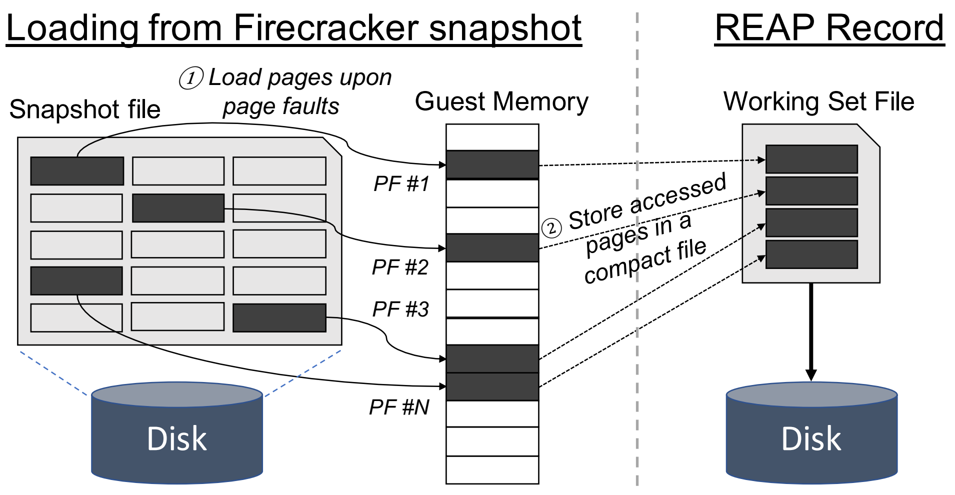
5.2 REAP: Record-and-Prefetch

The compact and stable working set of a function's guest memory pages, which instances of the given function access across invocations of the function, provides an excellent opportunity to slash cold-start delays by prefetching.

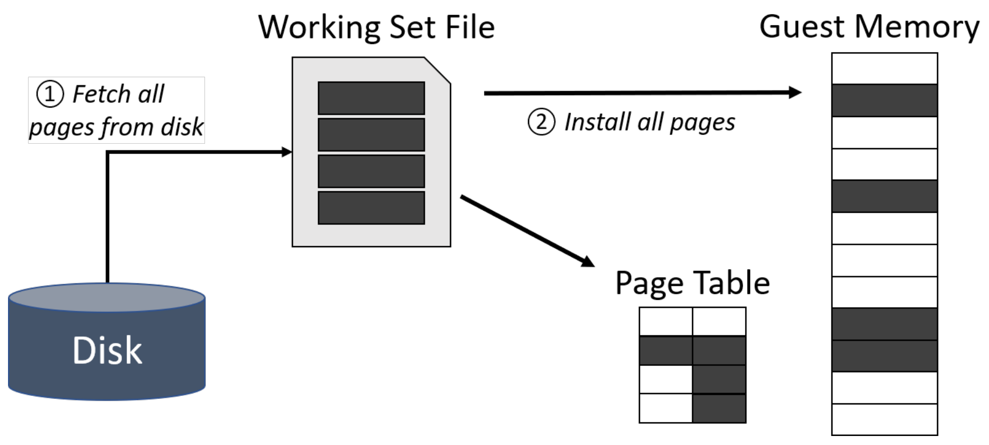
Based on this insight, we introduce Record-and-Prefetch (REAP), a light-weight software mechanism inside the vHive-CRI orchestrator to accelerate function invocation times in serverless infrastructures. REAP records a function's working set upon the first invocation of a function from a snapshot and replays the record to accelerate load times of subsequent cold invocations of the function by eliminating the majority of guest memory page faults. The rest of the section details the design of REAP.

5.2.1 REAP Design Overview

Given an existing function snapshot, REAP operates in two phases. During the *record* phase, REAP traces and inspects the page faults that a function instance triggers when accessing pages in the guest memory, identifying the positions of these pages in the backing guest memory file (Fig. 5.1a). After a function invocation is complete, REAP creates two files, namely the *working set (WS) file)* that contains a copy of all accessed guest memory pages in a contiguous compact chunk and the *trace file* that contains the offsets of the original pages inside the guest memory file. The contiguous compact WS file can be rapidly brought into physical memory in a single read operation, which greatly reduces disk and system-level overheads in the snapshot baseline that requires many disparate accesses to pages scattered across the guest memory file on disk.


Loading from Firecracker snapshot
REAP Record
① Load pages upon page faults
Snapshot file
Guest Memory
Working Set File
PF #1
PF #2
PF #3
PF #N
② Store accessed pages in a compact file
Disk
Disk

(a) REAP record phase.



(b) REAP prefetch phase.

Figure 5.1: REAP's two-phase operation.

After the completion of the record phase, all future invocations of the function enjoy accelerated load times as REAP's *prefetch* phase eagerly populates the guest memory from the WS file before launching the function instance (Fig. 5.1b). Upon an arrival of a new invocation, REAP fetches the entire WS file from disk into a temporary buffer in the orchestrator's memory and eagerly installs the pages into the function instance's guest memory region. REAP also populates the page table of the instance in the host OS. As a result, when the instance is loaded, the function executes without triggering page faults to the stable memory working set. Page faults to uniquely-accessed pages in a given invocation are handled by REAP on demand.

5.2.2 Implementation

REAP adheres to the following design principles, which facilitate adoption and deployment in a cloud setting: i) REAP is agnostic to user codebase; ii) REAP is independent of the underlying serverless infrastructure; iii) REAP is implemented entirely in userspace without kernel modifications; iv) REAP works efficiently in a highly multi-tenant serverless environment.

We implement REAP as a part of the vHive-CRI orchestrator that controls the lifecycle of all function instances. For each function, the vHive-CRI orchestrator tracks active function instances and performs the necessary bookkeeping, including maintaining the snapshot files and working set records. To accommodate the highly-concurrent serverless environment with many function instances executing simultaneously, it is a fully parallel implementation with a dedicated *monitor* thread for each function instance. Each monitor thread records or prefetches the working set pages and also serves page faults that are raised by the corresponding instance. In our prototype, the monitor threads are implemented as lightweight goroutines, which are scheduled by the Go runtime.

To implement the monitor, we use the Linux `userfaultfd` feature that allows a userspace program to handle page faults on behalf of the OS. In Linux, a target process can register a virtual memory region in anonymous memory and request a user-fault file descriptor, which can be passed to a monitor running as a separate thread or process. The monitor polls for page fault events that the OS forwards to the user-fault file descriptor. Upon a page fault, the monitor installs the contents of the page that triggered the page fault. The monitor is free to retrieve page contents from any appropriate source, such as a file located on a local disk or from the network. Furthermore, the monitor can install any number of pages before waking up the target process. Thanks to these features, the monitor can support both local and remote snapshot storage, and can eagerly install the content of the entire WS file at once.

Upon each function invocation for which there is no warm instance available, the vHive-CRI orchestrator launches the monitor thread in one of two modes: record, if no WS file is available for this instance, or prefetch, if a corresponding WS file exists.

5.2.2.1 Record Phase

The goal of the monitor during the record phase is to capture the memory working set for functions instantiated from snapshots. Before loading the VMM state from a snapshot, the hypervisor maps the guest memory file as an anonymous virtual memory region and requests a user-fault file descriptor from the host OS, passing this descriptor over a socket to the monitor thread of the vHive-CRI orchestrator. Then, the hypervisor restores the VMM and emulated devices' state and resume the virtual CPUs of the newly loaded function instance that can start processing the function invocation.

Every first access to a guest memory page raises a page fault that needs to be handled and recorded by the monitor. The monitor maps the guest memory file as a regular virtual memory region in the monitor's virtual address space and polls (using the `epoll` system call) for the host kernel to forward the page fault events, triggered by the instance. Upon receiving a page fault event, the monitor reads a control message from the user-fault file descriptor that contains the description of the page fault, including the address in the virtual address space of the target function instance. The monitor translates this virtual address into an offset that corresponds to the page location in the guest memory file. While serving the page faults, the monitor records the offsets of the working set pages in the trace file.

We augment the Firecracker hypervisor to inject the first page fault of each instance to the first byte of the instance's guest memory. Doing so allows file offsets for all of the following page faults to be derived by subtracting the virtual address of the first page fault. Using the file offset of the missing page, the monitor locates the page in the guest memory file and installs the page into the guest memory region of the instance by issuing an `ioctl` system call to the host kernel, which also updates the extended page tables of the target function instance. After the vHive-CRI orchestrator receives a response from the function, indicating that the function invocation processing has completed, the monitor copies the recorded working set pages, using the offsets recorded in the trace file, into a separate WS file (§4.3.3).

Note that the record phase increases the function invocation time as compared to the baseline snapshots due to userspace page fault handling. As such, REAP penalizes the first function invocation to benefit subsequent invocations. We quantify the recording overheads in §5.3.4.

5.2.2.2 Prefetch Phase

For every subsequent function invocation, the vHive-CRI orchestrator spawns a dedicated monitor thread that uses the WS file to prefetch the working set memory pages from disk into a buffer in the monitor's memory with a single `read` system call. Then, the monitor eagerly and proactively installs the pages into the guest memory through a sequence of `ioctl` calls, following which it wakes up the target function instance with another `ioctl` call.

As in the record phase, the monitor maps the guest memory file during every subsequent cold function invocation. After installing all the working set pages from the WS file, the monitor keeps polling for page faults to pages that are missing from the stable working set and installs them on demand, as in the record phase. Since the WS file captures the majority of pages that a function instance accesses during an invocation, only a small number of page faults needs to be served by the monitor on demand.

5.2.2.3 Disk Bandwidth Considerations

REAP's efficiency depends entirely on the performance of the prefetch phase and, specifically, how fast the vHive-CRI orchestrator can retrieve the working set pages from disk. Although a single commodity SSD can deliver up to 1-3 GB/s of read bandwidth, SSD throughput varies considerably depending on disk access patterns. An SSD can deliver high bandwidth with one large multi-megabyte read request, or with $>$10 4KB requests issued concurrently. For example, on our platform (§6.5), with a standard Linux `fio` IO benchmark [105] that issues a single 4KB read request, the SSD can deliver only 32MB/s, whereas issuing 16 4KB requests can increase the SSD throughput to 360MB/s. While concurrent reads deliver much higher bandwidth than a single 4KB read, the achieved bandwidth is still considerably below the peak of 850MB/s of our Intel SATA3 SSD.

We find that REAP achieves close to the peak SSD read throughput (533-850MB/s) by fetching the WS file in a single $>$8MB read operation that bypasses the host OS' page cache (i.e., the WS file needs to be opened with the `O_DIRECT` flag).

5.2.3 Discussion

REAP adheres to the design principles set out in §5.2.2. We implement REAP entirely in userspace as a part of the vHive-CRI orchestrator. It is written in 4.5K Golang LoC, including tests and benchmarks, and is loosely integrated with the industry-standard Containerd framework [4, 50, 131] via gRPC. The implementation does not require any changes to host or guest OS kernel. We add less than 200 LoC to Firecracker's Rust codebase, not including two publicly available Rust crates that we used, to register a Firecracker VM's guest memory with `userfaultfd` and to delegate page faults handling to the vHive-CRI orchestrator. Finally, the orchestrator follows a purely parallel implementation by spanning a lightweight monitor thread (goroutine) per function instance.

5.3 Evaluation

In this section, we describe the platform setup, including the host and the guest setups, and present REAP evaluation results. In our experiments, we follow the methodology that is described in §4.3.1, unless specified otherwise.

5.3.1 Evaluation Platform

We conduct our experiments on a 2×24-core Intel Xeon E5-2680 v3, 256GB RAM, Intel 200GB SATA3 SSD, running Ubuntu 18.04 Linux with kernel 4.15. We fix the CPU frequency at 2.5GHz to enable predictable and reproducible latency measurements. We disallow memory sharing among all function instances and disable swapping to disk, as suggested by AWS Lambda production guidelines [19, 15].

We use a collection of nine Python-based functions (Table 5.1) from the representative FunctionBench [86, 87] suite.[1] We also evaluate a simple `helloworld` function. The root filesystems for all functions are generated automatically by Containerd, using Linux device mapper functionality as used by Docker [60], from Linux Alpine OCI (Docker) images.[2] Functions with large inputs (namely `image_rotate`, `json_serdes`, `lr_training`, `video_processing`) retrieve their inputs from an S3 server [115] deployed on the same host.

We optimize virtual machines for minimum cold-start delays, similar to a production serverless setup, as in [19, 85]. The VMs run a guest OS kernel 4.14 without modules. Each VM instance has a single vCPU. We boot VM instances with 256MB guest memory, which is the minimum amount to boot all the functions in our study.

5.3.2 Understanding REAP Optimizations

We start by evaluating the cold-start latency of the `helloworld` function, whose short user-level execution time is useful for understanding serverless framework overheads. In addition to evaluating the baseline Firecracker snapshots and REAP as presented

[1]We omitted the microbenchmarks, MapReduce and Feature Generation because they require a distributed coordinator.

[2]The only exception is `video_processing` that uses a Debian image due to a problem with OpenCV installation on Alpine Linux.

Table 5.1: Serverless functions adopted from FunctionBench.

Name	Description
helloworld	Minimal function
chameleon	HTML table rendering
pyaes	Text encryption with an AES block-cipher
image_rotate	JPEG image rotation
json_serdes	JSON serialization and de-serialization
lr_serving	Review analysis, serving (logistic regr., Scikit)
cnn_serving	Image classification (CNN, TensorFlow)
rnn_serving	Names sequence generation (RNN, PyTorch)
lr_training	Review analysis, training (logistic regr., Scikit)
video_processing	Applies gray-scale effect (OpenCV)

in §6.3, we also study two additional design points that help justify REAP's design decisions. Specifically, we consider the following configurations.

Vanilla snapshots: This is the baseline configuration, which restores the VMM and the emulation layer in 50ms, then spends 182ms processing the function invocation (Fig. 5.2) that takes just 1ms for for a warm instance (Fig. 4.2). The large processing delay is directly attributed to the handling of page faults in the critical path of function execution. As `helloworld`'s working set is around 8MB, one can infer that vanilla snapshotting is only able to utilize 43MB/s of SSD bandwidth, i.e., $<5\%$ of the peak bandwidth on our platform.

Parallel Page Fault handling: This design (labeled "Parallel PFs" in Fig. 5.2) parallelizes page fault processing. It does so by using the trace file specifying the offsets of the pages comprising the stable working set, and deploys goroutines to bring in the associated pages, in parallel, from the guest memory file. For this and all the following configurations, we make 16 hardware threads available to vHive-CRI goroutines, managed by the Go runtime. Note that this configuration does *not* use the WS file.

We observe that parallelizing page faults reduces function invocation time by $1.9\times$ (to 118ms) by overlapping I/O processing and exploiting SSD's internal parallelism. Repeating the same calculation as for the baseline, we identify that the orchestrator uses only 130MB/s of SSD bandwidth, which is 15% of the maximum. This design point underlines that achieving high read bandwidth from the SSD is key to efficient page fault processing, and that lowering software overheads by itself is insufficient.

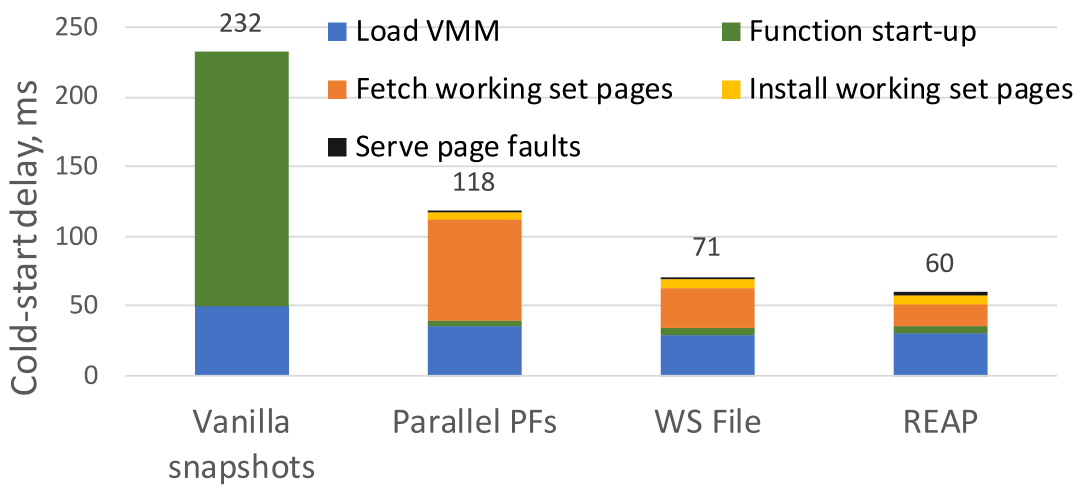


Figure 5.2: REAP optimization steps.

WS file: This design leverages the WS file (Sec 5.2.1), which enables fetching the entire stable memory working set of a function with a single IO read operation. The difference between this design point and REAP is that the former reads into the OS page cache (which is the default behavior in Linux), whereas REAP bypasses the page cache (§5.2.2.3). From the figure, one can see that fetching the pages from the WS file can be performed in 29ms, 3.1× faster than through parallel page-sized reads ("Parallel PFs" bar in Fig 5.2). This design point utilizes 275MB/s of SSD bandwidth.

REAP: The last bar shows the performance of the actual REAP design, as described in Sec 5.2.2.3, that fetches the working set pages from the WS file and bypasses the OS page cache. As the figure shows, retrieving the working set pages is accelerated by 2× (to 15ms) over the "WS File" design point that does not bypass the page cache. This highlights that while it's essential to optimize for disk bandwidth, software overheads also cannot be ignored. In this final configuration, REAP achieves 533MB/s of SSD bandwidth, which is within 37% of the 850MB/s peak of our SSD.

5.3.3 REAP on FunctionBench

Fig. 5.3 compares the cold-start delays of all the functions that we study with the baseline Firecracker snapshots and REAP prefetching. With REAP, all functions' invocations become 1.04-9.7× faster (3.7× on average). The fraction of time for restoring the connection from the orchestrator to the function's gRPC server shrinks by 45×, on average to a mere 4-7ms thanks to the stable working set for this core functionality that is prefetched by REAP.

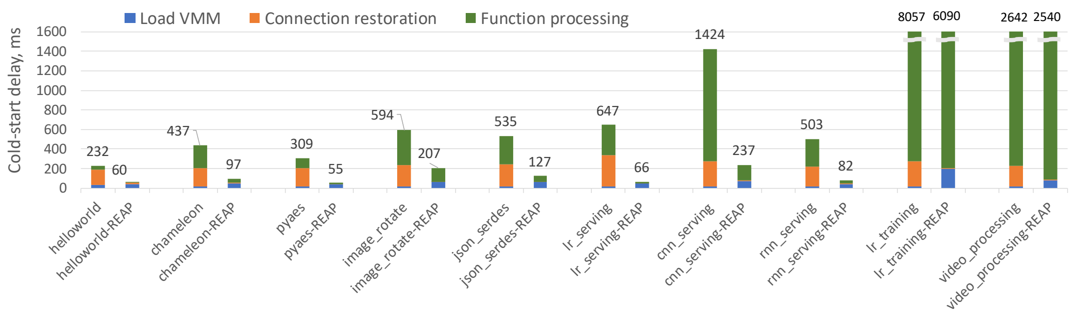


Figure 5.3: Cold-start delay with baseline snapshots (left bars) and REAP (right bars).

Although we find that REAP efficiently accelerates the actual function processing, functions with a large number of pages missing from the recorded working set benefit less from REAP. The function processing latency is reduced by 4.6×, on average, for all functions except `video_processing`. During the REAP record phase, the `video_processing` function takes a video fragment of a different aspect ratio than in the prefetch phase that, as we suspect, changes the way OpenCV performs dynamic memory allocation (e.g., uses buffers of different sizes), resulting in a different guest physical memory layout and, hence, different working sets. The orchestrator has to serve the missing pages one-by-one as page faults arise. However, the end-to-end cold-start delay for `video_processing` is nonetheless reduced as the longer function processing time is offset by faster re-connection to the function. We highlight, however, that functions with large inputs or control-flow that differs substantially across invocations may benefit less from REAP.

We repeat the same experiment in the presence of the invocation traffic to 20 warm, i.e., memory resident, functions and observe that the obtained data is within 5% of Fig. 5.3 results. Also, we measure the efficacy of REAP on the same server but store the snapshots on a 2TB Western Digital WD2000F9YZ SATA3 7200 RPM HDD, instead of the SSD, and observed a 5.4× speed-up (not shown in the figure), on average, with REAP over baseline snapshotting.

5.3.4 REAP Record Phase

REAP incurs a one-time overhead for recording the trace and the WS files. Upon the first invocation of a function, this one-time overhead increases the end-to-end function invocation time by 15-87% (28% on average). Since most functions that we study have small dynamic inputs, they exhibit relatively small overheads of 12-34%, with

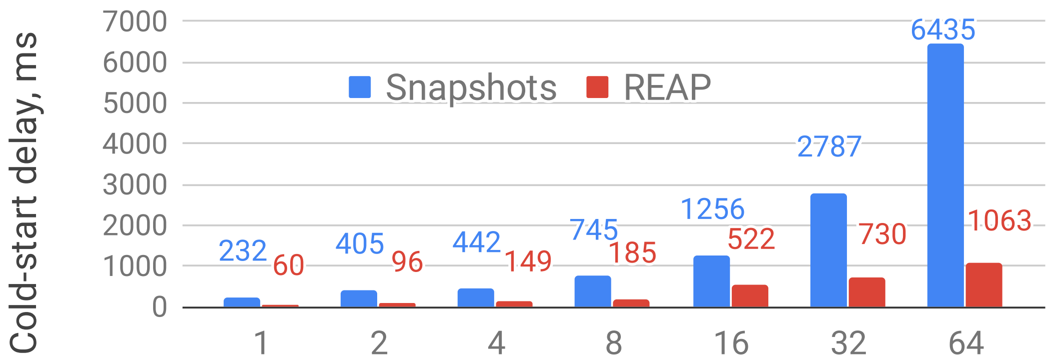


Figure 5.4: Average instance cold-start delay while sweeping the number of the concurrently loading instances.

`image_rotate` being an outlier with a performance degradation of 87%.

Because of the high speedups provided by REAP on all subsequent invocations of a function, and because the vast majority of functions execute multiple times [133], we conclude that REAP's one-time record overhead is easily amortized.

5.3.5 REAP Scalability

We demonstrate that REAP orchestrator retains its efficiency in the face of higher load. Specifically, we measure the average time that an instance takes to load from a snapshot and serve one function invocation when multiple independent functions arrive concurrently. We use the the `helloworld` function and consider up to 64 concurrent independent function arrivals. Fig. 5.4 shows the result of the study, comparing REAP to the baseline snapshots.

Concurrently loading function instances should be able to take advantage of the multi-core CPU and abundant SSD bandwidth (48 logical cores and 850MB/s peak measured SSD bandwidth in our platform). Thus, we expect that as the degree of concurrency increases, the average per-instance latency will not significantly increase thanks to the available parallelism. Indeed, REAP's cold invocation latency stays relatively low, increasing from 70ms to 185ms when the number of concurrent function arrivals goes from 1 to 8. By contrast, the baseline's per-instance invocation time shows a near-linear growth with the number of concurrently-arriving functions. We measure that the SSD throughput that the baseline instances are able to collectively extract is limited to mere 32MB/s for a single instance and 81MB/s for 64 concurrent instances.[3]

[3] We compute the SSD throughput per instance as the working set size divided by the average loading time.

Compared to the baseline, REAP is able to achieve 118-493MB/s, which explains its lower latency *and* better scalability. Starting from the concurrency degree of 16, REAP becomes disk-bandwidth bound and its scalability is diminished.

5.4 Discussion

5.4.1 REAP's Efficiency and Mispredictions

REAP's efficiency depends on how quickly the orchestrator can retrieve the guest memory pages from the snapshot storage and the percentage of the retrieved but unused pages. If the snapshots are located in a remote storage service (e.g., S3 or EBS), the retrieval speed depends on the amount of data to be moved and the latency and bandwidth of the network between the host and the service as well as the latency and bandwidth of the service's internal disks.

REAP reduces both the network and the disk bottlenecks by proactively moving a minimal amount of state. However, REAP may fetch a modest number of pages that are not accessed during processing of some invocations. Our analysis shows that the fraction of mispredicted pages during a cold invocation is close to the "Unique" pages metric, shown in Fig. 4.5, which is 3-39%. These mispredictions have no impact on system correctness. The cost of these mispredictions is a modest increase in SSD bandwidth usage, proportionate to the fraction of the mispredicted pages.

5.4.2 Applicability to Real-World Functions

Although REAP is applicable to the vast majority of functions, some functions may not benefit from REAP. For these functions, the additional working set and trace files may not be justified. Prior work shows that 90% of Azure functions are invoked less than once per minute, making these functions the primary target for REAP [133]. Functions that are invoked very rarely (e.g., 3.5% of functions are invoked less frequently than once per week) or more frequently than once per minute (and thus remaining warm) are unlikely to benefit from snapshot-based solutions. Also, REAP is ill-suited for the functions where the first invocation is not representative of future invocations although we do not observe such behavior in our studies. In this pathological case, the orchestrator can easily detect low working set pages usage and either repeat REAP's record phase or fall back to vanilla Firecracker snapshots for future invocations. For detection, the orchestrator could monitor the number of page faults that occur after the working set pages are installed, comparing this number to the number of pages in the working set.

5.4.3 Applicability to Other Isolation Technologies

We prototype REAP in Firecracker hypervisor but this approach seamlessly extends to other isolation technologies, such as container, language [147, 135], unikernel [43, 88, 102, 106], and gVisor [75, 64] based virtualization. Fundamentally, REAP requires guest memory to be managed as a set of virtual memory regions (or objects) on the host, which is the case for all of the aforementioned technologies. In this case, guest memory management can be offloaded to a monitor process via the `userfaultfd` mechanism, similarly to our prototype in Firecracker. However, we expect the technologies with simpler memory management, such as regular virtual machines that usually feature just one guest memory region, to benefit more from the REAP technique than the technologies with more complex memory management, e.g., language sandboxes. For example, Du et al. [64] show that a gVisor MicroVM (which is effectively a Golang-based language sandbox) exhibits a significant overhead upon a cold start, which is attributable to de-serialization of the snapshot state that is required before the guest memory file can be mapped into the main memory of the host.

5.4.4 Security Concerns

Similar to other snapshot techniques, spawning virtual machine clones from the same VM snapshot with REAP has implications for overall platform's security. In a naive snapshotting implementation, these VM clones may have an identical state for random number generators (i.e., poor entropy) and the same guest physical memory layout. The former problem may be addressed at the system level with hardware support for random number generation albeit the user-level random number generation libraries may remain vulnerable [3, 68]. The latter problem may lead to compromised ASLR, allowing the attacker to obtain the information about the guest memory layout. One mitigation strategy could be periodic re-generation of a snapshot (as well as the working set file and the trace file, for REAP). Alternatively, similarly to prior work on after-fork memory layout randomization [101], the orchestrator can dynamically re-randomize the guest memory placement while loading the VM's working set from the snapshot in the record phase of REAP. This would require modifying the guest page tables, with the hypervisor support, according to the new guest memory layout.

5.5 Related Work

5.5.1 Virtual Machine Snapshots

Originally, VM snapshots have been introduced for live migration before serverless computing emerged [46, 118]. The Potemkin project propose flash VM cloning to accelerate VM instantiation with copy-on-write memory sharing [145]. Snowflock extends the idea of VM cloning to instantiating VMs across entire clusters, relying on lazy guest memory loading to avoid large transfers of guest memory contents across network [95]. To minimize the time spent in serving the series of lazy page faults during guest memory loading, the researchers explore a variety of working set prediction and prefetching techniques [154, 153, 156, 91]. These techniques rely on profiling of the memory accesses *after* the moment a checkpoint was taken and inspecting the locality characteristics of the guest OS' virtual address space. Compared to these techniques, our work shows that serverless functions do not require complex working set estimation algorithms: it is sufficient to capture the pages that are accessed from the moment the vHive-CRI orchestrator forwards the invocation request to the function until the orchestrators receives the response from that function. Moreover, we find that extensive profiling may significantly bloat the captured working set, hence slowing down loading of future function instances, due to the guest OS activity that is not related to function processing.

5.5.2 Serverless Cold-Start Optimizations

Researchers have identified the problem of slow VM boot times, proposing solutions across the software stack to address it. Firecracker [19] and Cloud Hypervisor [2] use a specialized VMM that includes only the necessary functionality to run serverless workloads, while still running functions inside a full-blown, albeit minimal, Linux guest OS. Dune [38] implements process-level virtualization. Unikernels [43, 88, 102, 106] leverage programming language techniques to aggressively perform dead code elimination and create function-specific VM images, but sacrifice generality. Finally, language sandboxes, e.g. Cloudflare Workers [52] and Faasm [135], avoid the hardware virtualization costs and offer language level isolation through techniques such as V8 isolates [144] and WebAssembly [17]. Such approaches reduce the start-up costs, but limit the function implementation language choices while providing weaker isolation guarantees than VMs. REAP targets serverless workloads but remains agnostic to the hypervisor and the software that runs inside the VM.

Caching is another approach to reduce start-up latency. Several proposals investigate the idea of keeping pre-warmed, pre-initialized execution environments in memory and ready to process requests. Zygote [58] was introduced to accelerate the instantiation of Java applications by forking pre-initialized processes. The zygote idea has been used for serverless platforms in SOCK [121], while SAND [20] allows the reuse of pre-initialized sandboxes across function invocations. These proposals, though, trade-off low memory utilization for better function invocation latencies. REAP is able to deliver low invocation latencies without occupying extra memory resources when function instances are idle.

Prior work uses VM snapshots for cold-start latency reduction, although snapshots have been initially introduced for live migration and VM cloning before serverless computing emerged [46, 95, 118]. Both Firecracker [19, 5] and gVisor with its checkpoint-restore functionality [75] support snapshotting. The state-of-the-art snapshotting solution, called Catalyzer, improves on gVisor's VM offering three design options for fast VM restoration [64]. Besides the ”cold-boot” optimization discussed in §2.4.1, Catalyzer also proposes ”warm-boot” and ”sfork” optimizations that provide further performance improvements but require memory sharing across different VMs. In a production serverless deployment, memory sharing is considered insecure and is generally disallowed [19, 15].

Replayable execution aims to minimize the memory footprint and skip the lengthy code generation of the language-based sandboxes by taking a snapshot after thousands of function invocations, exploiting a similar observation as this work – that functions use a small number of memory pages when processing a function invocation [147]. However, when loading a new instance, their design relies on lazy paging similar to other snapshotting techniques [5, 64]. In contrast, our work shows that the working set of the guest memory pages of virtualization-based sandboxes can be captured during the very first invocation, and that all future invocations can be accelerated by prefetching the stable memory working set into the guest memory.

5.6 Conclusion

Serverless functions exhibit a small working set of the guest memory pages that remains stable across different function invocations. Based on this insight, we present the REAP orchestrator that records a function's working set of pages, upon the first invocation of the function, and speeds up all further invocations of the same function by eagerly prefetching the working set of the function into the guest memory of a newly loaded function instance.

Chapter 6

Expedited Data Transfers

In this chapter, we address the second performance problem we identify in §3 related to data communication across different functions. After analyzing the state-of-the-art data transfer methods, we describe Expedited Data Transfers (XDT), a serverless-native API-preserving fabric, which enables direct memory-to-memory data transfers obviating the need for using external, serverful services for data passing.

6.1 Introduction

Serverless computing is expressive enough to support various applications such as video encoding [130, 70], compilation [89, 69] and machine learning [83], each of which consists of several functions connected in a workflow. However, the stateless and ephemeral nature of function instances mandates that functions communicate any intermediate and ephemeral state across the different functions that comprise the application logic. Inter-function communication generally happens when one function, the *producer*, invokes one or more *consumer* functions in the workflow and passes inputs to them. Crucially, the instances of the consumer functions are not known by the producer at invocation time because they are picked by the cloud provider's load balancer and autoscaler components on demand. Also, for many serverless applications, the amount of data communicated across function instances can be large, measuring 10s of MBs or more; examples include video analytics [70, 69, 130, 129], map-reduce style and database analytics [125, 117, 124], and ML training [83].

As described in §2, cross-function communication can happen in one of two ways. The first is by *inlining* the data inside function invocation requests. Because invocation requests traverse the cloud provider's autoscaling infrastructure (i.e, the request *control* *plane*), providers limit the maximum size of inlined data to hundreds of KBs or a few MBs to mitigate the impact of large payloads on the forwarding logic along the request/response's path [26, 74]. The second inter-function communication approach is via an intermediate storage service (e.g., AWS S3 or Google Cloud Storage), which requires the producer function to first store the data, then invoke the consumer, and subsequently have the consumer retrieve the data from storage. The indirection through a storage layer overcomes the payload size limitation of inline transfers, but introduces significant latency overheads and adds cost for the storage.

Researchers have identified the problem of efficient serverless communication and have proposed several solutions. Some seek to improve the performance of storage-based transfers through the use of tiered storage, such as combining an in-memory storage layer (e.g., Redis) with a cold storage layer (e.g., S3) [103, 136, 129, 116]. While tiered storage can somewhat improve performance over a single storage layer, the general disadvantages of through-storage indirection remain. Others try to enable direct function-to-function communication, but do so in a way that is incompatible with the existing autoscaling infrastructure and may pose security risks [142, 150].

Our work focuses on the problem of seamless, high-performance serverless communication that is non-disruptive with existing serverless infrastructure. To that end, we introduce Expedited Data Transfers (XDT), a serverless communication substrate that allows direct communication between two function instances in a manner that is secure, flexible and compatible with the autoscaling infrastructure used by cloud providers. At the heart of XDT is an explicit separation of the control plane used for function invocation, which is tightly integrated with the autoscaling infrastructure, from the data transfer itself. In simplest terms, with XDT, the producer function buffers the data that needs to be transferred in its memory and transfers a secure reference to the data inline with the invocation to the consumer function. The consumer then directly *pulls* the data from the producer's memory.

More concretely, XDT defines a short-lived namespace of objects with the same lifetime as the function instance. This namespace can be accessed by subsequent function instances through secure references that do not expose the underlying infrastructure to the user code. XDT exploits the insight that the lifetime of individual function instances as controlled by the keep-alive policies, implemented to keep function instances active to reduce the chance of cold starts, exceeds the lifetime of intermediate state required across function invocations.

XDT naturally supports a variety of inter-function communication patterns, including producer-consumer, scatter (map), gather (reduce), and broadcast. By requiring only minimal modifications at the end-points of the existing function invocation control plane, XDT is fully compatible with the deployed autoscaling infrastructure. Moreover, unlike inline transfers, XDT is not limited to small transfer sizes. And unlike through-storage transfers, XDT avoids high-latency data copies to and from a storage layer and the associated financial cost of storage usage.

We prototype XDT in vHive/Knative [143] by extending Knative queue-proxy components with XDT support. The XDT-enabled queue proxies buffer to-be-transmitted objects at the producer side until one of the consumer function instances (chosen by the Knative autoscaling infrastructure) pulls the object, passing it to the consumer function's user code. Since we are unable to modify the proprietary components of a commercial cloud, we evaluate our proposal by deploying a XDT-enabled Knative cluster on an AWS EC2 node with fast access to AWS S3. Using a set of microbenchmarks, we show that XDT delivers superior performance versus through-storage transfers via S3 for all of the aforementioned communication patterns in serverless computing.

The main contributions of our work are as follows:

- We show that existing inter-function communication methods fall short of serverless demands for high performance and low cost. The most general serverless communication approach, through-storage transfers, carries latency and bandwidth overheads of 4.3× for objects of 100KB versus inline transfers, which are limited in size to at most few MBs.
- We introduce XDT, which uses control/data separation to pass a secure object reference to a consumer function instance as part of an invocation request, and delegates to the consumer pulling the data from the producer's memory. XDT supports a variety of inter-function communication patterns and is fully compatible with serverless autoscaling infrastructure.
- We demonstrate that XDT is flexible and fast with 1.8-12.3× lower latency and higher effective bandwidth versus through-storage transfers with S3 on AWS. Evaluation of XDT with three real-world serverless applications shows 1.14-2.71× end-to-end speedups.

6.2 Serverless Communication Requirements

We now list the requirements for an ideal inter-function data communication method.

1. *High performance:* low latency and high bandwidth across a full range of transfer sizes.

2. *Seamless integration with autoscaling:* inter-function communication should work naturally with the existing autoscaling infrastructure.

3. *API compatibility:* the communication method should require no or minimal changes to the user code and should support both passing-by-value and passing-by-reference APIs.

4. *Security:* must keep sensitive provider information, including IP addresses of function instances and the underlying topology, hidden from untrusted user code.

As discussed in Section 2.4.2, existing inter-function communication methods are unable to meet all of these requirements, prompting an alternative solution, which we introduce next.

6.3 Expedited Data Transfers (XDT)

6.3.1 Design Insights

We exploit three insights that lead us to the design presented below. The first insight concerns *control/data separation*. Inline transfers in today's serverless clouds transfer the data along with the function invocation message, which results in the inlined object traversing the entire control plane of the function invocation and forces providers to impose strict limitations on maximum size of inlined objects.

A better communication method would separate the control (function invocation) from the data transfer. Doing so would naturally unburden the control plane without impacting the functioning of the autoscaling infrastructure. The challenge is doing so without resorting to a storage service, which is what existing through-storage transfers rely on. We address this challenge with the help of the second insight.

The second insight is that the data that need to be transferred between instances are ephemeral, with lifetimes on the order of a few seconds [90, 89]. Hence, the data lifetime is much shorter than the keep-alive period of serverless functions, which is typically in the order of minutes, to increase the likelihood of using warm invocations [19, 133].

Based on the above, we make one final insight: instead of using a storage service to communicate data across function instances, a producer (upstream) instance can simply buffer the data in its own memory and have the consumer (downstream) instance pull from it. This insight forms the foundation for XDT, presented next.

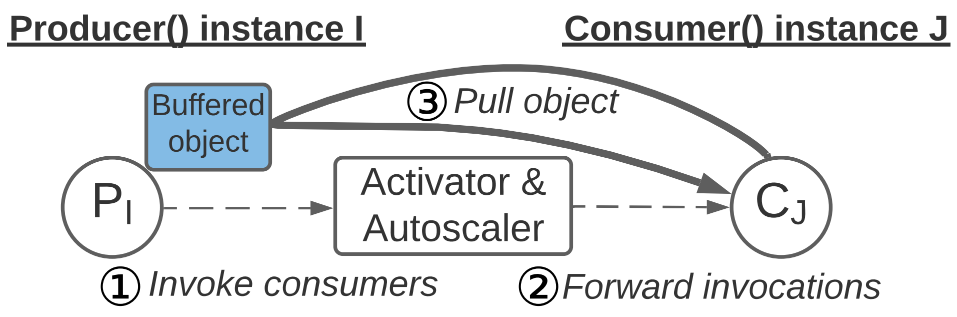


Figure 6.1: XDT architecture overview.

6.3.2 Design Overview

We introduce Expedited Data Transfers (XDT), a serverless-native high-performance data communication fabric that meets all four of serverless communication requirements (§6.2): high performance, compatibility with autoscaling, standard API for transferring data in serverless, and security.

Following the insights developed in §6.3.1, XDT splits the function invocation plane into a control and data planes. Crucially, the control plane is unchanged, matching the existing serverless architecture (Fig. 2.1), thus allowing the autoscaling infrastructure to take the load balancing decisions for each incoming invocation by steering the invocation to the least-loaded instances of a function. The control plane carries only the function invocation control messages, i.e., RPCs. The data plane is responsible for transferring the objects.

In simplest terms, a producer function instance in XDT buffers the data to be communicated to one or more consumer functions in its own memory and transfers a secure reference to the data inline with the invocation to the consumer function(s). The consumer(s) then directly *pull* the data from the producer's memory. XDT fundamentally replaces push-based data transfers, in which the producer pushes the data through the activator or through a storage layer, with a pull-based approach, whereby the consumer pulls the data directly after the control plane has made its decisions.

Fig. 6.1 describes XDT operation. Let us assume two serverless functions, a producer and a consumer, each of which may have any number of instances at any point in time. As in the case with existing communication methods, the producer logic invokes the consumer function while passing a data object as an argument. However, in contrast to the existing systems (§2.3), in XDT, consumer function invocations travel to the activator separately from their corresponding objects ①, which remain buffered at their source. After contacting the autoscaler as needed, the activator chooses the

API Call	Description
`rsp := invoke(URL, obj)`	Invoke a function
`ref := put(obj, N)`	Buffer an object locally
`obj := get(ref)`	Fetch a remote object

Table 6.1: XDT API description.

instance of the consumer function, to which the activator forwards the invocation for processing (2). Once the invocation arrives at the target instance, the instance can pull the object from the producer instance (3), using the secure reference enclosed in the invocation message.

6.3.2.1 XDT Programming Model

The XDT programming model features a minimalist yet expressive API (Table 6.1) that supports all three essential communication patterns, namely invoking a function (synchronously or asynchronously via a queue service, e.g., AWS SQS), scattering and broadcasting objects to several consumers, and gathering the output of several functions. The XDT API is fully compatible with the API supported by production clouds, such as AWS Lambda and S3's Boto3 [28].

The blocking call, `invoke()`, invokes a function by its `URL`, passing a binary data object `obj` by value. Upon invocation, the API of the XDT SDK is responsible for buffering the object at the producer side until the consumer function instance, chosen by the autoscaling infrastructure, pulls it. In this case, the consumer function starts processing *after* the object is transferred to the consumer instance.

Next, XDT supports the standard non-blocking (asynchronous) interface, which is similar to a common key-value store interface like in AWS S3 [28], namely `get()` and `put()` calls. In contrast to using a storage service, with XDT, the sender instance of the producer function can finish the invocation before one of the consumer instances retrieves the transmitted object. To de-couple the function invocation and the data transfer interfaces, XDT introduces *XDT references* as a first-class primitive. When the producer function calls `put()`, the runtime returns an XDT reference to a specific object while retaining an immutable copy of the object.[1] When the consumer needs to read this object, it calls `get()` that pulls the object from the remote server. Each reference is

[1]Note that during non-blocking transfers, the producer function's user code allocates the object, with the XDT SDK only holding references to it.

associated with a user-specified number of retrievals `N` of that object, which complete before the object can be de-allocated at the producer instance. From a user perspective, references are just opaque hashes that do not expose any information regarding the underlying provider infrastructure, and that can be neither generated nor manipulated by user code.

The above programming model allows to seamlessly port serverless applications, e.g., those implemented for AWS Lambda or Knative serverless platforms, with a set of the corresponding wrapper functions. To demonstrate the API's portability, we implemented XDT SDKs for applications written in Python and Golang and deployed in a Knative cluster.

6.3.2.2 XDT Semantics & Error Handling

Function invocations in modern serverless offerings, like AWS Lambda and Azure Functions, by default, provide the *at-most-once* semantics [71, 93, 96], i.e., a function invocation may execute not more than once even in the presence of a failure.[2] Hence, the provider is responsible for exposing the runtime errors to the user logic to handle them [30, 29, 114, 42]. Error handling logic may vary based on the function composition method. The user can compose the functions as a direct chain (e.g., the producer makes a blocking call to the consumer) or chain the functions in an asynchronous workflow. In the latter case, an *orchestrator* invokes the functions within the workflow. The orchestrator can be provider-based (e.g., AWS Step Functions [27] and Azure Durable Functions [112]) or a third function that drives other functions. To handle some failures, re-execution of several functions, i.e., a sub-workflow, may be required. In this case, the first function of the sub-workflow must be re-invoked with the same arguments as the original invocation. In this case, the user is responsible to pass the first function's context throughout the sub-workflow down to the function that can detect its failure.

Handling of XDT-related failures follows the same approach. We describe an XDT failure scenario in a two-function workflow with one producer function and one consumer function, which can be recursively generalized to an arbitrary workflow. Crucially, the lifetime of an XDT object is connected to the lifetime of the producer instance, thus a shutdown of a producer instance leads to immediate de-allocation of all the objects, retrievals of which have not started.

[2]The user can construct primitives with at-least-once semantics by combining primitives with at-most-once semantics and re-try logic. Prior work also shows constructing primitives with the exactly-once semantics [96].

For blocking invocations, i.e., the ones invoked with the `invoke()` API call, the producer instance stays alive waiting for the response from the consumer, and may decide to re-invoke the consumer invocation if the previous invocation returns an error. For the non-blocking invocations, an XDT transfer may fail if the producer instance is killed (e.g., due to exceeding the maximum invocation processing time) before the transmitted object is retrieved by a consumer instance. For example, it is possible that the producer function returns success before the transfer is complete, which is followed by the instance shutdown. However, in this case, the consumer function receives the corresponding error when executing XDT `get()`. We show that the invocation of the consumer can follow the at-least-once semantics approach. To guarantee correct execution of the entire workflow, the consumer needs to re-invoke the workflow starting from the producer function. Hence, the user code in the consumer function should forward this error to the corresponding entity (i.e., the orchestrator or the driver function) that can re-invoke the producer with the same, original arguments. For example, if AWS Step Functions orchestrator is used, the user can define a custom fallback function to handle a particular error code [30].

6.4 Implementation

We prototype XDT in vHive [143], an open-source framework for serverless experimentation that is representative of production clouds. vHive features the Knative programming model [11] where a function is deployed as a containerized HTTP server's handler (further referred to as *function server*), which is triggered upon receiving an HTTP request, i.e., RPC, sent to a URL assigned to the function by Knative. Each function instance runs in a separate Kubernetes pod atop a worker host (bare-metal or virtualized) in a serverless cluster.

6.4.1 XDT Prototype in vHive/Knative

We start by describing the implementation of the different software layers of the prototype, required to support blocking function invocations with XDT, followed by a discussion of support for the non-blocking XDT API.

6.4.1.1 XDT Software Development Kit (SDK)

XDT relies on an SDK to implement the API, bridging the user logic and the provider components that perform the transfer. At the producer instance's side, the SDK splits the original invocation request into two messages, namely a control message and an object, which comprises the transferred data. The SDK creates and adds an XDT reference to the gRPC request as an HTTP header. The reference comprises an encrypted string, containing the IP address of the pod where instance's function server is running, and the object key, which is unique for that pod. Encryption prevents the user code from obtaining the IP addresses of function instances.

At the consumer instance's side, the SDK reconstructs the original request, joining the control message and the object (after the latter has been pulled), before invoking the consumer function in the same way as with the vanilla serverless API.

6.4.1.2 Control and Data Planes

XDT extensively uses gRPC [8] both for the control and the dataplane to leverage the benefits of HTTP/2. Given that the XDT control plane messages are short, the control plane communicates over single-shot gRPC requests. In contrast, the dataplane needs to transfer large messages, for which it uses gRPC streaming. For streaming, we select the chunk size of 64KB based on the available performance guidelines [1]. We chose gRPC due to its support for a wide range of programming languages, high performance and full compatibility with the HTTP/2 protocol.

6.4.1.3 Provider Components Extension

XDT requires the following logic inside the provider's infrastructure. We extend the Knative queue proxy (QP)for object buffering (§2.3). QP is a minimal auxiliary provider container, written in Golang. It is deployed per function instance and shares the pod with the function server. The added logic increases the QP memory footprint by 2MB.

We also extend the QPs with two pools of pre-allocated memory buffers: one for buffering the outbound data (transferred from the corresponding function server), another for buffering the inbound data (the one arriving from other functions). Upon receiving an XDT transfer request from the producer function, the QP assigns one of the buffers from its pre-allocated pool to service the transfer. Once the transfer is finished, the corresponding buffer is returned back to the pool. A similar mechanism is used for buffering the inbound data.

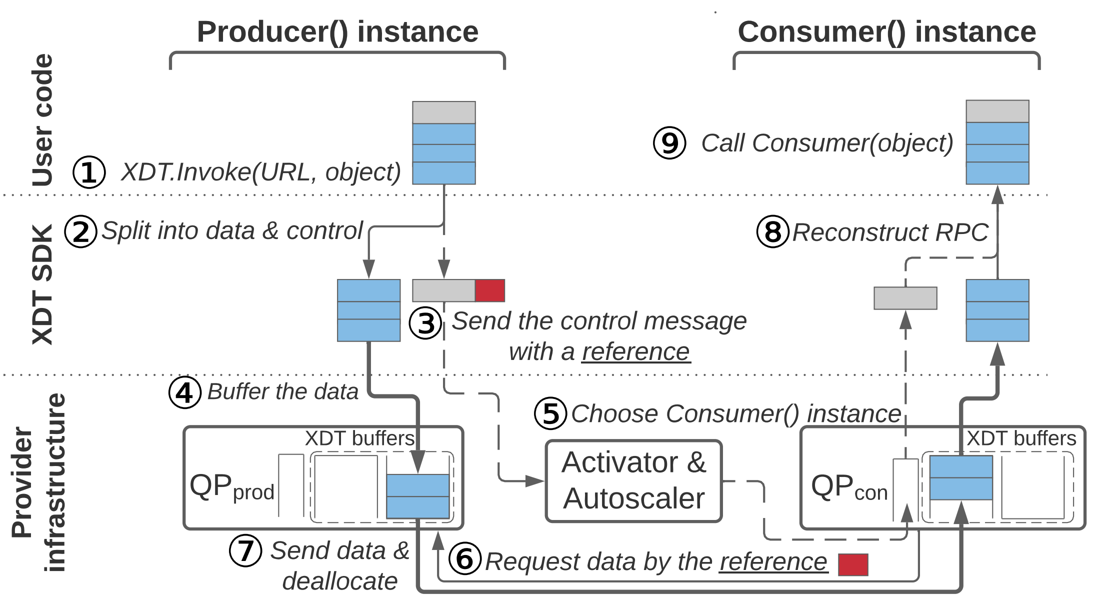


Figure 6.2: XDT operation in a single producer single consumer scenario (only the request path is shown). The dashed arrows show the control plane, the solid lines show the data plane, and the thick solid lines show data streaming in the data plane.

Because a QP, being a minimal provider container, might be online long before the function server during a cold start, we deploy the following performance optimization. We let the QP retrieve the object on behalf of the consumer function server, instead of the consumer SDK, to overlap retrieving the request with booting the function instance.

6.4.2 XDT Operation

6.4.2.1 XDT `invoke()` Operation

Fig. 6.2 shows the request path in the XDT infrastructure following an `invoke()` call. (1) when the caller function needs to call another function it invokes the SDK. (2) the SDK splits the request into two parts, the XDT object and the control plane message that carries the reference to the object. (3) the SDK sends the control message to the activator and (4) streams the object into a buffer inside the producer's QP (QP_{prod}). (5) the activator chooses the instance of the consumer and forwards the control message to the consumer's QP (QP_{con}). (6) QP_{con} extracts the reference from the header, decrypts the reference to extract IP address and the object key, and requests the data by sending a gRPC streaming request to the QP_{prod}, requesting the data by the object key. (7) QP_{prod} sends the data to the QP_{con} and de-allocates the chunks of the object when they are dispatched. (8) QP_{con} forwards the object to the SDK (9) that reconstructs the original request, and invokes the function handler. If the response is small, it follows the reverse control plane path through the two QPs and the activator.

We consider two alternative mechanisms for streaming the data from the producer function server via the two QPs to the consumer function server. One mechanism is *store-and-forward* (SF) streaming, in which each of the communicating components buffer the entire object before forwarding it to the component next in the chain. Although simple in implementation, the SF approach uses memory inefficiently, requiring deep buffers in each of the components. Another mechanism is *cut-through* (CT) streaming, which allows the chunks of a transmitted object to travel from the first to the last component without waiting for the entire object to be buffered in a single component. The CT approach is more memory-efficient, allowing shallow buffering in the components, albeit demanding a flow control for the entire entire chain of components.

6.4.2.2 XDT `get()` / `put()` Operation

Whereas `invoke()` is a synchronous call, the two other calls of the XDT API – `put()` and `get()` – are asynchronous. While the operation of `put()` and `get()` is similar to `invoke()`, there are a few important differences. The first difference is that `put()` returns an XDT reference for the object to the user logic. The producer function may pass this reference, like any other string field, to any function that belongs to the same user. Once the consumer function calls `get()` using the delegated reference, the SDK retrieves the object by sending a streaming gRPC request directly to the producer instance (i.e., to a gRPC server inside the SDK), using the IP address and the key in the reference.

The asynchronous `get()/put()` API can be used not only for invocations but also for large responses as well. The response path follows the control plan path in the reverse order and is used only with small (inline) replies, i.e., <6MB in AWS Lambda. In the case of a large reply, the XDT-enabled consumer creates a reference to the response object through a `put()` call and includes the reference in the response. Upon receiving the response, the producer can retrieve the response payload through a `get()` call.

6.4.3 Flow Control

The XDT design relies on the availability of pre-allocated buffers in both QP_{prod} and QP_{con} components to offer high performance data transfers. If buffers are unavailable, the system needs to engage a flow control mechanism to pace the sending components before the downstream buffers free up. Fortunately, gRPC streaming works on top of TCP and can rely on its flow control without any changes to the XDT logic, which only needs to buffer and forward the object's chunks along the component chain. Hence, if the number of transmitted objects exceeds the number of available buffers, the following transfers are paused, resulting in the user code blocking in the corresponding XDT API call.

6.5 Methodology

6.5.1 Evaluation Platform

Due to the close-source nature of commercial cloud infrastructure, we prototype and evaluate XDT in Knative [11]. We deploy a Knative cluster that features XDT-enabled queue-proxy containers on an AWS EC2 node, similarly to prior work [90, 103, 150], thus ensuring low access time to AWS S3. We use a single bare-metal `m5.metal` instance in the 'us-west-1' availability zone, to evaluate the baseline and the XDT-enabled serverless settings. This instance features Intel Xeon Platinum 8175 3.1GHz with 96 SMT cores, 384GB RAM, EBS storage, and a 25Gb/s NIC.

Using the vHive experimentation framework [143], we set up Knative 0.23 in a single-node Kubernetes 1.20 cluster [13], running all deployed functions, Knative autoscaling components, and Istio ingress [9] on a single physical node. In all experiments, we emulate a stable serverless workflow where enough active instances are present at all times – i.e., there are no cold start delays during the measurements. We achieve this by deploying functions with a fixed number of instances.

6.5.2 The vHive Measurement Framework

We employ the vHive measurement methodology [16], which supports both end-to-end benchmarking and detailed latency tracing, using OpenTelemetry [14] distributed tracing modules for traces collection and Zipkin [18] for latency breakdown visualization and analysis.

The vHive framework features a service, called invoker, that injects requests in a common format for all of the studied workloads and waits for the responses from the corresponding workflows, reporting the end-to-end delays. The user code of workloads is annotated with OpenTelemetry calls, which report various delays to a trace collector Zipkin server, including the internal overheads of Knative serverless infrastructure.

Unless specified otherwise, we report average end-to-end latency computed from 10 measurements. For microbenchmarks, which do not have any computational overheads except network processing, we report *effective bandwidth* of a data transfer as the size of the transferred object size divided by the measured end-to-end latency.

6.5.3 Baseline and XDT Configurations

Our baseline is the through-storage communication approach, which is unencumbered by transfer size limitations inherent in inline transfers. We use AWS S3 for this purpose, which matches the baseline configuration used in prior work [89, 90, 150, 125, 83]. Unless specified otherwise, we configure XDT queue proxies to use the cut-through streaming method while featuring enough buffers to avoid producer-side contention; namely, up to 32 1MB buffers for `invoke()` and up to 40 5MB buffers for `get()`/`put()`.

6.5.4 Microbenchmarks

We use a number of microbenchmarks, implemented in Golang 1.16, each of which evaluates one of the data transfer patterns commonly used in serverless computing (§6.3.2.1), namely producer-consumer (1-1), scatter, gather, and broadcast. All these patterns comprise various numbers of instances of the producer and the consumer functions communicating one or more objects from the former to the latter. From here on, by saying a producer (consumer), we mean a producer (consumer) function instance.

6.5.5 Real-World Workloads

Similarly to prior work [103, 90, 129, 89], we deploy and run three real-world data-intensive workloads widely used in serverless computing. Each workload is comprised of multiple functions, deployed with Knative Serving [12], that call each other using the blocking interface, i.e., a caller function waits for the callee to respond. Each of the workloads uses one or more data transfer patterns to communicate across functions. Both XDT and the S3-based baseline use the same communication API: `invoke()`, `get()` and `put()` (§6.3.2.1).

We configure the workloads to exploit maximum compute parallelism in each workflow. For example, in video analytics, the video decoder function invokes the recognizer function for each frame concurrently, leveraging as many parallel recognizer function instances as necessary. All workloads are containerized, ported to vHive/Knative, and will be released by the time of publication.

Note that our applications are distributed with function invocations running concurrently, e.g., the mapper and reducer invocations in the MapReduce benchmark. Hence, when inspecting the latency breakdown of their workflows with Zipkin, the critical path of its execution might be different for different configurations. For instance, if the critical path goes through a communication-dominant path in the baseline, then the critical path, and hence its latency breakdown, may change with XDT. We observe this behavior for the Stacking Ensemble Training and MapReduce workloads.

Video Analytics: This application is composed of three functions: the video streamer, frames decoder, and object recognizer, similarly to the setup in prior work [130]. This workload execution time depends on the efficiency of the scatter and the 1-1 patterns execution. The video streamer, implemented in Golang, sends a 2-second 30fps 1080p video fragment, originally residing in memory of the video streamer function, to the frames decoder. The decoder, implemented in Python 3, decodes all frames from the received video fragment, invoking the object recognizer function for every $5^{\text{-th}}$ decoded frame. The invocations to the recognizer are sent in parallel, allowing Knative to forward each of them to a separate function instance without queuing. Finally, the recognizer, implemented in Python 3, pre-processes the received frame and performs inference, using a pre-trained SqueezeNet 1.1 model [82, 126]. In all configurations, function invocations are performed via the `invoke()` API.

Stacking Ensemble Training: This is a distributed training application, implemented in Python 3. It fits the serverless programming model well due to its speed, low memory footprint, and low computational complexity [61, 59, 127]. This workload's execution is highly dependent on the efficiency of the broadcast and gather communication patterns.

The workload consists of 4 functions, namely the driver, the trainer, the reducer, and the meta-trainer. The driver orchestrates the entire workflow, by, first, saving a synthetically generated training dataset to AWS S3, via `put()`, and makes 16 concurrent invocations to the trainer function, resulting in 16 models trained in parallel, each on its own function instance. Each of these instances retrieve the dataset, via `get()`, which is equivalent to the driver broadcasting the dataset to these instances. Each trainer instance, then, performs model training with one of the four algorithms (linear SVR, lasso, K-nearest neighbors, and RandomForest), saving the trained model, via `put()`.

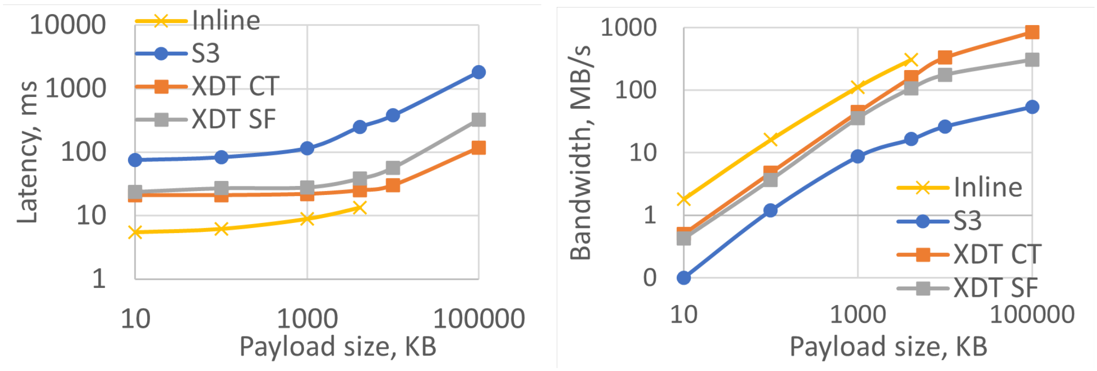
Once the driver receives responses for all the trainer invocations, it invokes the reducer that retrieves the trained models, in the gather pattern, via `get()`, and saves a joint model as a single object, via `put()`. Finally, the driver invokes the meta-trainer that retrieves the joint model and finishes the stacking ensemble's training by training the second-level (meta) model, saving it, via `put()`, followed by the driver retrieving the final model, via `get()`.

MapReduce: This is a workload whose execution time can be dominated by the data shuffling phase between the mapper and the reducer jobs in a modern serverless setting [89, 125]. This workload features a Python implementation of Aggregation Query from the representative AMPLab Big Data Benchmark [123] `1node` dataset. We scale this dataset down to 2% of the original size to model a large-scale setup, shuffling many small (<100KB) files from mappers to reducers, on a single AWS EC2 node. This model is conservative because the large-scale setup would have the same processing time, per input line, with more time spent data shuffling due to the larger data set. Empirically, we found that 40 mappers and 10 reducers is a balanced compute and shuffling configuration for the baseline system that shuffles data via S3.

The workload comprises three functions, namely the driver, the mapper, and the reducer. First, the driver invokes the mapper function once for each of the 40 input files in parallel. The mapper function fetches its input file from AWS S3 and produces one file per reducer, clustering its output by the `sourceIP` field. The mapper, then, saves its output via `put()`. After all mappers return their responses to the driver, it invokes the reducer function 10 times, each of which retrieves one file per mapper invocation, via `get()`. The final output of the query is written by the reducers to S3 in parallel.

6.6 Evaluation

We evaluate XDT in three steps. First, we conduct two sensitivity studies to find a best-performing configuration. Second, we study XDT scalability in various communication scenarios using microbenchmarks while measuring latency and bandwidth. Finally, we evaluate XDT against the baseline on three real-world serverless applications.



(a) Latency (lower is better) (b) Bandwidth (higher is better)

Figure 6.3: Average latency and effective bandwidth of the *1-1* workflow, sweeping the transferred object size.

6.6.1 XDT Sensitivity Studies

The implementation of XDT leaves a few design choices for empirical evaluation, namely the choice of the streaming mechanism – store-and-forward (SF) vs. cut-through (CT) – as well as the depth of the XDT buffers in the queue proxies. Both choices are important for supporting the `invoke()` API efficiently. We study the latency and effective bandwidth of `invoke()` in the single producer-consumer scenario (*1-1*) while varying the streaming mechanism and QP buffer depth.

6.6.1.1 The Streaming Mechanism

We start by investigating the efficiency of the SF and CT streaming mechanisms. Fig. 6.3 shows the latency and effective bandwidth of XDT configurations using SF and CT streaming, varying the size of the transferred objects from 10KB to 100MB. Next, we compare these results to the configurations that use AWS S3 and inline transfers.

We observe that both XDT configurations exhibit significantly lower latency and higher bandwidth than the configuration that uses S3. The gap between XDT and S3 grows when increasing the object size. The SF configuration delivers 3.2× and 5.6× lower latency when transferring 10KB and 100MB objects, respectively, yielding up to 306MB/s effective bandwidth vs. 117MB/s peak S3 configuration bandwidth. This demonstrates better bandwidth scaling of XDT with the size of the object, compared to S3.

Compared to SF, the cut-through streaming optimization accelerates XDT by 1.1× for 10KB objects and 2.8× for 100MB objects, yielding 3.6-15.7× lower latency

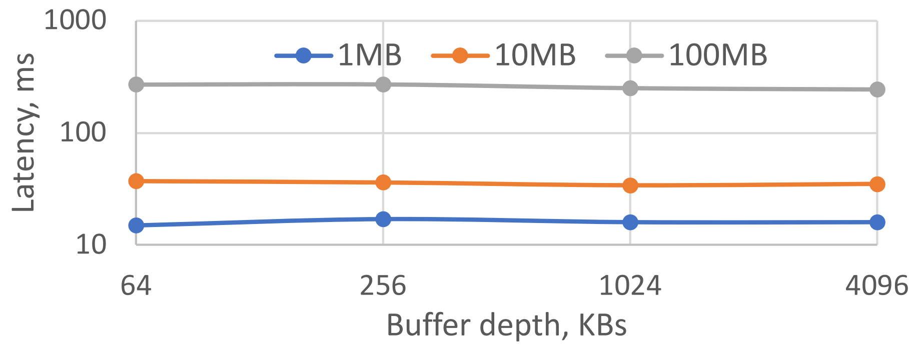


Figure 6.4: Average latency of XDT `invoke()` for objects of various size, as a function of queue proxy (QP) buffer depth.

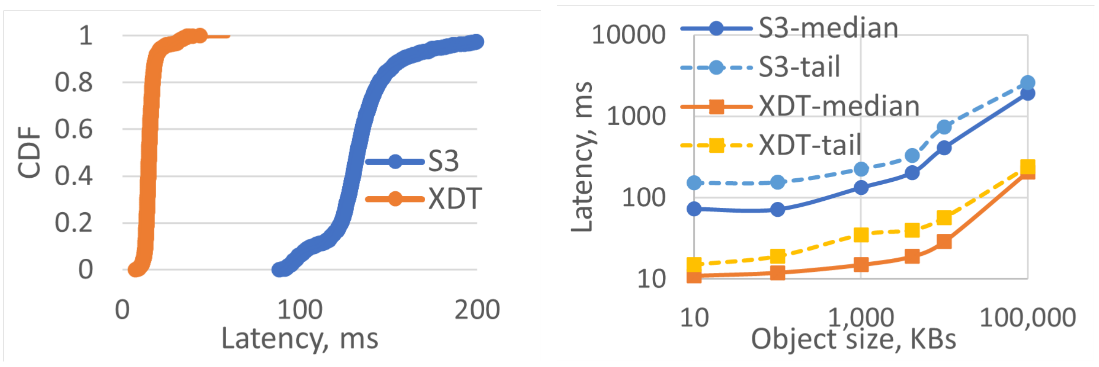
than with AWS S3. With a 100MB object, the XDT-CT is able to deliver 855MB/s bandwidth, which is within 13% of the bandwidth of vanilla gRPC streaming between two endpoints, communicating over a localhost interface. This result highlights the efficiency of the XDT-CT design, which we use in all further experiments referring to it simply as XDT.

Compared to XDT, inline transfers are 1.9-3.8× faster and have higher effective bandwidth. However, inline transfers impose a strict limitation on the maximum transfer size of <4MB in the gRPC/Knative setup and <6MB in AWS Lambda [26]. In contrast, XDT is not encumbered by object size limitations and supports large object transfers efficiently. Due to limitation on maximum object size for inline transfers, the rest of the evaluation does not consider this option. However, we note that for workloads that only transfer small objects and are limited to the 1-1 and scatter communication patterns, inline transfers represent a superior choice.

6.6.1.2 Memory Requirements for XDT

Next, we evaluate the added memory overhead for blocking XDT transfers, as non-blocking transfers do not require extra bookkeeping (§6.3.2.1). Fig. 6.4 shows the average transfer latency for three object sizes, namely 1MB, 10MB, and 100MB. We vary the buffer depth from 64KB, which holds a single object chunk, to 4MB, which holds 64 chunks per buffer.

Increasing the buffer size leads to a small decrease in the latency, leading us to conclude that the sensitivity of the XDT design to buffer depth is small. For instance, transferring 1MB object with the buffer sizes of 64KB and 4MB is performed in 15ms and 16ms, respectively. The latency gap for transferring a 100MB object is slightly



(a) Latency CDFs for 1MB objects.

(b) Median and tail latency, sweeping the object size.

Figure 6.5: Transfer latency cumulative distribution functions (CDFs), median and tail (99-th percentile) latencies for S3 and XDT in the *1-1* workflow.

larger, with 270ms for a 64KB buffer size and 243ms for a 4MB buffer.

Based on the obtained results, we configure XDT to use 1MB buffers in further experiments, as this size yields low latency for larger transfers, e.g., 250ms for a 100MB object, while imposing a relatively small memory overhead.

6.6.2 Microbenchmarks

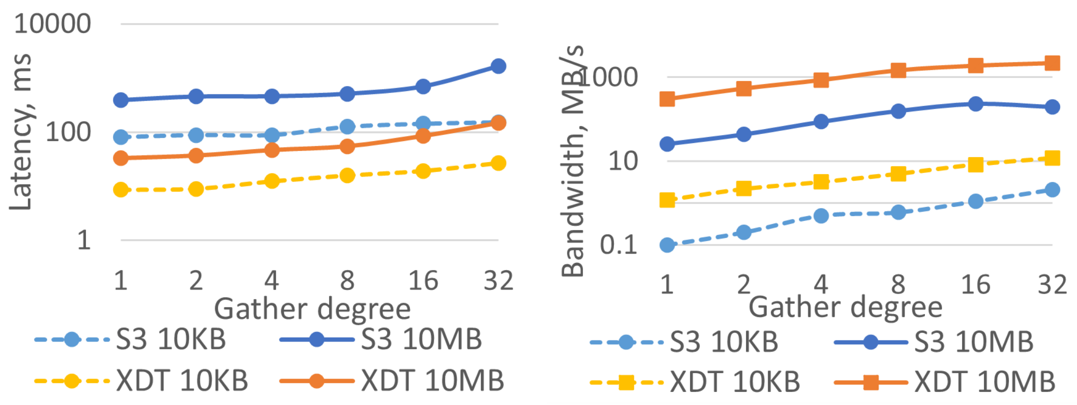
This subsection further quantifies the latency and effective bandwidth characteristics of XDT in common communication scenarios (§6.5.4): 1-1, gather, scatter, and broadcast.

6.6.2.1 Transfer Latency

Latency is a key metric for interactive, user-facing cloud services. We study the latency characteristics as a function of the transferred object size in the 1-1 producer-consumer scenario (Fig. 6.5).

Fig. 6.5a shows the latency cumulative distribution functions (CDFs) for XDT and AWS S3 transferring 1MB objects. We observe that XDT exhibits significantly lower median and tail (99-th percentile) latency, i.e., by 8.9× and 6.5× respectively, when compared to the configuration that uses S3. This indicates that the XDT architecture, and our implementation of it, is well-suited to latency-critical workloads, which are common in cloud.

Fig. 6.5b shows the average and the tail latencies of the two configurations while varying the size of the transferred objects. We notice that XDT consistently shows lower



(a) Latency (lower is better) (b) Bandwidth (higher is better)

Figure 6.6: Average latency and effective bandwidth of the *gather* workflow.

median and tail latencies, by 6.0-14.2× and 6.5-13.0× respectively, in comparison to S3.

6.6.2.2 Gather Communication Pattern

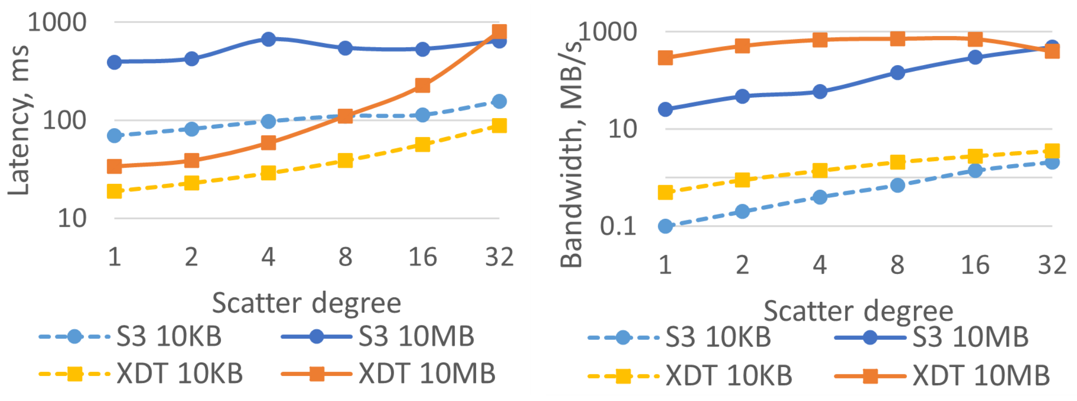
Gather, or reduce, is essential for applications with functions whose input is the output of several other functions and use the `put()`/`get()` API. We explore the latency and effective bandwidth of such transfers, sweeping the number of producers, i.e., the gather degree, from 1 to 32.

Fig. 6.6a shows the average latency of XDT and S3-based configurations while transferring small (10KB) and large (10MB) objects. For small objects, XDT achieves 5.6-9.3× lower latency compared to S3. XDT delivers effective bandwidth of up to 11.8MB/s vs. S3's peak of 2.1MB/s, as shown in Fig. 6.6b. For large object transfers, XDT delivers 8.2-12.3× lower latency, compared to S3, and provides effective bandwidth of up to 2175MB/s vs 230MB/s for S3.

6.6.2.3 Scatter Communication Pattern

Scatter, or map, is important when functions have a large fan-out of calls to other functions, passing the objects via the `invoke()` and the `put()`/`get()` APIs. We study the latency and effective bandwidth of these transfers (fig. 6.7), varying the number of consumers, i.e., the scatter degree, from 1 to 32.

Fig. 6.7a demonstrates the latency characteristics of XDT and S3-based systems in the scatter scenario. We observe that for small object transfers, XDT consistently shows 1.8-3.7× lower latency over the S3-based baseline. XDT delivers 0.5-3.6MB/s of



(a) Latency (lower is better) (b) Bandwidth (higher is better)

Figure 6.7: Average latency and effective bandwidth characteristics of the *scatter* workflow.

effective bandwidth vs. 0.1-2.1MB/s for the S3-based system, as depicted in Fig. 6.6b. For large transfers, XDT delivers 10.9-11.6× speedups for the low 1-4 scatter degree and more modest 2.4-4.9× speedups for the higher scatter degree of 8-16. For the scatter degree of 32, XDT matches the performance of the S3-based configuration, indicating a performance bottleneck in our prototype that requires investigation.

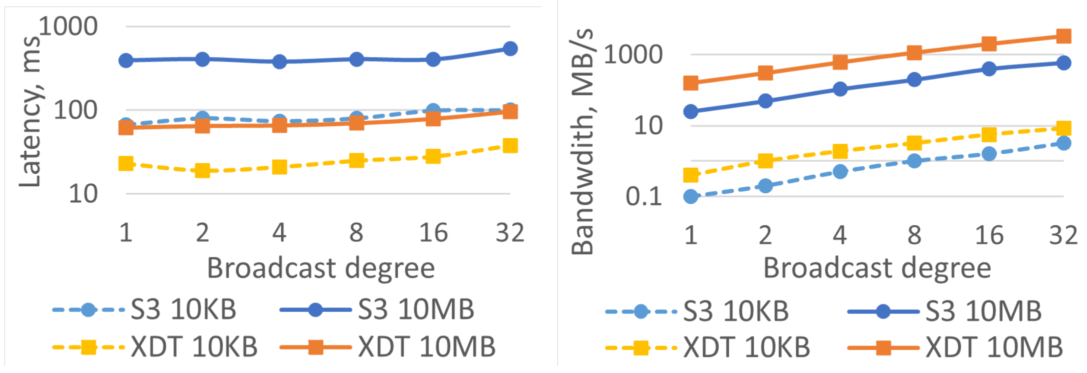
6.6.2.4 Broadcast Communication Pattern

Broadcast support is required for functions that distribute the same data among many consumers, via a single `put()` call followed by multiple `get()` calls with the *same* S3 key or XDT reference. We investigate the performance characteristics of the XDT and S3-based systems, sweeping the size of the transferred objects.

Fig. 6.8 demonstrates the latency and effective bandwidth characteristics of the XDT and S3-based configurations. For both small and large objects, XDT delivers 2.6-6.3× lower latency. The effective bandwidth of XDT is particularly high when transferring large objects, reaching 3331MB/s vs. peak 590MB/s delivered by S3.

6.6.3 Real-World Workloads

Next, we study three data-intensive applications (§6.5.5) and present their end-to-end latency along with a detailed breakdown of the sources of latency. After that, we present the estimation of cost reductions brought by XDT for developers of serverless applications.



(a) Latency (lower is better) (b) Bandwidth (higher is better)

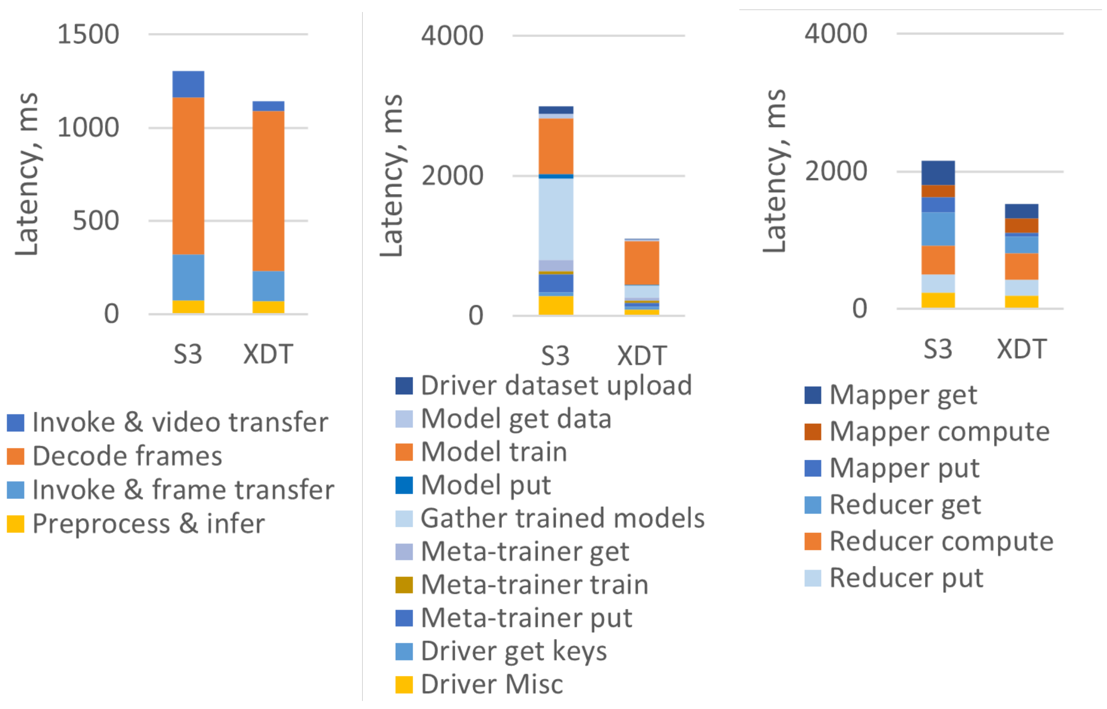
Figure 6.8: Average latency and effective bandwidth characteristics of the *broadcast* workflow.

6.6.3.1 Workload Performance Evaluation

Video Analytics spends 30% of its execution time in transferring the video fragment and the frames in the S3-based configuration. With XDT, this fraction decreases to 19%, yielding the overall speedup of 1.14× over the baseline. This speedup comes from 1.7× and 1.6× faster transmission of video and frames, respectively.

Stacking Ensemble Training spends 63% of execution time in data communication in the S3-based baseline. The largest fractions of data communication are the *gather trained models* and the *meta-trainer put* phases, accounting for 39% and 9% of the overall execution time. Using XDT allows to decrease both of these fractions to 12% and 3%, respectively, driving the overall data communication fraction down to 23%. Thus, XDT delivers a 2.71× speedup over the S3 baseline.

MapReduce shows 63% of execution time spent in communication for the S3 baseline. Moreover, 28% of the overall time is spent retrieving the original input from S3 and writing back the results to S3, which we do not optimize with XDT. The rest, i.e., 33% of time, are subject to XDT optimization. XDT allows to achieve 1.42× overall speedup vs. the S3-baseline. Note, however, that we plot the latency breakdown of the critical path across a highly parallel job, i.e., the longest-running mapper and reducer invocations. XDT's speedup is achieved due to a significant decrease in data shuffling, namely mapper-put and the reducer-get phases, which are sped-up by 3.9× and 2.0× respectively.



(a) Video Analytics. (b) Stacking Ens. Training. (c) MapReduce.

Figure 6.9: Latency breakdown of real-world workloads, deployed in XDT and S3-based systems.

6.6.3.2 Workload Cost Estimation for Developers

From the application developer perspective, the cost of executing a single invocation of a data-intensive serverless application consists of following two categories. The first fraction of the cost is a sum of `processing time` $\times$ `function memory footprint` product of each function invoked. For example, AWS Lambda charges application developers for `GB` $\times$ `seconds` while processing an invocation and a small fixed fee per invocation [25]. The second category is the storage cost, billed in `GB/month` as for AWS S3 [23].

Table 6.2 shows the costs associated with executing a single invocation of the three real-world applications we study. Using XDT instead of a storage service allows significant cost savings for developers, by reducing the processing time due to faster data transfers and removing the need for storage while passing ephemeral data across functions. For all three workloads, XDT reduces the invocation processing cost proportionally to cross-function communication phase acceleration in each function (§6.6.3.1). The overall cost of executing Video Analytics and Stacking Ensemble Training workloads is decreased by 43% and 51%, respectively, as both processing and

	S3 storage			XDT	Cost diff.
Workloads	Compute	Storage	**Total**	**Compute**	
Video analytics	53	23	**76**	**43**	**43%**
Stacking Ens. Training	159	41	**200**	**98**	**51%**
MapReduce	362	823	**1185**	**241**	**80%**

Table 6.2: Cost estimation (in $USDx10^6$) of executing a single invocation for an S3 storage based configuration vs. a configuration that uses XDT for data transfers, assuming AWS Lambda [25] and AWS S3 [23] pricing models.

communication becomes faster. Executing the MapReduce workload is particularly cheaper with XDT than for the baseline configuration, by 80%, due to the large amount of ephemeral data passed from the mapper function invocations to the reducer function invocations in the baseline storage-based configuration. With XDT, these ephemeral data can be passed directly between the instances of the mapper and reducer functions, imposing much lower costs.

6.7 Discussion

Prior work [90, 125, 103, 135, 136] considers a number of ephemeral storage service designs, aiming to provide high-performance transfers without imposing a large cost. Hence, some works [90, 125, 136, 116, 129] feature a multi-tier design where the first tier comprises a fast, but pricey, in-memory store while the second tier is slow and cost-optimized. To reduce the cost of the pricey in-memory tier, some proposals [90, 103, 116] employ machine-learning techniques in the control plane for intelligent object allocation, increasing the complexity of serverless infrastructure. Another line of work employs direct inter-function communication approaches [150, 142] – by exposing the IP addresses of function instances to the user code – that increases the attack surface and places the burden of load balancing and scaling on the user.

With XDT, it is possible to obviate the need for ephemeral storage *and* achieve the performance of an in-memory data store – the peak storage performance configuration – at the cost that is negligible compared to storage. By design, the XDT data-plane's performance is equivalent to a single direct data transfer initiated by the consumer side. While prior work shows [90, 89, 125] that the usage of storage can be prohibitively expensive due to its pricey in-memory tier. In contrast, the cost of transmitting an object over XDT is equivalent to the cost of 1MB bookkeeping per concurrent blocking transfer (§6.6.1.2) for the duration of the transfer; and the data transferred via the non-blocking API requires no extra bookkeeping at all (§6.3.2.1). Moreover, both performance and cost of XDT scales naturally with the number of instances that perform XDT transfers at any moment of time.

6.8 Related Work

XDT separates the control and data paths for inter-function communication and leverages a pull-based approach to scale transmission bandwidth by avoiding centralized bottlenecks. The idea of control and data plane separation is widely applied in the areas of software-defined networks and storage. Crab [92] and Prism [79] follow a similar separation to reduce the load on L4 and L7 load balancers, respectively. Delos [37] leverages a similar idea when solving virtual consensus in replicated systems, by separating the VirtualLog control plane from Loglets, which are pluggable data plane implementations).

Prior works [135, 149, 136, 116] consider extending serverless with a distributed shared memory (DSM) tier and pass references over the DSM around instead of data objects. In contrast to these proposals, the data objects transmitted via XDT are immutable, avoiding the complexity of supporting data consistency models.

Our work speeds up serverless application execution by rapidly moving data to compute. Alternatively, Shredder [155] suggests running compute operations directly at the storage tier. SAND [20] accelerates data communication proposing a hierarchical messaging bus, which also relies on serverless function instances co-location. Kayak [151] and Bhardwaj et al. [39] investigate the balance between moving data vs. moving compute, suggesting hybrid schemes to combine both. Despite the potential efficiency gains, today's commercial systems, e.g., AWS Lambda, tend to avoid serverless function co-location, as such policies may lead to hotspots [19, 36], instead relying on statistical multiplexing across the server fleet. Other works [72, 133] propose keep-alive policies to minimize the number of cold function invocations.

6.9 Conclusion

In modern serverless clouds, data-intensive applications' performance, which heavily depends on several functions communicating in a workflow, suffers from significant communication delays. We show that serverless communication methods used in production clouds and optimizations proposed by researchers fall short of serverless communication demands. In response, we introduce XDT, a high-speed API-preserving communication method that integrates seamlessly with the existing autoscaling infrastructure that underpins cloud deployments. XDT leverages control/data separation and secure references to provide low latency and high bandwidth in a variety of communication scenarios. A vHive-based XDT prototype accelerates real-world serverless applications by 1.14-2.71$\times$ over a production baseline.

Chapter 7

Conclusions and Future Work

This dissertation has shown that data movement is the key problem that limits the performance of today's serverless cloud architecture, due to its fundamental separation of compute and data management. To address this problem, we propose a data-centric architecture that unlocks fast, resource-efficient serverless clouds. Our work takes a holistic approach, embodying a comprehensive performance analysis that quantifies the data-movement problem, identifies the underlying cloud infrastructure subsystems that limit overall performance, and proposes two steps that implement the proposed data-centric serverless cloud architecture.

To analyze the performance of state-of-the-art commercial clouds, we propose a performance analysis methodology and build an open-source serverless benchmarking framework, called *STeLLAR*. STeLLAR enables accurate performance characterization of serverless deployments. STeLLAR is provider-agnostic and highly configurable, allowing the analysis of both end-to-end and per-component performance with minimal instrumentation effort. Using STeLLAR, we study three leading serverless clouds and reveal that data movement significantly reduces the overall performance of a modern serverless cloud. In particular, we identify that the long cold-start delays of launching new function instances and slow cross-function communication limit the overall performance.

To enable further performance analysis, we introduce an open-source full-stack framework called vHive, which integrates cutting-edge production technologies from the leading serverless providers in a complete, representative platform. To demonstrate vHive's utility, we further study the breakdown of cold-start latencies and pinpoint the root cause of the slow cold starts. We identify that this performance overhead occurs because of a long series of page faults, arising due to lazy paging in the host operating system. We also find that serverless functions operate over stable memory working sets across function invocations.

Using the insights obtained from the studies with vHive, we construct *REAP*, a light-weight record-and-prefetch mechanism, that captures the locations of each function's working-set memory pages in storage at the first function invocation and prefetches them into memory upon all following invocations of the same function. As a result, REAP allows to eliminate most of page faults upon a cold start and achieve significant speedups.

Finally, we address the data communication bottleneck by proposing an API-preserving serverless-native *XDT* fabric. XDT enables direct function-to-function transfers, where a trusted component of the sender function buffers the payload in its memory and sends a secure reference to the receiver, which is picked by the load balancer and autoscaler based on the current load. Using the reference, the receiver instance pulls the transmitted data directly from sender's memory. Effectively, XDT allows to achieve the performance of an in-memory transfer without imposing additional costs, as in the case of using a storage service.

Driven by the insights obtained with STeLLAR and vHive, the REAP snapshots and the XDT communication method pave the way for a high-performance and resource-efficient *data-centric* serverless cloud architecture, which co-designs serverless autoscaling capabilities and careful data management. In comparison to prior art, REAP demonstrates that low-latency cold starts may be achieved with conventional virtualization technologies, like AWS Firecracker, without compromising security like with the solutions that rely on memory sharing. Compared to the state-of-the-art cloud systems that require the developers to compose their applications out of serverless functions and traditional, "serverful" storage services for ephemeral data transfers, XDT allows to develop data-intensive applications in a pure serverless manner – without using any external services and the associated costs.

7.1 Future Directions

The serverless paradigm introduced a new programming model for cloud computing, starting the fundamental shift in cloud architecture. This thesis identifies and addresses the performance problems of state-of-the-art serverless systems, making serverless clouds more reactive to traffic changes and adding fast communication support for data-intensive applications. However, making serverless clouds energy-efficient leaves many system design challenges, providing opportunities for future research in cloud programming models, hardware acceleration, and data-locality optimizations.

7.1.1 Revisiting the programming model

Currently, developers structure their applications manually, by experimenting and adjusting the data-flow graph of their application based on the empirical data they collect. This task is onerous due to several reasons. First, modern cloud architecture is distributed and features a deep software stack. Second, the bulk of the stack, both serverless and classic Backend-as-a-Service infrastructure, is obscured from the service developers that have little visibility into the characteristics of their applications that matter for achieving predictable performance. Finally, the providers have no visibility into the code of the microservices and serverless functions that comprise the applications as well as the workload characteristics (e.g., bursty traffic), complicating timely autoscaling and cloud resources provisioning. It is necessary to devise a programming model and an optimizing compiler techniques for decomposing an application into a data-flow graph of autoscaling components (e.g., serverless functions and other services), and scheduling mechanisms to enable cloud application programming with performance guarantees.

7.1.2 Hardware offloads

To date, serverless software stack runs entirely on CPUs, leading to poor performance and large energy footprints. For example, monitoring and balancing load of cloud resources upon *every* serverless function invocation wastes a lot of CPU cycles. Another type of overhead lies in moving data, such function images or MicroVM snapshots from disaggregated storage to the target hosts when spawning new function instances. Offloading these operations to in-network processing units, smart NICs, RDMA, and in-CPU data streaming accelerators, some of which are already available in a modern datacenter, promises significant performance and energy gains. To support this shift, it is paramount to devise novel abstractions for exposing the capabilities of this hardware to cloud infrastructure and the user code inside serverless functions, while retaining the monitoring capabilities and providing high security as in today’s CPU-only clouds. It is important to develop new hardware primitives and programming model extensions, enabling a high-performance and energy-efficient system design.

7.1.3 Data locality aware scheduling

Modern serverless schedulers make no assumptions on data locality, placing functions independently from the location of their input data origin and disregarding the data dependencies across functions. There are three types of data in the context of serverless computing: function images used for instances initialization, input data ingested into a serverless application, and the ephemeral data transmitted across functions. It is key to explore the possibility of extracting this application-specific information at runtime or providing programming interfaces to communicate necessary information from the application programmers to the serverless scheduler. Using this information, serverless scheduler can implement intelligent policies and mechanisms, minimizing data movement while timely adjusting the amount of allocated resources and balancing the load across the system.

Bibliography

[1] Best message size for streaming large payloads. Available at https://github.com/grpc/grpc.github.io/issues/371.

[2] Cloud Hypervisor. Available at https://github.com/cloud-hypervisor.

[3] Entropy for Clones. Available at https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/random-for-clones.md.

[4] Firecracker-containerd. Available at https://github.com/firecracker-microvm/firecracker-containerd.

[5] Firecracker snapshotting. Available at https://github.com/firecracker-microvm/firecracker/blob/master/docs/snapshotting/snapshot-support.md.

[6] Fission: Open Source, Kubernetes-Native Serverless Framework. Available at https://fission.io.

[7] Fn project. Available at https://fnproject.io.

[8] gRPC: A High-Performance, Open Source Universal RPC Framework. Available at https://grpc.io.

[9] Istio. Available at https://istio.io.

[10] Kata Containers. Available at https://katacontainers.io.

[11] Knative. Available at https://knative.dev.

[12] Knative Serving. Available at https://knative.dev/docs/serving.

[13] Kubernetes: Production-Grade Container Orchestration. Available at https://kubernetes.io.

[14] OpenTelemetry: An Observability Framework for Cloud-native Software. Available at `https://opentelemetry.io`.

[15] Production host setup recommendations. Available at `https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md`.

[16] vHive Benchmarking Methodology. Available at `https://github.com/ease-lab/vhive/blob/main/docs/benchmarking/methodology.md`.

[17] Webassembly. Available at `https://webassembly.org`.

[18] Zipkin. Available at `https://zipkin.io`.

[19] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 419–434, 2020.

[20] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 923–935, 2018.

[21] Amazon. A Demo Running 4000 Firecracker MicroVMs. Available at `https://github.com/firecracker-microvm/firecracker-demo`.

[22] Amazon. Amazon EC2 Instance Types. Available at `https://aws.amazon.com/ec2/instance-types`.

[23] Amazon. Amazon S3 Pricing. Available at `https://aws.amazon.com/s3/pricing`.

[24] Amazon. AWS Lambda Function Scaling. Available at `https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html`.

[25] Amazon. AWS Lambda Pricing. Available at `https://aws.amazon.com/lambda/pricing`.

[26] Amazon. AWS Lambda Quotas. Available at `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`.

[27] Amazon. AWS Step Functions. Available at `https://aws.amazon.com/step-functions`.

[28] Amazon. Boto3 documentation. Available at `https://boto3.amazonaws.com/v1/documentation/api/latest/index.html`.

[29] Amazon. Error Handling and Automatic Retries in AWS Lambda. Available at `https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html`.

[30] Amazon. Error Handling in Step Functions. Available at `https://docs.aws.amazon.com/step-functions/latest/dg/concepts-error-handling.html`.

[31] Amazon. Using Container Images with Lambda. Available at `https://docs.aws.amazon.com/lambda/latest/dg/lambda-images.html`.

[32] Apache. OpenWhisk. Available at `https://openwhisk.apache.org/`.

[33] AWS re:Invent. A serverless journey: AWS Lambda under the hood, 2019.

[34] Azure Lessons. How Much Memory Available For Azure Functions. Available at `https://azurelessons.com/azure-functions-memory-limit/`.

[35] Baidu. The application of Kata Containers in Baidu AI Cloud. Available at `https://katacontainers.io/collateral/ApplicationOfKataContainersInBaiduAICloud.pdf`.

[36] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan Narayanaswamy. FirePlace: Placing Firecraker Virtual Machines with Hindsight Imitation. *Proceedings of the 34th Workshop on Machine Learning for Systems at NeurIPS 2020*, 3, 2021.

[37] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. Virtual Consensus in Delos. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 617–632, 2020.

[38] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–348, 2012.

[39] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. Adaptive Placement for In-memory Storage Functions. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 127–141, 2020.

[40] Ricardo Bianchini. Serverless in seattle: Toward making serverless the future of the cloud. Available at `https://acmsocc.github.io/2020/keynotes.html`.

[41] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCárthaigh, and Andrei Sandu. Restoring Uniqueness in MicroVM Snapshots. *CoRR*, abs/2102.12892, 2021.

[42] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Serverless Workflows with Durable Functions and Netherite. *CoRR*, abs/2103.00033, 2021.

[43] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the 2020 EuroSys Conference*, pages 32:1–32:15, 2020.

[44] Michael J. Cafarella, David J. DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. DBOS: A Proposal for a Data-Centric Operating System. *CoRR*, abs/2007.11112, 2020.

[45] CBINSIGHTS. Why serverless computing is the fastest-growing cloud services segment. Available at `https://www.cbinsights.com/research/serverless-cloud-computing`.

[46] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[47] Cloud Native Computing Foundation. CRI-O: Lightweight container runtime for kubernetes. Available at `https://cri-o.io`.

[48] Cloudflare Blog. It's Go Time on Linux. Available at https://blog.cloudflare.com/its-go-time-on-linux.

[49] ComputerWeekly.com. Storage: How Tail Latency Impacts Customer-facing Applications. Available at https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications.

[50] Containerd. An Industry-Standard Container Runtime with an Emphasis on Simplicity, Robustness and Portability. Available at https://containerd.io.

[51] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. *CoRR*, abs/2012.14132, 2020.

[52] CouldFlare. Cloudflare workers. Available at https://workers.cloudflare.com/.

[53] Alexandros Daglis, **D. Ustiugov**, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. SABRes: Atomic Object Reads for In-Memory Rack-Scale Computing. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, 2016.

[54] Daniel Krook. Five minute intro to open source serverless development with OpenWhisk. Available at https://medium.com/openwhisk/five-minute-intro-to-open-source-serverless-development-with-openwhisk-328b0ebfa160.

[55] Datadog. The State of Serverless 2020. Available at https://www.datadoghq.com/state-of-serverless-2020.

[56] Datadog. The State of Serverless 2021. Available at https://www.datadoghq.com/state-of-serverless.

[57] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[58] Android Developers. Overview of memory management. Available at https://developer.android.com/topic/performance/memory-overview.

[59] Federico Divina, Aude Gilson, Francisco Goméz-Vela, Miguel García Torres, and José F Torres. Stacking Ensemble Learning for Short-term Electricity Consumption Forecasting. *Energies*, 11(4):949, 2018.

[60] Docker. Use the device mapper storage driver. Available at `https://docs.docker.com/storage/storagedriver/device-mapper-driver`.

[61] Jie Dou, Ali P Yunus, Dieu Tien Bui, Abdelaziz Merghadi, Mehebub Sahana, Zhongfan Zhu, Chi-Wen Chen, Zheng Han, and Binh Thai Pham. Improved Landslide Assessment Using Support Vector Machine with Bagging, Boosting, and Stacking Ensemble Machine Learning Framework in a Mountainous Watershed, Japan. *Landslides*, 17(3):641–658, 2020.

[62] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, **D. Ustiugov**, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The Mondrian Data Engine. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017.

[63] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, **D. Ustiugov**, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios N. Pnevmatikatos. Algorithm/Architecture Co-Design for Near-Memory Processing. *ACM SIGOPS Operating Systems Review*, 2018.

[64] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, pages 467–481, 2020.

[65] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019.

[66] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. A

Review of Serverless Use Cases and their Characteristics. *CoRR*, abs/2008.11110, 2020.

[67] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. Serverless Applications: Why, When, and How? *IEEE Softw.*, 38(1):32–39, 2021.

[68] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael M. Swift. Not-So-Random Numbers in Virtualized Linux and the Whirlwind RNG. In *IEEE Symposium on Security and Privacy*, pages 559–574, 2014.

[69] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 475–488, 2019.

[70] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 363–376, 2017.

[71] Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of The 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 174–178, 1999.

[72] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 386–400, 2021.

[73] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite

for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, pages 3–18, 2019.

[74] Google. Google Cloud Functions Quotas. Available at `https://cloud.google.com/functions/quotas`.

[75] Google. gVisor. Available at `https://gvisor.dev`.

[76] Google Cloud. Configuring warmup requests to improve performance. Available at `https://cloud.google.com/appengine/docs/standard/python/configuring-warmup-requests`.

[77] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach. BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms. *CoRR*, abs/2102.12770, 2021.

[78] Hacker News. `clock_gettime()` Overhead. Available at `https://news.ycombinator.com/item?id=18519735`.

[79] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the Pain. In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 535–549, 2021.

[80] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[81] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[82] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡1MB model size. *CoRR*, abs/1602.07360, 2016.

[83] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards Demystifying Serverless Machine Learning Training. In *SIGMOD Conference*, pages 857–871, 2021.

[84] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 ACM Symposium on Cloud Computing (SOCC)*, pages 445–451, 2017.

[85] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*, pages 158–164, 2019.

[86] Jeongchul Kim and Kyungyong Lee. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.

[87] Jeongchul Kim and Kyungyong Lee. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*, page 477, 2019.

[88] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 61–72, 2014.

[89] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding Ephemeral Storage for Serverless Analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 789–794, 2018.

[90] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 427–444, 2018.

[91] Thomas Knauth and Christof Fetzer. DreamServer: Truly On-Demand Cloud Services. In *Proceedings of the 7th ACM International Systems and Storage Conference (SYSTOR)*, pages 9:1–9:11, 2014.

[92] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, pages 193–207, 2020.

[93] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[94] Kubeless. Kubeless: The kubernetes native serverless framework. Available at `https://kubeless.io`.

[95] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 2009 EuroSys Conference*, pages 1–12, 2009.

[96] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, 2015.

[97] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. Analyzing Open-Source Serverless Platforms: Characteristics and Performance. *CoRR*, abs/2106.03601, 2021.

[98] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential Serverless Made Efficient with Plug-in Enclaves. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, pages 14–19, 2021.

[99] Linux programmer's manual. Userfaultfd. Available at `https://man7.org/linux/man-pages/man2/userfaultfd.2.html`.

[100] Paul Litvak. How We Escaped Docker in Azure Functions. Available at `https://www.intezer.com/blog/research/how-we-escaped-docker-in-azure-functions`.

[101] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[102] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, pages 461–472, 2013.

[103] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 285–301, 2021.

[104] Pascal Maissen, Pascal Felber, Peter G. Kropf, and Valerio Schiavoni. FaaSdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 73–84, 2020.

[105] Linux man page. fio. Available at `https://linux.die.net/man/1/fio`.

[106] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 218–233, 2017.

[107] Marc Brooker at AWS re:Invent 2020. Deep Dive into AWS Lambda Security: Function Isolation. Available at `https://www.youtube.com/watch?v=FTwsMYXWGB0&t=782s`.

[108] Artemiy Margaritov, **D. Ustiugov**, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd International Symposium on Microarchitecture, (MICRO)*, 2019.

[109] Artemiy Margaritov, **D. Ustiugov**, Boris Grot, and Edouard Bugnion. Towards Virtual Address Translation via Learned Page Table Indexes. Workshop on ML for Systems at the 32nd Conference on Neural Information Processing Systems (NeurIPS), 2018.

[110] Artemiy Margaritov, **D. Ustiugov**, Amna Shahab, and Boris Grot. PTEMagnet: Fine-grained Physical Memory Reservation for Faster Page Walks in Public

Clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[111] Market Reports World. Serverless Architecture Market by End-Users and Geography - Global Forecast 2019-2023, 2019. Available at `https://www.marketreportsworld.com/serverless-architecture-market-13684687`.

[112] Microsoft. What are Durable Functions? Available at `https://aws.amazon.com/step-functions`.

[113] Microsoft. Azure functions, 2019. Available at `https://azure.microsoft.com/en-gb/services/functions`.

[114] Mikhail Shilkov. Making Sense of Azure Durable Functions. Available at `https://mikhail.io/2018/12/making-sense-of-azure-durable-functions`.

[115] MinIO. Kubernetes native, high performance object storage. Available at `https://min.io`.

[116] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the 2021 EuroSys Conference*, pages 228–244, 2021.

[117] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD Conference*, pages 115–130, 2020.

[118] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference*, pages 391–394, 2005.

[119] Goncalo Neves. Keeping functions warm – how to fix AWS Lambda cold start issues. Available at `https://serverless.com/blog/keep-your-lambdas-warm`.

[120] Stanko Novakovic, Alexandros Daglis, **D. Ustiugov**, Edouard Bugnion, Babak Falsafi, and Boris Grot. Mitigating Load Imbalance in Distributed Data Serving with Rack-scale Memory Pooling. *ACM Transactions on Computer Systems (TOCS)*, 2019.

[121] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 57–70, 2018.

[122] OpenNebula. OpenNebula + Firecracker: Building the future of on-premises serverless computing. Available at `https://opennebula.io/opennebula-firecracker-building-the-future-of-on-premises-serverless-computing`.

[123] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[124] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD Conference*, pages 131–141, 2020.

[125] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 193–206, 2019.

[126] Pytorch. SqueezeNet. Available at `https://pytorch.org/hub/pytorch_vision_squeezenet`.

[127] Smitha Rajagopal, Poornima Panduranga Kundapur, and Katiganere Siddaramappa Hareesha. A Stacking Ensemble for Network Intrusion Detection Using Heterogeneous Datasets. *Security and Communication Networks*, 2020, 2020.

[128] Allison Randal. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *ACM Comput. Surv.*, 53(1):5:1–5:31, 2020.

[129] Francisco Romero, Gohar Irfan Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. *CoRR*, abs/2104.13869, 2021.

[130] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. *CoRR*, abs/2102.01887, 2021.

[131] Samuel Karp. Deep dive into firecracker-containerd. Available at `https://speakerdeck.com/samuelkarp/deep-dive-into-firecracker-containerd-re-invent-2019-con408`.

[132] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1063–1075, 2019.

[133] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 205–218, 2020.

[134] Mikhail Shilkov. Serverless: Cold start war. Available at `https://mikhail.io/2018/08/serverless-cold-start-war`.

[135] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 419–433, 2020.

[136] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020.

[137] Bernd Strehl. Lambda serverless benchmark. Available at `https://serverless-benchmark.com`.

[138] **D. Ustiugov**, Theodor Amariucai, and Boris Grot. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021.

[139] **D. Ustiugov**, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios Pnevmatikatos. Design Guidelines for High-Performance SCM Hierarchies. In *Proceedings of the 4th International Symposium on Memory Systems (MEMSYS)*, 2018.

[140] **D. Ustiugov**, Plamen Petrov, M. R. Siavash Katebzadeh, and Boris Grot. Bankrupt Covert Channel: Turning Network Predictability into Vulnerability. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT), co-located with USENIX Security*, 2020.

[141] The Linux Foundation Projects. Open Container Initiative. Available at `https://opencontainers.org`.

[142] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, pages 16–29, 2020.

[143] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 559–572, 2021.

[144] V8. Isolate class reference. Available at `https://v8docs.nodesource.com/node-0.8/d5/dda/classv8_1_1_isolate.html`.

[145] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 148–162, 2005.

[146] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 443–457, 2021.

[147] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the 2019 EuroSys Conference*, pages 39:1–39:16, 2019.

[148] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 133–146, 2018.

[149] Stephanie Wang, Benjamin Hindman, and Ion Stoica. In reference to RPC: it's time to add distributed memory. In *Proceedings of The 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*, pages 191–198, 2021.

[150] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *Proceedings of the 10th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2021.

[151] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of the 18th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 633–651, 2021.

[152] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, pages 30–44, 2020.

[153] Irene Zhang, Tyler Denniston, Yury Baskakov, and Alex Garthwaite. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 1–12, 2013.

[154] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast restore of checkpointed memory using working set estimation. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE)*, pages 87–98, 2011.

[155] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*, pages 1–12, 2019.

[156] Jun Zhu, Zhefu Jiang, and Zhen Xiao. Twinkle: A fast resource provisioning mechanism for internet services. In *Proceedings of the 2011 IEEE Conference on Computer Communications (INFOCOM)*, pages 802–810, 2011.