Serverless Datacenter Applications

DISS. ETH NO. 30387

Serverless Datacenter Applications

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

MICHAL WAWRZONIAK

Master of Science in Computer Science, Princeton University

accepted on the recommendation of

Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Ana Klimović, co-examiner
Prof. Dr. Timothy Roscoe, co-examiner
Prof. Dr. Rodrigo Bruno, co-examiner

2024

Abstract

Serverless computing platforms, such as Function-as-a-Service (FaaS), have been gaining popularity and are considered by some as the future of cloud computing. Serverless applications benefit from highly elastic resources that are fast to instantiate, scale to thousands of instances and are based on fine-grained resource accounting. These attractive resources are bundled with the FaaS programming model, based on restricted event-triggered functions that applications can compose into complex workflows. This can provide benefits such as autoscaling and reduced management overhead. However, limitations such as the restricted function environment, limited execution duration, or restricted networking can be obstacles for some applications, such as distributed data analytics. Furthermore, these resources are not directly available to classic datacenter applications based on the network-of-hosts programming model.

This thesis proposes to unbundle the serverless resources from the programming model, and demonstrates that it is possible to use existing publicly available FaaS infrastructure to transparently provide serverless resources to datacenter applications. Boxer system is developed that, transparently to the application, provides the classic network-of-hosts model based on publicly available FaaS resources to enable serverless datacenter applications. It shows how such serverless datacenter applications can benefit from augmented ephemeral elasticity and can be instantiated on per-request granularity.

It is demonstrated that unmodified long-running datacenter applications can be transparently augmented with ephemeral elasticity based on FaaS, reducing the overprovisioning needed as application resources can scale faster in response to demands, such as bursty workloads. An unmodified microservice benchmark shows that its applications, when using Boxer-provided ephemeral elasticity, scale as quickly as with overprovisioned virtual machines. It is also shown that ephemeral elasticity can significantly reduce the failure recovery times in unmodified distributed datacenter applications as new resources can be

quickly provisioned compared to virtual machines. Furthermore, serverless datacenter applications can also be instantiated on per-request granularity, in particular, it is shown how distributed off-the-shelf data analytics systems can be instantiated on per-query granularity using publicly available FaaS resources instead of submitting queries to a long-running pre-selected and configured data processing system.

Zusammenfassung

Serverless Computing-Plattformen wie Function-as-a-Service (FaaS) erfreuen sich zunehmender Beliebtheit und werden von manchen als die Zukunft des Cloud-Computing angesehen. Serverless Anwendungen profitieren von hochelastischen Ressourcen, die sich schnell instanziieren lassen, auf Tausende von Instanzen skalieren lassen und auf einer feinkörnigen Ressourcenabrechnung basieren. Diese attraktiven Ressourcen sind mit dem FaaS-Programmiermodell gebündelt, das auf eingeschränkten, ereignisgesteuerten Funktionen basiert, die Anwendungen zu komplexen Workflows zusammenstellen können. Dies kann Vorteile wie automatische Skalierung und reduzierten Verwaltungsaufwand bieten. Einschränkungen wie die eingeschränkte Funktionsumgebung, die begrenzte Ausführungsdauer oder die eingeschränkte Vernetzung können jedoch Hindernisse für einige Anwendungen darstellen, wie z. B. verteilte Datenanalysen. Darüber hinaus sind diese Ressourcen für klassische Rechenzentrumsanwendungen, die auf dem Network-of-Hosts-Programmiermodell basieren, nicht direkt verfügbar.

Diese Arbeit schlägt vor, die serverless Ressourcen vom Programmiermodell zu entbündeln, und zeigt, dass es möglich ist, vorhandene öffentlich verfügbare FaaS-Infrastrukturen zu verwenden, um Rechenzentrumsanwendungen transparent serverless Ressourcen bereitzustellen. Es wird ein Boxer-System entwickelt, das transparent für die Anwendung das klassische Netwokr-of-Hosts-Modell auf der Grundlage öffentlich verfügbarer FaaS-Ressourcen bereitstellt, um serverless Rechenzentrumsanwendungen zu ermöglichen. Es wird gezeigt, wie solche serverless Rechenzentrumsanwendungen von erweiterter flüchtiger Elastizität profitieren und auf Anfragegranularität instanziiert werden können.

Es wird gezeigt, dass unveränderte, lang laufende Rechenzentrumsanwendungen transparent mit flüchtiger Elastizität auf der Grundlage von FaaS erweitert werden können, wodurch die erforderliche Überbereitstellung reduziert wird, da Anwendungsressourcen als Reaktion auf Anforderungen, wie z. B. stoßweise Arbeitslasten, schneller skaliert

werden können. Ein unveränderter Microservice-Benchmark zeigt, dass seine Anwendungen bei Verwendung der von Boxer bereitgestellten flüchtigen Elastizität genauso schnell skalieren wie mit überbereitgestellten virtuellen Maschinen. Außerdem wird gezeigt, dass flüchtige Elastizität die Wiederherstellungszeiten nach Fehlern in unveränderten verteilten Rechenzentrumsanwendungen erheblich verkürzen kann, da neue Ressourcen im Vergleich zu virtuellen Maschinen schnell bereitgestellt werden können. Darüber hinaus können serverless Rechenzentrumsanwendungen auch auf Anfragegranularität instanziiert werden. Insbesondere wird gezeigt, wie verteilte Standard-Datenanalysesysteme auf Anfragegranularität instanziiert werden können, indem öffentlich verfügbare FaaS-Ressourcen verwendet werden, anstatt Abfragen an ein lang laufendes, vorab ausgewähltes und konfiguriertes Datenverarbeitungssystem zu senden.

Contents

1 Introduction

1.1 Introduction

Serverless, most commonly offered as Function-as-a-Service (FaaS) computing platforms, provide applications with resources that are highly elastic, quick to instantiate, accounted at fine granularity, and without the need for explicit run-time orchestration (e.g., AWS Lambda [AWS17] cold instantiation times can be expected under 200ms [ABI+20], while billing is at 1ms granularity [AWS01]). This combination of properties underpins the success and popularity of the serverless FaaS paradigm. The interest is such that it has been claimed that it could be the next step in the evolution of cloud computing [SSSK+21], is offered by all major cloud providers [AWS17, Mic17, Goo01], and implemented by open source projects [Apa17, Ope24, Kna29].

However, these attractive resources are bundled with a restrictive programming model. FaaS applications are expected to be factored into collections of event-triggered functions that serverless platforms invoke in response to specified events. Functions can be composed into workflows [AWS01b, Azu01], or composition can be achieved by functions explicitly invoking other functions or generating appropriate events. Furthermore, serverless functions are constrained to limited lifetimes (e.g., currently 15 minutes on AWS Lambda [aws18]), have no persistent state, and lack direct function-to-function communication.

While well-suited and popular [SFG+20, JHA+23] for simple event-driven workloads (such as resizing an uploaded image), the restricted model of FaaS proved challenging for more complex applications.

1.1.1 Custom Serverless Systems

The potential of FaaS infrastructure motivated the development of many custom experimental serverless systems that provide functionality analogous to existing datacenter (VM-based) applications, but that worked around the limitations of the publicly available serverless platforms.

Many focused on embarrassingly parallel data analysis workloads [JPV+17] for computational imagining, solar physics, object recognition, feature extraction, distributed sort, lock-free stochastic gradient descent computation, distributed video processing [FWS+17a, AIVP18], software compilation, and unit tests [FRI+19]. Distributed system for executing large-scale dense linear algebra programs [SKV+20], ML training and serving [CFT+19, JGL+21, WDH+22], and distributed data analytics [MMA20, PCFDM20, PVS19] were explored.

A major bottleneck for many of these custom systems using the publicly available serverless infrastructure is the lack of direct communication among functions, which has wide-ranging implications for application design and performance. Systems such as Sprocket [AIVP18] and gg [FRI+19] relied on S3 [Ama17] cloud storage to transfer data between workers. Lambada [MMA20] and Starling [PCFDM20] propose sophisticated exchange operators using S3. Locus [PVS19], PyWren [JPV+17], NumPyWren [SKV+20] combined S3 with another faster storage layer based on ElasticCache [Red03]. Communicating through cloud storage adds significant overhead, so these systems proposed, among other things, complex ways to reduce the overhead and sophisticated optimizations to minimize the amount of data exchanged. Mu [FWS+17a], instead of using cloud storage, used a VM-based rendezvous server to relay messages between function workers. However, a VM-based rendezvous server introduces another scalability bottleneck and additional infrastructure that needs to be managed, defeating the objective of serverless.

1.1.2 Custom Serverless-Hybrid Systems

Another important use of the elastic serverless resources is to leverage them to quickly add resources to a long-running VM-based system, e.g., in response to load spikes. Several custom datacenter systems have been designed (or modified) to temporarily take advantage of serverless function resources this way. Beehive [ZWT+23] is a system based on modified JVM that can offload certain closures from a long-running JVM process to serverless functions to absorb load spikes. Sponge [SUE+23] is a custom stream processing engine that, in response to load spikes, can redirect some processing from long-running VM-based system to serverless functions. Similarly, Pixels-Turbo [BSA23] augments a long-running VM-based data analytics system with specialized serverless workers to accelerate processing in response to sudden load spikes. Crackle [PFCM23] models a custom unified query engine that can execute both on top of VMs and serverless functions to minimize the overall cost.

1.1.3 Custom Serverless Platforms

Despite their advantages, there is a growing belief that today's FaaS serverless platforms are not adequate, especially for tasks such as general data processing [HFG+19, WLZ+18a] since running queries often requires features that are missing, such as caching, support for direct communication among functions, and persistent state. One approach involves redesigning serverless platforms from scratch and developing an entirely new FaaS platform to be run on VMs. Built with a focus on transactional workloads, Anna [WSH19] is an elastic caching system built on top of a pre-existing key-value store [WFLH18] that can be used to implement stateful functions platform [SWL+20a] and transactional causal consistency for serverless functions [WSH20a]. Nightcore [JW21] is another custom FaaS runtime that focuses on eliminating overheads present in other serverless platforms to reduce function invocation latencies and sustain higher invocation rates to support microsecond-scale microservices that can include chained functions invocation patterns. Faasm [SP20] is another custom stateful serverless runtime that provides additional abstractions to functions, including shared memory, using WebAssembly [HRS+17] to enforce memory isolation. Although such experimental platforms may have more appropriate foundations to implement target applications, such as primitives for exchanging data between functions, these

custom platforms likely sacrifice completeness, lack maturity relative to the publicly available platforms, and have no clear path to deployment at scale.

1.2 Serverless Datacenter Applications

As discussed above, *custom serverless systems* benefit from the publicly available serverless platforms but can be limited by their restrictions, such as the function-to-function communication bottleneck. They also lack features and maturity relative to their datacenter (VM-based) application counterparts (e.g., Lambada is not as complete as Apache Spark.)

Custom serverless-hybrid systems augments specialized systems with elasticity available on serverless platforms. This approach is also limited by restrictions of the available serverless platforms, furthermore it does not generalize to unmodified off-the-shelf datacenter applications.

Lastly, *custom serverless platforms* are of research value and can provide a direction for the future, at the same time, they are unlikely to have a path to deployment at the scales that would allow full evaluation and use by other (possibly experimental) systems.

Given these challenges, this thesis proposes an alternative approach of ***serverless datacenter applications***, unmodified datacenter applications with serverless properties, and it demonstrates that **without any additional priveledges, it is possible to extend and leverage publicly available serverless platforms to run unmodified distributed datacenter applications** to augment long-running applications with ephemeral elasticity and to instantiate them on per-request granularity.

This demonstrates that the resources commonly bundled with the event-triggered-function model of FaaS can be unbundled and made available to datacenter applications based on the network-of-hosts model (Chapter 2).

1.3 Structure of the Dissertation

Chapter 2: Boxer System - describes the design and implementation of the Boxer system. Boxer is a system that both extends the existing publicly available serverless platform AWS Lambda with direct function-to-function networking, and provides the network-of-hosts model on top of functions. With Boxer, unmodified networked datacenter applications can run on top of serverless functions and VMs/containers.

The system is described mostly as it is as of writing this thesis. However, the design and implementation of the system evolved with time as the project progressed. The initial prototype was used for work in [WMBA21] to provide function-to-function networking to a custom version of Lambada [MMA20] system. However, since then, to support various unmodified applications, almost every component has changed, the NAT-traversal procedure changed twice, the process instantiation procedure evolved to include dynamic membership, and non-blocking connect, accept, and epoll became supported, which allowed latency-sensitive and high performance distributed applications to work as expected. Other details like role-based hostname resolution and file name renaming were also added when needed. Boxer was first used to show that it can run the DeathStarBench [GZC+19] microservices benchmark, then Apache Zookeeper, both used to demonstrate the ephemeral elasticity (Chapter 3.) Later, it was developed further so it could execute unmodified distributed systems such as Apache Drill and Apache Spark to show the possibility of running off-the-shelf datacenter applications such as data processing systems in AWS Lambda environment (Chapter 4.)

Chapter 3: Fast Ephemeral Elasticity for Datacenter Applications - makes a case for ephemeral elasticity to reduce resource overprovisioning. It then follows with detailed network microbenchmarks demonstrating that all the necessary network connectivity to realize ephemeral elasticity is present. Next experiments show ephemeral elasticity realized using Boxer to dynamically scale DeathStarBench to absorb short and unpredictable load spikes. The last experiment shows fast adding a Zookeeper node running in a function to temporarily compensate for a failure and reduce the time of lower availability for a long-running VM-based Zookeeper cluster. The experiments show that using Boxer and ephemeral elasticity can reduce datacenter overprovisioning.

Chapter 4: Per-request Systems - demonstrates using Boxer to instantiate distributed datacenter systems on per-request granularity. First, the benefits of Boxer-provided direct function-to-function networking are shown with microbenchmarks and the comparison of running TPC-H benchmark using Lambada based on S3 exchange operator and stream networking based on Boxer function-to-function networking. This is followed by a detailed set of TPC-H benchmarks for running distributed Apache Spark and Apache Drill in AWS Lambda using Boxer. Demonstrating that complex and mature datacenter applications can be used in serverless functions, removing the necessity of building serverless specific equivalents (e.g., feature-complete Apache Spark can be run instead of building systems like Lambada from the ground up) and that achievable performance is comparable to a class of VMs. The chapter concludes with a description of a design for a prototype that motivates one of the use cases of the per-request systems paradigm, which chooses what systems to instantiate based on a query. The use case is motivated by showing how much time and cost savings could be achieved by an oracle choosing only between two systems (Spark and Drill) and the number of workers.

1.4 Publications

The thesis material is based on the following publications:

- [WMBA21] M. Wawrzoniak, I. Muller, R. Bruno, and G. Alonso.
  "Boxer: Data Analytics on Network-enabled Serverless Platforms"
  In Conference on Innovative Data Systems Research (CIDR'21). 2021.

- [WBKA23] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso.
  "Ephemeral Per-query Engines for Serverless Analytics"
  In Joint Proceedings of Workshops at 49th International Conference on Very Large Data Bases (VLDB'23), Workshop on Serverless Data Analytics (SDA'23). 2023.

- [WMB+24] M. Wawrzoniak, G. Moro, R. Bruno, A. Klimovic, and G. Alonso.
  "Off-the-shelf Data Analytics on Serverless"
  In Conference on Innovative Data Systems Research (CIDR'24). 2024.

- [WMB+22] M. Wawrzoniak, I. Muller, R. Bruno, A. Klimovic, and G. Alonso
  "Short-lived Datacenters"
  arXiv:cs.DS:2202.06646, 2022.

- [WBKA24b] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso
  "[WIP] Imaginary Machines: A Serverless Model for Cloud Applications"
  In Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies, SESAME '24, Association for Computing Machinery. 2024.

- [WBKA24a] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso
  "Boxer: FaaSt Ephemeral Elasticity for Off-the-Shelf Cloud Applications"
  arXiv:cs.DS:2407.00832, 2024.

In addition, running and configuring the Apache Spark and Apache Drill TPC-H benchmarks in Chapter 4 was performed as part of Gianluca Moro's MS Thesis [Mor23].

2

Boxer System

2.1 Introduction

This chapter describes the Boxer system that is at the core of this thesis and is used in the following chapters. The chapter first describes the relevant aspects of today's serverless FaaS platforms, their relative resource scales in Section 2.2 and in Section 2.3 the differences between the FaaS event-triggered-function model and the classic network-of-hosts model of datacenter applications. Then Section 2.4 lists the concrete requirements and assumptions this places on Boxer. Lastly, the design and implementation of Boxer in AWS Lambda are described in Section 2.5 and Section 2.6, respectively.

In this thesis, *datacenter applications* refer to the most popular and 'classic' type of cloud applications that have evolved with or for cloud platforms based on networks of servers, virtual machines, or containers. Some of the canonical examples of such datacenter applications include Nginx [Ngi24] proxy, Apache Spark [ZXW+16] data analytics platform, or Memcached [Mem03] object cache, Apache Flink [Apa01] stream-processing, and many others that form the software foundation of the cloud application stack.
The term *serverless applications* can refer to different technologies, in this thesis, it refers more narrowly to applications designed to run on function-as-a-service (FaaS) platforms, such as AWS Lambda.

Serverless datacenter applications refers to the combination of properties of both types of applications. Serverless datacenter applications are unmodified datacenter applications that can also benefit from properties usually associated with serverless applications, such as high elasticity and fine-grained resource accounting. Although publicly available cloud infrastructure, such as AWS [Clo15], does not directly support such an option, some of the underlying resources that would enable it are already accessible in the form of microVMs [ABI+20] available through FaaS service such as AWS Lambda [AWS17].

Boxer system, described in this chapter, aims to enable serverless datacenter applications on a publicly available cloud, spanning long-running network VMs and serverless (FaaS) functions. It is a form of an interposition layer between cloud applications and platforms, or a *cloud overlay*, that can span AWS EC2 VMs and AWS Lambda, intercepts application execution, and emulates the network-of-hosts environment that applications expect when deployed in a conventional VM/container environment.

2.2 Relative Resource Scales

Relative to the available VMs, functions in today's FaaS platforms are considered lightweight and resource-restricted. They are small, the largest available AWS Lambda today has only 10GB of memory and 6 vCPUs [aws20]. From this perspective, running off-the-shelf datacenter applications that are designed to run natively on much larger VMs may seem like a mismatch.

However, it is worth considering that when some of the most popular datacenter systems used today were originally designed and developed, their target environment was closer to today's AWS Lambda functions than to today's AWS EC2 VMs. For example, Zookeeper became an Apache Software Foundation project in 2008 [Ini03] and was already commonly used by then, the Spark project started just a year later in 2009 [Apa03]. At that time, newly announced AWS EC2 High-CPU instance types [aws23a], c1.medium had 1.7GB of memory and 2 vCPUs, and c1.xlarge had 7GB of memory and 8 vCPUs. When Zookeeper was commonly used, and just before the Spark project started, the AWS EC2 'High-CPU Extra Large Instance' VM had comparable CPU and memory resources to today's AWS Lambda function.

While the available VMs and supporting platforms have significantly grown, the underlying system architecture of common datacenter applications (e.g., Zookeeper, Spark)

has remained largely unchanged. This indicates that (possibly with some amount of reconfiguration and tuning), datacenter applications can be a good match for contemporary *networked* functions of today's FaaS platforms.

2.3 Incompatible Execution Models

Datacenter applications and serverless applications are based on different programming models. Although both most often rely on Linux-based system stack, datacenter applications use the *network-of-hosts* model, while serverless applications rely on *event-triggered-functions* model (both described below.) This mismatch prevents existing datacenter applications from transparently and efficiently using the combination of VM/container and FaaS infrastructures to take advantage of the resources provided by serverless platforms.

2.3.1 Network-of-Hosts Model

The network-of-hosts model refers here to the classic and dominant datacenter model that has been used by datacenter applications for decades. Datacenter applications are built around the concept of concurrently long-running interconnected hosts that execute a collection of application processes that communicate via the network. Initially, hosts were based on network-connected physical servers, with the advent of cloud computing, they often became based on virtual machines or containers. Application processes running on hosts can be reached using various network addresses and names. Concurrently running application processes can use the network to exchange data to collectively 'compose' into the applications. The applications also receive external service requests via an externally accessible network.

In cloud platforms, from the application perspective, the networking is still predominantly based on IP addresses, hostnames, and L4 port numbers. The dominant process type is based on Linux processes, and the most common way the networking is exposed to it is via POSIX-like socket interfaces [pos24].

2.3.2 Event-Triggerend-Functions Model

The event-triggered-functions model refers here to the model promoted by the more recent FaaS serverless platforms such as AWS Lambda. Serverless applications are factored

into collections of functions that execute in response to named events. The functions are expected to be stateless and have limited execution time (e.g., up to 15 minutes in AWS Lambda and 30 minutes in Azure Functions.) Furthermore, the functions have restricted networking capabilities, they can access external services, but cannot access other concurrently executing functions or receive externally initiated network data. Serverless applications receive external service requests as events (that can contain a limited payload). The serverless functions can be 'composed' into complex applications by arranging them into event-driven dataflow graphs with auxiliary systems such as AWS Step Functions [AWS01b].

Publicly available FaaS platforms, such as AWS Lambda, base function instances on a restricted Linux environment, where each function can execute a collection of restricted Linux processes. The functions benefit from fast start times [ABI+20] and high invocation parallelism, making them attractive for highly burstable loads.

At first glance, these programming models appear to be very different, however, with the right perspective, these models might be thought of as dualities of each other, perhaps not unlike the classic thread-based and event-based programming models [LN79].

2.4 System Requirements and Assumptions

To achieve the vision of serverless datacenter applications, the Boxer system needs to satisfy the following requirements:

Support existing publicly available platforms: To test the feasibility and to avoid making unrealistic assumptions, Boxer must run on a public cloud platform available today. It must not assume any more privileges than those available to a regular tenant of a publicly available cloud platform. For example, it can use AWS EC2 as the virtual machine platform and AWS Lambda FaaS as the serverless platform.

Application transparency: To make the technique general and broadly applicable to a large set of existing applications, Boxer must not rely on modifying or specializing applications. When Boxer runs generic applications in environments for which they were not designed (e.g., FaaS, or combination of FaaS and VMs), the environment must be transparently emulated to match what the unmodified applications expect. Note that, the applications should function correctly without any modifications, but the goal is not

to provide complete environment emulation, such that the applications cannot detect any difference between their native environment and the environment emulated by Boxer.

Efficiency: Any introduced overhead (e.g., to emulate some aspects of the execution environment) must be sufficiently small to not violate the performance objectives or timing constraints of the user application. For example, the system must not reduce the level of application concurrency, and it must provide network connections to the application faster than its connection timeout to avoid getting trapped in connect-retry loops. Beyond not violating such assumptions, reducing system overhead is particularly important as FaaS instances can be relatively small, this makes system overhead play a proportionally greater role.

Orchestration compatibility: Cloud application deployments depend on orchestration systems. If Boxer required a new or modified orchestration system, it would reduce its usability and generality. Thus, there should be a way to use Boxer with unmodified popular orchestration systems, such as Docker Compose [Doc27], to pave a path to a possible straightforward adoption in practice.

System Assumptions: It is assumed that the same tenant uses Boxer and the guest application, so there is no incentive for the application code to escape the mechanism Boxer uses. It is assumed that the guest applications are *cooperative*, preventing applications from circumventing the mechanisms provided is not a goal at this point.
Boxer also does not aim to provide isolation between processes or applications, it is assumed that this is already provided by the underlying platform.
Lastly, the current system prototype does not need to provide complete transparency (e.g., applications can determine that they are running in Boxer, and Boxer does not nest). Currently, Boxer does not aim to cover all the unusual corner cases that do not affect the functioning of typical target applications.

2.5 Boxer System Design

The following sections provide an overview of the Boxer system and the reasoning that guided the design.

Boxer must provide the required *network-of-hosts* execution model to applications on top of the *event-triggered-functions* model of the underlying serverless platform. Furthermore,

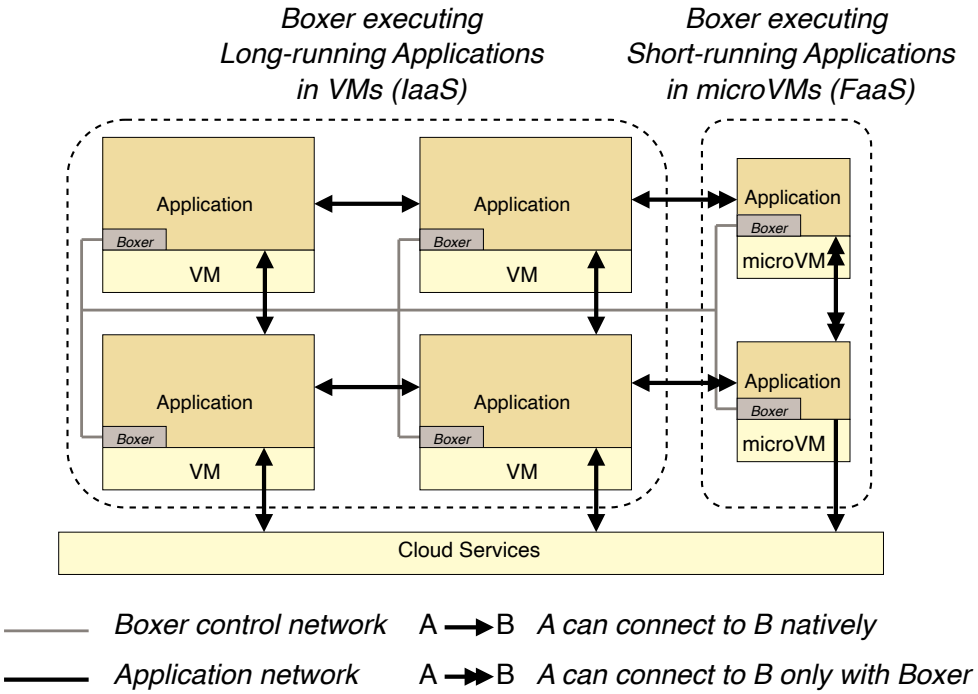

Figure 2.1: *An unmodified networked datacenter application spanning VMs and ephemeral microVMs under unified interface. The long-running components of the application are running in VMs and are (temporarily) augmented with ephemeral FaaS microVM-based resources using Boxer.*

it must be provided in a unified way across resources of serverless functions and long-running virtual machines/containers, if both types of resources are used. Since neither the applications nor the underlying platforms can be modified, Boxer is designed as an interposition layer placed between the cloud applications and the platforms, i.e., a form of *cloud overlay* (Figure 2.1). To achieve this, Boxer selectively *intercepts* the execution of the guest applications running on top of it, and uses the platform resources below to *emulate* the expected environment for the application.

2.5.1 Intercepting the Guest Application Execution

As described in the requirements, the system cannot rely on any modifications to the guest applications; therefore, the interposition must be enforced dynamically at runtime and not

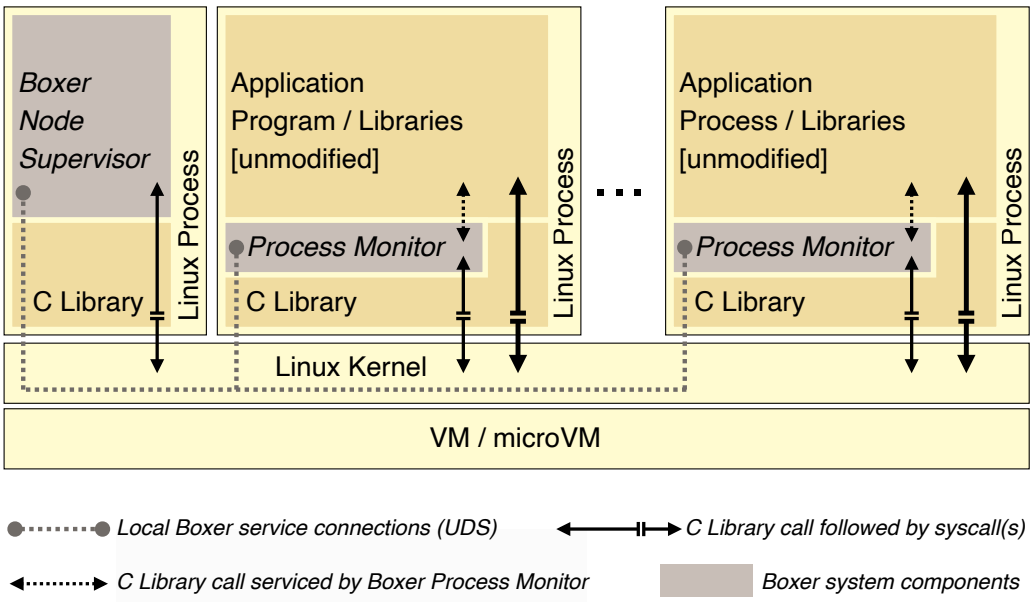

Figure 2.2: *Boxer node (VM, microVM, or a container) running Boxer Node Supervisor and application processes with Boxer Process Monitors selectively intercepting C Library calls.*

by modifying the application code or binaries. At the same time, target platforms include microVMs of publicly available FaaS services (AWS Lambda) that provide a restricted Linux-based environment. These restrictions eliminate approaches to trap application executions that rely on hardware-accelerated nested virtualization, modifying the Linux kernel, loading kernel modules, or using eBPF [eBP24] functionality. On the other hand, unprivileged userspace virtualization techniques based on full dynamic translation (e.g., QEMU [Bel05]) add too much overhead to be viable [WMUW19]. Furthermore, the system cannot rely on other standard methods used to intercept system calls of Linux processes, such as those based on `ptrace` or `seccomp` system calls, or Syscall User Dispatch [Sys17], because their use is restricted in the unprivileged environment of the publicly available FaaS platforms (AWS Lambda).

Given this combination of constraints, the choice was made to selectively intercept the application execution at the system C Library function call level. The interception of the calls is implemented by controlling the dynamic linking of application processes as they are started (described in §2.6.2). Hence, the current system targets applications

that dynamically link with the system C Library and do not directly issue system calls that Boxer must intercept. Compared to the other trapping approaches, interposition at the function call level incurs a negligible performance penalty of an additional function call [YTAI23], supporting the requirement for low system overhead. Although dynamic linking with system C Library is the standard way that applications are constructed, other techniques (e.g., based on lightweight selective dynamic binary rewriting) are worth investigating further to broaden the scope of the interception mechanism while keeping minimal performance penalty.

To minimize the overall performance overhead of the emulation, Boxer aims to limit the intercepted surface area to a minimum. The system is designed to reduce the number of intercepted functions and to delegate as much functionality directly to the underlying platform as possible. This also improves the fidelity of the emulation since fewer mechanisms must be re-implemented. In particular, Boxer leaves signals and memory management directly to the underlying platform, and the interception of system C Library calls is limited to the control path operations only. Most significantly, intercepting data path calls (e.g., `send`, `write`, `recv`, `read`, `sendfile`) and I/O notification calls (e.g., `epoll`, `select`) was avoided entirely, providing no-overhead performance for those operations.

Figure 2.2 shows the components running on each node in a distributed application with Boxer. The Boxer Process Monitor (Section §2.6.1) is the system component responsible for intercepting the necessary system C Library calls. It is loaded by the Node Supervisor (Section §2.6.2) into every guest application process. The Process Monitor is limited to a thin shim that interacts with the local Node Supervisor that provides services needed for the emulation.

2.5.2 Emulating the Unified Network-of-Host Model

Emulating the unified *network-of-hosts* model for all nodes participating in an instance of a Boxer network (VMs, containers, microVMs of FaaS) requires providing additional services to the guest applications. Boxer exposes all nodes, including the ephemeral FaaS microVMs, as networked hosts to the guest application. The guest application processes running on each of the participating hosts expect the ability to establish network connectivity between application processes running on all other hosts. To provide network connectivity between different hosts, Boxer must provide network transport between the hosts, manage network addresses, and provide hostname resolution. In the current version

of Boxer, these services are provided by sub-services of the Boxer Node Supervisor.

The separation of the functionality between the Node Supervisor and the Process Monitors was chosen for multiple reasons. First, the emulation for each guest process is not independent of other processes on the same host. Multiple processes share network namespace, can share file descriptors that correspond to the same network socket, or other local resources that require coordination. Therefore, emulation cannot be performed for each process independently, but must be coordinated with emulation for all processes on a host. Delegating control over possibly shared emulated resources to the local Node Supervisor provides a single point of control, greatly simplifying the mechanism. Secondly, implementing the additional logic required to provide connectivity to remote hosts in the Process Monitor can change the I/O patterns of the guest application, requiring a more complex mechanism to preserve transparency. For example, in guest applications using non-blocking I/O based on `epoll` mechanism, additional file descriptors would need to be added to the interest list, requiring `epoll_ctl` and `epoll_wait` functions to be intercepted to provide transparency. This would result in an additional complex mechanism implementing I/O signaling and would introduce overhead in the data path of the guest application. As described in Section §2.6.3, delegating the network transport setup to the Node Supervisor service reduces the complexity, does not require intercepting the guest application's data path, and improves the emulation fidelity by reducing the number of interfaces to be emulated. Furthermore, aggregating the services for all guest processes in an external process simplifies maintaining and sharing of the control network between Boxer nodes.

This separation of functionality between stateless and thin Process Monitors and a Node Supervisor providing the services to all local processes requires a communication channel capable of request-response commands, sending file descriptors, and signaling some I/O notifications with minimal interference with the guest application. Unix domain sockets are used for the former two because they can be used to send file descriptors between processes. For the latter, a technique of delivering the required signals to Process Monitors using marked local stream sockets was developed (they are later referred to as *signal sockets*) that requires minimal mechanism in the Process Monitor and is compatible with the guest using blocking and non-blocking I/O. In the future, some functionality may be handled by shared memory (e.g., hostname or filename map tables.)

2.6 System Implementation

This section describes the implementation of Boxer node components, Process Monitor (hereinafter PM) (§2.6.1), the Node Supervisor (hereinafter NS) (§2.6.2), and how they interact to provide the environment emulation. This is followed by a description of how Boxer networks are formed from a collection of Boxer nodes and what services are provided. The section ends with a description of how datacenter applications can be used with Boxer and standard container orchestration tools (§2.6.8).

2.6.1 Process Monitor (PM)

Boxer PM is responsible for selectively intercepting the execution of guest application processes. The reason for intercepting the execution of processes is to transparently emulate the required application environment. Every guest process, as it is loaded to be executed by Boxer, is first dynamically linked with the PM library. The library exports a set of symbols normally exported by the system C Library used by the underlying operating system. The library is linked between the platform system C Library and the application program and application libraries (Figure 2.2). This provides an interposition layer where the PM library can intercept the guest application execution by selectively intercepting system C Library calls made by the guest application processes. Currently, the intercepted calls are for stream socket (`socket`, `bind`, `connect`, `listen`, `accept`), network address and hostname resolution (`getaddrinfo`, `uname`), and file access (`open`, `close`). Together with their variants and companion functions, 17 functions are intercepted (additional functions are intercepted for application tracing purposes only.)

Notably, the selective interception avoids intercepting any data path functions such as `read`, `recv`, `send`, `write`, etc., or I/O notification functions such as `epoll`, `select`, etc. The intervention in the execution of the guest processes is limited to control plane operations for establishing network connections and network and file system naming. Once network connections are established or files are opened, there is no additional overhead during the data processing operations.

The functionality implemented in the PM is kept to a minimum. No state is persisted between intercepted calls, with most of the functionality handled by the NS services. PMs access their local NS by exchanging messages on a named Unix domain socket, referred to as *service connections* (shown in Figure 2.2).

For some of the intercepted calls, e.g., `getaddrinfo`, the functionality of PM is mainly limited to just parsing the arguments, sending an appropriate request message (in this case `NameLookupReq`) to the NS, and then formatting and returning the results to the caller.
For other intercepted functions, the procedure before sending a request on the service connections is more involved. For example, when handing an intercepted `accept` function, first, a non-blocking native accept call must be made before potentially sending an accept request to Boxer NS. This is because a local connection may be ready to be accepted and immediately returned to the caller [1] Alternatively, a *signal connection* (explained in Section 2.6.3) from Boxer NS may be accepted. Boxer NS may create a *signal connection*, a local connection from a reserved address , only used to trigger a matching I/O event delivery to the guest program if Boxer has a ready connection to be accepted by a guest process. The PM will first accept the signal connection, discard it, and then proceed to send the service request message to the NS (in this case `AcceptReq`) to attempt to receive a socket from Boxer (as a file descriptor sent over the service connection) and then return it to the guest program as the return value from the original `accept` call.
Handling `bind` and `connect` also requires more setup before issuing service requests, but the main mechanisms are implemented as NS services, leaving the PM stateless and relatively simple.

2.6.2 Node Supervisor (NS)

The Node Supervisor (NS) is an unprivileged process that runs in every node (VM, container, or microVM) participating in the Boxer network (Figure 2.2). The NS is responsible for managing the local guest application processes, maintaining the control network with other nodes in the Boxer network, and running the network service and the coordinator service.

When a Boxer node starts, the NS is the first process to start. In AWS Lambda FaaS, it is started directly by the platform runtime. When it is run in a container, it is started as the container entrypoint (`ENTRYPOINT` in Dockerfile), and in VMs, it can be started using a variety of platform-dependent methods. The NS starts the specified guest application programs with the configured arguments and environment variables and preloads all of their guest application processes with the Process Monitor (PM) library (§2.6.1). It then

[1] The future version of Boxer will handle local connections through NS to improve completeness (there are subtle corner cases currently not handled) and further reduce PM complexity.

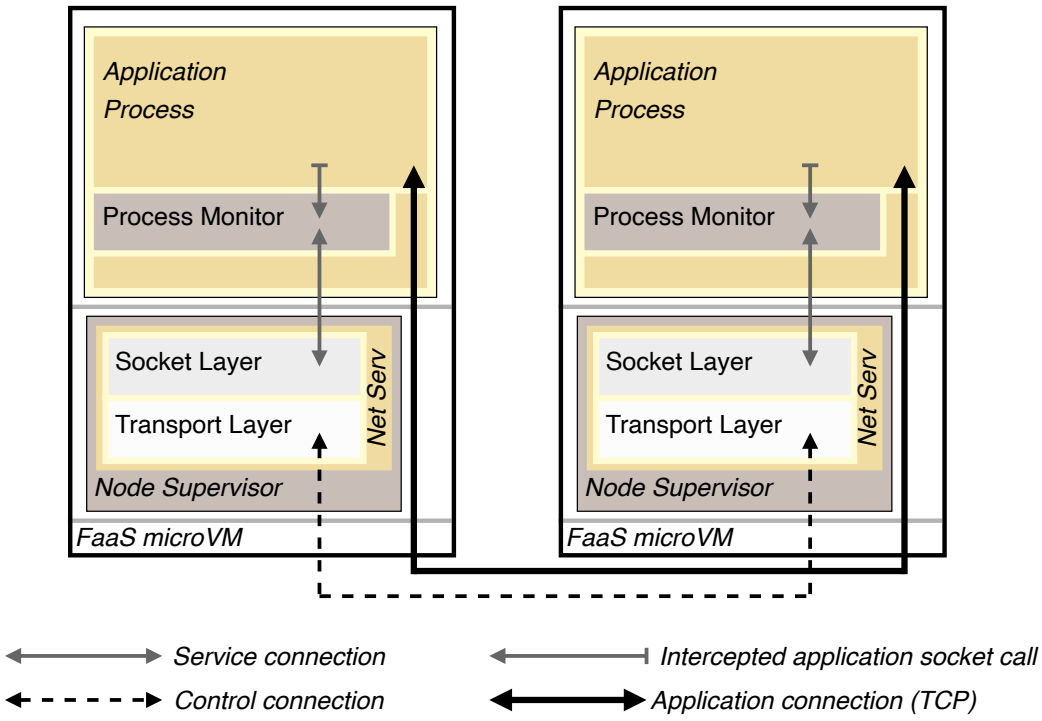

Figure 2.3: *Network Service is composed of the Socket Layer and the Transport Layer used to setup network connectivity between remote application processes. Node Supervisor does not process the network data path of application processes.*

listens for the PMs to open local service connections and issue requests. Unix domain sockets are used for the service channels, allowing some service requests to include file descriptors, as is necessary for creating new sockets for the application processes (described below).

The secondary role of the NS is to bootstrap and maintain the control network with other remote NSs. The control network is used to send and receive commands between remote nodes, including network setup requests. Currently, the control network is based on direct node-to-node connectivity that supervisors establish as nodes join the network. Internally, the supervisor forwards the local and remote requests to one of its local services, such as the networking or coordination service discussed in the following sections.

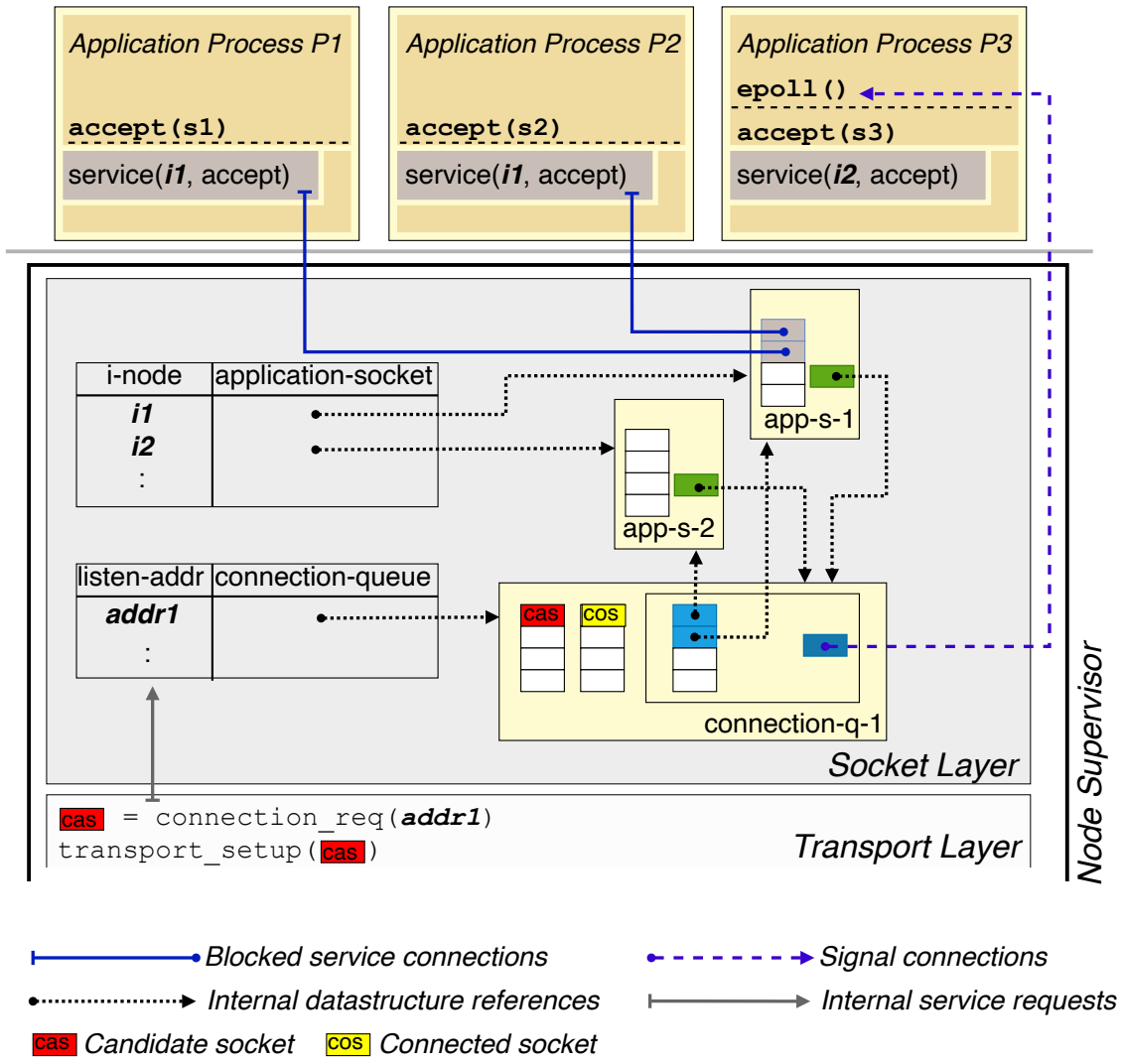

Figure 2.4: *A sample configuration state of core internal data structures of the stream socket layer for listening sockets showing the* `accept` *path of 3 processes. Detailed description in Section 2.6.3.*

2.6.3 Network Service

Conceptually, the NS's network service is composed of two layers: the socket layer and the transport layer (Figure 2.3). The socket layer provides the mechanism for creating and managing guest application sockets. The network transport layer (not meant to be interpreted with the OSI model terminology) provides network data delivery service

between Boxer nodes that can back the sockets created by the socket layer. This section first describes the NS's socket layer and then the transport layer.

2.6.3.1 Socket Layer

The socket layer interacts with the process monitors of the guest application processes from above and the transport layer from below. Currently, Boxer supports stream sockets, as these are the most commonly used in the target datacenter applications, however, other types may be added in the future.

In the case of the active side (the side initiating stream connections), when a PM intercepts a relevant socket call, e.g., `connect` call to establish a stream connection to a destination in the Boxer network, it will send a request to the network service to initiate the connection to the remote destination Boxer node. The socket layer will then request the connection from the transport layer. Once the connection is established, the application process will be unblocked with the correctly configured socket, and the guest application can proceed unaware of the behind-the-scenes process. Supporting only such an optimistic scenario would require minimal mechanisms in the socket layer. However, to support unmodified datacenter applications, Boxer stream socket layer must implement a mechanism to support the sufficiently complete socket interface with error handling, including non-blocking I/O, and interactions with other system features, such as the ability to share sockets between different guest processes (which guest applications of interest do use.)

To illustrate the approach Boxer takes to handle the more complex scenarios of the stream socket interface, consider Figure 2.4. The figure shows a subset of internal data structures on the passive side (the side listening for connections from the active side) of the socket layer. The figure illustrates the state configured for two listening guest application sockets (`app-s-1`, `app-s-2`). One of the sockets (`app-s-1`) is shared between two guest application processes `P1` and `P2`. The guest processes refer to the socket by file descriptors `s1` and `s2`, and are both waiting in blocking `accept` call to receive new connected sockets (or appropriate error codes). Process `P3` has a different socket (`app-s-2`) and uses non-blocking I/O to accept new connections. It is blocked on `epoll` call using a previously configured interest list, using `epoll_ctl`, that contains a file descriptor associated with the application socket `app-s-2`. However, in the example, both listening sockets are bound to the same local address `addr1` (such as `0.0.0.0:8080`).

This seemingly simple example illustrates a complex interaction of features used by datacenter applications that is not uncommon. Applications often use multiple processes to handle requests that arrive on new connections accepted on the same external address, processes send socket file descriptors between each other (for example a main process sends configured sockets to worker processes), high-performance systems are commonly based on non-blocking I/O event notification mechanisms, while at the same time blocking I/O is still used for simple control mechanisms. Boxer must support such scenarios to run unmodified datacenter applications.

Implementing a complete socket layer with a set of compatible userspace interfaces that is performant and compliant enough is challenging and requires extensive engineering effort, especially in the target constraint execution environment of FaaS microVMs. This practical challenge is addressed in Boxer by the strategy to reuse as many existing mechanisms of the host environment as possible and only implement minimal interventions to achieve the needed functionality. This surgical approach is a way to rely on the host environment to handle as much of the complex interfaces as possible to preserve compatibility and minimize overhead. For example, Boxer's strategy of *selective interposition* and *emulation gadgets* lead to supporting non-blocking sockets while avoiding implementing any non-blocking I/O notification mechanisms such as `epoll` that are known to be very complex to implement correctly and are critical for application performance.

The socket layer keeps track of all stream sockets used by the guest processes. It maintains the mapping between i-nodes and application sockets in the `application-socket-table`, which can be used to uniquely identify each socket in the system. When a process monitor sends a service request to the network service (e.g. for `accept` call of the guest process), it may first need to look up the i-node associated with the relevant socket and use that in the request. The socket layer can then use the i-node to map it to the unique socket data structure (referred to as *shadow socket*) corresponding to the real application socket in the underlying system. In Figure 2.4, both processes `P1` and `P2` request `accept` service on the same i-node value `i1`, which maps their requests to the same listening socket entry. Because these two requests are blocking, the PMs will also block waiting on the responses. When servicing an `accept` request for an application socket, the socket layer first checks if any new connected sockets are already available to be returned to the application. Such pending connections can start to accumulate after the guest application process calls `listen` on a passive socket and sets the pending connection queue size. If there are no new connections available, the socket layer adds the service connections to the accept-

queue of the app-s-1 socket record, keeping the processes blocked. The accept queue holds the active service connections from the process monitors of the blocked processes. The accept queue will be drained once there are matching new connections that can be passed back to the blocked PMs using the service connections, which will then return the new sockets to the guest processes. Each listening socket record contains a reference to a connection-queue that will accumulate new matching connections, in the Figure 2.4 example, all sockets point to the same connection-queue connection-q-1. Connection-queue structures are created when guest processes create listening sockets bound to a new address (their queue size is updated with listen requests.) They are added to the connect-queue-table that is indexed by the listening address. When a guest process creates a listening socket that is bound to an address that is already associated with an exisiting connection-queue, it reuses the same connection-queue (e.g. this allows guest applications to use SO_REUSEPORT socket option.) In the discussed example, there is only one connection-queue because both sockets app-s-1 and app-s-2 are listening on the same address addr1.

When the transport layer (discussed below) makes a connection request to the socket layer, it specifies the connection destination address in the request (connection_req(addr1) in the example of Figure 2.4.) This address is then used to lookup a matching connection-queueu, if one is found, it means that there is a guest process listening for such connection, and an appropriate transport setup may continue using a candidate socket (cs). If there is no match, or the pending connection queueu is full, the request is denied, and the transport layer can propagate the error to the active side, potentially resulting in the (remote) client process receiving a connection refused error.

As new connections in a connection-queue become ready, when candidate sockets can be returned to the application processes, the references to the matching listening sockets are used to return the new sockets to the blocked processes on the accept queues (e.g., process P2 on the accept queue of socket app-s-1.) The PMs are then unblocked returning the new sockets to the guest application.

However, to handle non-blocking accept requests by the guest processes, the socket layer must be able to deal with the I/O event notification facilities (such as epoll.) Boxer manages to avoid intercepting any such notification functions by using a technique based on *signal-sockets*. When a new pending connection is available, the corresponding connection queue can create a new *signal-connection* to the local address that is bound to the real application socket that the guest process is listening on. The local listening address of

the real socket is setup by the program monitor when it intercepts the *bind* call of the guest application process. This is also the socket that the guest process (oblivious to what is actually happening behind the scenes) will add to its I/O notifications (e.g., with epoll_ctl) to be notified by the kernel if there are new connections to accept (on what it thinks it is listening on). The signal-connection is configured to emulate triggering this event. When the corresponding signal-connection is made, appropriate I/O events will be set, and if the guest process chooses to call *accept*, the process monitor will hide (and discard) the signal-connection and then make a request to the node supervisor's socket layer to retrieve the real new connection from the appropriate connection-queue (or return the appropriate errro code if none are already left, since other requests might have drained the queue while this signaling cycle was in progress.) This somewhat strange contraption of using signal-sockets, proved very valuable by enabling the necessary nonblocking I/O functionality while at the same time avoiding reimplementing the complex logic of event notification functions such as epoll, poll, or select.

The above example showed some of the elements of the mechanism used to accept new stream sockets. It is important to keep in mind that these *emulation gadgets* can only work as a collection of appropriat matching mechanisms emulating socket creation, binding, listening, connecting, and closing, all together working in concert to emulate the application's expected semantics, using only unprivileged interposition.

2.6.3.2 Transport Layer

The transport layer implements the setup of data delivery for the guest application sockets that are managed by the Boxer socket layer (described above). It is important to emphasize that the transport layer only performs the *setup* of the network transport and is not involved in the data path. Currently, Boxer implements direct TCP, TCP-NAT-Traversal, and IP-Forwarding-Proxy transports. Other transports for stream sockets, such as those based on S3, S3 Express, DynamoDB, QUIC, or other intermediary services or overlays, are planned for the future (some may require additional functionality implemented in the form of local proxy processes, similar to the sidecar pattern in microservices.) The transport layer implementations rely on the Boxer control network to exchange necessary messages between nodes to configure remote connectivity. The control network is described below in Section 2.6.4, what matters here is that all Boxer nodes can exchange control

messages between each other. For example, the TCP-NAT-traversal transport that Boxer uses in AWS Lambda, exchanges messages with remote Boxer nodes to establish new TCP connections that traverse the intermediate NAT devices. Once the transport setup is completed, the socket layer can make it available to the guest application processes. An overview of the mechanisms of TCP-NAT-Traversal and IP-Forwarding-Proxy transports are provided below.

TCP-NAT-Traversal Transport

This section describes the TCP-NAT-traversal transport used by Boxer in AWS Lambda. In AWS Lambda, like in other FaaS services, individual functions cannot accept externally initiated network traffic, which, as discussed previously, is one of the main limitations of the FaaS programming model. In particular, functions can initiate TCP connections to external destinations, however, functions cannot accept remote TCP connections. This prevents functions from using TCP networking to provide stream sockets between concurrently executing functions. The exact mechanisms that are used by the service providers to implement these network restrictions are not publicly known and are subject to change, however based on network analysis, the behavior AWS Lambda appears to be consistent with the use of NAT devices. To describe how Boxer TCP-NAT-Traversal transport works, first, a brief overview of NATs is given, approaches to traverse NATs, observations of properties of NATs used in AWS Lambda, and then the approached taken by Boxer is described.

NAT (Network Address Translation) is a common networking technique used in data center networks composed of multiple logically isolated IP address spaces. NAT devices map addresses from one (private) address space to another (public) address space to provide transparent routing between hosts in those different address spaces. The technique dates back to 1995 when it was initially proposed as a temporary solution to shortages in available IP address space [EF94]. With time, it became clear that, like so many temporary solutions in the space, NAT devices were becoming one of the core tools in use, so further efforts began to standardize the techniques and terminology [HS99], with recommendations for TCP based NAT behavior described in 2006 [FGB+08] and with further updates continuing later [PPB+16]. There are many NAT variants, differing in how permissive they are, how address spaces are mapped, or in details of what protocol-level (packet) behavior they exhibit. In general, hosts behind a NAT can initiate new network traffic to routable

hosts outside of their NAT, however, they cannot receive new (uninitiated) network traffic from the outside. Based on the traffic initiated by the internal hosts behind a NAT, the NAT devices build up necessary mapping tables to perform the address translations of the network traffic. If a NAT device does not have an active valid mapping for network traffic arriving from an outside network, the traffic is likely not permitted to pass.

As NAT devices became popular, peer-to-peer (P2P) systems were gaining momentum as well. P2P systems often aimed to utilize end-user hosts connected behind NAT devices. This lead the P2P research community to develop several now classic NAT traversal techniques [FSK05, J.L05].

The goal of the NAT traversal techniques (also referred to as NAT hole punching) is to provide network connectivity between hosts connected behind NAT devices that otherwise would not be able to send network traffic between each other. The basic idea is to use the hosts behind the NAT devices to generate network traffic that would update the NAT devices' mappings in such a way that the NAT devices would then permit the specially addressed network traffic between the two previously isolated hosts to pass through. The TCP NAT traversal techniques rely on the availability of a rendezvous server that is accessible by the hosts behind NAT(s). The rendezvous server is then used for signaling, to determine the correct network addresses to use, and to exchange the necessary information between the end hosts to attempt the NAT traversal. In principle, how the required network traffic is generated does not matter. The most straightforward way for the end hosts is to send and receive the appropriate network packets directly. However, many of the use cases of the P2P applications cannot assume that the applications have sufficient privileges to send and receive raw network packets. Because of this, the TCP NAT traversal protocols describe how to use standard userspace stream socket API, in a non-standard way, to generate appropriate network traffic. The network traffic, to configure the NAT devices, and then to establish a TCP connection that traverses those devices, is all generated from unprivileged userspace programs. Depending on the properties of the NAT devices, the operating systems used, and the application requirements, different NAT traversal protocol variations may be used.

To find the best strategy for Boxer TCP-NAT-traversal in AWS Lambda functions, it is helpful to characterize the NAT devices used by AWS Lambda. Unfortunately, the NAT configuration used by AWS Lambda is not publicly defined, and the network, the

microVMs, and the NATs can interact in arbitrarily complex ways, possibly including non-standard vertical integrations. This yields a large configuration space that non-exhaustive measurements might not completely cover, and the behavior might change under different scales or other unusual internal conditions. Furthermore, the configuration can change with time, and the observations must be performed from a regular cloud user's available restricted and unprivileged vantage points (e.g., AWS Lambda functions cannot observe their raw network traffic.)

AWS Lambda functions are permitted to initiate new TCP connections to outside services, but cannot accept incoming TCP connection requests. In particular, AWS Lambda functions have their local private address spaces, and NAT devices provide a transparent mapping to external addresses used for communicating with external services. Based on (non-exhaustive) observations of the AWS Lambda network, it can be concluded that the AWS Lambda NATs can be treated as Basic Outbound NATs (according to RFC 2663 [HS99] terminology) and that they are Port Restricted Cone NATs (using RFC 3489 [RHMW03] classification,) and that they have predictable external IP addresses. The NAT-expressed external IP addresses of a function microVM instance are predictable in the sense that the observable external IP address for a function stays the same for all connections (on all ports) during that function's execution. In addition, the number of external IP addresses available for AWS Lambda functions is sufficiently large that it is simple to instantiate (using selective rejection) sets of functions (at least on the order of 100s of functions) where all of the functions have unique NAT external IP addresses. At the same time, when this is done, the NAT external TCP ports match the internal ports of the microVMs (this matches the Basic Outbound NAT definition.) The NAT devices permit traffic to internal addresses from external addresses only if there was previously traffic sent from the internal address to the external address, and this filtering is performed using internal and external IP addresses and TCP port numbers (matching Port Restricted Cone NAT classification.)

The NAT devices appear to permit TCP SYN packets to traverse the NAT for already initiated connections, and do not reply with TCP RST packets in response to receiving TCP SYN packets for a flow not (yet) configured in the NAT translation table (this seemingly minor detail plays a role in choosing a traversal protocol.)

Some variants of the standard approaches described in [FSK05, J.L05] can be used to establish TCP connections between AWS Lambda functions, however, these approaches do

not directly match Boxer's requirements, and their reliability varies. Consequently, Boxer uses the NAT traversal approaches differently, and the protocols used are currently further specialized to the AWS Lambda NATs, taking advantage of the observed properties mentioned above. The primary source of the mismatch is how these standard approaches are intended to be used. Although the protocols aim to establish TCP connectivity between two hosts, these approaches do not seek to preserve the standard networking APIs for the applications using them. They are meant to be used by specialized P2P applications, or applications that can be modified to use the specialized interfaces, and not unmodified networked systems (that Boxer aims to support) that expect to use the network using the socket API in the standard way.

Although the standard approaches rely only on standard unprivileged userspace networking APIs, their use is very different. For example, to establish a single TCP connection traversing a NAT, the standard approaches require the application to perform multiple additional socket calls on the application's socket, establish connections to the rendezvous server, exchange control messages, and handle additional (expected) errors. The applications need to `accept` and `connect` concurrently, requiring additional sockets and I/O signaling, further complicating the emulation of the standard socket semantics. Although a mechanism described in [J.L05] suggests a way to control which hosts are the active and passive sides of a connection, to do it reliably requires additional signaling and the RST response from the NAT, which is not available in AWS Lambda. Falling back on timing-based heuristics adds complexity, increases latency, and reduces reliability. Experiences with preserving standard stream socket interface with such challenges supported Boxer's approach of moving the TCP hole punching mechanism (almost) entirely outside of the guest processes and into the network service of the Node Supervisor. With this separation, it is possible to provide the expected semantics of the unmodified guest application processes, while at the same time, the NAT traversal transport is unconstrained in the mechanisms it uses since it runs in the separate Node Supervisor process.

With the separation of the mechanism providing socket semantics to the guest applications and the actual NAT traversal protocol, there is greater freedom (and reduced complexity) in choosing how to provide the network connections traversing the NATs. Based on the understanding of the AWS Lambda NAT properties, Boxer implementation converged on using a simple protocol based on simultaneous TCP open [rfc81] that does not require using an external relay server for each connection.

As described above, because all functions in the Boxer network are selected to have unique and stable external IP addresses, and their external ports match their internal ports. Once the external IP address is determined, each node can locally determine the externally visible IP address and TCP port for a fully bound local stream socket. This ability to determine the external addresses locally allows Boxer not to require contacting external relays while attempting to establish new connections traversing the NAT devices. When a guest application process attempts to make a connection using the `connect` socket function, it is intercepted by the Boxer socket layer and then, assuming the destination address is in the Boxer network, passed to the transport layer to start establishing the TCP connection traversing the NAT. Before the transport layer can start, the socket first needs to be fully bound so that the local (ephemeral) port, that would normally be assigned by the TCP stack during connection establishment, is already known. It needs to be known before (possibly) calling `connect` on the socket so that the control messages containing the full address (described below) can be sent to the remote node before attempting the connection. For this reason, if the local port is not yet assigned (the application could use the socket in a way that already assigns the local port), the socket layer uses `bind` with unspecified local address to force the kernel to assign an ephemeral port to the (not yet connected) socket.

Once the connecting (active) side knows the local ephemeral port, the complete externally visible address 4-tuple (srcIP,srcPort,dstIP,dstPort) of the future TCP connection is known (without relying on connecting to an external rendezvous host.) At this point, the active side can begin the simultaneous TCP open NAT traversal procedure. First, using the control network, `PUNCH(srcIP,srcPort,dstIP,dstPort)` message is sent to the destination Boxer node.

On receiving the message, the transport layer of the destination node first attempts to match the request to a connection queue in the socket layer (`connection_req(addr1)` in Figure 2.4). If a guest process on the destination node has already created a listening socket with a matching address, then there will be a match. If the pending connection queue is not full, the transport layer will receive a new candidate socket (`cas` in Figure 2.4) bound to the local listening address matching the requested external destination address, and the transport can proceed further. The transport layer then proceeds with the setup by sending back the `ACK(srcIP,srcPort,dstIP,dstPort)` message back to the source node, and uses (non-blocking) `connect` on the candidate socket using the external source

address (srcIP,srcPort) of the connection as the destination address (function transport_setup(cas) in Figure 2.4).

When the source node receives the ACK(...) message, it then also uses (non-blocking) connect socket function on the previously prepared socket using the external destination address of the destination node (dstIP,dstPort) as the destination address argument.

This relatively simple procedure is sufficient to reliably establish TCP connections traversing the AWS Lambda NATs. At the TCP-level, both sides start with TCP active open, with TCP stacks initially entering the SYN_SENT state after sending the initial SYN packets. As one of the SYN packets is permitted to pass through the NAT, the receiving TCP stack moves to SYN_RCVD and sends back SYN-ACK which the NAT also lets through, moving the receiver to ESTABLISHED state and then responding with ACK resulting with both sides in ESTABLISHED TCP stack states.

Because AWS Lambda NATs do not generate RST (or ICMP error) packets in response to the unsolicited SYN packets, there is no need for additional error handling and retry logic to handle this case. As the socket becomes writable, Boxer socket layer promotes the candidate socket to a connected socket. On the destination (passive) side, the socket can be passed up to a guest application process if it calls accept on a matching listening socket. On the source (active) side, if the connection was initiated via blocking connect call, the appropriate process must be unblocked, and the application continues oblivious to the method used to establish the connection.

Notice that the active side, before starting to establish the connection, waits for an acknowledgment message from the destination Boxer node confirming that there is a socket listening on a matching destination address. This is because the destination side can also respond with a RST(...) message, for example if there is no matching listening socket created by any of the guest processes. When this message is received, the connection procedure must be aborted, and the appropriate error code must be delivered to the application process.

The main advantages of this simple approach are that it is robust and does not require an external relay. The connection establishment microbenchmarks 3.3.1 based on thousands of TCP connections, did not encounter a failed connection attempt. Furthermore, all the established TCP connections had symmetric throughput, which was not always true for other NAT traversal approaches. If the network configuration changes in the future or

Boxer is used with other FaaS platforms, the protocols may need to be adapted, either by updating the existing procedures or adding new protocols.

IP-Forwarding-Proxy Transport

As an example of a slightly different transport implementation to the above NAT traversal protocol, Boxer can also forward network connections through an intermediary VM node. The forwarding is performed at the IP level, but in the future, other variants could tunnel the traffic in other ways.

In this transport, similarly to the NAT traversal protocol (above), the active side sends a connection request message to the passive side using the control channel (possibly specifying a custom IP proxy). However, assuming that there is a listening socket bound to the address requested by the active side, the passive side first uses the control channel to send a message to the Boxer node running on the proxy VM.The Boxer node running on the proxy VM, on receipt of the message, configures the forwarding. Currently, the forwarding is configured as IP-level forwarding in the Linux kernel using iptables [Net30]. The inserted iptable rules use L4 addresses for demultiplexing and forwarding from the active side to the passive side and vice-versa.

Once the Boxer node running on the proxy responds with the acknowledgment that the IP routes were inserted successfully, and the proxy destination port numbers, the passive side then forwards back the acknowledgment to the original active side. It establishes the connections using the IP proxy as the destination address (instead of the source address). The active side, once it receives the positive acknowledgement, also proceeds with connection establishment using the IP proxy address as the destination address.

This kernel-level forwarding not only does not require any userspace processing on the datapath, but it also does not require any kernel-level TCP stack processing; all processing is performed at the IP level. This network connectivity method represents the lower-bound overhead method for providing network connectivity between two functions using a VM intermediary. It is useful for comparisons to what could be achieved by any of the many other projects based on userspace forwarding proxies. An additional value of this transport is that it is a way to peek into the network traffic of AWS Lambda functions. From the vantage point of the functions, the visibility on the network is very limited, the environment is restricted to stream semantics sockets, but no network-level observability is possible. Using the IP forwarding proxy, the network flows between functions can be observed at

the proxy VM local ethernet level (since the VMs provide kernel access). This provides details of IP and TCP network traffic between functions, which can help with debugging, tunning, and provide additional insights about the networking configuration.

2.6.4 Coordinator Service

All Boxer nodes run a coordinator service that listens for network membership updates, maintains its local membership set, and propagates updates to other Boxer nodes in the network. The membership set contains records for node-ids, role-ids, node addressable IP addresses, and the (optionally) assigned names for all nodes in the network. Every service instance allows other nodes to become children and propagate the updates. Currently, the seed node is used as the root of the propagation tree. The coordinator service is used during the initialization process (Section 2.6.5) by the networking service to establish the control connections. It is also used to determine when to start the guest application execution and to provide the application with the initial list of peer addresses (Section 2.6.6).

When a new node joins the network, its coordinator service learns the addresses of all other nodes in the network. Concurrently, coordinator services of all other nodes in the network are updated with the address of the newly joined node. The Boxer networking service of every node listens for membership updates and treats these events as signals that it is time to establish direct control connections with the new node. The membership updates contain the expected external address of the Boxer network control service. At that point, all nodes have enough information to establish new control connections that may traverse the NAT. Boxer nodes establish the control TCP connections between the new node and the rest of the nodes in the network. This procedure is repeated for every joining node, resulting in N×N connectivity through TCP connections, one for each pair of nodes in the network. The control connections are sufficient to exchange commands between the Boxer nodes in the network.

Names assigned to nodes in a Boxer network are transparently available to the guest applications. Guest processes that use standard system C Library name resolution functions that are intercepted by the program monitor, such as `getaddrinfo`, will transparently query the coordinator service for matches. If the coordinator service produces no matches, the name resolution is forwarded to the underlying host and follows the standard path. Other than the assigned names, the coordinator resolver also provides some canonical hostnames that can ease application configuration. Hostname `node-ID` will always resolve

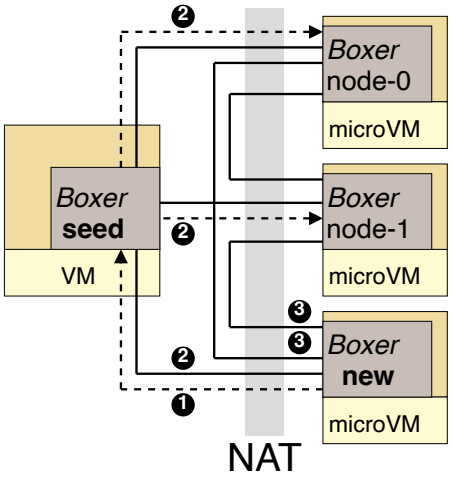

Figure 2.5: *New Boxer node $Boxer_{new}$ in FaaS function joins already connected nodes $Boxer_{node-0}$, $Boxer_{node-1}$ running in FaaS functions and seed node $Boxer_{seed}$ running in a virtual machine. Solid lines represent control connections, dashed lines messages sent, numerical labels the stage of the join protocol when message is sent or connection is established.*

to the IP address of the Boxer node assigned node id `ID`. Similarly, hostname `role` will resolve to an IP address of a node assigned role `role`. If there are multiple nodes assigned role `role` then names `role-N` resolve to the IP address of the `N`-th node with the role `role` ordered by node id order (e.g., `worker-3` resolves to Boxer node assigned role `worker` with the third lowest node ID among the nodes with that role.)

The Boxer can be configured to listen to the coordination service and only start executing its guest application when a certain number of nodes are present in the network or when a minimum number of nodes with specified names are present (e.g., only when a predefined number of workers are ready). When the supervisor starts the guest application, it populates a set of local files with a list of other nodes, names, and node ids and the node id of the local node. Some guest applications can use these static files as part of their configuration. In addition to the static files, guest applications can use a local Unix domain socket interface to connect to the coordination service and stream dynamic membership updates. Alternatively, local `boxctl` program can be used to stream updates to standard output, which can be useful in shell scripts.

2.6.5 Joining a Boxer Network

One of the nodes in a Boxer network is designated as the Boxer *seed* node and is used during the network initialization (Figure 2.5) procedure. Since FaaS functions cannot connect to other functions before Boxer is initialized, the seed node must run outside AWS Lambda. Currently, the seed is run in an AWS EC2 instance, but it could be run anywhere that is routable from Boxer processes attempting to join the network. All Boxer nodes must be configured with the address of the seed node to use, usually by setting the `SEED_ADDR` environment variable.

The seed node performs three functions. First, based on the initial connection made by the joining node, the seed determines the externally observable address of the new node. It then informs the joining node of the observed address. If the node is connecting from AWS Lambda, this is the external address assigned by the NAT gateway.

Second, the network node ID is assigned to the joining node by the seed based on the join order. The coordinator service of the seed updates all other Boxer nodes in the network with the new node information. It also supplies the joining node with the list of all other nodes that already joined. This triggers all of the nodes to establish the control connections with the new node.

Third, the seed node selects the set of Boxer nodes that will compose the network. Some of the nodes attempting to join may be prevented from joining the network. If a new Boxer node attempts to join a network using an external address already used by another node, the join attempt is rejected, and the joining Boxer node terminates immediately (in the case of a AWS Lambda function, the function terminates.) This is done so that all nodes on the network have unique external addresses. The TCP-NAT-traversal protocol used in AWS Lambda (Section 2.6.3.2) relies on this property to avoid using external relays to establish each TCP connection. Because AWS Lambda does not guarantee that the exact number of requested functions will be started in parallel, and because of the possibility of functions being rejected when joining the network, when starting larger AWS Lambda based networks, a small number of additional functions is requested to compensate for the possibly rejected functions.

Another reason the seed node may reject joining nodes is based on the network configuration. If the network was configured to be of a specific size, once the desired node count is reached, all further join attempts are rejected. Similarly, the network can also be defined to contain a specific combination of nodes with particular role (e.g., 32 nodes with role

`worker` and 1 node with role `controller`). Once there are enough nodes in the network with a particular role, all further nodes joining with that role will be rejected. When rejected, the functions terminate immediately.

Once all nodes join the network, there is no further need for the seed node, and it can terminate. On the other hand, if the network is configured to have a dynamic size, the seed node must stay running to allow new nodes to join at any time.

2.6.6 Starting Guest Application

By default, Boxer nodes will execute the configured guest programs (with configured arguments and environment variables) immediately after they join the network. This is suitable for applications that expect to have dynamic membership (e.g., the number of worker nodes can vary during execution.) However, some applications expect complete static network configuration before their execution begins. To make using such applications easier, Boxer provides an option to specify the size of the network before nodes begin execution of the guest application. Once the specified size is reached, Boxer populates a set of local files with the complete network configuration and starts the specified application program. The specified program may be a user-supplied shell script that then generates the necessary application specific configuration files based on the full knowledge of the network configuration, and only then starts the actual application program.

2.6.7 Utilities

Boxer provides additional useful utilities and options that were added while experimenting with more applications. One such utility is the ability to transparently remap file system names visible to guest applications. This is useful when applications expect hard-coded path names that are unavailable or restricted in FaaS. Based on intercepting some of the file system C Library calls (e.g., `open`) in the Process Monitor, Boxer allows users to specify file system paths that should be redirected to different paths.

Boxer uses this mechanism to redirect the application's accesses to some '/etc/' configuration files that are read-only in FaaS environment (e.g., Boxer replaces '/etc/resolv.conf' with custom resolver configurations.)

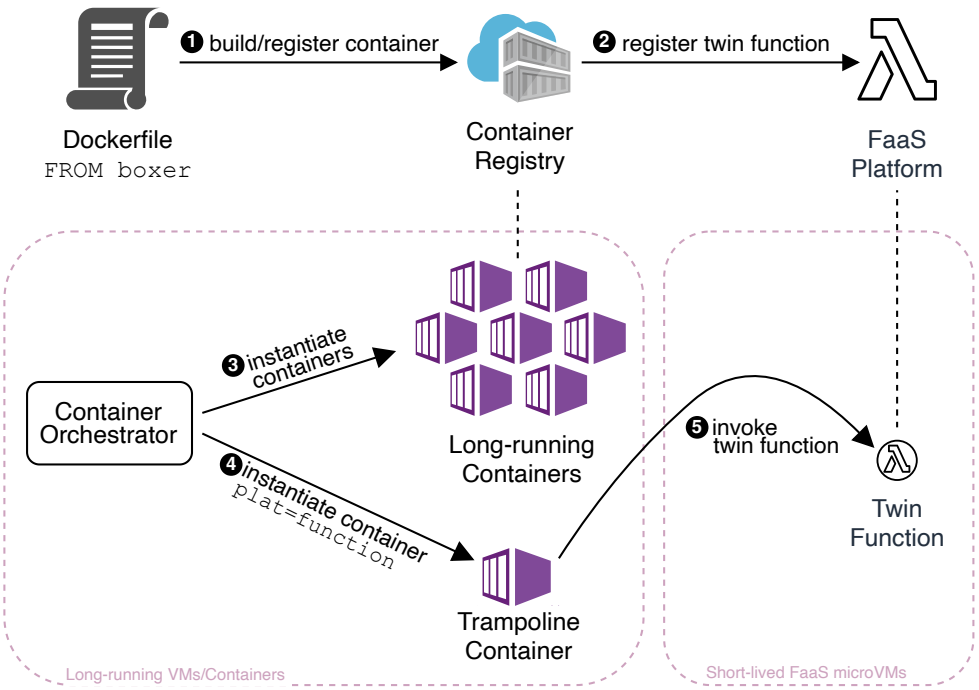

Figure 2.6: *Container Orchestration with Boxer.*

2.6.8 Container Orchestration with Boxer

Container-based orchestration systems, such as Kubernetes [Kub15], Docker Swarm [Doc15], or Docker Compose [Doc27], are a common way to deploy and manage conventional datacenter applications. Therefore, integrating Boxer with these existing orchestration frameworks improves usability, lowers the barrier to adoption, and reduces the level of Boxer-specific customization needed. However, applications leveraging Boxer have to be started through the Boxer supervisor, and the FaaS functions that Boxer relies on do not follow the same life cycle as conventional container-based datacenter applications. To bridge this gap, Boxer versions of commonly used base container images are available. These images can be used to transparently define application containers as if Boxer was not present. *Trampoline container* technique is used to invoke both Boxer functions or containers via the same container orchestration systems.

The Boxer containers are standard containers with the addition of the appropriate Boxer binaries and having replaced the container entrypoint with a Boxer entrypoint. The Boxer entrypoint first starts Boxer and then executes the original base container entrypoint with

the specified command and environment variables. This results in the original container entrypoint being transparently started with the same arguments and environment variables except running via Boxer.

Second, trampoline containers are context-sensitive containers that start the container execution differently depending on their environment (Figure 2.6). When the orchestrator (e.g., Docker Compose) starts a Boxer application container, the container execution follows the standard Boxer application startup steps described above. However, when the orchestrator starts the same container but specifies target platform to be a function (by setting the environment variable `plat` to `function`), the container entrypoint does not start the Boxer application in the container. Instead, it collects the environment variables and the specified run command and invokes the corresponding twin function, passing the serialized environment as the invocation event, which is then used by twin function's Boxer to join the overlay and start the appropriate container entrypoint running in FaaS. The container that was started by the container orchestrator stays active, and can be used to forward events from the application running in FaaS function, such as console logs, that then can also be processed by the container orchestration tools, even when the application is actually running in a remote FaaS function.

When the FaaS platform invokes the corresponding function, in the cold case, it starts to execute the entrypoint that starts the function runtime that then receives the invocation event. Based on the environment sent in the execution event, the function handler starts Boxer supervisor, restores the environment variables from the original container and starts the specified command (in the warm case, the function runtime is not started, only the function handler is executed.) This results in a new Boxer node being added and the application running in the function, not in the original container. The original container remains running as a *trampoline container*, receiving logs, waiting for the function container (refered to as the *twin container*) to terminate, so then it can terminate, signaling termination to the container orchestrator.

The above protocol requires that the container's twin function is registered with the FaaS platform. Fortunately, on FaaS platforms such as AWS Lambda, functions can also be registered using OCI container format [BDGP23]. Boxer base containers are both AWS Lambda functions and regular containers; if they are started by container orchestrator they can start the application or be a trampoline container, and when they are started by the FaaS platform, they behave as function. This means that the user only needs to

configure their application using the base Boxer container once, and then it can be both used as a Boxer container with a container orchestrator running in regular containers, and can be registered with FaaS as the corresponding twin function.

The experiments in Section 4.3 use this technique to run the same Apache Spark and Apache Drill containers using Docker Compose in AWS EC2 VMs and AWS Lambda functions.

2.7 System Limitations

The current Boxer system prototype is in development and has numerous limitations and some design decisions were motivated by expedience and do not represent the best solutions. However, it is also worth noting that it is a prototype that moved the needle from impossible to possible. Before Boxer, running unmodified distributed datacenter applications using serverless FaaS platforms was not considered possible. Boxer fulfilled its role by showing this as a possibility and demonstrating a new serverless datacenter application paradigm. However, the prototype did not (yet) reach the level of a complete or optimal solution; further iterations will be required to approach that level. Although many complex and unmodified applications can run on Boxer, the functionality of the Process Monitor interposition layer is known to be incomplete in terms of scope and correctness. The system has been developed by incrementally experimenting with adding the functionality needed to support new and more complex applications, not by going through the specifications and deriving a (potentially provably) correct solution. Because of this, it currently provides the functionality required to support many systems of interest, but not necessarily systems that do not follow the paths exercised by the systems tested so far. At the same time, the systems that have been used with Boxer already exercise a large functionality surface area, to the point that experimenting with new systems often does not require any updates. For example, this was the case with Apache Spark. When Apache Spark was tested with Boxer, it worked on the first try.

2.8 Related Work

Systems have been proposed to bypass the kernel for the application's data path operations to avoid the performance overheads imposed by the kernel. Many of these systems follow the libraryOS structure of shifting the functionality traditionally provided by a monolithic kernel (like Linux) into the application context and providing it with access to lower-level resources (Arrakis [PLZ+14], IX [BPK+14], Junction [FCS+24]). These systems use the kernel as their control plane, while (some of) the data plane is shifted to the application and exposed via custom or traditional POSIX(-like) interfaces. Similarly, mTCP [JWJ+14] exposed to applications with an alternative BSD-like socket interface to TCP stack implemented in userspace using packet I/O. Boxer nodes share some design structure with these systems in that its process monitor runs as a library in the application and exposes standard POSIX(-like) interfaces, and it relies on (manipulating) the underlying kernel's control-plane operations. However, Boxer runs in the restricted and unprivileged environment of FaaS (and AWS Lambda in particular), so it cannot access lower-level resources, and therefore, paradoxically, instead of implementing the data path in the process monitor, it works hard to stay away from interfering with data path to avoid adding any additional overhead. If Boxer could access lower level resources, then implementing full transport mechanisms in process monitor could provide performance benefits. Boxer also shares some design with Slim [ZZZ+19] that also intercepts stream socket creation operations in order to redirect them to an overlay-based implementation. With similar interposition Mbox [KZ13] uses privileged seccomp/eBPF, to prevent sandboxed processes from accessing the host filesystem to layer an overlay filesystem on top of the host filesystem. Although with security focus, gVisor [gVi17] is a system that emulates some system calls in userspace of the application process, and delegates some (more privileged) functionality to a neighboring process, similarly in structure to Boxer's Process Monitor and Node Supervisor.

Without the requirement of supporting unmodified datacenter applications, other FaaS specialized systems have explored enabling general computation on FaaS by providing GPU support [Nuc23, KJK+18, SAC+20], familiar concurrency APIs [ZFPS20], transactional workflows [ZCC+20, SSL+16], atomicity guarantees over shared storage [SWC+20], and handling timeouts by checkpointing and generating continuation functions [ZFPS20]. Other optimizations such as locality-oriented scheduling [FS22], cold-start reduction [OYZ+18], and memory footprint optimizations [SJS+22] are orthogonal to Boxer as it is implemented

on top of such serverless architectures but could potentially could benefit from such platform optimizations.

Also without emulating the network-of-hosts model, a number of projects addressed function networking limitations by using intermediaries to initiate connections and to relay messages between functions. mu [FWS+17b] proposed a framework for parallel computation and communication across buffers and relaying messages between functions. Others [KWS+18, WFLH18, PVS19, MMA20, PCFDM20] leveraged external storage to exchange data between functions. Projects such as InfiniCache [WZM+20] and gg [FRI+19] use a proxy-based approach. Nat-hole-punching in AWS Lambda has been leveraged to provide function communication primitives by [CBCH23], but not to transparently provide network-of-hosts model to datacenter applications.

2.9 Conclusion

This chapter motivated and presented the design and implementation of the Boxer system. The serverless event-triggered-functions model of FaaS and the classic network-of-hosts model of datacenter applications were described and compared. It argued that, in terms of scale, the contemporary FaaS resources are not dissimilar to those available recently as virtual machines, however, today these resources are exposed under event-triggered-functions model, and therefore not available to unmodified datacenter applications. It was then shown, that with Boxer, it is possible to expose the resources of an existing publicly available FaaS platform under network-of-hosts model, and therefore provide a unified interface that can span classic virtual machines and serverless functions making them transparently available to classic distributed datacenter applications. The description of Boxer implementation shows how the system achieves this using publicly available FaaS resources, without requiring any additional privileges, transparently to applications, avoiding datapath overheads, all while providing compatibility with unmodified orchestration systems.

The next chapter continues to demonstrate how, by leveraging Boxer, unmodified distributed datacenter applications can transparently run across long-running virtual machines and short-lived serverless functions to access the elasticity of serverless resources. It is shown that leveraging the fast ephemeral elasticity based on resources of FaaS platforms can reduce resource overprovisioning for datacenter applications.

3

Fast Ephemeral Elasticity for Datacenter Applications

3.1 Introduction

This chapter motivates and proposes a new approach of improving elasticity available to off-the-shelf datacenter applications. It demonstrates that by using Boxer to provide the uniform network-of-hosts model that spans long-running virtual machines and short-lived FaaS functions, off-the-shelf datacenter applications can transparently benefit from *fast ephemeral elasticity* based on FaaS, and therefore reduce the need for resource overprovisioning. The chapter shows that without re-architecting the applications, the ephemeral elasticity provided by Boxer enables significant performance and cost savings for off-the-shelf applications with, e.g., recovery times over 5x faster than EC2 VM instances and absorbing load spikes comparable to overprovisioned EC2 VM instances.

This section continues with introducing the background on available elasticity in datacenters and existing approaches used to address its challenges. Section 3.2 characterizes the mismatch between the level of elasticity that datacenter applications require (based on an application request trace), and what is available (based on measurements of VM/container instantiation times). It then motivates the use of combination of long-running VM resources and short-lived FaaS resources to match the application demands in a cost-efficient

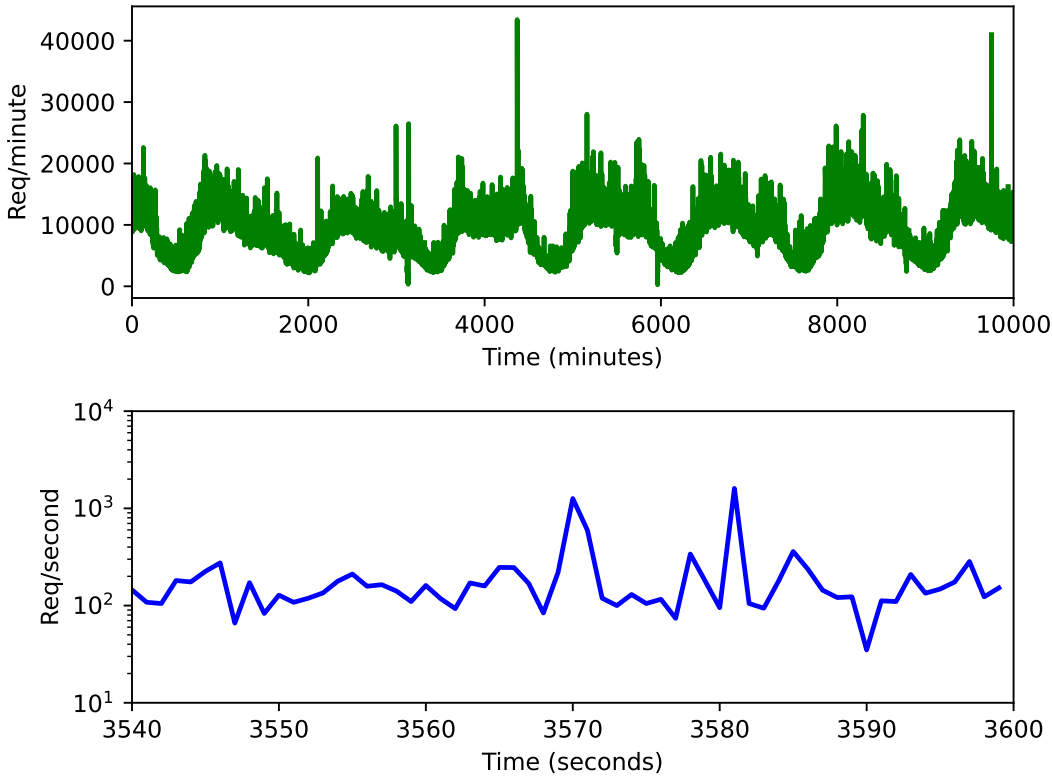

Figure 3.1: *Reddit requests over 7 days (top) and 1 minute (bottom). Extracted from a public Reddit 2015 trace.*

way. Section 3.3 demonstrates that it is possible to achieve fast ephemeral elasticity with Boxer, first with a series of microbenchmarks validating network properties across different types of links, and then by showing elasticity improvements with unmodified microservices benchmark (DeathStarBench) and Apache Zookeeper cluster.

Elastic resource allocation is a key feature of cloud computing [AFG+10]. Cloud users rent virtual machines (VMs) on-demand to meet the resource requirements of their applications. However, the granularity of elasticity offered by today's virtual machines is insufficient to react to sudden load spikes or VM failures that latency-sensitive cloud applications commonly experience. For example, Figure 3.1 shows that the request rate for a Reddit web service application varies up to *two orders of magnitude* within *five seconds*. Meanwhile, instantiating just a VM or allocating new resources for a 'fast starting' container in the cloud takes *tens of seconds* (Section 3.2.1).

Since conventional cloud infrastructure is slow to respond when load spikes or failures occur, users often resort to overprovisioning resources to provide the illusion of higher elasticity [TBD+20, DK14, CBM+17, WXY+22]. This is expensive for users as they rent and pay for more and/or larger VMs than they really need. Widespread overprovisioning is also costly for cloud providers, who despite techniques like over-committing and harvesting slack resources [TBD+20, BDR+21, AGF+20], still need to power significantly more machines than necessary to support the aggregate load [LYX+17a]. For example, the 2019 Borg traces show that CPU and memory utilization is only ∼60%, even when the provider overcommits resources [TBD+20].

Function-as-a-Service (FaaS) platforms, such as AWS Lambda [AWS17] and Azure Functions [Mic17] offer highly elastic compute pools that automatically scale based on the number of tasks that users invoke. The "serverless" execution model of these platforms simplifies resource allocation, scales resources on demand, and offers fine-grained billing favorable for short tasks. Under the hood, FaaS platforms execute tasks in lightweight VMs designed to boot quickly (e.g., 100s of milliseconds [ABI+20]). However, although it offers high elasticity, current FaaS cloud platforms couple the fast-booting VMs with an event-triggered programming model and a constrained execution environment that makes them unfit to run general-purpose off-the-shelf cloud applications [HFG+19]. Existing FaaS platforms, force applications to be written as collections of short-lived, stateless functions, which cannot accept network connections while executing [HFG+19, CIMS19, SSSK+21, ZFPS20].

Previous work tries to overcome these obstacles by implementing point solutions for various use cases, such as data analytics [MMA20, PCFDM20], stream processing [SUE+23], video processing [FWS+17b], and machine learning training [JGL+21]. Other solutions address some of the limitations of FaaS (e.g., function-to-function communication) but still require re-architecting large software stacks to adapt to the FaaS programming model [LLNB23, WSH19, KWS+18, FRI+19, WMBA21, CBCH23].

This chapter shows how off-the-shelf cloud applications can transparently benefit from *fast ephemeral elasticity* with FaaS. The focus is on ephemeral usage of FaaS to absorb load spikes and accommodate sudden failure recovery, rather than running an entire application from start to end since traditional long-running VMs still provide a cost advantage compared to FaaS [Pri17, MMA20, PCFDM20] and are suitable to serve steady application load. The aim is to seamlessly run applications across traditional long-running VMs and FaaS instances without requiring changes to applications. The key challenge is providing a familiar distributed programming model (i.e., POSIX-style network-of-hosts) — which

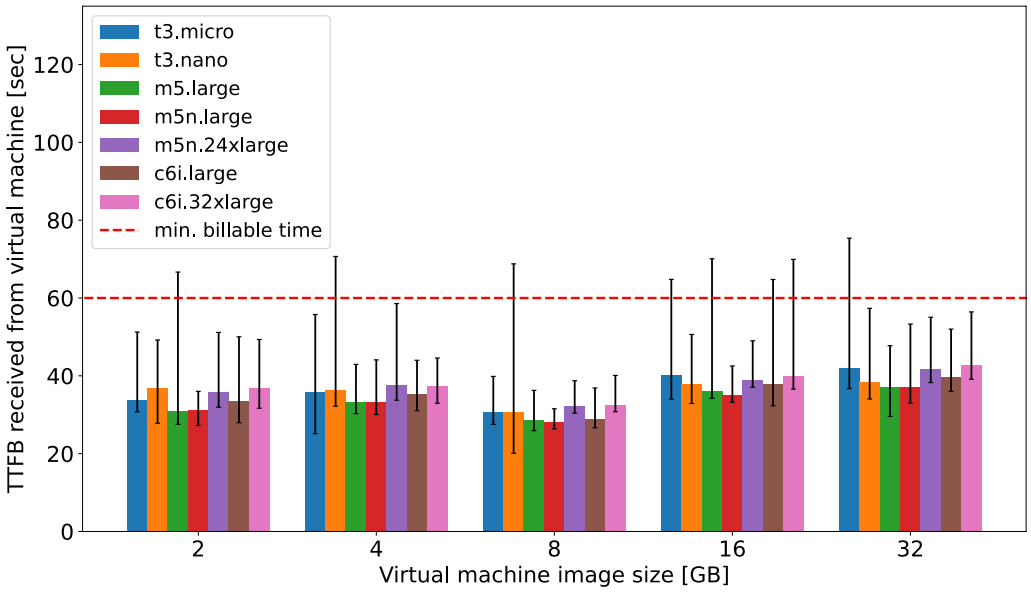

Figure 3.2: *Median instantiation times of AWS EC2 VMs service, error bars are min and max values. Details in Section 3.2.1*

generic cloud applications expect — on top of existing FaaS platforms. The Boxer system (Chapter 2) is used to address this challenge. Boxer, transparently to the applications, provides the necessary unified network-of-hosts environment that can span VM/container and FaaS function resources. Because of this Boxer can be used to quickly add networked microservice instances running in FaaS to augment long-running instances running in VMs to absorb load bursts, as is shown using an unmodified microservice application (DeathStarbench elasticity benchmark described in Section 3.3.2). Similarly, it can be used to quickly (temporarily) replace failed nodes in distributed system with Lambda instances (Zookeeper node replacement in Section 3.3.3).

3.2 The elasticity dream: are we there yet?

To characterize the elasticity requirements of cloud applications and understand the limitations of VM and container-based cloud infrastructure, a public Reddit trace [May19] that includes user requests per second is analyzed as an example of a web-based microser-

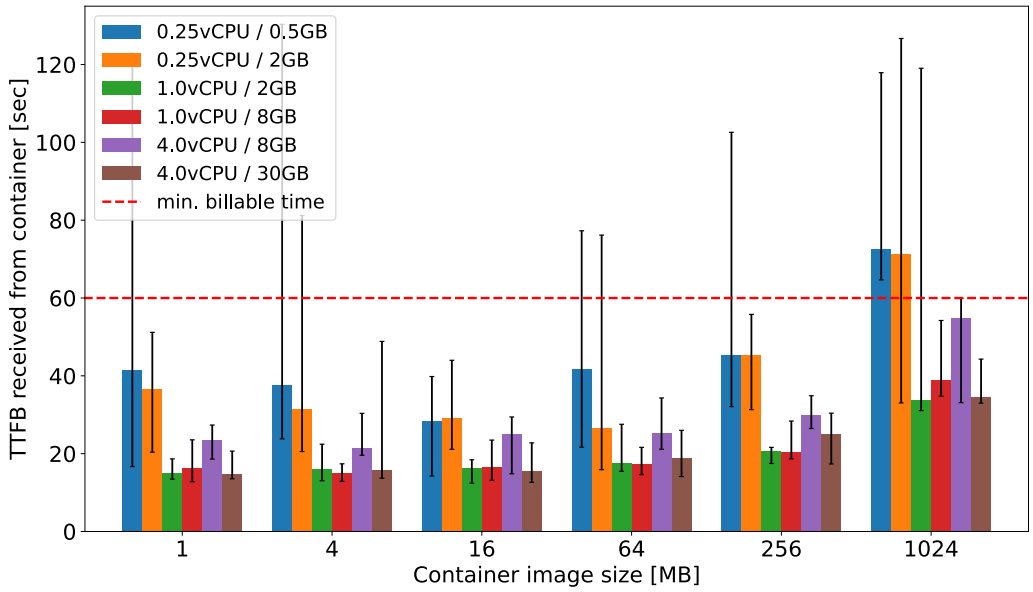

Figure 3.3: *Median instantiation times of AWS Fargate/ECS container service, error bars are min and max values. Details in Section 3.2.1*

vice application. Two subsets are extracted from the dataset: a 7-day trace containing the number of requests per minute, and a 1-hour trace containing the number of requests per second (Figure 3.1), from which two key observations can be made:

Observation #1: the 7-day trace (top plot in Figure 3.1) displays an evident daily pattern for which the infrastructure can be scaled over the course of minutes and hours. For such course-grained load variations, high infrastructure elasticity is not required;

Observation #2: when looking at the 1-minute trace, significant workload burstiness can be seen. Unlike the 7-day trace, the 1-minute trace requires highly elastic or highly overprovisioned infrastructure to be able to serve the workload changes of more than an order of magnitude in a few seconds.

This leads to the conclusion that real workloads benefit from two different elasticity granularities: coarse-grain elasticity to scale the entire infrastructure over the period of minutes and hours, and fine-grain elasticity to serve unpredictable user request bursts at the second scale. The next section demonstrates that existing cloud infrastructure cannot

cost-effectively satisfy both types of elasticity. In the following section, the concept of *fast ephemeral elasticity* is proposed, a solution for cost-effective and highly elastic cloud infrastructure.

3.2.1 Virtual Machine and Container Elasticity

Conventional virtual machine or container service deployments offered by cloud providers have significant, often deeply intertwined, inefficiencies. The issue is illustrated by the experiments that measure the time to first byte (TTFB) received from instantiations of virtual machines (Figure 3.2) of different image sizes and virtual machine types, and containers (Figure 3.3) with different resources sizes (vCPU, memory) and container image sizes. The measurements are for the time from issuing a local (in the same availability zone and VPS) instantiation request to receiving back the first one-byte UDP packet sent from the instantiated purpose-built minimal container/virtual machine image. The experiment is repeated 10 and 32 times for each of the ECS and EC2 configurations, respectively. As the data shows, real-world VM and container services, such as AWS EC2 and AWS Fargate, take on the order of 10s of seconds to allocate new resources, initialize, and return the first byte of data to a user. And that without including the additional time needed to instantiate the application intended to run in the VM or container. Note that although containers can be 'fast starting,' the cloud services providing them (AWS Fargate) still need to allocate additional resources for them, adding to the container instantiation times. These long initialization times make it difficult for applications to respond quickly to unpredictable load spikes or node failures. As a consequence, users commonly overprovision resources, which underutilizes expensive hardware infrastructure, e.g., memory utilization in cloud deployments is typically between 50 and 55% [RTG+12, SHY+18] and very rarely exceeds 80% [LYX+17b].

This leads to the conclusion that VM and container-based deployments are suitable for slowly evolving loads (that change in minutes and hours, e.g., in the 7-day trace in Figure 3.1), but not for high, unpredictable load bursts (e.g., in the second range seen in the 1-minute trace of Figure 3.1).

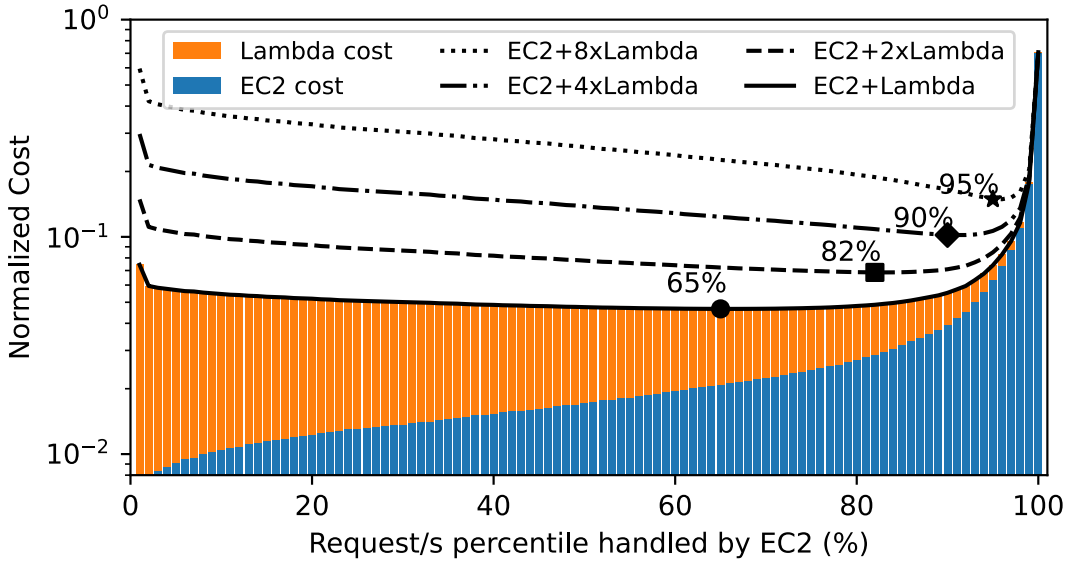

Figure 3.4: *Reddit deployment cost for different EC2 capacities using Lambda to handle requests that exceed capacity. (Section 3.2.2)*

3.2.2 Fast Ephemeral Elasticity

The proposed solution is the concept of *fast ephemeral elasticity*: running applications across both VM/containers as well as FaaS. The goal is to have a single orchestrated application deployment that takes advantage of both types of infrastructure: one for predictable load, and the other for request bursts. The remainder of this section estimates the potential of fast ephemeral elasticity to reduce resource overprovisioning and reduce the cost of running applications on the cloud. To do so, a cost analysis is conducted to compare using AWS Lambda to accommodate load bursts with overprovisioned AWS EC2 VMs. The cost of each deployment, including the cost of the EC2 baseline infrastructure and Lambdas for accommodating bursts, can be calculated as follows:

$$\sum_{t=0}^{T}\left[\frac{\beta}{\alpha}\times \$_{\text{EC2}} + max\left(0, \frac{\delta_t - \beta}{\gamma}\times \$_{\text{Lambda}}\right)\right]$$

where β is the number of requests served by EC2 VMs; α and γ the throughput per core of EC2 and Lambda (measured for Deathstarbench microservice in §3.3.2). $\$_{\text{EC2}}$ and $\$_{\text{Lambda}}$ are the cost per second per core (the costs are based on cost-efficient `c6g.2xlarge` VM and a 2GB Lambda); δ_t the load (number of requests) at time t.

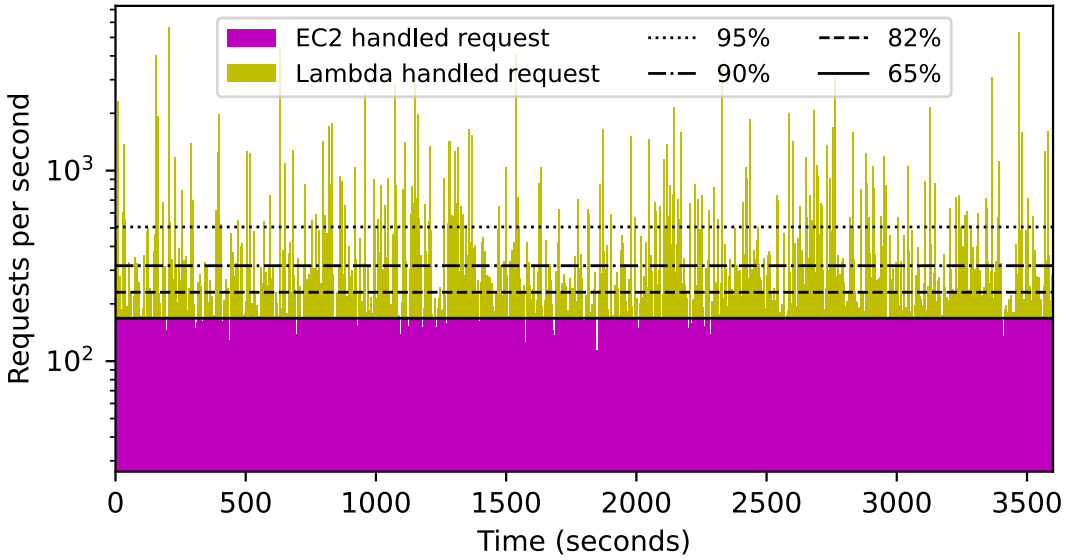

Figure 3.5: *Reddit 1-day trace (bottom) showing requests handled by EC2 and Lambda to minimize cost while providing capacity to handle all requests (c100). Handing 65% of all requests corresponds to 3% of the maximum observed request/s. (Section 3.2.2)*

Figure 3.4 presents the normalized total deployment cost per hour for the Reddit trace with a varying percentage of capacity served by EC2 instances (β goes from 0 to the maximum number of requests at any moment). If no capacity is handled by EC2, then all requests are served by Lambda instances, leading to a high cost per hour. On the other hand, if all requests are handled by EC2, a significant amount of overprovisioning is required to handle all request bursts, also leading to a high cost. The deployment that minimizes cost and therefore resource overprovisioning is obtained by combining EC2 and Lambda instances (approximately 65% of the capacity handled by EC2), thereby demonstrating the potential of the fast ephemeral elasticity idea. If more Lambda resources are necessary to process requests (because of inflexible resource allocation options, additional memory, or networking requirements,) the total cost increases, and best capacity allocation shifts towards greater VM allocation proportion (e.g., 82% for 2x Lambda per-request requirements.)

Figure 3.5 illustrates the best capacity allocation between EC2 and Lambda at 65% level, when the request rate is below 65% of the maximum the long-running VM capacity handles it, when it is above, additional ephemeral capacity based on Lambda is used to scale up temporarily. The additional horizontal lines illustrate if 82%, 90%, and 95% of the load

	c100	c99	c95	c90
EC2 + Lambda	93.42%	75.53%	43.40%	21.86%
EC2 + 2xLambda	90.31%	65.03%	25.71%	5.87%
EC2 + 4xLambda	85.60%	50.08%	7.17%	no-saving
EC2 + 8xLambda	78.95%	31.35%	no-saving	no-saving

Table 3.1: *Estimated cost savings relative to different EC2 provisioning levels (c100, c99, c95, c90) based on Reddit trace.*

was handled by the long-running VMs, corresponding to the optimal choices for 2x, 4x, and 8x Lambdas needed to service requests.

Table 3.1 shows estimated cost reduction when using ephemeral elasticity relative to different levels of EC2 VM overprovisioning. Even when EC2 is provisioned to handle only 95% of maximum requests per second (c95), and 2 × Lambdas are needed to service requests, the estimated cost reduction of using ephemeral elasticity is over 25%. At some point, ephemeral elasticity does not provide savings relative to VM overprovisioning. For example, if VM overprovisioning is to handle only 90% of the maximum requests per second (c90) and 4x or more Lambdas are needed to service the requests, using ephemeral elasticity no longer provides cost reduction.

3.2.3 Assumptions

Realizing ephemeral elasticity is based on several assumptions. First, it is assumed that individual sub-services of the target applications (e.g., individual worker nodes, quorum members nodes, microservice nodes) can be scaled up and down by adding and removing service nodes. This is a standard paradigm in microservice architectures.

Second, it must be assumed that the application's long-term persistent state is either stored in the long-running VMs or in remote cloud storage, not in the short-lived ephemeral workers. This also matches the standard microservice paradigm where stateful components are factored out from the stateless services.

3.3 Fast Ephemeral Elasticity with Boxer

Fast ephemeral elasticity is unavailable today because FaaS, VMs, and containers operate under two different programming models. FaaS is based on event-triggered functions, while VMs are based on the classic network-of-hosts model. As described in Chapter 2, the Boxer system can be used to unbundle FaaS microVMs from the event-triggered programming model, and provide a unified network-of-hosts environment that can span long-running virtual machines, containers, and FaaS functions. Therefore, to achieve ephemeral elasticity, datacenter applications are run on Boxer. The long-running services use VMs or containers for the coarse-grained elasticity, and FaaS functions for the fine-grained elasticity. Because the applications run on Boxer, FaaS functions and VMs can be dynamically added to the network while preserving the illusion to the application that they are all just regular hosts on the network.

To demonstrate the ability of Boxer to provide ephemeral elasticity to datacenter applications, unmodified DeathStarBench [GZC+19] is run using Boxer on AWS EC2 and AWS Lambda (Section 3.3.2). DeathStarBench is a suite of cloud microservice benchmarks deployed using container networks that mimic how large-scale, complex distributed applications are often deployed in the cloud. Later in Section 3.3.3, Zookeeper benchmark is used to demonstrate how Boxer helps to recover from node failures quickly. Before the DeathStarBench and Zookeeper benchmarks are described, microbenchmarks for the Boxer provided network are studied.

3.3.1 Microbenchmarks

To confirm that the properties of Boxer provided networking are compatible with the ephemeral elasticity use case, microbenchmarks for throughput, latency, and connection establishment are performed.

This section summarized the network characteristics observed in AWS EC2 and AWS Lambda using Boxer and TCP-NAT-Traversal. It shows what a typical application running in AWS Lambda with Boxer can achieve in terms of TCP throughput and latency. Unless noted otherwise, all measurements were performed in AWS us-west-2 region with Lambda functions with 3008MB of memory and m4.large EC2 VM instances.

Boxer enables TCP connectivity between AWS Lambda serverless functions and allows

outside hosts, running outside of AWS Lambda to initiate TCP connections to serverless functions. As described in Section 2.6.3 this is achieved by traversing the NAT gateways between hosts. During development some unfavorable network conditions were observed when Boxer used different methods to establish connectivity, e.g., not symmetric throughput. The target cloud applications usually expect symmetric network properties between end-points. However, given that Boxer traverses an unknown network of middle-boxes that may impose arbitrary network filtering or throttling rules, the properties of the various scenarios the middle-boxes could differentiate, based on ordering of packets, timing or types of end-hosts must be verified. Thus, in the evaluation six connection types are distinguished:

1. *Function-to-Function* connections are established by Boxer between a pair of AWS Lambda functions,
2. *VM-to-Function* are connections initiated by Boxer running in an EC2 VM to an AWS Lambda also running Boxer,
3. *Function-to-VM* connections are initiated by Boxer running in AWS Lambda to an EC2 VM also running Boxer,
4. *Function-to-VM-native* are connections initiated from AWS Lambda to EC2 VM without the use of Boxer (this scenario is allowed by default on AWS Lambda) as it is used as a baseline,
5. *VM-to-VM* are connections established using Boxer between a pair of EC2 VMs and
6. *VM-to-VM-native*, used as a baseline, are vanilla connections established between a pair of EC2 VMs without Boxer.

This evaluation also shows the versatility of Boxer and the many configurations in which it can be deployed.

3.3.1.1 Throughput Analysis

The TCP throughput of different connection types between pairs of hosts (VMs or functions) is benchmarked by running an unmodified iperf3 [ipe10] tool as a Boxer application (or natively for the native connection types). 32 non-overlapping pairs of functions are

TCP connection type	Mean		Median		Std.		Min		Max	
	Forwd	Rev	Forwd	Rev	Forwd	Rev	Forwd	Rev	Forwd	Rev
Function to Function	622.57	622.63	626.88	628.59	24.32	25.81	564.25	561.16	680.63	678.36
VM to Function	428.28	426.03	429.03	428.09	1.69	4.01	420.31	415.62	429.16	429.18
Function to VM	410.31	427.05	422.64	428.74	26.58	3.61	335.70	414.39	429.06	430.33
Function to VM (native)	427.77	426.67	429.04	428.62	3.16	4.93	412.74	399.35	429.07	429.07
VM to VM	428.89	428.96	428.96	428.95	0.55	0.29	424.78	427.50	429.01	430.39
VM to VM (native)	429.02	429.06	429.03	429.07	0.15	0.02	428.43	429.02	429.65	429.09

Table 3.2: *TCP throughput (m4.large us-west-2)*

TCP connection type	Mean		Median		Std.		Min		Max	
	Forwd	Rev	Forwd	Rev	Forwd	Rev	Forwd	Rev	Forwd	Rev
Function to Function	621.48	621.44	628.42	627.95	23.87	24.28	560.61	560.99	674.14	677.28
VM to Function	622.98	624.10	610.70	623.97	26.59	32.31	595.92	566.07	680.64	681.22
Function to VM	623.18	621.78	637.01	640.82	26.99	32.22	564.00	570.34	658.91	664.03
Function to VM(nativ)	624.13	622.39	639.62	622.06	27.75	31.24	566.61	571.08	657.34	668.00
VM to VM	4684.53	4706.22	4744.35	4735.99	144.29	103.36	4034.14	4263.13	4789.13	4787.11
VM to VM (native)	4770.42	4695.97	4786.23	4740.02	39.46	159.41	4571.73	3903.98	4787.20	4789.87

Table 3.3: *TCP throughput (m5.large eu-west-3)*

instantiated for a 60 seconds period for each scenario. In each pair, one function runs iperf3 in server mode, and one in client mode. The iperf3 client function connects to the listening server to begin the configured benchmark. When configured in the forward mode, the client side generates TCP traffic, when configured in the reverse mode, the server side generates the TCP traffic. Both forward and reverse throughput are measured to verify that the underlying network does not apply different network policies in different directions (as observed in some scenarios during development.) The achieved throughput is reported on the receiving side at 1-second intervals.

Table 3.2 presents throughput statistics for different connection scenarios. The sustained average throughput is 622Mbit/s in forward and reverse direction between a pair of AWS Lambda functions running Boxer (Function-to-Function). The variance level is low; the throughput is steady and sustained throughout the connection. An additional experiment verified that the throughput can be sustained throughout the maximum lifetime of an AWS Lambda function (currently 15 minutes). The observed throughput between a pair of VMs is 429Mbit/s if Boxer is used or not, demonstrating that Boxer adds no data-plane overhead (after a connection is established). The throughput between functions and VMs

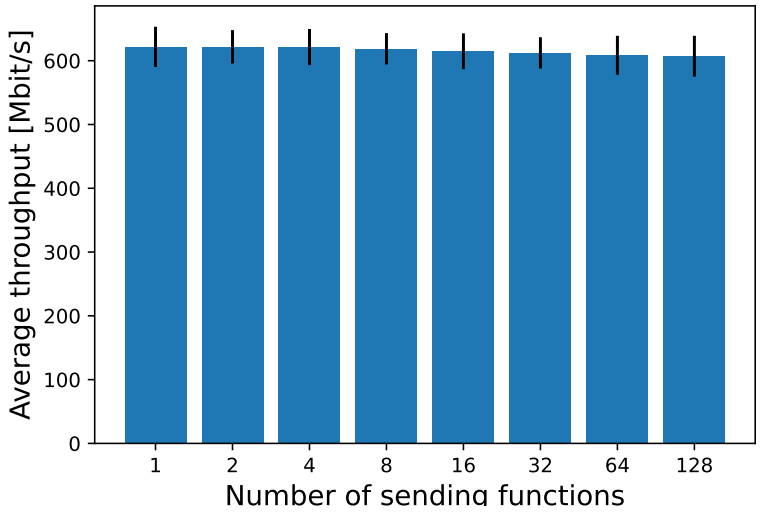

Figure 3.6: *Comparison of aggregate receive throughput as the number of sending functions varies. Maximum is 621.69Mbits/s at 1 sending function and minimum at 607.04Mbit/s at 128 sending functions, black lines represent standard deviation.*

is similar (410-428Mbit/s) and symmetric in all connection scenarios. In this case, it is limited by the throughput of the VMs (m4.large) network.

Table 3.3 shows the same benchmark scenarios but using a higher-bandwidth VM network (m5.large instances): the upper-bound on the throughput between AWS Lambda and VMs is the same as the throughput of the AWS Lambda internal network of 621Mbit/s (the experiment is performed in a different AWS region, and AWS Lambda function-to-function throughput is the same). The achieved TCP throughput between VMs and functions matches that observed by others [MMA20] between functions and some AWS services such as S3.

To gain further insight about the enforced bandwidth limits, an additional load-testing experiment was performed by concurrently sending data from multiple functions to one. The server executing in one function listens for connections from clients executing in N other functions. Each client establishes one TCP connection to the server and attempts to saturate the connection by sending data in a tight loop. At 1 second interval, the server records aggregate bytes received from all of the clients. The number of clients is varied from 1 to 256 functions and each configuration is run over a 5-minute interval. Figure 3.6 presents the averages of the aggregated received throughput at the server after removing

the initial and final 30 seconds of the experiment measurements. The maximum observed throughput is 621.69Mbits/s with 1 sending function and minimum of 607.04Mbit/s at 128 sending functions. The degradation of less than 3% can be attributed to the overhead associated with handling multiple connections. This leads to the conclusion that the ingress TCP bandwidth limits imposed on AWS Lambda functions do not depend on the number of sending functions and that the available bandwidth is comparable to that of regular instances.

The only resource parameter that can be adjusted for AWS Lambdas is the amount of memory, which then proportionally determines the vCPU share allocated to the function. The achievable throughput was measured with different memory allocated to functions to determine if the memory setting influences network properties. The throughput did not vary with memory settings of 512MB, 3008MB, and 10240MB. In all of the experiments in this section, the first 2 seconds are excluded from calculating the statistics, as a burst of throughput can be observed during that period, which would skew the steady-state statistics.

3.3.1.2 Latency Analysis

TCP connection type	Mean	Median	Std.	Min	Max
Function to Function	694.23	758.00	289.52	202.00	2769.00
VM to Function	547.84	457.00	194.29	244.00	2471.00
Function to VM	520.10	436.00	189.53	244.00	2372.00
Function to VM (native)	547.76	449.00	187.70	241.00	2071.00
VM to VM	193.69	188.00	43.34	150.00	2165.00
VM to VM (native)	197.62	194.00	28.18	153.00	1862.00

Table 3.4: *TCP latency (m4.large us-west-2)*

TCP latency of the six connection types described above was measured. For each connection type, 32 non-overlapping pairs of hosts (AWS Lambda functions or VMs) were instantiated. Each host executed a benchmarking program, and every pair created a single TCP connection with Nagle's algorithm disabled by both hosts. The host assigned the client role implements a TCP echo client that initiates a connection to the assigned server host. After accepting the connection from the client, the server function initiates 128 rounds of ping-pong exchanges of a 1024-byte message and measures the total time.

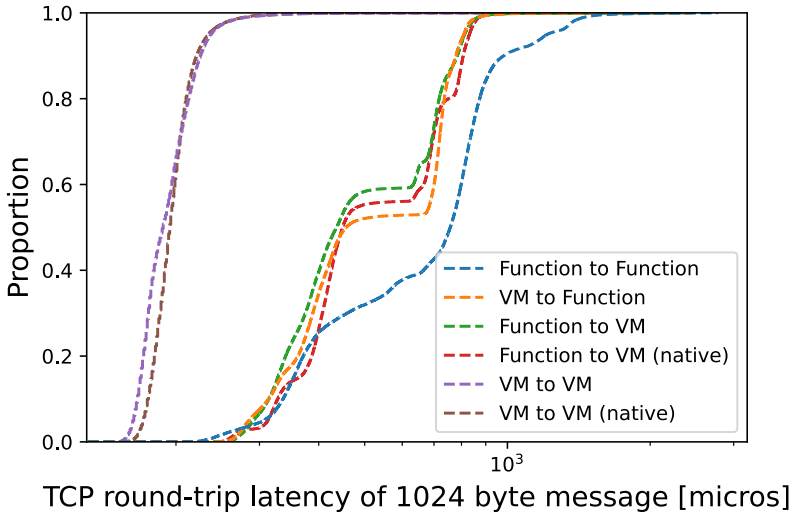

Figure 3.7: *Empirical CDF of TCP round-trip latencies between 32 distinct nodes pairs (of VMs and AWS Lambda functions) echoing 1024 byte message, repeated for 6 connection scenarios.*

This measurement is repeated 1024 times using the same connection, and then the connection is closed. Table 3.4 lists the summary statistics of observed round-trip latencies for different connection types, and Figure 3.7 shows the eCDF of the latencies measured.

Boxer's TCP connections between pairs of functions have a mean round-trip latency of 694μs. These connections also show significant variability in latencies, with a range between 202μs and 2769μs, as can be seen in the eCDF plot. The main source of the observed variance is not due to the variance within each TCP connection but due to the variance between different TCP connections. This suggests that the network between different Lambda functions, from the latency perspective, is not uniform, and different instantiation patterns result in different latencies. This makes sense as the distance between the machines where functions are deployed plays a role in determining the latency.

The latency measurements of the VM-to-VM-native connections, without the use of Boxer, and VM-to-VM connections that are established by Boxer show very close round-trip latencies of 198μss and 194μs, and follow a very similar distribution which can be seen in the eCDF plots of the two connection types. This further shows that there is no data plane overhead of Boxer provided connections once they are established. The observed latency of connections between Lambda functions and VMs have similar mean round-trip

latencies ranging from $520\mu s$ to $622\mu s$, and all follow a similar distribution, as seen in the eCDF plot. Perhaps surprisingly, the connections provided by Boxer have a slightly lower round-trip latency, which could be attributed to different processing of the traffic by the network due to its different initial packet signature (at this point, this was not investigated further.)

TCP connection type	Mean	Median	Std.	Min	Max
Function to Function	922.40	1013.00	349.83	197.00	2934.00
VM to Function	1235.53	1231.00	29.09	1164.00	1566.00
Function to VM	1040.12	1035.00	34.68	985.00	1516.00
Function to VM (native)	1233.36	1227.00	58.91	1174.00	2811.00
VM to VM	514.57	513.00	9.74	499.00	620.00
VM to VM (native)	364.11	362.00	9.08	353.00	488.00

Table 3.5: *TCP latency (μs). VMs are m5.large instances in eu-west-3 region.*

Measurements for the same latency microbenchmarks performed in eu-west-3 region using m5.large instances are reported Table 3.5. It can be observed that both the AWS Lambda function-to-function latency and AWS EC2 native VM-to-VM latencies are higher than compared to their counterparts in us-west-2 region based on m4.large instances. Although the relative patterns are similar, the difference in network properties between different regions is worth noting.

The observed mean round-trip latency of Boxer enabled TCP connections between pairs of functions is 3.51× (and 2.53×) greater than latencies observed on AWS m4.large/us-wast-2 (m5.latge/eu-west-3) VM-to-VM native TCP connections.

3.3.1.3 Connection Establisment Analysis

The time required for the application to have an established TCP connection is measured for the six connection types described above. For each connection type, 32 host pairs are instantiated (AWS VMs or Lambda functions, depending on the scenario), and one host in each pair is assigned to be the client and the other to be the server. The time-to-first-byte(TTFB) measurement is recorded by the client. The client starts a timer, attempts to connect to its server, and waits to read 1 byte of data from the server. Once it receives the 1 byte, it stops the timer, records the duration, and closes the connection. The server accepts connections and replies with 1 byte right after it accepts a new connection. This is

TCP connection type	Mean	Median	Std.	Min	Max
Function to Function	2735.21	2625.00	10001.00	890.00	1033112.00
VM to Function	1981.38	2153.00	7909.74	821.00	1011171.00
Function to VM	2086.03	2239.00	6124.57	882.00	1015205.00
Function to VM (native)	1378.56	1244.00	8027.04	382.00	1012935.00
VM to VM	1067.24	1034.00	166.09	894.00	7402.00
VM to VM (native)	407.81	345.00	284.37	258.00	6416.00

Table 3.6: *TCP connection establishment (μs). VMs are m4.large instances in us-west-2 region.*

TCP connection type	Mean	Median	Std.	Min	Max
Function to Function	3353.28	3600.00	1341.97	1007.00	33404.00
VM to Function	3407.12	3329.00	785.44	3130.00	21619.00
Function to VM	2360.65	2284.50	1042.51	2121.00	34312.00
Function to VM (native)	2487.87	2441.00	404.89	2341.00	10225.00
VM to VM	1238.97	1226.00	98.37	1134.00	2907.00
VM to VM (native)	1440.78	1419.00	229.24	1341.00	6512.00

Table 3.7: *TCP connection establishment (μs). VMs are m5.large instances in eu-west-3 region.*

repeated 1024 times by each pair, the measurements are reported in Table 3.6 for `m4.large` VM instances and in Table 3.7 for `m5.large` VM instances.

Unlike TCP throughput and latency, Boxer does have overhead compared to native TCP connection times. The mean time to establish a function-to-function connection is 2735μs. The overhead can be seen in comparison of establishment time for VM-to-VM-native connections of 408μs vs VM-to-VM connections that are established by Boxer of 1067μ. This is due to the additional round trip necessary to contact the destination Boxer to request that the connection setup and wait for the acknowledgment before proceeding locally, or in case of an error response, to forward the error to the application (for example in case when there is nothing listening at the destination address.) This additional signaling adds to the connection setup times and can also be observed in the connection times between functions and VMs. However, given that there are no alternative native connections for function-to-function and VM-to-function connection types, the additional latency can be considered as acceptable.

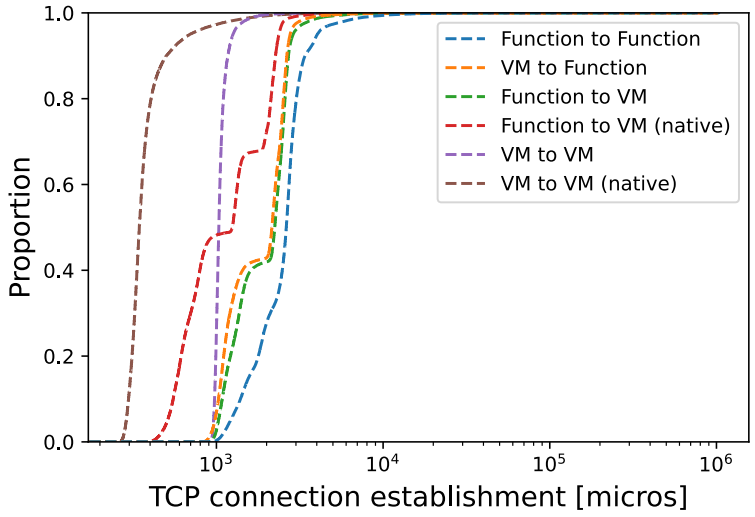

Figure 3.8: *Empirical CDF of TCP connection establishment times. Time-to-first-byte (TTFB) for different connection types measured in microseconds between 32 distinct pairs of hosts establishing 1024 TCP connections each.*

The eCDF of the TTFB times for the different scenarios in Figure 3.8 shows that the comparable scenarios with and without Boxer connection setup follow similar distribution, but are shifted by additional delay due to Boxer connection setup. Function-to-function connections and connection types that cross the AWS Lambda and EC2 networks have maximum TTFB times observed to be over 1 second, including a connection scenario that does not involve Boxer to establish TCP connections (Function-to-VM-native.) This suggests that there may be packet loss or network congestion in the network and it is not a direct consequence of the procedure Boxer uses to establish the connections. As it can be seen from the eCDF, such extreme connection times are rare, in the case of Function-to-Function connection type, 99.9 quantile for TTFB is 20654μs. Compared to the alternative of communicating through storage, such latencies are acceptable, especially considering that in all of the above experiments performed, all connections were successfully established.

The connection establishment times have an additional significance for Boxer node startup times. When a Boxer node joins the network, it first must contact the seed node and wait for a message reply on that connection, and only then it can start the application (unless

it is configured to wait for other nodes to join too.) The Boxer node startup time is dominated by the network latency, in the case of the node starting in a function, this would be the Function-to-VM-native latency, since the seed runs in VM and the first connection does not need to preform NAT traversal. Analogous, for the nodes starting in VMs the relevant network latency is VM-to-VM-native.

3.3.2 Running DeathStarBench on Boxer

This section focuses on benchmarking DeathStarBench's *socialNetwork*, which offers a social network service to users and is organized using three microservice layers:

1. Front-end layer, implemented using an NGINX webserver,
2. Logic layer, implemented using stateless Thrift services that communicate through RPCs,
3. Caching and storage layer, implemented with MongoDB [Mon03] and Memcached [Mem03] instances.

In *socialNetwork*, user requests are received by the front-end layer (NGINX web server) and then routed to one of the services in the logic layer. Depending on the user request, the logic layer may also perform one or multiple requests to the caching and storage layers. Since the logic layer is stateless (i.e., it contains no internal persistent state), it can be deployed on AWS Lambda (up to maximum function duration).

No modifications were required to the application code to deploy DeathStarBench on AWS Lambda with Boxer. The benchmark was only modified to i) use hostnames instead of fixed local IPs (for example, replace `192.168.1.7` by `nginx-thrift`), and ii) run all of the components of the front-end and logic layers using Boxer. Executing all of the front-end and logic layer components with Boxer ensures that the creation of new connections is handled by the Boxer network, allowing not only the front-end layer to establish connections to services running in the logic layer (VM-to-Lambda connections) but also components in the logic layer to establish connections between them (Lambda-to-Lambda connections).

Methodology

To evaluate the performance impact of moving the logic layer to Lambda using Boxer, four deployments (referred to as *EC2*, *Boxer-EC2-only*, *Boxer*, *Fargate*) are used, one baseline and three exercising Boxer in different ways.

1. *EC2* - All components deployed as EC2 VMs, this representys the baseline,
2. *Boxer (EC2-only)* - All components deployed as VMs in EC2, but the components of the front-end and logic layers use Boxer. This deployment measures the performance overhead of using Boxer,
3. *Boxer* - A mixed deployment with front-end, caching, and storage layers deployed as VMs, and logic layer deployed using Lambdas.
4. *Fargate* - A mixed deployment with the logic layer using AWS Fargate container service.

Two workloads included in the DeathStarBench suite are used to measure the throughput and latency of the end-to-end system. The read workload that issues requests to read a user timeline in the socialNetwork, and a write workload that creates follow relationships between users. Both workloads are generated using the `wrk` [wrk09] tool, which builds and issues requests to the front-end layer. The performance of both workloads (read and write) is reported separately as each workload stresses the Boxer overlay in a different way. The read workload primarily transfers data from the caching and storage layer (VMs), to the logic layer (VMs or Lambdas), and then to the front-end layer (VMs). The write workload operates in the opposite direction.

All experiments in this section were conducted using us-east-2 region. All VMs use a base Amazon Linux 2 [Ama10]. For front-end, caching and storage layers, `t3a.micro` instances were used due to the memory requirements of the services included in these layers. For the logic layer, when deployed in VMs, `t3a.nano` instances were used. Each Lambda is configured with 2048MB of memory (experimentally determined that in us-east-2, the performance of a 2048MB Lambda is similar to a t3a.nano VM instance). For Fargate, containers were deployed also with 2048MB of memory and 1.0 vCPU unit (this configuration is also the one that yields faster container startup time, see Figure 3.2).

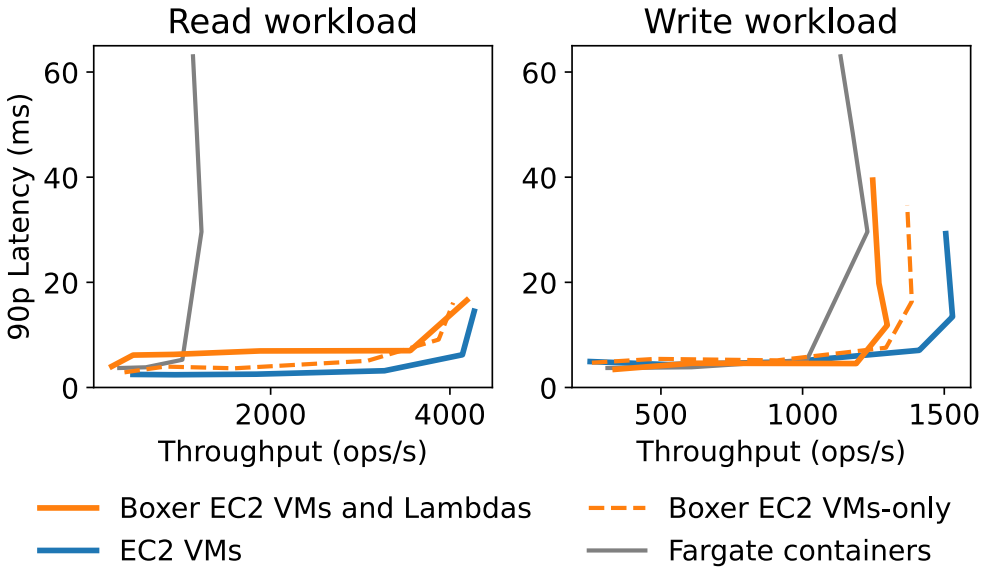

Figure 3.9: *DeathStarBench results in a static deployment.*

Overhead of using Boxer

This section analyzes the impact of using Boxer to deploy the DeathStarBench social Network application and move the application logic layer to AWS Lambda. Figure 3.9 shows the results for both read and write workloads across the four different types of deployments. For each workload, the average throughput and 90th percentile latency with an increasing load in the system were collected. Boxer introduces only a small overhead. For the read workload, the EC2 deployment becomes saturated at 3270 ops/s while the Boxer-EC2-only becomes saturated at 3070 ops/s. For the same data points, the 90p latency of a single request for the EC2 and Boxer-EC2-only deployments are 3.18 ms and 5.07 ms, respectively. Note that these latencies are measured end-to-end, thus include multiple internal microservice to microservice requests. The write workload demonstrates similar results. The EC2 and Boxer-EC2-only deployments become saturated at 1411 ops/s and 1294 ops/s, with latencies of 7.07 ms and 7.56 ms, respectively.

A similar analysis is used to measure the overhead of launching the logic layer services in AWS Lambda by comparing the Boxer-EC2-only and Boxer deployments. Figure 3.9 shows that for the read workload, the Boxer deployment saturates at 3556 ops/s with a 90p latency of 7 ms. For the write workload, the same deployment saturates at 1189 ops/s and with a 90p latency of 4.55 ms.

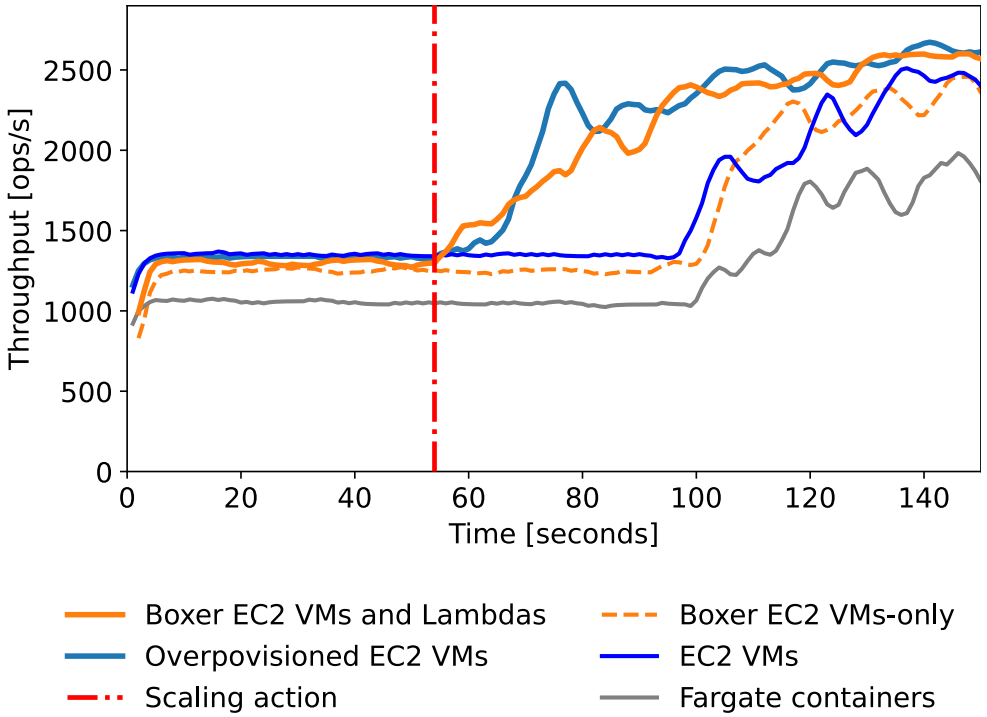

Figure 3.10: *DeathStarBench write workload comparing scaling with different elastic deployments (Section 3.3.2).*

Using Boxer incurs a small performance overhead arising from the interception and management of connections between the multiple services. Moving services to Lambda also incurs a small overhead due to the different ways CPU and network are allocated to VMs and Lambdas. One could increase the memory budget assigned to lambdas to increase their vCPU allocation and thus close the gap between Boxer-EC2-only and Boxer.

Elasticity through Boxer

This section demonstrates how Boxer can provide *ephemeral elasticity* to increase the elasticity of microservices running on VMs(and containers) by leveraging serverless FaaS platforms. The ephemeral elasticity is show with the dynamic deployment experiment based on the DeathStarBench benchmark. In the experiment, initially, all logic layer services are deployed on VMs, when the load increases, additional logic layer services are allocated to handle the increased load. The resources for the additional services are allocated either using AWS EC2 VMs, AWS Fargate serverless containers, or AWS Lambdas functions (enabled by Boxer). In addition, measurements of an overprovisioned VM de-

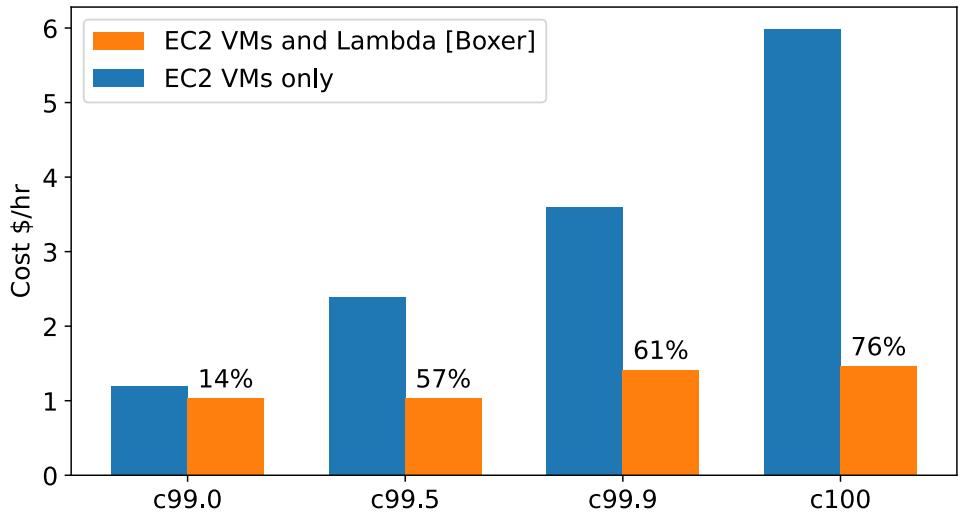

Figure 3.11: *DeathStarBench logic layer absolute cost and cost reduction based on 1-day Reddit trace sample (Section 3.3.2).*

ployment (*Overp. EC2*) are provided as the baseline. In the overprovisioned deployment, already allocated and running VM resources are added to the pool of workers. The goal is to compare the level of elasticity provided by the different deployment types.

Figure 3.10 presents a throughput trace for the different deployment scenarios. Throughput is measured using wrk [wrk09] by looking at how many requests the front-end layer can handle per second. The tool dynamically increases the throughput based on the perceived system capacity. After approximately 55 seconds (dashed vertical line), a scaling action is taken to add a total of 12 workers to the pool of workers in the logic layer (one extra replica for each service in the logic layer). Using EC2 and Fargate takes approximately 45 seconds to fully deploy all new workers (t=100s). While using Lambda and overprovisioned EC2 VMs scale almost immediately (approximately 1 second, t=56s). Using Boxer to accommodate bursts reduces the time to add new workers to the pool by approximately 45× compared to EC2 and Fargate, providing comparable performance to VM-based overprovisioning.

Figure 3.11 compares the cost for using EC2 VM-based overprovisioning and Boxer-enabled ephemeral elasticity (using EC2 and Lambda). Based on a 1-day Reddit trace sample and the DeathStarBench throughput benchmarks (Figure 3.9), the necessary VMs for the logic layer to be overprovisioned to handle at least 99.0, 99.5, 99.9, and 100 percentile of requests/s in the trace (EC2-only) are calculated. The cost of allocating a single VM

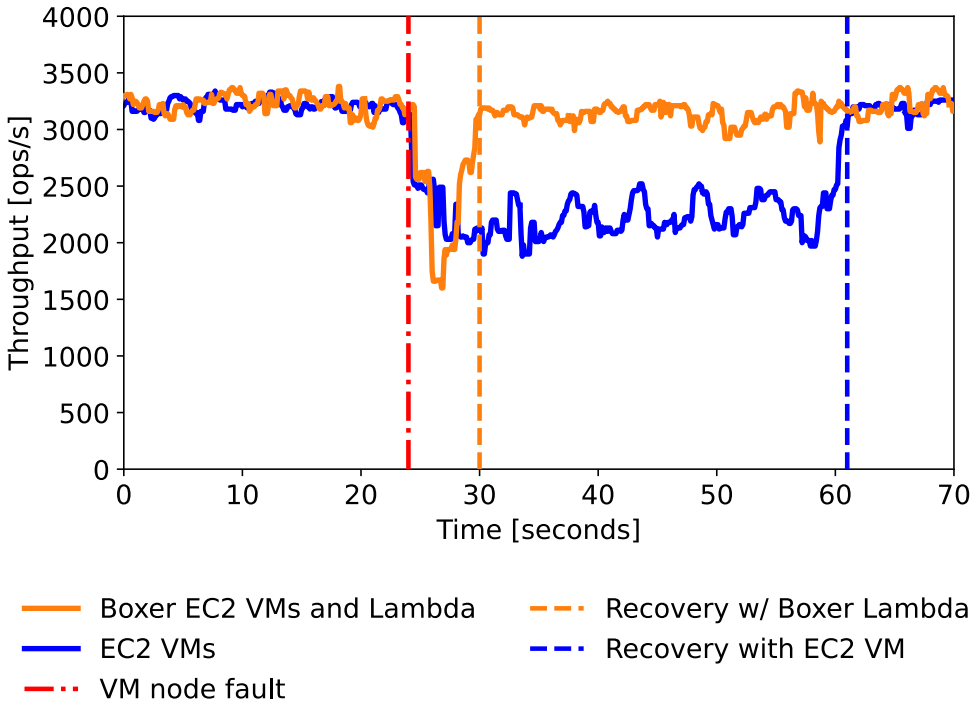

Figure 3.12: *Recovering from node crash in a 3-node EC2 Zookeeper cluster using EC2 and Lambda using Boxer.*

instance for each logic layer service in VMs and on-demand dynamically scaling up using Boxer to Lambdas to absorb load bursts is compared. The cost reduction for using Boxer-provided ephemeral elasticity ranges from 14% to 76%, depending on the target capacity levels.

3.3.3 Elastic Fault Tolerance in Zookeeper

Boxer provided ephemeral elasticity can also be used to reduce downtime (or duration of reduced availability) due to recovery from node crashes. Minimizing node downtime is crucial in highly dependable systems such as Zookeeper as read throughput drops and system guarantees may become compromised if additional faults happen before the first one is recovered. Moreover, having larger Zookeeper clusters to accommodate more faults is not common, as write throughput degrades with additional replicas.

To demonstrate this scenario, a 3-node Zookeeper [HKJR10] cluster is deployed on EC2. Using this cluster, one of the nodes is forcibly shutdown and recovery from the fault is performed using either a newly allocated EC2 VM or a Lambda function with Boxer.

Zookeeper nodes use `t3a.micro` VMs and Lambdas with 2048 MBs. Zookeeper is configured to allow dynamic reconfiguration, i.e., to automatically adapt the quorum every time a new node leaves or joins the network. Boxer is used to transparently allow a Zookeeper node deployed in a Lambda instance to join the quorum. In this recovery scenario, the rapidly deployed Zookeeper node in the short-lived Lambda function stays active only while a more permanent replica is being instantiated. The read-only workload based on the Zookeeper Benchmark [Zoo29] is used in the experiment.

Figure 3.12 shows an execution trace of the workload throughput through time. After approximately 25 seconds, one of the Zookeeper nodes is shutdown, and a new instance, either based on a new EC2 VM or Boxer-enabled Lambda, is started to replace the failed node. Using EC2 VM resources requires 37.0 seconds to recover from the fault. While using ephemeral elasticity based on Boxer and AWS Lambda, only 6.5 seconds were needed for fault recovery. This demonstrated a 5.7× improvement in Zookeeper node recovery time.

3.4 Discussion

The elasticity bottleneck shifts from resource allocation to the application. By leveraging the ephemeral elasticity, datacenter applications can now quickly gain access to new resources; however, that does not necessarily mean that the applications can leverage the resources as quickly as they become available. For example, load balancers, or controllers, may be configured to rebalance their workload among workers at too high intervals. In some cases, this may be as simple as changing configuration parameters, but going from 10s seconds of granularity to 100s of milliseconds in some applications may require substantial changes.

Another aspect of datacenter applications that is amplified by availability of fast ephemeral elasticity is the application startup times. Application startup times can dominate the time needed for the application to start using the newly allocated resources. These systems were designed as long-running services, and startup times were not optimized. However, this scenario may be addressed by starting applications from snapshots, which recently became available in AWS Lambda [aws23b].

3.5 Related Work

Fast ephemeral elasticity based on the uniform interface to VMs and FaaS instances provided by Boxer allows off-the-shelf datacenter applications to adapt to node failures and unpredictable load spikes with reduced level of overprovisioning and in a cost-effective way. This is similar to other systems that utilize VMs and FaaS instances to accommodate failures and load spikes. ML inference serving in MArk [ZYWY19] uses VMs due to their cost-effectiveness and FaaS to quickly add resources when the system must scale. Beehive [ZWT+23] offloads time-consuming closures from JVM-based applications to FaaS during load spikes. Pixels-Turbo [BSA23] is query engine that by default processes queries in VMs and to accelerate processing unpredictable load spies invokes additional FaaS functions. Cackle [PFCM23] is a query execution engine services persistent demand using VMs, and rapid demand spikes using FaaS functions. Microless [CZL+23] is a custom platform based on Kubernetes that aims to support hybrid deployment of microservices between VMs and serverless functions to improve cost efficiency. Boxer pursues a similar vision, but rather than just accelerating particular domain-specific systems, it aims to leverage publicly available FaaS for the elasticity acceleration of unmodified off-the-shelf cloud applications.

Improving elasticity by making compute and memory resources more fungible is an active area of research [RPA+23, RLF+23, AGF+20]. However, with Boxer, cloud users can benefit from high elasticity and significantly reduce overprovisioning for their applications without waiting for cloud providers to evolve and optimize their underlying infrastructure.

3.6 Conclusion

This chapter motivated and demonstrated the feasibility of providing *fast ephemeral elasticity* to off-the-shelf datacenter applications. It showed that Boxer can provide the necessary *network-of-hosts* model that can uniformly span long-running virtual machines and short-lived FaaS functions. It showed that this augmented elasticity can be made available to unmodified datacenter applications on publicly available FaaS infrastructure without any privileged access.

With microbenchmarks it demonstrated that the Boxer provided networking in FaaS is comparable to that available to some EC2 VM types, and that it is compatible with the requirements for providing fast ephemeral elasticity.

Based on benchmarks of unmodified datacenter applications, it also demonstrated that the availability of such fast ephemeral elasticity provides elasticity-fill that can significantly reduce the level of overprovisioning required to react to unpredictable dynamic load and for failure recovery.

The next chapter explores the possibility of instantiating datacenter applications on per-query granularity, with focus on serverless data analytics.

4

Per-request Systems

4.1 Introduction

This chapter examines the possibility of instantiating complete datacenter systems on per-request granularity, enabling the *Per-Request Systems* paradigm. The per-request systems paradigm refers here to allocating resources and instantiating a complete, possibly distributed, datacenter system in response to a user request. After processing the request, the system can be shut down, and resources can be released. [1]

This chapter focuses on per-request system paradigm in the context of serverless data processing systems. As the first step, in Section 4.2, Lambada [MMA20], a purpose-built serverless data processing system, is significantly accelerated by leveraging the Boxer-provided direct networking between concurrently executing functions; it is transformed into a networked serverless data processing system. Second, in Section 4.3, the Boxer system is further developed to go beyond custom-built serverless data processing systems to support popular unmodified off-the-shelf datacenter distributed data processing engines, such as Apache Spark [ZXW+16] and Apache Drill [Apa20], on publicly available serverless platform. Third, Section 4.4 describes the Ephemeral Per-query Engines paradigm and proposes MetaQ, a system that aims to instantiate the best query engine and configuration

[1] Of course, optimizations, such as caching and batching, can play an important role in this model too.

for each query, pointing to the additional unique benefit of per-request systems, where the system can be specialized to a particular request.

The ability to instantiate complete datacenter systems on per-request granularity contrasts with the standard *service-oriented* paradigm dominating contemporary datacenters, where many user requests are processed by long-running systems using long-term resource allocations. Conceptually, in the per-request system paradigm, the *system is attached to the request* sent to a platform that dynamically instantiates the system to process the request instead of sending requests to a running system. Although at very different scales, this resembles the concept dating back to Active Networks [TW96], where network data packets (analogous to requests) also contained the code to be used to process them (analogous to systems) by the receiving routers.

Time scales are of significance. The per-request systems become viable once the resource allocation (and system instantiation) are small enough not to interfere with the expected request processing latencies. Of course, today, this may not yet apply in the domain of microsecond-scale RPC requests. However, at the scales of end-user requests, where human perceivable latencies are in 50-150 millisecond range [ZKK03], and publically accessible FaaS microVMs can be allocated in similar range [ABI+20] (and in 10s of milliseconds warm), the per-request systems paradigm is becoming a possibility.

In the end-user per-request paradigm, users can 'bring all their services with them' at per-request granularity. This may remove the necessity to aggregate many requests (and, therefore, multiple end-users) into single long-running concentrated services. Consequently, this paves the path to a world where each user can be in control of their own choice of service stack, e.g., each end user can have their own social media services, search engines, AI systems, providing independence from the multi-user service providers of today (a vision suggested already [PZ17, JPV+17]). Furthermore, the consequences on the cloud services landscape can be similar, as systems services become instantiated at per-request granularities (e.g., data processing engines), system services offerings in the cloud, and by the cloud providers, may become unnecessary and less desirable. Possibly eventually returning the cloud systems to the role of low-level resource providers for various low-level state (and caching), computation and communication resources; and with time reducing them to (possibly public) commodity utilities.

Although the long-term path remains speculative, this chapter demonstrates a concrete step in that direction by demonstrating Boxer as a system to support the per-request system paradigm in the context of serverless data processing, where users' requests take

the form of SQL queries, and the per-request instantiated systems are query engines. The utility of this scenario has been already demonstrated for occasional and interactive data analytics [MMA20, PCFDM20].

4.2 Networked Serverless Data Processing

This section demonstrates how, using Boxer, a purpose-built serverless data processing systems can be transformed into a networked serverless data processing systems. Based on Lambada [MMA20] serverless data processing system, the effects of accelerating such a system by providing networking support are measured.

Serverless data processing systems such as Starling [PCFDM20] and Lambada [MMA20] have shown that using serverless platforms for data processing is possible and useful for occasional and interactive use. However, recent work has also shown that for data processing applications (regardless of whether it is OLTP, OLAP, or ML), existing serverless platforms are inadequate [HFG+19, MBK+20] and additional services are needed in practice, often to address the lack of communication capabilities between functions [KWS+18, MMA20, PCFDM20]. This section demonstrates how Boxer-provided function-to-function communication can be used to implement serverless data processing more efficiently than previously possible. The benchmarks show a speedup as high as 11 × in TPC-H queries over systems that use cloud storage to communicate across functions.

Boxer extends Lambada [MMA20] serverless data processing system with support for TCP-based stream communication. The new implementation is not only simpler and less specific to a particular environment, it also significantly outperforms the original version based on data exchanges through cloud storage: Most TPC-H queries are both faster and less expensive by a factor of 4 × and 6 ×, with some queries enjoying an improvement as high as 11 × and 15 × in running time and cost reduction, respectively.

4.2.1 Lambada Before Networking

Lambada [MMA20] is a data analytics system built for serverless functions that relies on S3 to exchange data between functions using an exchange operator that is among the most efficient published so far.

In Lambada, only a small component of the system referred to as *driver* resides on the laptop or workstation of the user. It consists of the user interface, query optimizer, and compiler, as well as part of the query execution layer. All other components are serverless components that are fully managed by the cloud provider. In particular, Lambada uses serverless functions as workers of the execution layer, which communicate among themselves and with the driver through shared serverless storage, i.e., a combination of message queues (*SQS*), cloud storage (*S3*), and key-value stores (*DynamoDB*).

The driver translates queries into executable plans and starts executing them. After some potential pre-processing, it launches a potentially large number of serverless functions in order to execute data-parallel plan fragments representing the bulk of the work. Queries typically read one or more input tables from cloud storage and either store the result again on cloud storage or return their intermediate results to the driver, which may post-process them before returning them to the user. For more details about Lambada, refer to the original paper [MMA20].

4.2.2 Alternatives for Data Shuffling

Distributed data processing systems rely on efficient shuffling of data between nodes to produce results. Because serverless functions cannot communicate directly with each other, serverless data processing systems resort to a variety of alternative mechanisms to shuffle data among functions. The main alternatives for transferring data between functions are based on using cloud-based storage services such as Amazon S3 or auxiliary VM-based intermediary data forwarding services.

Leveraging a VM-based intermediary service to exchange data can require additional infrastructure to be managed, increase the request latency times, and become a scalability bottleneck, especially for parallel query processing. For data processing, where the amount of data being exchanged can be large, such as a multi-staged exchange operator, this is an inefficient approach due to the amount of data copying involved.

For data analytics, an alternative to implementing such extra services is to exchange data through cloud storage using an exchange operator writing to and reading from permanent storage services. In Starling [PCFDM20], intermediate data is exchanged between functions using Amazon's S3 [Ama17]. The high latency of the service is hidden by using parallel read requests per function and, for actual data shuffling requiring all-to-all communication, the ability to read only a fraction of a file that S3 provides is used so that

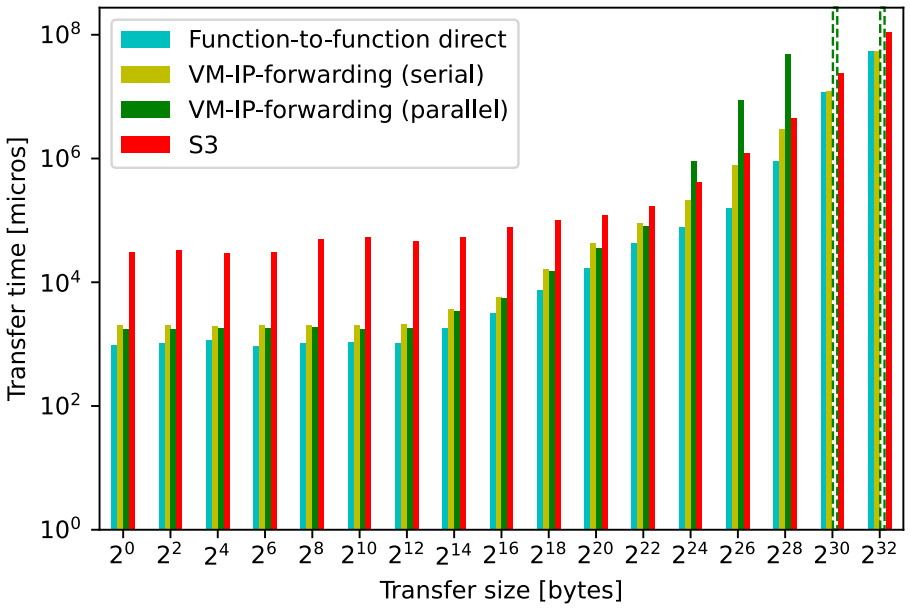

Figure 4.1: *Data transfer times (RTT/2) between function pairs using different transport methods. The median times of 16 transfer pairs are reported for each configuration. Dashed bars represent unfinished transfers due to function timeouts. All functions are 10GB and VMs are m4.xlarge*

the amount of data to be moved is minimized. Lambada [MMA20] also uses S3 but uses a more sophisticated exchange operator. These two systems have the advantage of not requiring any external service but, despite their efforts to minimize the overhead, their performance is still limited by that of S3, both systems show that data analytics over serverless today is only cost-efficient at a low query throughput.

As a comparative illustration of the challenges of the data transfer between functions, consider the microbenchmark comparing data transfer times between function pairs using different methods available today for AWS Lambda functions. Figure 4.1 shows absolute times for transferring different data amounts between function pairs using S3 cloud storage, VM-based intermediary, and function-to-function direct transfers using Boxer. To illustrate the possibility of an intermediary becoming a bottleneck, the VM-based intermediary measurement is subdivided into a configuration where all function pairs use one intermediary in parallel (labeled 'parallel') and a measurement where the intermediary is used by only one function pair at a time (labeled 'serial'). The VM-based intermediary

is based on functions setting up TCP flows between each other using a VM (AWS EC2 m4.xlarge) as an intermediary that forwards the traffic at the IP level using kernel-level forwarding (described in Section 2.6.3.2). Because this forwarding does not involve transferring data to userspace or using the TCP stack on the forwarding VM, it can be seen as representing the lower-bound of any intermediary service that would use a userspace-based mechanism (as all known proposed solutions do.)

First, notice that transfer times based on S3 cloud storage are significantly slower for all transfer sizes. Secondly, notice the difference between the two VM-based intermediary-based approaches, as the transfer sizes become greater the need to share the available bandwidth of the intermediary VM becomes the bottleneck, first increasing the transfer times, and eventually slowing down to the point where there is not enough bandwidth available to complete the transfers before the AWS Lambda maximum function duration time. Most significantly, notice that the direct (Boxer-based) function-to-function transfers are the fastest for all transfer sizes and do not suffer the intermediary-based solution's scalability problems. With the largest transfer sizes, the relative advantage of the direct function-to-function transfer times compared to transfer-dedicated intermediary VM (serial) is reduced as the dominating factor becomes the bandwidth limits imposed on the functions.

4.2.3 Lambada with Networking

Lambada serverless data processing system is used to demonstrate the feasibility and benefits of using Boxer to accelerate serverless data analytics with networking. First, as the baseline, Lambada is used to execute TPC-H benchmark queries using the S3 cloud storage-based exchange operator, and then it is compared to running the same benchmark while using Boxer function-to-function communication in its exchange operator.

4.2.3.1 Baseline Lambada using S3

In the original version, which is used as the baseline, Lambada uses serverless cloud services for indirect communication among the workers. In particular, it implements operators that need to shuffle data among the workers such as joins, grouping, reduction, and sorting, using a purpose-built exchange operator that communicates through files on cloud storage. The basic approach is to write all data that needs to be sent from a particular

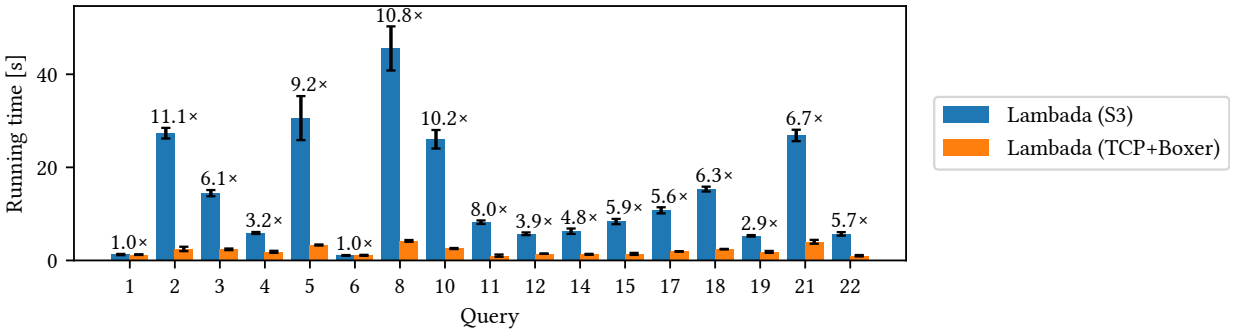

Figure 4.2: *Running time of TPC-H queries using Lambada based on TCP+Boxer or S3 for communication.*

worker to another one into a file whose name is based on the two worker IDs, such that the receiver can poll on that file until it exists and read its content. While this approach makes it possible for workers to communicate without a direct network connection, its basic version does not scale to a large number of workers: Since it involves a quadratic number of files (one for each pair of workers), it runs into service limits of the cloud storage system and incurs a quadratic request cost. To overcome this issue, Lambada does the exchange in two rounds, each within a sub-group of $\sqrt{P}$ of the P workers, which requires only $\mathcal{O}(P \cdot \sqrt{P})$ accesses to cloud storage, and combines all data sent by each worker into a single file, reducing the number of write accesses to just P. Refer to [MMA20] for details.

4.2.3.2 Networked Lambada using Boxer

To measure the impact of Boxer on query processing, Lambada was extended with a traditional TCP-based exchange operator that uses the interfaces provided by Boxer. This operator reads its upstream data, partitions it into buffers (one for each other worker), and sends each buffer to its target worker as soon as it is full using a regular TCP connection. In order to reduce the number of connections, one connection among all operators in a query for every $\langle sender, receiver \rangle$ pair was reused. Neither the design nor the implementation of this operator is specific or even aware of the serverless setup. In each worker, Boxer loads the process monitor library into Lambada's query processing engine (as described in Section 2.6), and the engine uses standard stream socket interfaces.

The two versions of Lambada are evaluated and compared using TPC-H at Scale Factor 10. As in the original paper [MMA20], the data was generated in a dictionary-encoded form and the queries were formulated against the dictionary codes in order to avoid implementing string support. The data was stored in Snappy-compressed Parquet files on S3, totaling about 3.2 GB. For the workers, serverless functions with 2 GiB main memory were used, which gave them the timeslices of slightly more than one vCPU. Lambada misses features to run the full benchmark, the missing features are unrelated to the network layer and are limitations of Lambada: Queries 9, 13, 16, and 20 need string or null support and Queries 7 and 20 need a broadcast operator or support for multiple stages.

All experiments used 64 function workers and a single-level exchange for both variants, which simplified the setup and interpretation of the experiments. In order to isolate the effect of the communication mechanism, an artificial barrier at the beginning of each inner plan fragment was added (implemented using Boxer in both versions) and the time was measured from the moment the first worker passed that barrier until the last worker completed its plan fragment. Except for using two different versions of the exchange operator, the plan fragments run by both versions were exactly the same. All reported measurements represent averages of five runs.

4.2.3.3 Query Execution Time

Figure 4.2 shows the running time of the two variants. As the plot shows, Lambada is consistently and significantly faster using direct communication via Boxer than using indirect communication via S3 cloud storage. The numbers above the bars indicate the speed-up of the former over the latter. The performance difference is mostly in the order of $4\times$ and $6\times$ and has a strong correlation with the number of exchange operators in the data-parallel plan fragments: Queries 1 and 6 do not exchange any data and thus have the same running time with both variants as expected. Queries 4, 12, 14, 15, 19, and 22 have a single join, i.e., two exchange operators, or a join and an aggregation, each, so the speed-up of the two variants is mostly in the range of $4\times$ and $6\times$. Queries 3, 10, 11, 17, and 18 use four to seven instances of the exchange operator, so the TCP-based version tends to have a greater advantage. Queries 2, 5, 8, and 21 use ten or more exchange operators and therefore the performance gain is the largest for these queries.

The performance difference comes from two main effects: First, while indirect communication requires the sender to complete all writing (per exchange operator) to cloud storage

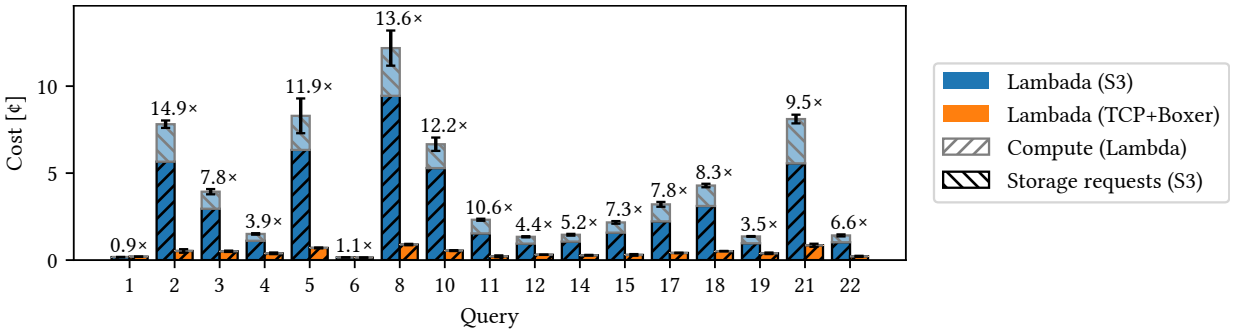

Figure 4.3: *Monetary costs of Lambada on TPC-H using TCP+Boxer or S3 for communication.*

before the receiver can start reading any data, the direct communication can be done in a streaming fashion using reasonably small chunks. The second effect is due to the higher latency of accessing cloud storage compared to function to function communication via Boxer, which explains why the benefit of direct communication is greater for more complex queries.

4.2.3.4 Monetary Query Cost

The cost of the CPU time used by the workers running in serverless functions as well as the request cost for the cloud storage, which are mainly done by the exchange operator, are also analyzed. The other costs of running the query (1) request cost to the serverless functions and (2) storage cost of temporary results on cloud storage, are both relatively minimal, so they are excluded here. Except for the seed Boxer coordinator (Section 2.6), there is no cost of setting up infrastructure or keeping it running between queries.

Figure 4.3 shows the costs of running the same queries as above. The numbers above the bars indicate the cost reduction of using communication through Boxer compared to using cloud storage. The numbers are strongly correlated to the running time: using Boxer reduces the costs by a factor between $4\times$ and $6\times$ for most queries and significantly more for the more complex ones. This is not surprising since the costs are dominated by the CPU time of the serverless functions. However, the request costs of accessing S3 have a significant share as well (GETs and PUTs to S3 are not for free and the total cost accumulates due to the many accesses needed to implement an exchange operator for parallel query processing); often in the order of a quarter and up to a third (for Query

21). Interestingly, the request costs alone are higher than the total costs using Boxer for almost all queries.

4.2.4 Future Opportunities

The increasing elasticity of the cloud makes it possible to use thousands of cores for jobs that take just a few seconds to complete. These extremely wide but short-lived jobs create challenges for many system components, even if Boxer makes direct networking among the workers possible. In particular, failures or at least delays become increasingly likely with the "width" of a job but must be dealt with at extremely short time scales. This is true for the start-up of large numbers of workers, the connection setup among these workers (which requires quadratic overall work for a fully connected network that Boxer currently used), the efficient handling of large numbers of connections at each worker, and many more. Improvements could be based on generic solutions such as indirect routing, sparser connection topologies, such as the grid-based topology used in Lambada's two-level exchange operator, or hedged computations and/or communication, i.e., redundant executions of latency-critical phases where only the first to complete contributes to the final result. This points to the possibility of the application informing the substrate layer of its relaxed constraints. For example, instead of always providing all-to-all connectivity, Boxer could allow Lambada to specify the network topology it will require in advance. Lambada, based on the knowledge of the query plans, may determin that a fully connected network topology is not necessary (e.g., trees are probably a better, more natural option for query processing), leading to relaxed requirements.

4.2.5 Summary

This section demonstrated how Boxer can be used to implement and accelerate a modified version of Lambada [MMA20] serverless data processing system. Comparing the performance of using the original S3-based exchange operator and using Boxer-based network communication, significant performance gains can be observed for most of the queries.

4.3 Serverless Off-the-shelf Data Processing

Motivated by the research indicating that serverless is ill-suited for data analytics due to limitations of the current platforms [HFG+19, WLZ+18b], there is a growing amount of work on serverless data analytics along two main lines. One approach is to build new engines specifically designed to work around the limitations of existing serverless platforms, like Lambada [MMA20] considered in the previous section, and others [PCFDM20, JGL+21, WDH+22]. Another approach involves extending existing serverless platforms, often with VM-based services, to better support data analytics workloads on serverless infrastructure [RCG+21, WFLH18, SWL+20a, KWS+18].

All these efforts provide valuable insights on serverless data analytics but also implicitly, by constructing new engines, they give up on the many existing distributed data processing platforms (e.g., Spark, Flink, Drill, etc.) and by extending serverless or modifying platforms, depart from actual commercial offerings of serverless platforms. In practice, this amounts to having to completely redesign data processing engines from scratch and/or building on the assumption that open source research serverless platforms will be more efficient and feature-complete compared to the native serverless offerings from cloud providers.

This section explores and proposes an alternative solution for data analytics on serverless: to provide the functionality required to *run off-the-shelf distributed data processing platforms (e.g., Apache Spark or Drill) on top of existing commercial serverless platforms (AWS Lambda [AWS17]).*

Although practically challenging, doing so is based on two insights: that serverless FaaS platforms can be extended without modifying the underlying infrastructure, and that contemporary publicly available serverless functions are already similar in memory size and computational power to what VMs were at the time when most popular data analytic platforms were initially developed (see Section 2.2).

To test the feasibility of the idea, Boxer (Section 2.5) is used to transparently provide the environment required by the engines. Boxer makes the function environment appear like standard networked containers or VMs to the unmodified data processing engine processes. The mechanism does not involve any changes to the serverless platform and provides a complete interface that enables running unmodified distributed data engines.

In this section, Boxer is used to execute unmodified Apache Spark [ZXW+16] and Apache Drill [Apa20] on AWS Lambda and run the TPC-H benchmark. The performance obtained is comparable to that observed when running the base systems on a comparable class of conventional EC2 VMs, thereby proving that the approach is a viable alternative to having to completely redesign either the engines or the serverless platforms.

4.3.1 Approaches to Enable Data Analytics on Serverless

There are two main approaches to enable data analytics on serverless. One is to design new engines specific to serverless; the other is to develop more amenable serverless platforms. Of course, a combination of both approaches is also possible. Examples of the former approach are Lambada [MMA20] and Starling [PCFDM20]. Both propose query engines atop unmodified serverless platforms that use a varied number of techniques to work around the limitations of serverless platforms (mainly optimizing communication through storage and speeding up start-up times for many functions). These systems have inspired other work extending the ideas to, e.g., machine learning [JGL+21, WDH+22]. Pixels-Turbo [BSA23] augments a long-running VM-based data analytics system with specialized serverless workers to accelerate processing in response to sudden load spikes. Crackle [PFCM23] models a unified query engine that can execute both on top of VMs and unmodified serverless platforms to reduce overall cost. Similarly, Sponge [SUE+23] is a custom stream processing engine that executes on VMs and unmodified serverless platforms to redirect some processing in response to load spikes.

Regarding alternative serverless platforms, the focus has been on providing, e.g., stateful function services [SWL+20b, AFK19, WSH20b] or extending serverless with, e.g., ephemeral storage solutions [KWS+18, PVS19] that offer a fast data plane option for serverless functions.

Neither of these approaches take advantage of existing mature data analytics systems (e.g., Apache Spark, Drill) and existing commercial serverless platforms (e.g., AWS Lambda). As a result, existing applications need to be rewritten to use the new platforms. It is also questionable whether the new serverless platforms will ever be as competitive and efficient as the native solutions of cloud providers.

The approach put forward here is based on a simple observation: serverless worker instances are already similar to small VMs (some are based on microVMs [ABI+20]). Supporting existing data analytics engines in serverless is, thus, a problem of ensuring that

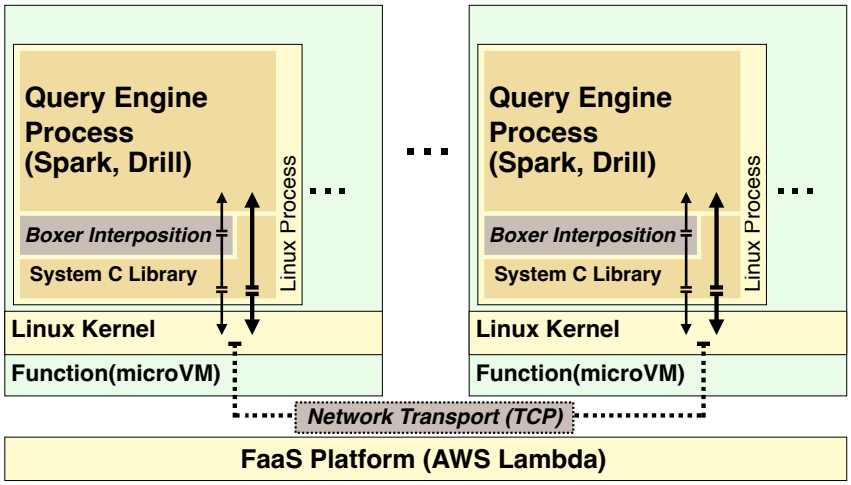

Figure 4.4: *Unmodified distributed query engines executing in parallel networked AWS Lambda functions.*

the environment available in serverless workers is sufficiently similar to the one available in serverful VM/container instances. With that, one should be able to use existing distributed data processing engines without changing either the underlying serverless platform nor the engines themselves.

4.3.2 Distributed Query Engines in Serverless Environments

The serverless FaaS paradigm is based on fine-grained, event triggered computations that are composed into dataflow graphs. Computations are based on Linux processes in both serverful VMs and serverless functions. From that perspective, applications such as distributed query engines running on VMs should, in principle, be able to run in serverless functions. Unfortunately, the environment and computation composition model of serverless platforms significantly differs from that of conventional VMs (see Section 2.3). Mainly, distribution in serverless platforms is based on an event-based dataflow composition model, while in conventional VM applications, like distributed query engines, it is based on parallel networked processes. Because of this, conventional distributed engines do not work on serverless platforms as they assume access to direct communication with other concurrently executing remote processes.

To bridge this gap, in the work described in this section, Boxer system is used to transparently provide existing data engines with the required model of network of parallel exe-

cuting processes on top of existing FaaS platforms. Doing this goes beyond just enabling networking between processes, as it has a number of consequences on the execution environment. Below are outlined the main functions that Boxer provided that were important in making Apache Spark and Apache Drill run in AWS Lambda. Figure 4.4 shows system components needed to execute unmodified query engine processes in AWS Lambda.

Parallel function execution

Unmodified distributed query engines typically assume a static configuration of distributed processes continuously exchanging data. However, in serverless, function scheduling is based on events triggering function executions (control of available concurrency of the underlying resources is exposed in some commercial offerings [AWS27]). This gives the FaaS platforms more freedom for scheduling while preserving the semantics of the dataflow computation, however, it does not match the environment that distributed query engines assume of a collection of active processes running in parallel.

To address this mismatch, the Boxer's initialization mechanism (Section 2.6) was used to create an engine-specific pool of function instances running in parallel. The pool of functions combines functions of different types (roles in Boxer terminology) as needed (e.g., worker nodes, head nodes, auxiliary systems, etc.). Once the required combination of parallel functions is instantiated, the query engine processes are started in the function pool, matching the model that query engines expect.

Application transparency

To support unmodified query engines, Boxer exposes the altered function environment transparently to the query engine processes by selectively intercepting their execution (Section 2.6.1). For example, when a query engine process attempts to connect to a named process running in a remote function, instead of returning an error as it would happen within a serverless function, the name resolution and connect calls are intercepted, network addresses are provided and a connection setup is performed (see below), returning a valid stream socket connected to the named remote function. The query engine process is unaware of the additional connection setup performed and proceeds just as if it would have connected to a remote host process. Boxer does not intercept any data plane operations of the query engine processes, all data writes and reads are processed without the interposition layer being involved.

In addition to the transparency from the perspective of the individual query engine processes, transparency at the system orchestration level is preserved. Query engines are composed of sub-services (e.g, head nodes, worker nodes, meta-data services, coordination services, etc.) that are commonly managed using container orchestration tools, such as Kubernetes [Kub15], Docker Swarm [Doc15], or Docker Compose [Doc27]. To allow for orchestration-level transparency, query engine container images are derived from Boxer base images that can be run via native container runtimes or can execute as FaaS instances. All experiments presented in this section were orchestrated using Docker Compose that either started containers using container runtimes on EC2 VMs (baseline) or as AWS Lambda instances (unmodified engines on serverless).

Network transport

In order for the query engine processes executing in different functions to communicate using the expected socket interface, a network transport mechanism must be provided. Most query engines rely on stream socket semantics to communicate, commonly implemented by TCP protocol (datagram semantics are not used by any of the query engines that were considered here.) Unfortunately, commercially available serverless FaaS platforms do not provide a mechanism to establish TCP connections where the endpoints are different function instances. For this reason, Boxer NAT-traversal transport (Section 2.6.3) was used to provide the connectivity between query engine processes running on different functions.

4.3.3 Evaluating Serverless Apache Spark and Apache Drill

This section describes the feasibility of the proposed approach and the initial measurements of two unmodified existing distributed query engines, Apache Spark and Apache Drill, running on a commercially available serverless FaaS platform, AWS Lambda. Comparative TPC-H benchmark measurements are reported for the two query engines executing in AWS Lambda functions and AWS EC2 virtual machines, showing that distributed query execution performance in serverless functions is comparable to that of a class of EC2 instances. The overhead of the per-query engine initialization times is examined, and future improvements are proposed.

Both Apache Spark and Apache Drill support large-scale analytics with distributed query execution. In the conventional mode of operation, the systems are run on long-running

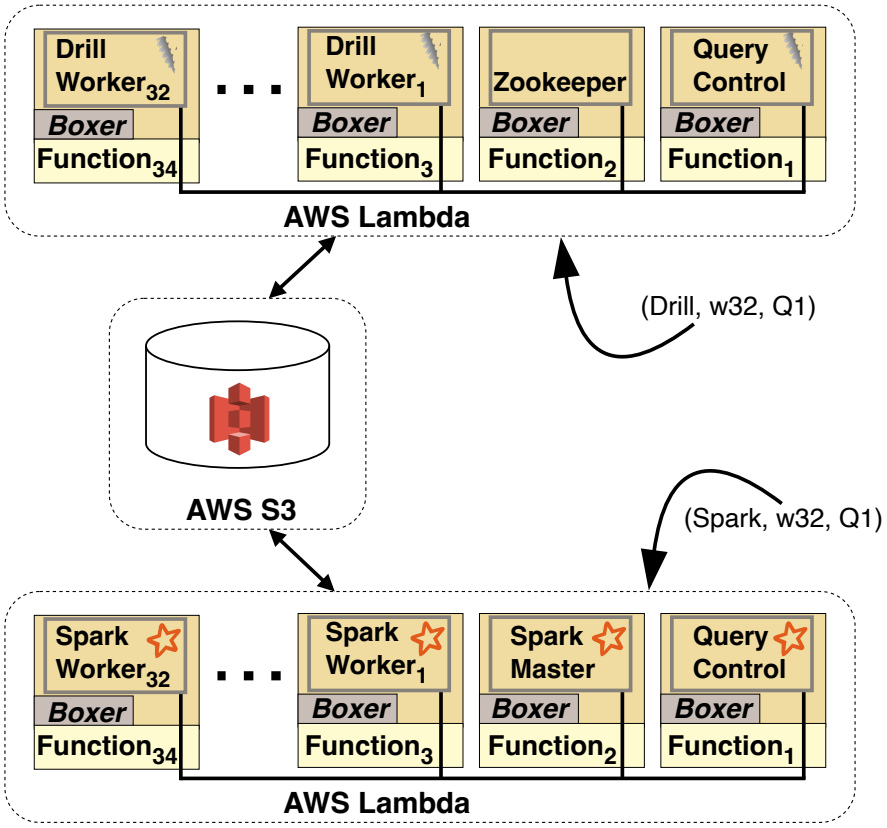

Figure 4.5: *Experimental setup: Example of (bottom) Apache Spark (32 workers, master, and query control node) in AWS Lambda instantiated to execute query Q1 on data stored in AWS S3. Apache Drill (top) (32 workers, Apache Zookeeper instance, query control node) instantiated in AWS Lambda to execute query Q1 on AWS S3.*

dedicated clusters or virtual machines. In contrast to that environment, here the systems are intended to be used for serverless data analytics. Therefore, these engines are run using Boxer (Section 2.5) in short-lived networked AWS Lambda serverless functions. Following the per-request systems paradigm, complete Apache Spark and Apache Drill instances are instantiated when there is a query to be executed and shut down immediately after producing results.

4.3.3.1 Experimental setup

The experiments presented here use the TPC-H dataset at scale factors(SF) 10, 30 and 100 with data stored in Parquet format, compressed with Snappy, and split into 100MB

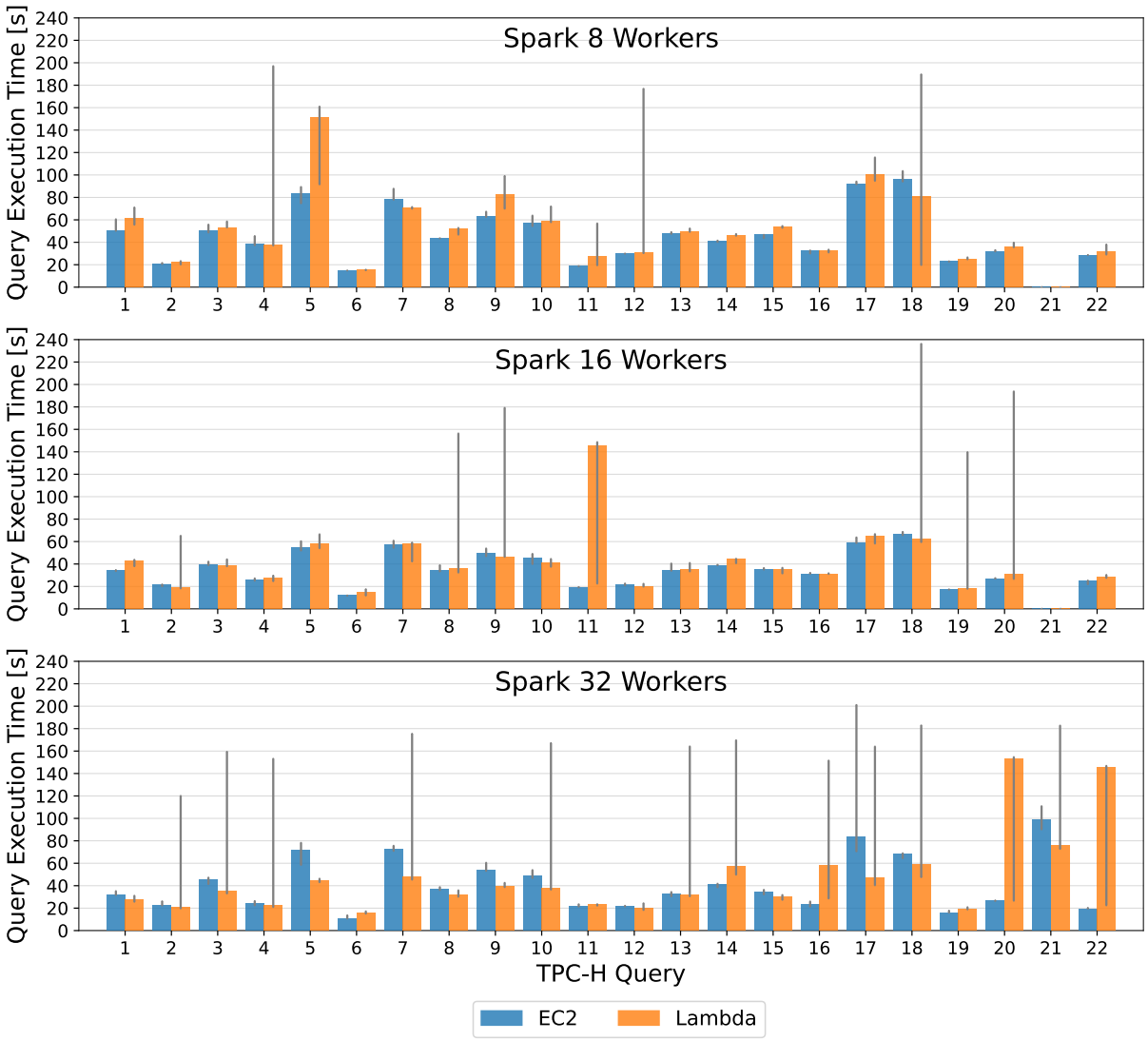

Figure 4.6: *TPC-H (scale factor 100) query execution times for unmodified Apache Spark running in AWS Lambda and AWS EC2. Median times of 3 executions of each query for each configuration, error bars are min. and max. times.*

partitions stored on AWS S3. The resulting compressed dataset sizes range from 3GB for SF-10 to 32GB for SF-100, with the largest relation of SF-100 containing 600 million rows.

The AWS Lambda measurements are based on functions with 10GB of memory and 6 virtual cores, the largest AWS Lambda functions currently available. All query engine nodes and auxiliary functions (explained below) execute in their own AWS Lambda functions

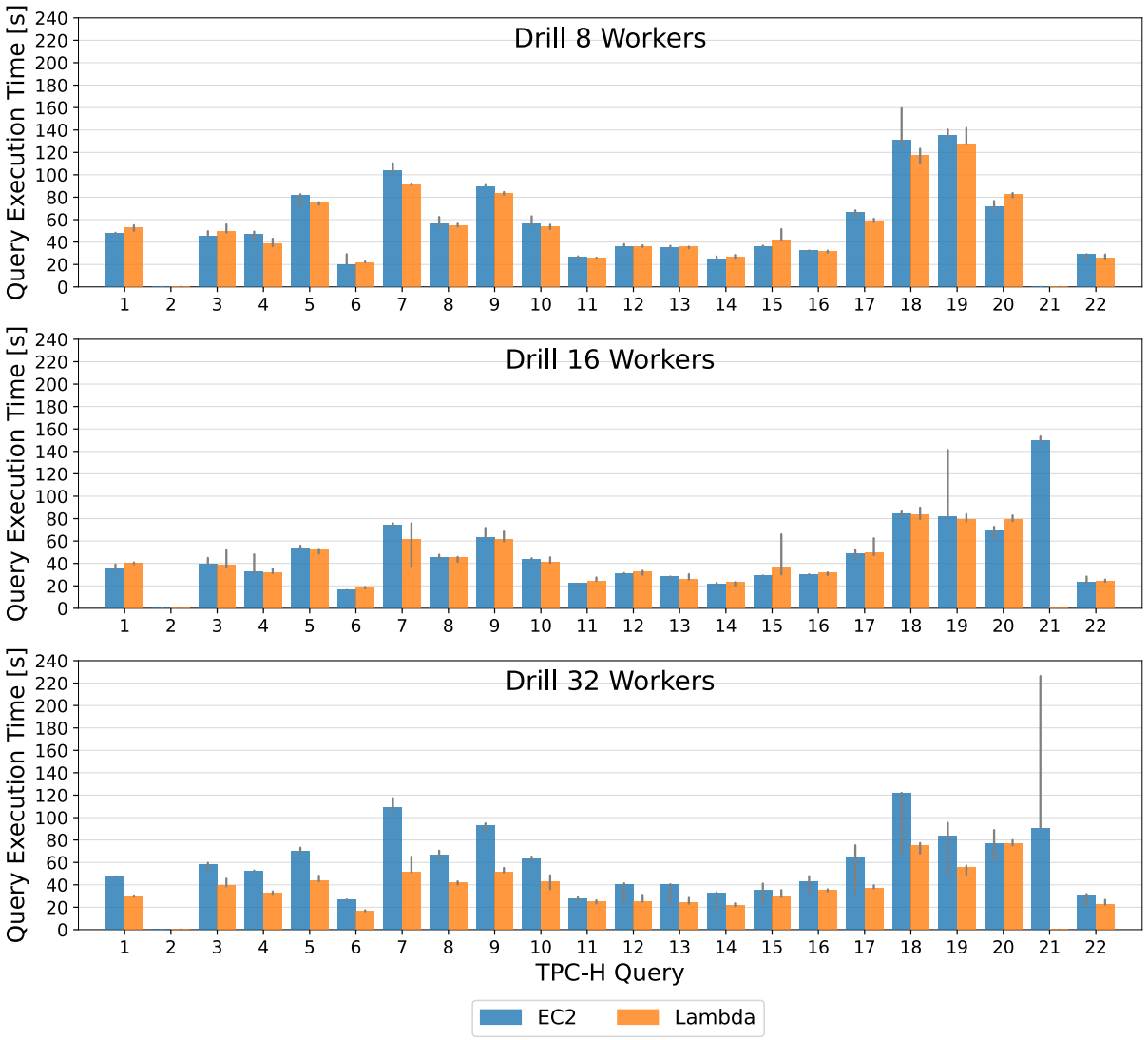

Figure 4.7: *TPC-H (scale factor 100) query execution times for unmodified Apache Drill running in AWS Lambda and AWS EC2. Median times of 3 executions of each query for each configuration, error bars are min. and max. times.*

(Figure 4.5). In the Apache Spark experiment, there are 8, 16, or 32 worker nodes and a master node. In the case of Apache Drill, there are 8, 16, or 32 worker nodes and a single-node Apache Zookeeper [HKJR10] instance (also instantiated in a dedicated function) that Drill relies on for coordination. In addition to the above nodes, every query engine instantiation includes a query control function that is responsible for waiting for

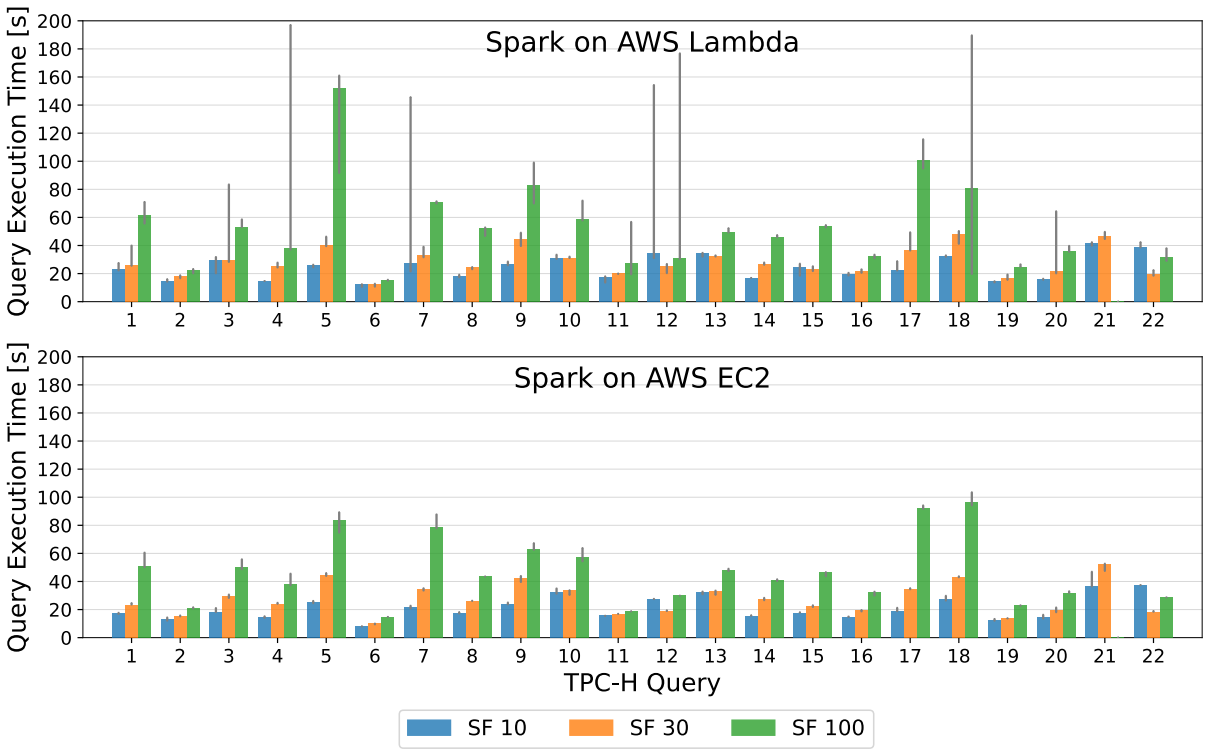

Figure 4.8: *TPC-H query execution times for different scale factors (SF) for unmodified Apache Spark running in AWS Lambda and AWS EC2 with 8 workers. Median times of 3 executions of each query for each configuration, error bars are min. and max. times.*

the query engine to be ready, submitting the specified query, and waiting for the result.

For each TPC-H query measured, all of the above-described query engine instance functions are instantiated, a single query is executed, and then all of the functions are terminated. Each TPC-H query configuration is repeated 3 times. In this experiment, the effects of using warm functions and caching are factored out. To ensure that each query is started in fresh cold functions, after each query, the function registrations are reset so that the platform considers them as new functions, guaranteeing cold starts.

To relate the measurements in AWS Lambda to the virtual machine environment, the AWS EC2 measurements are performed in a symmetric way. Instead of AWS Lambda functions, AWS EC2 virtual machines are instantiated. For each query execution, a new set of EC2 virtual machines is instantiated, a single query is executed, and then the virtual machines are deleted. All measurements are based on t2.2xlarge EC2 instances configured

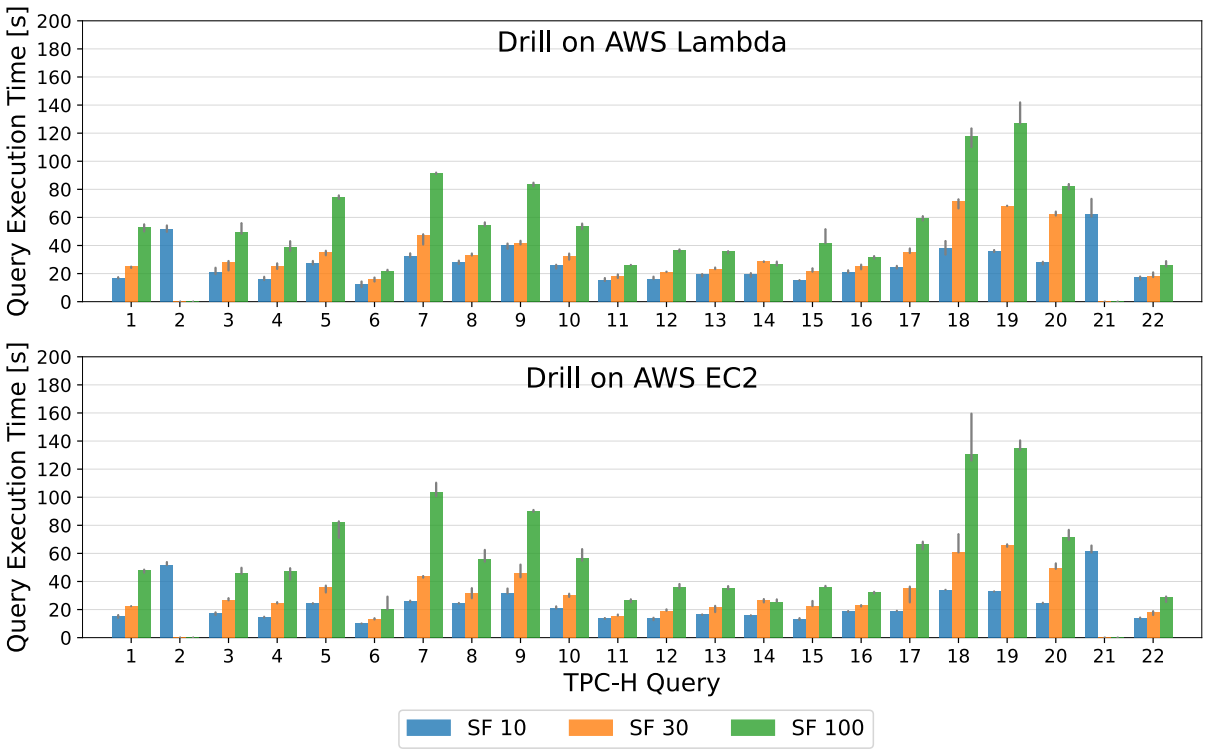

Figure 4.9: *TPC-H query execution times for different scale factors (SF) for unmodified Apache Drill running in AWS Lambda and AWS EC2 with 8 workers. Median times of 3 executions of each query for each configuration, error bars are min. and max. times.*

in unlimited mode (to avoid possible CPU throttling) and limited by the Linux kernel to 6 virtual cores and 10GB of memory (from 8 virtual cores and 32GB of memory). Virtual machines configured this way provide similar performance to the AWS Lambda functions used (also with 6 virtual cores and 10GB of memory).

All of the described nodes are configured as container images derived from Boxer base images. Containers derived from the Boxer base images can be used by conventional container orchestration tools, or they can be selectively run as AWS Lambda functions. Using this functionality, all of the benchmarks used the same container images on EC2 instances as AWS Lambda and used Docker Compose [Doc27] as the container orchestrator for both.

All of the measurements are based on Spark version 3.32, Drill version 1.21.1, Zookeeper 3.7.1, and Amazon Corretto 17 JVM package. The systems are not performance-tuned, the

measurements represent close to out-of-box unoptimized configurations. All measurements were performed on AWS eu-central-1 region using x86_64 CPU architectures.

4.3.3.2 TPC-H measurements

The per-query TPC-H measurements on AWS EC2 and AWS Lambda are shown for Apache Spark in Figure 4.6, and Figure 4.8; and for Apache Drill in Figure 4.7, and Figure 4.9. The main observations from this experiment are that

(1.) *Unmodified Apache Spark and Apache Drill distributed query engines can run in AWS Lambda serverless functions*,

(2.) *the distributed query execution performance in AWS Lambda is comparable to using a class of networked EC2 virtual machines.*

Figure 4.6 and Figure 4.7 show observed query execution times for TPC-H SF-100 for different numbers of worker nodes (8, 16, 32) for Spark and Drill respectively. Figure 4.8 and Figure 4.9 show query execution times for different scale factors (SF-10, SF-30, SF-100) all using 8 worker nodes for Spark and Drill respectively. Comparing the measurements based on VMs and functions, following observations can be made:

Completeness: First, except for a single case (described below), the same sets of queries complete successfully on EC2 VMs as on Lambda functions.

Spark completed all queries on AWS Lambda and EC2 in the SF-10 and SF-30 measurements and SF-100 with 32 worker nodes. However, other SF-100 configurations with fewer than 32 workers did not complete all queries. This is because when Spark was configured with only 8 or 16 worker nodes for SF-100, query 21 could not complete due to the workers running out of memory. The same issue was observed on both platforms.

Apache Drill, in all shown configurations except for SF-10, was unable to complete query 2 due to the workers running out of memory. The same reason prevented Drill from completing query 21 using 8 workers for SF-30 and SF-100. The same issues were observed on both platforms. However, in the case of SF-100 in 16 and 32 worker configurations, Drill was able to complete query 21 on EC2 but not on Lambda. This difference is due to AWS Lambda functions having a limited number of file descriptors available (1024 in total.) For these two configurations, Apache Drill workers attempted to create a sufficiently large

number of temporary files that the file descriptor limit was reached and execution terminated. No Drill configuration option was found that would reduce this unusually large number of temporary files. This is the only observed case where the query execution was not possible using AWS Lambda functions but was possible using AWS EC2 VMs with similar resources.

Dispersion: Second, depending on the data processing system, query execution times can show a larger degree of dispersion in Lambda compared to EC2.

There is a significantly larger degree of dispersion in measurements for Spark running in Lambda compared to that in EC2. However, this difference is not shared by both systems. For Drill, the observed dispersion levels are similar in Lambda and EC2.

The dispersion level for Spark increases with the number of workers used, the number of queries where the maximum execution times is greater than 2× the median execution time is 4, 7 and 11 for 8, 16, and 32 workers configurations respectively (Figure 4.6). The dispersion level does not show a significant dependence on the scale factor for measurements with 8 workers (Figure 4.8). As of this writing, the mechanism that produces the increased dispersion that is specific to Spark on Lambda has not yet been identified. However, it is worth observing that even in the cases when the difference in median times between Spark in EC2 and Lambda is significant, the minimum time in Lambda approaches that of EC2, suggesting that if the high variance can be addressed, the query execution times will consistently approach those of EC2.

Performance: The query execution times of Spark and Drill vary but overall are comparable in AWS Lambda functions and EC2 VMs.

The sums of the query execution times for all TPC-H queries at SF-100 are reported in Table 4.1 and Figure 4.10. Times for each of the 3 executions are summed over all queries, except for queries 2 and 21, which are excluded from the summations because not all configurations completed these two queries. The median of the sums for Spark in AWS EC2 is always faster than in Lambda (between 1.2× and 1.4×), where the high-variance executions contributed significantly to the aggregate differences. If only the aggregate of the minimum times were considered (Figure 4.6), the difference would be significantly reduced, showing that if the variance in Spark executions can be addressed, the absolute times between EC2 and Lambda would be even closer. Apache Drill, exhibiting much

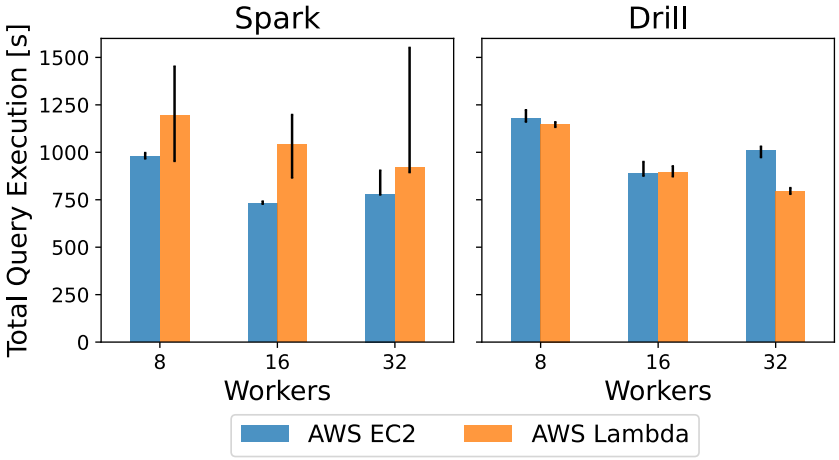

Figure 4.10: *Total time to run all queries (excludes Q2 and Q21) for SF-100.*

Eng.	Workers	Plat.	Median	Mean	Min	Max
Spark	8	EC2	978.76	980.91	965.91	998.08
Spark	8	Lambda	1195.08	1200.31	952.53	1453.30
Spark	16	EC2	731.26	733.52	726.85	742.45
Spark	16	Lambda	1039.12	1034.42	865.39	1198.75
Spark	32	EC2	780.48	820.46	775.76	905.14
Spark	32	Lambda	922.02	1122.65	893.66	1552.27
Drill	8	EC2	1178.38	1187.34	1159.89	1223.75
Drill	8	Lambda	1149.52	1147.24	1131.93	1160.28
Drill	16	EC2	886.40	904.11	874.92	950.99
Drill	16	Lambda	891.85	897.18	871.70	927.99
Drill	32	EC2	1007.71	1003.96	972.75	1031.41
Drill	32	Lambda	794.73	795.77	779.20	813.37

Table 4.1: *Total time to run all TPC-H queries (excludes Q2 and Q21) for SF-100.*

less variance, shows the EC2 and Lambda median of sum times to be much closer; for 16 workers configuration, the difference is minimal (under 1.01×) in favor of EC2, and in the case of 8 and 32 workers, the median of the sum times for Drill is faster in Lambda than in EC2 configuration, with 1.02× and 1.26× speedup respectively.

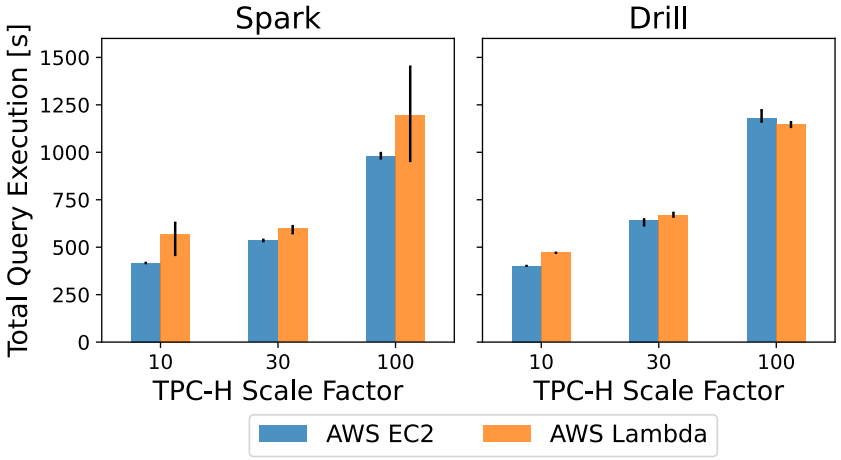

Figure 4.11: *Total time to run all TPC-H queries (excluding Q2 and Q21) using 8 worker nodes.*

		Spark			Drill		
SF	**Workers**	**EC2**	**Lambda**	**L.%**	**EC2**	**Lambda**	**L.%**
SF-10	8	2.35	2.10	0.89	2.94	2.44	0.83
SF-30	8	1.83	1.99	1.09	1.83	1.72	0.94
SF-100	*8*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
SF-100	16	1.34	1.15	0.86	1.33	1.29	0.97
SF-100	32	1.25	1.30	1.03	1.17	1.45	1.24

Table 4.2: *Speedups relative to SF-100 with 8 workers to run all TPC-H queries as scale factor is decreased (to SF-30 and SF-10) or as the number of workers is increased (to 16 and 32). L.% is the fraction of the EC2 speedup achieved by Lambda.*

Scaling: Relative scaling properties of Spark and Drill are similar in AWS Lambda and AWS EC2. Scaling query engines by the number of worker nodes (Figure 4.10) or by the data set scale factor (Figure 4.11) does not result in degenerate scaling properties that could be observed if, for example, the network bandwidth of the Lambda functions was shared.

Table 4.2 lists relative speedups achieved by decreasing the scale factor or increasing the number of workers. The speedups for Spark and Drill on both EC2 and Lambda platforms are relative to query execution times of all queries (except Q2 and Q21) for SF-100 and using 8-worker nodes. The top two rows show the speedup achieved as the scale factor is

decreased to SF-30 and SF-10, and the bottom two rows show the speedups observed as the number of workers is increased to 16 and 32 (e.g. the speedup achieved by Spark on Lambda by decreasing the scale factor from SF-100 to SF-30 while keeping the number of workers constant at 8 is 1.99). The fraction of the speedup achieved by Spark on Lambda relative to EC2 ranges from 0.86 to 1.09, and for Drill from 0.83 to 1.24.

The difference in throughput between Lambdas and the selected EC2 instances can be a factor in these scaling (and absolute times) differences. The throughput between the EC2 instances used is 975Mbit/s (median), and between the Lambda instances, it is 629Mbit/s (median,) both with low variance over time. However, in the first 5 seconds of Lambda execution, the temporary throughput can be observed as high as 2.8Gbit/s. As the number of Lambdas increases, for some of the queries, this temporary burst may help to accelerate a larger fraction of the execution, in other cases, the higher steady-state throughput may be advantageous.

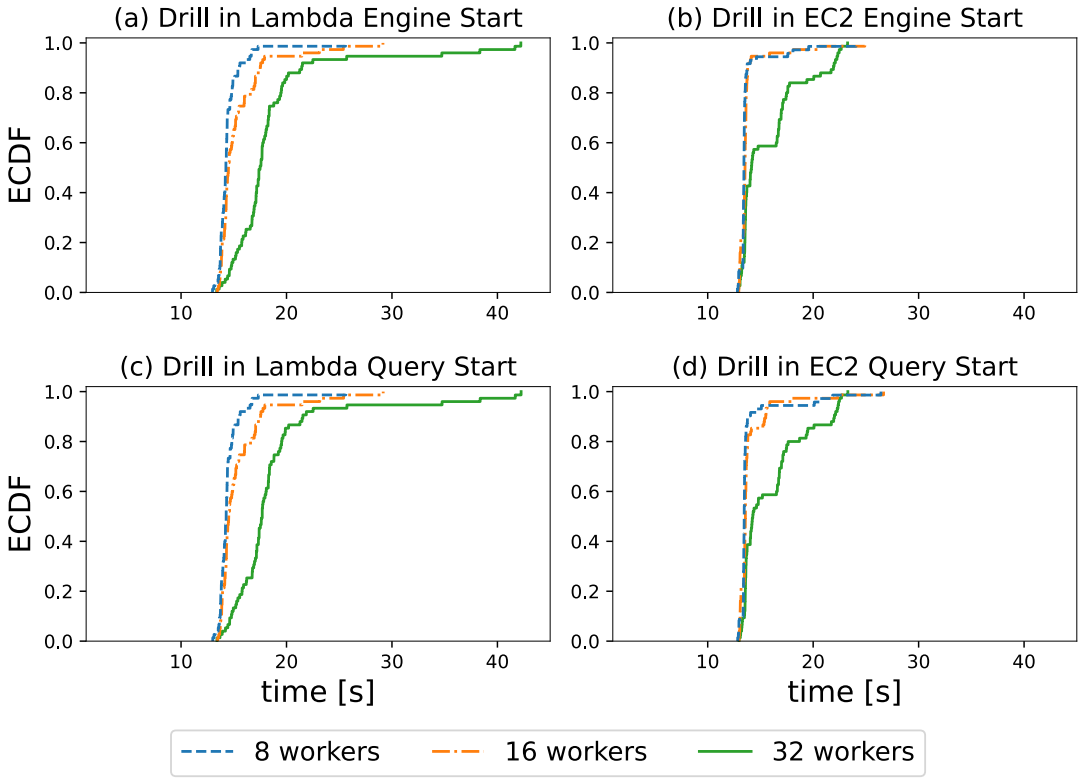

Figure 4.12: *Apache Drill engine initialization times (details in Section 4.3.3.3).*

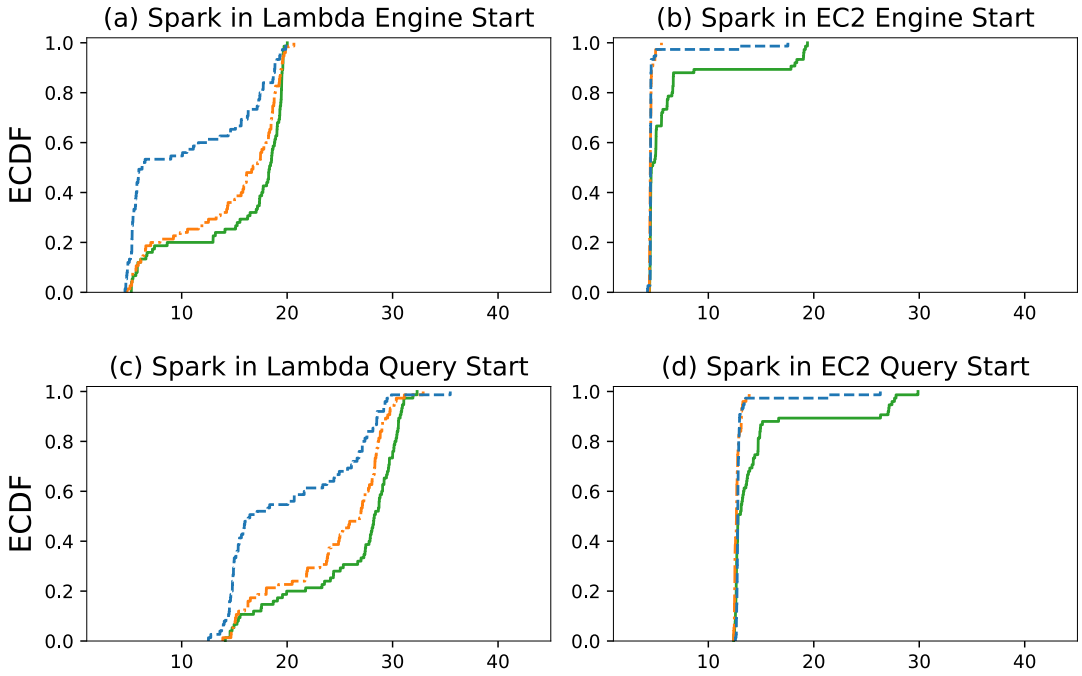

Figure 4.13: *Apache Spark engine initialization times (details in Section 4.3.3.3).*

4.3.3.3 Query engine initialization

Engine	Workers	Median	Mean	Std Dev	Min	Max
Spark	8	16.46s	20.25s	6.16s	12.54s	35.47s
Spark	16	26.99s	24.50s	5.25s	13.90s	32.90s
Spark	32	28.27s	26.19s	5.32s	14.14s	32.33s
Drill	8	14.28s	14.54s	1.53s	13.01s	25.62s
Drill	16	14.53s	15.46s	2.61s	13.46s	29.17s
Drill	32	17.70s	18.77s	5.40s	13.40s	42.25s

Table 4.3: *Time to submit a query in Apache Spark and Apache Drill in AWS Lambda. Corresponds to Figure 4.13(c) and 4.12(c).*

Relative to long-running query engines, executing queries in serverless functions involves additional query engine startup as the engines are instantiated. In the limit, a new query engine instance can be started to run each query, or an engine may need to be restarted if the function duration limit is reached (currently, AWS Lambda functions have 15 minutes

duration limit). The measured times required to start Apache Spark and Apache Drill instances from the moment the requested set of AWS Lambda functions is available to when a query can be submitted for execution are listed in Table 4.3.

The query engine initialization can be further subdivided into two stages: the time to start the engine and the time to start processing queries ('Engine Start' and 'Query Start' respectively in Figure 4.13 and Figure 4.12). The time to start the engine ('Engine Start') includes the time for the master node (in Spark) or Zookeeper (in Drill) to start and for all the worker nodes to start and join. At this point, the query engine is considered started but not yet ready to begin query execution. In the next stage ('Query Start'), each engine needs to be appropriately initialized so that it can execute the query. In the case of Drill, the table location catalog must be updated to reflect the TPC-H tables stored in S3, and only then the query is submitted for execution. In the case of Spark, when the cluster is started, it is ready to allocate executors on its workers for Spark applications. Depending on the Spark API used, different initialization procedures can follow; in the experiments presented here, a single executor per worker was allocated and Spark SQL Scala API was used to configure the TPC-H tables and submit the query.

As expected, as the number of workers increases, so does the time to start the engine and begin the query execution. In the case of Apache Drill in AWS Lambda, the median times to start query execution range from $14.28s$ to $17.70s$ for 8 to 32 workers (Table 4.3). However, the observed maximum times are significantly higher, up to $42.25s$. The distribution of these times, Figure 4.12(c), shows that only a small fraction of the Drill initialization times have such high start times, and can be seen from Figure 4.12(a) that the long tail propagates from the time needed for workers to start. One possible source of this is the system waiting for straggler workers to join. Since this delay is before the query execution starts, one possible solution is to start more workers than required. Once the required number of workers joins, the remaining slower workers can be released.

In the case of Apache Spark in AWS Lambda, median times to start queries also increase with worker count, from $16.46s$ to $28.99s$ for 8 to 32 workers. However, compared to Apache Drill, a higher degree of dispersion can be seen Figure 4.12(c). Similarly, this is most likely contributed by the time required for the workers to join Figure 4.13(a). Based on comparison with the same measurement in EC2 Figure 4.13(b), it can be seen that this level of dispersion is specific to AWS Lambda, as Spark engine start times in EC2 show significantly lower dispersion. The approach mentioned above of starting a small number of extra workers may be less appropriate here.

A promising and more general solution to the overhead of the query engine startup time is to use function snapshots. AWS Lambda already provides a method of starting functions from previously taken function snapshots [aws23b]. Perhaps this could be adapted to reduce the query engine initialization times. To execute a query, functions would be configured to start from an already initialized query engine ready for query execution.

4.3.4 Limitations and Opportunities

This section discusses the limitations of the current system and briefly explores promising research directions that the off-the-shelf serverless data processing approach opens up.

Startup times: As shown in Section 4.3.3.3, existing data analytics platforms (together with the underlying JVM runtime) can have long initialization times. These initialization times might seem to negate the benefits of using elastic serverless platforms. However, since this is a problem that is shared by many other serverless applications, serverless platforms now offer functionality to accelerate initialization times by starting functions from previously taken snapshots [aws23b]. Further development of the system will explore how to take advantage of this technique to accelerate query engine initialization times. It needs to be investigated how to leverage the AWS Lambda snapshots to snapshot and restore collections of networked functions. This will raise a number of questions. First, what to snapshot - how specific do the engine configurations have to be when they are snapshotted, e.g., do different snapshots have to be created for engines with different worker counts, or can the number of workers to restore be dynamically selected. Second, at what point in the query engine initialization phase to take the snapshots, e.g., to make Spark available to all types of applications snapshot should be taken at the engine start time. However, from the difference between Figure 4.13(a) and (c), it can be seen that to use Spark SQL interface would still result in overhead for the allocation of the appropriate executors to submit the query. Snapshots could be created for different sets of executor instantiations, so in the case of Spark SQL, the query can be submitted right after restoring the snapshots. These questions need to be studied to successfully leverage the already available snapshotting technology to reduce the initialization overhead.

Restricted use cases: Use of off-the-shelf data processing engines does not overcome the limitations that have an impact on which use cases are currently a good fit for serverless. Serverless platforms currently impose a fixed time limit (e.g., AWS Lambda function invocations cannot take longer than 15 minutes) that restricts serverless use cases

to focus on fast-executing queries. Furthermore, serverless workers usually are not as flexible compared to VMs with regard to resource bundles as providers typically offer a small set of resource configurations (e.g., AWS Lambda only allows users to select the amount of memory, CPU is allocated proportionally). The current pricing model also narrows down the workload patterns that are cost-efficient compared to long-running engines [PCFDM20, MMA20].

More off-the-shelf engines: Boxer system used in this study has limitations that continue to be addressed. Some of the failures experienced during the experiments need to be investigated to determine if they are due to errors in the implementation or if there are additional mechanisms that need to be implemented to improve robustness and scalability. Extending support to a wider range of data processing engines, in particular, Trino [Tri20], Databend [Dat20], and Clickhouse [Cli20], will likely provide new insights.

VM-FaaS hybrid: The proposed approach can be extended to other settings beyond just running individual (or sets of) queries. For long-running query engines, sudden bursts of query volume can quickly lead to congestion and, as a result, long response times. How to use serverless workers as an extension of off-the-shelf serverful query engine deployments to quickly accommodate rapid workload fluctuations? In this scenario, serverless networked functions are used to deploy additional worker instances of a query engine that are used only for the duration of the workload spike. By using serverless functions to accommodate such workload bursts, serverful infrastructure overprovisioning can be significantly reduced. Recently, Pixels-Turbo [BSA23] system described a custom system where serverless workers can be instantiated to execute sub-plans to accelerate query execution of VM-based system. However, in the approach presented there, specialized serverless query executors had to be implemented and integrated with Pixels [BA22] system, and not taking advantage of function-to-function networking. This type of rapid scaling into networked serverless functions is not specific to query engines, other types of data processing systems (and beyond) can benefit from this paradigm. For example, stream processing systems such as Apache Flink [Apa01] can reduce their level of overprovisioning if they can temporarily scale out to networked Lambda functions to quickly absorb load spikes while handling steady-state load using the more cost-effective VMs. In contrast to using off-the-shelf streaming system, the recently proposed Sponge [SUE+23] streaming system has been specifically developed to leverage serverless functions to absorb load spikes.

Locality-aware instances: The ability to instantiate off-the-shelf data analytics systems in serverless also has the potential to help take advantage of data locality by facilitating

executing (sub)queries closer to the data. Long-running VM-based systems could not afford the flexibility to be fully or partially instantiated close to the queried data on per-query granularity. Using Boxer, a long-running VM-based system may be transparently augmented by short-lived serverless workers instantiated close to a remote data source as queries that reference it arrive. Depending on the workload, this may mean instantiating enough new workers to execute the query fully or just enough to perform appropriate data reduction close to the data source. This can be especially beneficial if the workload is composed of queries that reference diverse data sources close to FaaS platforms, such as in different regions of a cloud provider or even across different cloud providers. Boxer has not yet been experimented with as a wide-area overlay or across different providers, but this will be explored, possibly taking advantage of some recent cost, latency, and throughput optimization studies [JKW+23, RBB+22].

Caching: Depending on the workload, it can be beneficial to augment the serverless environment with caching. A number of specialized serverless caching solutions have been proposed [AGL+23, SWL+20a, MBN+21], some being able to leverage publicly available serverless platforms [WZM+20, RCG+21]. Instead of relying on serverless-specific solutions that work around serverless limitations, leveraging Boxer, unmodified off-the-shelf distributed caching systems such as Memcached [Mem03] or Redis [Red03] could be instantiated alongside data analytics platforms in Boxer networked serverless functions environment. Beyond improving single-system performance, especially when multiple queries fetch common remote data, such caches can be leveraged to reduce data movement when chaining executions of multiple different systems together in serverless functions. For instance, when one system is used for a preprocessing step, followed by data analytics queries based on a different system, and then machine learning tasks requiring yet another system, all leveraging the caching systems running in the networked serverless functions for passing the intermediate state between the stages. Such pipelines could be composed of a combination of specialized serverless systems and off-the-shelf systems running in dynamic sets of networked functions.

Worklets: Finally, serverless functions have more limited resources compared to currently available virtual machine options. They can also be scaled down to very small execution environments (AWS Lambda as low as 128MB of memory and fractional vCPUs) that, for some workloads, might be the right size. Given that the existing engines target virtual machine environments, there may be an interesting opportunity to consider supplementing the existing engine workers with variants that are meant to be short-lived, lighter weight,

possibly more limited in functionality, and more specialized to their target functions and the resource-constrained execution environments of serverless functions. Introducing these specialized engine 'worklets', initially ranging from specialized configurations of full workers to eventually even single-operator micro-workers, is a promising trajectory on the path to minimize further the gap between the existing query engines and the flexibility of serverless.

4.3.5 Summary

This section proposed a new direction for serverless data analytics — using existing unmodified off-the-shelf data analytics engines on existing unmodified serverless infrastructure. The feasibility of the approach was validated by transparently running two such distributed off-the-shelf engines (Apache Spark and Apache Drill) on a commercially available serverless platform (AWS Lambda). Initial results demonstrate that these serverless deployments have comparable performance to those based on a class of networked VMs, charting a new path to bridge the gap between existing distributed query engines and serverless.

4.4 Ephemeral Per-query Engines

The previous two sections challenged the common assumption that queries need to be submitted to a pre-configured, already running engine, and demonstrated the possibility of dynamically instantiating a chosen data processing engine upon query submission by leveraging serverless resources of FaaS platforms. The possibility was first demonstrated here using a custom networked serverless data processing engine (Section 4.2) and then further reinforced (in Section 4.3) by demonstrating the ability to instantiate unmodified off-the-shelf distributed query engines (Apache Spark and Apache Drill) on commercially available serverless FaaS platform. The engines were benchmarked with TPC-H queries by instantiating the distributed engines on demand for each query.

Building on these demonstrated possibilities, this section elaborates further on the significance of such functionality for data processing and proposes the radically new design as *Ephemeral Per-Query Engines* (EPQE), i.e., query engines dynamically instantiated for each query and discarded upon completion. The ultimate goal is to be able to select the optimal engine, resources, and configuration on a per-query basis, to eliminate the inefficiencies of using all-purpose configurations and resource overprovisioning. It is followed

by a description of the design of the MetaQ prototype system that aims to realize this vision, points to further experimental evidence supporting the idea, and discusses its many practical implications.

4.4.1 Motivation

Operating a long-running query engine has several limitations. First, it generates costs even if it is idle. Second, most distributed query engines lack elasticity, which leads to deployments being over-provisioned to cope with potential peak loads [VMA+20, TB16, DLNK16]. Third, as workload diversity increases, each query might benefit from a different configuration and engine deployment (e.g., involving accelerators, caches, parallelism level, etc.), resulting in the engine often running in a less than optimal setting for most queries [AI15, AYB+21].

In the EPQE paradigm, given a query, a query engine is instantiated, potentially selected from a variety of engines, in the best possible configuration and deployment for the query. The query is executed by the engine, and upon completion, the engine is shut down (unless there is a reason to keep it running, like a similar query arriving while the engine is active). This eliminates the need for dynamic elasticity in the engine. Every query gets an engine deployed on the resources it needs (e.g., nodes, memory, bandwidth, CPUs). This also simplifies engine deployment (since the engine can be instantiated specifically for the query at hand, e.g., maximizing data source locality) and removes the need for auto-tuning [AYB+21] of long-running engines (the engine settings need to be optimized only for the given query, which allows for more specialized and efficient solutions [ZLG+17]). The approach also eliminates the problem of idle resources since if there is no query, there is no engine running. Finally, another crucial aspect of the idea is the possibility of selecting among different data processing engines on a per-query basis. This opens up the opportunity to use different engines depending on factors like data types (e.g., relational, semi-structured, graphs), file formats used (e.g., Arrow, Parquet, CVS, JSON, etc.), expected performance (e.g., based on previous profiling), feature set (e.g., availability of required statistical functions), or suitability to the overall task (e.g., when the query is a step in an ML pipeline). The idea resembles unikernel operating systems [MMR+13] where, for each application, a specialized operating system is constructed (e.g. from a library operating system [KBL+21]) and instantiated, already optimized for the application.

The vision of EPQE is enabled by the emergence of Function as a Service (FaaS). In serverless computing, users deploy and invoke fine-grain functions on-demand [SSSK+21, CIMS19]. There are three main characteristics of serverless that can help in realizing the EPQE idea. First, thanks to lightweight VM system infrastructure [ABI+20, APV22], functions can be instantiated quickly. For example, in AWS Lambda [AWS17], function cold start initialization latency is ∼200ms. Such fast resource instantiation times allow starting a new engine for a query without contributing significantly to the overall execution time for many queries. Second, individual functions can be deployed with different CPU and memory configurations. Furthermore, thousands of functions can be instantiated in parallel. Such a level of resource availability and configurability allows for the possibility to right-size and right-configure engines at per-query granularity. Finally, FaaS platforms provide fine-grained resource accounting (e.g., AWS Lambda users pay at microsecond granularity), aligning the costs of the EPQEs to the work done, and cost can play a role in deciding which engine to instantiate for each query.

4.4.2 MetaQ Prototype Design

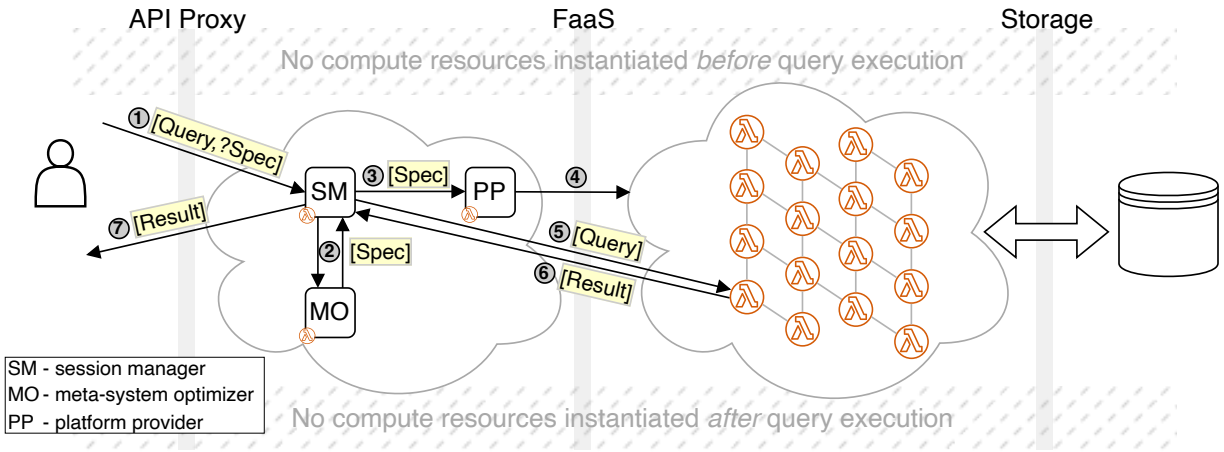

Figure 4.14: *MetaQ system prototype mode of operation (see Section 4.4.2 for details).*

In this section, a design of a prototype system, called MetaQ, to support EPQE paradigm is outlined.

MetaQ has three main components: *session manager* (SM), *platform provider* (PP), and *meta-system optimizer* (MO). The session manager oversees end-to-end query exe-

cution and its resources, including handling communication with the client. The platform provider orchestrates the required resources and configures the environment required for the query engine execution. The meta-system optimizer is used to determine the complete specification of the resources, the query engine to be used, its configuration, and possibly engine-specific query rewriting. In cases when users specify the complete specification, the meta-system optimizer can be bypassed since the execution is fully specified.

Figure 4.14 illustrates query execution in MetaQ. To execute a query, a user (step ①) starts MetaQ and specifies the query and (optionally) the specifications of the query engine and resources to use for the query execution. MetaQ launches as a serverless FaaS function that can be instantiated on demand via a request to an API proxy service of a cloud provider (such as AWS API Gateway [AWS01a]).

MetaQ begins by instantiating the session manager (SM) for the given query. If the user-supplied specifications of the query engine or resources are not complete or are left underspecified, then (step ②) the meta-system optimizer (MO) is used to choose all of the missing specifications. The specification has three elements:

(a) the initial resource allocation (e.g., where, how, and how many configured AWS Lambda functions should be started),

(b) the query engine to use (such as Apache Drill, Apache Spark, Trino [Tri20], etc.),

(c) the configuration of the query engine instantiation, including auxiliary systems such as Zookeeper [HKJR10] (e.g., mapping of engine executors onto the resources, configuring engine settings, required storage plugins, etc.)

Once the complete specification is determined, it is used to instruct the platform provider (PP) (step ③) to instantiate and configure the specified resources and then start the configured query engine processes (and any auxiliary systems). The platform provider (step ④), using the specification of initial resource allocation (a), requests the resources from the underlying platform, such as networked FaaS functions, configures their networking, and assigns necessary names, roles, and ids to function instances. The query engine specification (b) determines which function (or container) images are instantiated from the available catalog. Finally, before the platform provider starts the query engine, the specification of the query engine configuration (c) is used to populate the necessary configuration files and environment variables for the query engine.

Once the engine is started and ready to process queries, the session manager (step ⑤) submits the user query and awaits the results from the execution engine. When the query execution completes, the session manager retrieves the results (step ⑥) and returns them to the user (step ⑦). When the query execution completes, all of the resources are released, and the system scales back to zero.

It is assumed that the persistent data is stored in standard formats (such as Parquet, ORC, Avro, CSV) and is available through cloud storage services compatible with the common query engines (such as S3 or EBS). The set of distributed query engines considered is restricted to ones that can be used in such networked shared-disk configurations.

Although it is shown how MetaQ can run in FaaS environments, its design is not tied to them. For example, MetaQ's components (SM, MO, PP) could execute locally on the user's computer and then could provide (a subset of) standard client protocols that many distributed query engines often expose (such as PostgreSQL standard wire protocol or JDBC). Independently, there could be different platform providers (PP) giving access to different types of resources for query execution, from the user's local resources (useful for smaller workloads) to serverless container services such as AWS Fargate or future serverless platforms that may provide access to heterogeneous hardware accelerators.

4.4.3 Feasibility study

Preliminary feasibility studies of MetaQ used a version of Boxer (Section 2.5) as the platform provider (PP) and used AWS Lambda functions to run off-the-shelf query engines. For the initial validation, it was assumed that the user specified the complete system specification (resource allocations, query engine specification, and configuration) along with every query. This bypassed the meta-engine optimizes (MO), which will be the focus of future research. These early feasibility studies used TPC-H benchmark queries, and the validation results closely follow those already reported in the previous section (Section 4.3), demonstrating running off-the-shelf query engines with Boxer on AWS Lambda, so they are not repeated here.

A key and unique aspect of the EPQE and MetaQ is the possibility of choosing a different engine and configuration for each query. How to choose the right engine and configuration to execute a given query? The meta-engine optimizer can consider many dimensions when making the choice, and studying this process can lead to exciting new research

	SF-10			SF-30			SF-100		
Query	**Eng**	**WC**	**D.%**	**Eng**	**WC**	**D.%**	**Eng**	**WC**	**D.%**
1	**Drill**	8	1.40	**Drill**	16	1.06	**Drill**	32	1.01
2	Spark	**16**	1.02	Spark	**8**	1.01	Spark	**16**	1.09
3	Spark	8	1.00	Spark	16	1.00	Spark	32	1.00
4	Spark	8	1.00	Spark	16	1.00	Spark	32	1.00
5	Spark	**16**	1.06	Spark	16	1.00	Spark	32	1.00
6	Spark	**4**	1.39	Spark	**8**	1.21	Spark	**16**	1.07
7	Spark	8	1.00	Spark	16	1.00	Spark	**16**	3.67
8	Spark	8	1.00	Spark	16	1.00	Spark	32	1.00
9	Spark	**4**	1.13	Spark	16	1.00	Spark	32	1.00
10	**Drill**	**16**	1.35	**Drill**	16	1.07	Spark	32	1.00
11	**Drill**	**4**	1.43	Spark	**4**	1.16	Spark	32	1.00
12	**Drill**	8	2.14	Spark	16	1.00	Spark	32	1.00
13	**Drill**	8	1.85	**Drill**	16	1.64	**Drill**	32	1.50
14	**Drill**	**16**	1.17	**Drill**	16	1.82	**Drill**	32	2.69
15	**Drill**	**16**	1.72	Spark	16	1.00	**Drill**	32	1.00
16	Spark	**1**	1.25	Spark	16	1.00	Spark	32	1.00
17	Spark	8	1.00	**Drill**	16	1.01	**Drill**	32	1.26
18	Spark	**16**	1.08	Spark	16	1.00	Spark	32	1.00
19	Spark	**4**	1.12	Spark	16	1.00	Spark	**16**	1.02
20	Spark	8	1.00	Spark	16	1.00	Spark	32	1.00
21	Spark	**16**	1.04	Spark	16	1.00	Spark	32	1.00
22	**Drill**	**16**	2.68	**Drill**	16	1.29	**Drill**	32	1.02

Table 4.4: *Best configurations based on median query execution time of engine(Eng) and workers count(WC) for each TPC-H query at scale factors SF-10, SF-30, SF-100. The choice is based on the median of three query executions of each configuration considered. The dominant configurations for queries at SF-10 is Spark with 8 workers, SF-30 is Spark with 16 workers, for SF-100 is Spark with 32 workers. Speedup relative to the dominant configurations based on medians of execution times is in column D.%. Bold shows better configuration parameters that the dominant configuration (based only on speedup).*

opportunities. However, it is first worth validating that worthwhile benefits can be gained from such optimization opportunities.

Fortunately, given only the limited preliminary measurements from experiments running off-the-shelf query engines in AWS Lambda (Section 4.3), it can be observed that just considering the selection of one of two engines (between Apache Spark and Drill) and the number of workers (from 8, 16, 32) on per-query granularity can lead to significant relative performance gains and cost savings compared to using a fixed configuration for all queries that would be chosen for a long-running engine.

Table 4.4 shows that selecting an engine and worker count based on a query and scale factor can provide significant speed and cost improvements relative to choosing the single best (dominant) configuration for each scale factor. The dominant configuration in Table 4.4 refers to the configuration that produces best execution times if used for all queries of a given scale factor. If an oracle chose the best configuration for a long-running query engine, it would select the dominant configuration. For example, for scale factor 10, the dominant configuration is to choose Apache Spark with 8 worker nodes. However, considering individual queries, there are significantly better configurations, and if choices are made on per-query granularity, an optimizer may attempt to select them. For example, for query 11, it is better to choose Apache Drill with 4 worker nodes over the dominant configuration of Apache Spark with 8 worker nodes, leading to 1.43 speedup and half the number of worker nodes used.

These initial results suggest that, indeed, the notion of instantiating a different engine depending on the query can be beneficial. This opens up very interesting research questions in terms of how an optimizer could decide which system and configuration to use.

4.4.4 Use Cases and Opportunities

Data Lakes: Data Lakes refer to collections of heterogeneous data that needs to be processed in a variety of different ways. The problem with this notion is that the processing is also highly heterogeneous, and it is the user who is responsible for handling it. Lakehouses is a new iteration of the concept that incorporates the data processing as a first-class citizen and provides support for different engines, languages, etc., while automating as much as possible the task of matching data to engines and tools [ZGXA21].

MetaQ is well-suited to Lakehouses as it enables dynamically selecting the engine and processing tools on the fly, and this can be done on the basis such as data types, data

sizes, type of query, user requirements, or cost, etc. Furthermore, the per-query engine vision enables an intriguing possibility: sharing of auxiliary data structures across engines (indexes, partitions, zone maps, etc.) as well as creating a general infrastructure that is engine agnostic (e.g., a main memory caching layer for data to avoid having to retrieve it from slow storage every time or a results cache). Such infrastructure exists, but it is typically system specific. MetaQ opens up the possibility of seeing these aspects as orthogonal to the actual engine. In the extreme, all common modules of query engines could become serverless components dynamically added to an engine as it is instantiated with the query-specific functionality.

The Meta-Engine: EPQE unlock a number of opportunities when it comes to selecting the most appropriate engine for each query. This can be done in a very simple manner by, for instance, asking the user to specify which engine to use. However, going one step further and automating the selection process by building an end-to-end query system that handles this is most interesting. In a scenario where users write queries in an engine-agnostic syntax (for example, in a declarative language such as SQL), MetaQ's meta-system optimizer could inspect the query and determine which engine is the most efficient given the data types, its type (static or streaming), the type of operations required, etc. This leads to cross-engine optimizations, such as picking the engine that is faster to perform a given operation provided by several engines. The main research question is how to derive meta-system optimizer policies. One possible approach is to extend the domain of automatic configuration systems [AYB+21] with the additional tasks of choosing not just configuration parameters for a query engine, but also the choice of the query engine itself and resource allocation based on the query considered, eventually realizing the vision of vertically integrated per-query optimization.

Autoscaling Per-query Deployments: With a new deployment being launched and shut down per query, it is now possible to optimize the deployment where the engine will run for every query. Such deployment configuration could determine the amount of resources used, such as the CPU and/or memory budget. Such configuration could be inferred by analyzing the query and data inputs to estimate the amount of data that would be processed and, therefore, the amount of compute and memory necessary to finish the query within a particular time frame. From another perspective, it is now possible to dynamically find tradeoffs between execution time and price for each query. This tradeoff could also be exposed to users as a way to prioritize interactive queries over batch workloads.

4.4.5 Summary

Distributed data processing engines are ubiquitous in the cloud and are at the heart of many modern applications. These engines often require to have a fixed underlying infrastructure to run in the form of pre-allocated VMs, virtual vetworks, and other services provided by the cloud. This results in inefficiencies that are difficult to address: overprovisioning, coarse resource allocation, generic engine configurations, low utilization, etc. This section introduced the idea of ephemeral per-query engines: selected query engines dynamically instantiated when a query arrives and removed when it terminates. The initial feasibility experiments are encouraging, showing that existing engines can be sufficiently quickly instantiated on demand to run a single query and that choosing a system and configuration on per-query granularity can provide significant performance and cost benefits.

4.5 Conclusion

This chapter demonstrated the per-request systems paradigm in the context of data processing systems. Based on different stages of development of Boxer (Chapter 2), first, Lambada, a custom serverless data analytics system was significantly accelerated by enabling function-to-function networking, expanding the range of workloads that fit the paradigm. Second, off-the-shelf, unmodified distributed data analytics systems were shown to function in a commercially available FaaS serverless platform, expanding the set of practically available data processing functionality that fit the paradigm. Lastly, it proposed a system design to take advantage of the new possibilities enabled by the paradigm of choosing entire systems, configurations, and resources on per-request granularity.

As described throughout the chapter, in the context of data processing, there are many challenges and opportunities originating from the per-request systems approach. Many of them will likely generalize to other domains. The per-request systems paradigm can be the basis of a new promising research agenda.

5

Conclusion

This thesis demonstrates that it is possible to provide an execution environment on top of a publicly available serverless FaaS platform that can transparently run unmodified distributed datacenter applications. It shows that it is possible to unbundle the event-triggered-function model from FaaS resources and instead expose the resources via the network-of-hosts model. Boxer system was designed to provide direct function-to-function networking and to expose a unified network-of-hosts model that can span AWS Lambda functions and VMs/containers, while not requiring additional platform privileges.

The uniform network spanning long-running VMs and dynamically added AWS Lambda functions enabled *unmodified* long-running datacenter applications to benefit from ephemeral elasticity. DeathStarBench experiment scaling with functions, showed performance comparable to VM-based overprovisioning, and up to 76% cost reduction. Ephemeral elasticity used for Zookeeper fault recovery experiment showed 5.7 × improvement in failure recovery time when using AWS Lambda over using only EC2 VMs. These experiments not only demonstrated the benefits of fast ephemeral elasticity, but also showed that, in contrast to prior work, using Boxer, it can be available to unmodified datacenter applications, having the potential to significantly reduce the need for resource overprovisioning in datacenters, reducing cost and possibly improving overall datacenter energy efficiency.

The direct function-to-function networking enabled by Boxer accelerated Lambada serverless data analytics system, showing TPC-H query execution time improvements between 4× and 6× for most queries (some over 10×) compared to the time required when using

the default S3 cloud storage-based exchange operator. Cost savings were similar, with cost reductions between 4× and 6× for most queries. Many serverless systems that suffer from function-to-function data shuffling bottlenecks will likely benefit from Boxer-enabled networking.

Extensive experiments running TPC-H benchmarks using unmodified Apache Spark and Apache Drill on AWS Lambda using Boxer demonstrated that unmodified, mature, feature-complete, and complex datacenter systems can be instantiated on a per-request granularity using serverless resources. Furthermore, the measured TPC-H performance in AWS Lambda is comparable to that measured in a class of EC2 VMs. With Boxer, unbundling the programming model from the serverless resources provides a way to use feature-complete, mature, high-performance distributed systems in serverless environments on per-request granularity. This enables the possibility of infrequent and interactive use of such systems, reducing dependence on long-running services that aggregate requests (and users.) Perhaps this may also reduce the urgency of inventing yet another experimental and incomplete serverless system that does something that an existing datacenter system already does.

Finally, the analysis of the Apache Spark and Apache Drill per-TPC-H-query measurements reveals that only choosing on a per-query basis which of the two engines to use and how many workers to instantiate leads to significant performance speedup compared to using the static best configuration (over 2× for some queries). This new possibility for (meta)optimization is enabled by the ability to instantiate complete systems on per-request granularity. As the space of possible systems to choose from, their configurations, and resources to allocate increases, the (meta)optimization problem may lead to new sources of efficiency in the future.

While enabling serverless datacenter applications with Boxer, this thesis demonstrates tangible benefits today and points to promising opportunities further on the horizon.

List of Tables

List of Figures

Bibliography

[ABI+20] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. "Firecracker: Lightweight Virtualization for Serverless Applications." In *NSDI*. 2020.

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. "A View of Cloud Computing." *Commun. ACM*, vol. 53, no. 4, 50–58, 2010.

[AFK19] A. Akhter, M. Fragkoulis, and A. Katsifodimos. "Stateful Functions as a Service in Action." *Proc. VLDB Endow.*, vol. 12, no. 12, 1890–1893, 2019.

[AGF+20] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms." In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 735–751. 2020.

[AGL+23] M. Abdi, S. Ginzburg, C. Lin, J. M. Faleiro, I. Goiri, G. I. Chaudhry, R. Bianchini, D. S. Berger, and R. Fonseca. "Palette Load Balancing: Locality Hints for Serverless Functions." In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. ACM, 2023.

[AI15] A. Augusta and S. Idreos. "JAFAR: Near-Data Processing for Databases." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, p. 2069–2070. 2015.

[AIVP18] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. "Sprocket: A Serverless Video Processing Framework." In *Proceedings of the ACM Symposium on Cloud*

Computing, SoCC '18, pp. 263–274. Association for Computing Machinery, New York, NY, USA, 2018.

[Ama10] Amazon Linux 2. "https://aws.amazon.com/amazon-linux-2/.", 2020-12-10.

[Ama17] Amazon S3. "https://aws.amazon.com/s3.", 2023-08-17.

[Apa20] Apache Drill. "https://drill.apache.org/.", 2022-10-20.

[Apa03] Apache Spark history. "https://spark.apache.org/history.html.", 2023-12-03.

[Apa01] Apache Flink. "https://flink.apache.org/.", 2023-03-01.

[Apa17] Apache OpenWhisk. "https://openwhisk.apache.org/.", 2023-08-17.

[APV22] L. Ao, G. Porter, and G. M. Voelker. "FaaSnap: FaaS Made Fast Using Snapshot-Based VMs." In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, p. 730–746. Association for Computing Machinery, New York, NY, USA, 2022.

[AWS01] AWS Lambda changes duration billing granularity from 100ms down to 1ms. "https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-changes-duration-billing-granularity-from-100ms-to-1ms/.", 2020-12-01.

[aws18] "AWS Lambda enables functions that can run up to 15 minutes.", 2018.

[aws20] "AWS Lambda now supports up to 10 GB of memory and 6 vCPU cores for Lambda Functions.", 2020.

[aws23a] "Amazon EC2 now provides High-CPU instance types.", 2023.

[aws23b] "Improving startup performance with Lambda SnapStart.", 2023.

[AWS01a] AWS Fargate – Serverless compute for containers. "https://docs.aws.amazon.com/apigateway/.", 2023-03-01.

[AWS01b] AWS Step Functions. "https://aws.amazon.com/step-functions/.", 2023-03-01.

[AWS27] AWS Lambda provisioned concurrency. "https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html.", 2023-07-27.

[AWS17] AWS Lambda. "https://aws.amazon.com/lambda.", 2020-08-17.

[AYB+21] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo. "An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems." *Proc. VLDB Endow.*, vol. 14, no. 7, 1241–1253, 2021.

[Azu01] Azure Durable Functions. "https://learn.microsoft.com/en-us/azure/azure-functions/durable/.", 2023-03-01.

[BA22] H. Bian and A. Ailamaki. "Pixels: An Efficient Column Store for Cloud Data Lakes." In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pp. 3078–3090. 2022.

[BDGP23] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka. "On-demand Container Loading in AWS Lambda." In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 315–328. USENIX Association, Boston, MA, 2023.

[BDR+21] N. Bashir, N. Deng, K. Rzadca, D. Irwin, S. Kodak, and R. Jnagal. "Take It to the Limit: Peak Prediction-Driven Resource Overcommitment in Datacenters." In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, p. 556–573. Association for Computing Machinery, New York, NY, USA, 2021.

[Bel05] F. Bellard. "QEMU, a fast and portable dynamic translator." In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, p. 41. USENIX Association, USA, 2005.

[BPK+14] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency." In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 49–65. USENIX Association, Broomfield, CO, 2014.

[BSA23] H. Bian, T. Sha, and A. Ailamaki. "Using Cloud Functions as Accelerator for Elastic Data Analytics." *Proc. ACM Manag. Data*, vol. 1, no. 2, 2023.

[CBCH23] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler. "FMI: Fast and Cheap Message Passing for Serverless Functions." In *Proceedings of the 37th Inter-*

national Conference on Supercomputing, ICS '23, p. 373–385. Association for Computing Machinery, New York, NY, USA, 2023.

[CBM+17] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms." In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, p. 153–167. 2017.

[CFT+19] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. "Cirrus: A Serverless Framework for End-to-End ML Workflows." In *SoCC*. 2019.

[CIMS19] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. "The Rise of Serverless Computing." *Commun. ACM*, vol. 62, no. 12, 44–54, 2019.

[Cli20] ClickHouse. "https://clickhouse.com/.", 2023-06-20.

[Clo15] Cloud Computing Services - Amazon Web Services. "https://aws.amazon.com/.", 2022-04-15.

[CZL+23] J. Cheng, Y. Zhao, Z. Li, Q. Chen, W. Cui, and M. Guo. "Microless: Cost-Efficient Hybrid Deployment of Microservices on IaaS VMs and Serverless." In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 2303–2310. 2023.

[Dat20] Databend. "https://databend.rs/.", 2023-06-20.

[DK14] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management." In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2014.

[DLNK16] S. Das, F. Li, V. R. Narasayya, and A. C. König. "Automated Demand-Driven Resource Scaling in Relational Database-as-a-Service." In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, p. 1923–1934. 2016.

[Doc15] Docker Swarm. "https://docs.docker.com/engine/swarm/.", 2022-04-15.

[Doc27] Docker Compose. "https://docs.docker.com/compose/.", 2023-07-27.

[eBP24] eBPF. "https://ebpf.io/.", 2024-05-24.

[EF94] K. B. Egevang and P. Francis. "The IP Network Address Translator (NAT)." RFC 1631, 1994.

[FCS+24] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, I. Goiri, S. Elnikety, R. Fonseca, and A. Belay. "Making Kernel Bypass Practical for the Cloud with Junction." In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 55–73. USENIX Association, Santa Clara, CA, 2024.

[FGB+08] B. Ford, S. Guha, K. Biswas, S. Sivakumar, and P. Srisuresh. "NAT Behavioral Requirements for TCP." RFC 5382, 2008.

[FRI+19] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers." In *USENIX ATC*. 2019.

[FS22] A. Fuerst and P. Sharma. "Locality-Aware Load-Balancing For Serverless Clusters." In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22. 2022.

[FSK05] B. Ford, P. Srisuresh, and D. Kegel. "Peer-to-Peer Communication Across Network Address Translators." In *USENIX ATC*. 2005.

[FWS+17a] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads." In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 363–376. USENIX Association, Boston, MA, 2017.

[FWS+17b] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads." In *NSDI*. 2017.

[Goo01] Google Cloud Functions. "https://cloud.google.com/functions.", 2020-12-01.

[gVi17] gVisor. "https://gvisor.dev/.", 2024-08-17.

[GZC+19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." In *ASPLOS*. 2019.

[HFG+19] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. "Serverless Computing: One Step Forward, Two Steps Back." In *CIDR*. 2019.

[HKJR10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems." USENIX ATC'10. 2010.

[HRS+17] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. "Bringing the web up to speed with WebAssembly." *SIGPLAN Not.*, vol. 52, no. 6, 185–200, 2017.

[HS99] M. Holdrege and P. Srisuresh. "IP Network Address Translator (NAT) Terminology and Considerations." RFC 2663, 1999.

[Ini03] Initial ZooKeeper code contribution from Yahoo! "https://issues.apache.org/jira/browse/ZOOKEEPER-1.", 2023-12-03.

[ipe10] iperf3. "https://software.es.net/iperf/.", 2020-12-10.

[JGL+21] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang. "Towards Demystifying Serverless Machine Learning Training." In *Proceedings of the 2021 International Conference on Management of Data*, pp. 857–871. 2021.

[JHA+23] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker. "How Does It Function? Characterizing Long-term Trends in Production Serverless Workloads." In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, p. 443–458. Association for Computing Machinery, New York, NY, USA, 2023.

[JKW+23] P. Jain, S. Kumar, S. Wooders, S. G. Patil, J. E. Gonzalez, and I. Stoica. "Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware

Overlays." In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 1375–1389. USENIX Association, Boston, MA, 2023.

[J.L05] E. J.L. "TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem." In *Carnegie Mellon University, Tech. Rep, ISRI-05-104*. 2005.

[JPV+17] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. "Occupy the Cloud: Distributed Computing for the 99%." In *SoCC*. 2017.

[JW21] Z. Jia and E. Witchel. "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices." In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 152–166. 2021.

[JWJ+14] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems." In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pp. 489–502. USENIX Association, Seattle, WA, 2014.

[KBL+21] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici. "Unikraft: Fast, Specialized Unikernels the Easy Way." In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, p. 376–394. Association for Computing Machinery, New York, NY, USA, 2021.

[KJK+18] J. Kim, T. J. Jun, D. Kang, D. Kim, and D. Kim. "GPU Enabled Serverless Computing Framework." In *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 533–540. 2018.

[Kna29] Knative. "https://knative.dev/.", 2024-05-29.

[Kub15] Kubernetes. "https://kubernetes.io/.", 2022-04-15.

[KWS+18] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics." In *OSDI*, p. 427–444. 2018.

[KZ13] T. Kim and N. Zeldovich. "Practical and Effective Sandboxing for Non-root Users." In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 139–144. USENIX Association, San Jose, CA, 2013.

[LLNB23] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt. "Doing More with Less: Orchestrating Serverless Applications without an Orchestrator." In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 1505–1519. USENIX Association, Boston, MA, 2023.

[LN79] H. C. Lauer and R. M. Needham. "On the duality of operating system structures." *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, 3–19, 1979.

[LYX+17a] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. "Imbalance in the cloud: An analysis on Alibaba cluster trace." In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2884–2892. 2017.

[LYX+17b] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. "Imbalance in the cloud: An analysis on Alibaba cluster trace." In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 2884–2892. 2017.

[May19] May 2015 Reddit Comments. "https://www.kaggle.com/datasets/kaggle/reddit-comments-may-2015.", 2023-09-19.

[MBK+20] I. Müller, R. Bruno, A. Klimovic, J. Wilkes, E. Sedlar, and G. Alonso. "Serverless Clusters: The Missing Piece for Interactive Batch Applications?" In *SPMA*. 2020.

[MBN+21] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana. "OFC: An Opportunistic Caching System for FaaS Platforms." In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, p. 228–244. Association for Computing Machinery, New York, NY, USA, 2021.

[Mem03] Memcached. "https://memcached.org/.", 2023-12-03.

[Mic17] Microsoft Azure Functions. "https://azure.microsoft.com/en-us/services/functions.", 2020-08-17.

[MMA20] I. Müller, R. Marroquín, and G. Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure." In *SIGMOD*. 2020.

[MMR+13] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: Library Operating Systems for the Cloud." In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, p. 461–472. Association for Computing Machinery, New York, NY, USA, 2013.

[Mon03] MongoDB. "https://www.mongodb.com/.", 2023-12-03.

[Mor23] G. Moro. "Evaluating Data Processing Datacenter Applications in Serverless Environments." Master thesis, ETH Zurich, Zurich, 2023.

[Net30] Netfiler/iptables Project. "https://www.netfilter.org/projects/iptables/.", 2024-05-30.

[Ngi24] Nginx. "https://nginx.org/.", 2024-05-24.

[Nuc23] Nuclio Serverless Platform. "https://nuclio.io/.", 2023.

[Ope24] OpenFaaS: Serverless Functions, Made Simple. "https://www.openfaas.com/.", 2024-05-24.

[OYZ+18] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers." In *USENIX ATC*. 2018.

[PCFDM20] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. "Starling: A Scalable Query Engine on Cloud Functions." In *SIGMOD*. 2020.

[PFCM23] M. Perron, R. C. Fernandez, M. Cafarella, and S. Madden. "Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools." In *SIGMOD*. 2023.

[PLZ+14] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. "Arrakis: The Operating System is the Control Plane." In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 1–16. USENIX Association, Broomfield, CO, 2014.

[pos24] "IEEE/Open Group Standard for Information Technology–Portable Operating System Interface (POSIX™) Base Specifications, Issue 8." *IEEE/Open Group Std 1003.1-2024 (Revision of IEEE Std 1003.1-2017)*, pp. 1–4107, 2024.

[PPB+16] R. Penno, S. Perreault, M. Boucadair, S. Sivakumar, and K. Naito. "Updates to Network Address Translation (NAT) Behavioral Requirements." RFC 7857, 2016.

[Pri17] Prime Video Switched from Serverless to EC2 and ECS to Save Costs. "https://www.infoq.com/news/2023/05/prime-ec2-ecs-saves-costs/.", 2023-08-17.

[PVS19] Q. Pu, S. Venkataraman, and I. Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure." In *NSDI 19*. 2019.

[PZ17] S. Palkar and M. Zaharia. "DIY Hosting for Online Privacy." In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI. 2017.

[RBB+22] N. H. Rotman, Y. Ben-Itzhak, A. Bergman, I. Cidon, I. Golikov, A. Markuze, and E. Zohar. "CloudCast: Characterizing Public Clouds Connectivity." *CoRR*, vol. abs/2201.06989, 2022.

[RCG+21] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini. "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications." In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, p. 122–137. Association for Computing Machinery, New York, NY, USA, 2021.

[Red03] Redis. "https://redis.io/.", 2023-12-03.

[rfc81] "Transmission Control Protocol." RFC 793, 1981.

[RHMW03] J. Rosenberg, C. Huitema, R. Mahy, and J. Weinberger. "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)." RFC 3489, 2003.

[RLF+23] Z. Ruan, S. Li, K. Fan, M. K. Aguilera, A. Belay, S. J. Park, and M. Schwarzkopf. "Unleashing True Utility Computing with Quicksand." In

Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS '23, p. 196–205. Association for Computing Machinery, New York, NY, USA, 2023.

[RPA+23] Z. Ruan, S. J. Park, M. K. Aguilera, A. Belay, and M. Schwarzkopf. "Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes." In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 1409–1427. Boston, MA, 2023.

[RTG+12] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis." In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12. Association for Computing Machinery, New York, NY, USA, 2012.

[SAC+20] K. Satzke, I. E. Akkus, R. Chen, I. Rimac, M. Stein, A. Beck, P. Aditya, M. Vanga, and V. Hilt. "Efficient GPU Sharing for Serverless Workflows." In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, HiPS '21, p. 17–24. 2020.

[SFG+20] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider." In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218. USENIX Association, 2020.

[SHY+18] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, and C. Li. "ROSE: Cluster Resource Scheduling via Speculative Over-Subscription." In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 949–960. 2018.

[SJS+22] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella. "Memory deduplication for serverless computing with Medes." *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[SKV+20] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman. "Serverless linear algebra." In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, p. 281–295. Association for Computing Machinery, New York, NY, USA, 2020.

[SP20] S. Shillaker and P. Pietzuch. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing." In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433. USENIX Association, 2020.

[SSL+16] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren. "Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent." In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016.

[SSSK+21] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson. "What Serverless Computing is and Should Become: The next Phase of Cloud Computing." *Commun. ACM*, vol. 64, no. 5, 76–84, 2021.

[SUE+23] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun. "Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks." In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 301–314. USENIX Association, Boston, MA, 2023.

[SWC+20] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro. "A Fault-Tolerance Shim for Serverless Computing." In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. 2020.

[SWL+20a] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. Gonzalez, J. M. Hellerstein, and A. Tumanov. "Cloudburst: Stateful Functions-as-a-Service." *PVLDB*, 2020.

[SWL+20b] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. "Cloudburst: Stateful Functions-as-a-Service." *Proc. VLDB Endow.*, vol. 13, no. 12, 2438–2452, 2020.

[Sys17] Syscall User Dispatch. "https://docs.kernel.org/admin-guide/syscall-user-dispatch.html.", 2024-05-17.

[TB16] Z. Tan and S. Babu. "Tempo: Robust and Self-Tuning Resource Management in Multi-Tenant Parallel Databases." *Proc. VLDB Endow.*, vol. 9, no. 10, 720–731, 2016.

[TBD+20] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes. "Borg: The next Generation." In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. 2020.

[Tri20] Trino. "https://trino.io/.", 2023-06-20.

[TW96] D. L. Tennenhouse and D. J. Wetherall. "Towards an Active Network Architecture." *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 2, 5–17, 1996.

[VMA+20] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. "Building an Elastic Query Engine on Disaggregated Storage." In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, p. 449–462. USENIX Association, USA, 2020.

[WBKA23] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso. "Ephemeral Per-query Engines for Serverless Analytics." In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB) 2023*, vol. 3462 of *Workshop on Serverless Data Analytics (SDA'23)*. 2023.

[WBKA24a] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso. "Boxer: FaaSt Ephemeral Elasticity for Off-the-Shelf Cloud Applications." *arXiv:cs.DS:2407.00832*, 2024.

[WBKA24b] M. Wawrzoniak, R. Bruno, A. Klimovic, and G. Alonso. "Work in Progress: Imaginary Machines: A Serverless Model for Cloud Applications." In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies*, SESAME '24. Association for Computing Machinery, New York, NY, USA, 2024.

[WDH+22] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi. "Serverless Data Science - Are We There Yet? A Case Study of Model Serving." In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. 2022.

[WFLH18] C. Wu, J. M. Faleiro, Y. Lin, and J. M. Hellerstein. "Anna: A KVS for Any Scale." In *ICDE*. 2018.

[WLZ+18a] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. "Peeking behind the Curtains of Serverless Platforms." In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18. 2018.

[WLZ+18b] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. "Peeking Behind the Curtains of Serverless Platforms." In *Annual Technical Conference (USENIX ATC 18)*, pp. 133–146. Boston, MA, 2018.

[WMB+22] M. Wawrzoniak, I. Müller, R. Bruno, A. Klimovic, and G. Alonso. "Short-lived Datacenters." *arXiv:cs.DS:2202.06646*, 2022.

[WMB+24] M. Wawrzoniak, G. Moro, R. Bruno, A. Klimovic, and G. Alonso. "Off-the-shelf Data Analytics on Serverless." In *CIDR*. 2024.

[WMBA21] M. Wawrzoniak, I. Müller, R. Bruno, and G. Alonso. "Boxer: Data Analytics on Network-enabled Serverless Platforms." In *CIDR*. 2021.

[WMUW19] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl. "From Hack to Elaborate Technique—A Survey on Binary Rewriting." *ACM Comput. Surv.*, vol. 52, no. 3, 2019.

[wrk09] wrk - a HTTP benchmarking tool. "https://github.com/wg/wrk.", 2021-10-09.

[WSH19] C. Wu, V. Sreekanti, and J. M. Hellerstein. "Autoscaling Tiered Cloud Storage in Anna." *PVLDB*, 2019.

[WSH20a] C. Wu, V. Sreekanti, and J. M. Hellerstein. "Transactional Causal Consistency for Serverless Computing." In *SIGMOD*. 2020.

[WSH20b] C. Wu, V. Sreekanti, and J. M. Hellerstein. "Transactional Causal Consistency for Serverless Computing." In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, p. 83–97. Association for Computing Machinery, New York, NY, USA, 2020.

[WXY+22] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. "MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters." In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 945–960. 2022.

[WZM+20] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng. "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache." In *USENIX FAST*. 2020.

[YTAI23] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro. "zpoline: a system call hook mechanism based on binary rewriting." In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 293–300. USENIX Association, Boston, MA, 2023.

[ZCC+20] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. "Fault-tolerant and transactional stateful serverless workflows." In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 1187–1204. 2020.

[ZFPS20] W. Zhang, V. Fang, A. Panda, and S. Shenker. "Kappa: A Programming Framework for Serverless Computing." In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, p. 328–343. 2020.

[ZGXA21] M. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics." In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[ZKK03] H.-J. Zuberbühler, H. Krueger, and A. Kündig. "Delay perception thresholds in human-computer interaction. fundamentals for CSCW-applications." Swiss Federal Institute of Technology Zurich, Institute of Hygiene and Applied Physiology, Zürich, 2003. XVII International Annual Occupational Ergonomics and Safety Conference; Conference Location: Munich, Germany; Conference Date: 2003.

[ZLG+17] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. "BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning." In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, p. 338–350. Association for Computing Machinery, New York, NY, USA, 2017.

[Zoo29] Zookeeper Benchmark. "https://github.com/brownsys/zookeeper-benchmark.", 2024-05-29.

[ZWT+23] Z. Zhao, M. Wu, J. Tang, B. Zang, Z. Wang, and H. Chen. "BeeHive: Sub-Second Elasticity for Web Services with Semi-FaaS Execution." In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, p. 74–87. Association for Computing Machinery, New York, NY, USA, 2023.

[ZXW+16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. "Apache Spark: A Unified Engine for Big Data Processing." *Commun. ACM*, vol. 59, no. 11, 56–65, 2016.

[ZYWY19] C. Zhang, M. Yu, W. Wang, and F. Yan. "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving." In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019.

[ZZZ+19] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson. "Slim: OS Kernel Support for a Low-Overhead Container Overlay Network." In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 331–344. USENIX Association, Boston, MA, 2019.