Boxer: FaaSt Ephemeral Elasticity for Off-the-Shelf Cloud Applications

Michael Wawrzoniak[1], Rodrigo Bruno[2], Ana Klimovic[1], Gustavo Alonso[1]
[1]Systems Group, Dept. of Computer Science, ETH Zurich
[2]INESC-ID/Técnico, U. Lisboa

Abstract

Elasticity is a key property of cloud computing. However, elasticity is offered today at the granularity of virtual machines, which take tens of seconds to start. This is insufficient to react to load spikes and sudden failures in latency sensitive applications, leading users to resort to expensive overprovisioning. Functions as a Service (FaaS) provides significantly higher elasticity than VMs, but comes coupled with an event-triggered programming model and a constrained execution environment that makes them unsuitable for off-the-shelf applications. Previous work tries to overcome these obstacles but often requires re-architecting the applications. In this paper, we show how off-the-shelf applications can transparently benefit from *ephemeral elasticity* with FaaS. We built Boxer, an interposition layer spanning VMs and AWS Lambda, that intercepts application execution and emulates the network-of-hosts environment that applications expect when deployed in a conventional VM/container environment. The ephemeral elasticity of Boxer enables significant performance and cost savings for off-the-shelf applications with, e.g., recovery times over 5x faster than EC2 instances and absorbing load spikes comparable to overprovisioned EC2 instances.

1 Introduction

Elastic resource allocation is a key feature of cloud computing [13]. Cloud users rent virtual machines (VMs) on-demand to meet the resource requirements of their applications. However, the elasticity granularity 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 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* (Figure 2).

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 [18, 19, 46, 49]. 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 [12, 14, 46], still need to power

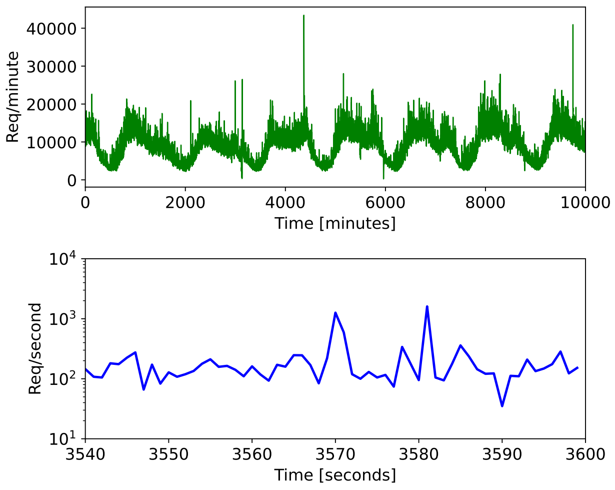
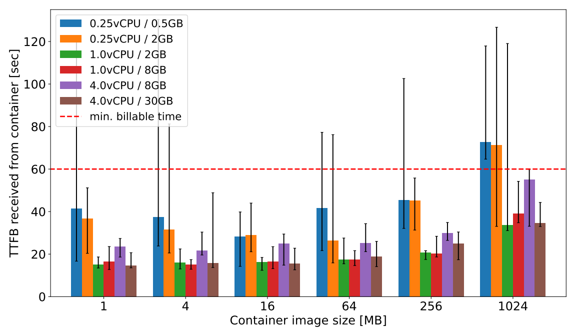


Figure 1. Reddit requests over 7 days (top) and 1 minute (bottom). Extracted from a public Reddit 2015 trace.

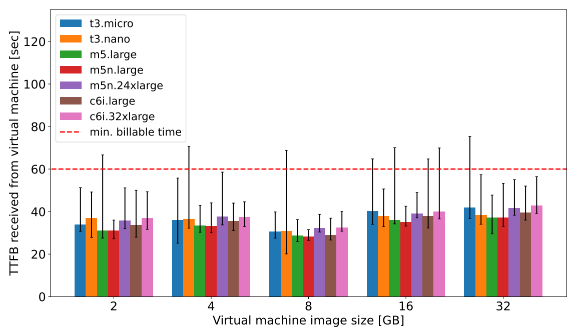
significantly more machines than necessary to support the aggregate load [30]. For example, the 2019 Borg traces show that CPU and memory utilization is only ~60%, even when the provider overcommits resources [46].

Function as a Service (FaaS) platforms, such as Azure Functions [7] and AWS Lambda [4] 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 [11]). 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 cloud applications off-the-shelf [24]. Existing FaaS platforms, force applications to be written as collections of short-lived, stateless functions, which cannot accept network connections while executing [16, 24, 41, 55].

Previous work tries to overcome these obstacles by implementing point solutions for various use cases, such as data analytics [32, 34], stream processing [43], video processing [21], and machine learning training [26]. Other solutions



(a) AWS Fargate/ECS container service



(b) AWS EC2 virtual machines

Figure 2. Median instantiation times of containers and VMs services, error bars are min and max values. Details in Section 2.1

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 [17, 20, 28, 29, 48, 52].

In this work, we show how off-the-shelf cloud applications can transparently benefit from *ephemeral elasticity* with FaaS. We focus 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 [9, 32, 34] and are suitable to serve steady application load. Our 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 generic cloud applications expect — on top of existing FaaS platforms. We achieve this by designing an interposition layer (now available to users, eventually supported by the cloud provider) deployed on top of existing FaaS platforms to emulate the necessary network, file system, and name resolution functionality. We have implemented such interposition layer, *Boxer*, on top of AWS Lambda. Boxer does not require changes to the application and integrates with traditional infrastructure orchestration tools such as Docker Compose. Boxer intercepts system C Library function calls and emulates the necessary network-of-hosts environment (network, file system, name resolution) that applications expect when deployed in a VM/container environment. We show that Boxer can be used to quickly absorb load bursts in an unmodified microservice application (DeathStar benchmark).Similarly, we show how an unmodified Zookeeper quorum running on EC2 can be quickly restored by replacing a failed node with a Lambda instance in about 6 seconds while doing the same with VMs takes close to one minute. These results demonstrate the potential of Boxer to provide ephemeral elasticity in a transparent manner.

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, we analyze a public Reddit trace [6] that includes user requests per second as an example of a web-based microservice application. From the dataset we extract two subsets: a 7-day trace containing the number of requests per minute, and a 1-hour trace containing the number of requests per second (Figure 1), from which we draw two key observations:

Observation #1: the 7-day trace (bottom plot in Figure 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, we find significant workload burstiness. Unlike the 7-day trace, the 1-minute trace requires highly elastic or highly overprovisioned infrastructure to be able to serve workload changes of more than an order of magnitude in a few seconds.

We conclude 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. Next, we demonstrate that no existing cloud infrastructure can cost-efficiently satisfy both types of elasticity. To bridge this gap, we propose the idea of *ephemeral elasticity*, a solution for affordable and highly elastic cloud infrastructure.

2.1 Virtual Machine and Container Elasticity

Conventional virtual machine or container service deployments offered by cloud providers have significant, often deeply intertwined, inefficiencies. In Figure 2 we explore the issue through experiments that measure the time to first byte (TTFB) received for different image sizes, container resources sizes (vCPU, memory) and virtual machine types.

We measure 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 on the VM. 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. This long initialization time makes 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% [36, 45] and very rarely exceeds 80% [31].

From here we conclude that VM and container-based deployments are suitable for slowly evolving loads (that changes in minutes and hours, e.g., in the 7-day trace in Figure 1), but not for high, unpredictable load bursts (e.g., in the second range seen in the 1-minute trace of Figure 1).

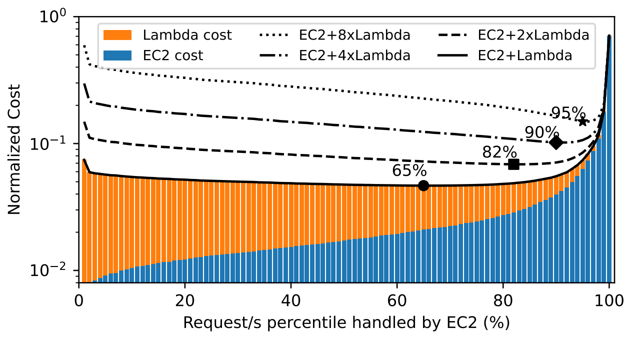
2.2 Ephemeral Elasticity

We propose the concept of *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. In this section, we estimate the potential of ephemeral elasticity to reduce resource overprovisioning and reduce the cost of running applications on the cloud. To do so, we conduct a cost analysis 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:

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

where β is the number of requests served by EC2 VMs; α and γ the throughput per core of EC2 and Lambda (measured for Deathstar microservice in §6.2). $\$_{\text{EC2}}$ and $\$_{\text{Lambda}}$ are the cost per second per core (we base on `c6g.2xlarge` VM and a 2GB Lambda); δ_{t} the load (number of requests) at time t.

Figure 3 (top) 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, 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 ephemeral elasticity idea. If more Lambda resources are necessary to process requests (because of inflexible resource allocation options, additional memory, or networking requirements,) total cost increases and best capacity allocation shifts (e.g., 82% for 2x Lambda per-request requirements.) Figure 3 (bottom) ilustrated the



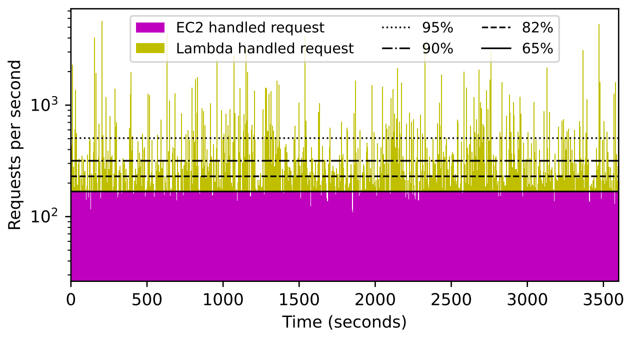


Figure 3. Reddit deployment cost (top) for different EC2 capacities using Lambda to handle requests that exceed capacity. 1-day Reddit trace (bottom) showing requests handled by EC2 and Lambda to minimize cost while providing capacity to handle all requests (c100), 65% of total requests corresponds to the level of 3% of the observed maximum. (Section 2.2)

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

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 temporairly. Table 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 $2 \times$ Lambdas are needed to service requests, the estimated cost reduction is over 25%.

2.3 Incompatible Datacenter Execution Models

Ephemeral elasticity is not available today as FaaS, and VMs and containers, operate under two different models:

Network-of-hosts is the classic and dominant datacenter model that has been in use for decades. Systems are built around the concept of long-running interconnected hosts that execute a collection of application processes that communicate via the network. Processes can be reached using various addresses and names, predominantly based on IP addresses, hostnames, and L4 port numbers.

Event-triggered functions is the model promoted by FaaS services. Systems are composed of a collection of functions executed in response to events. These functions can be composed into complex systems by arranging them into event-driven dataflow graphs. With their fast starting time and high invocation parallelism, they are suitable to highly burstable loads as the one depicted in Figure 1. However, functions are expected to be stateless, have limited networking, and limited execution time (e.g., up to 15 minutes in Amazon Lambda and 30 minutes in Azure Functions) [24, 41, 48] and are more expensive than a similar VM instances.

Takeaway: The mismatch between execution models prevents existing cloud applications from transparently and efficiently combining FaaS and VM/container infrastructure to achieve ephemeral elasticity.

3 System Requirements and Assumptions

To achieve the vision of fast ephemeral elasticity, we identify the following requirements for our Boxer prototype:

Work with existing platforms: To test feasibility and to avoid making unrealistic assumptions, Boxer should run on a 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, we 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), the environment must be transparently emulated to match what unmodified applications expect.

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 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 very small; this makes system overhead play a proportionally larger 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 [5], to pave a path to adoption in practice.

We assume that Boxer and the application are used by the same tenant so that there is no incentive for the application code to escape the mechanism used by Boxer. We assume that the guest applications are *cooperative*, we do not aim to prevent applications from circumventing the mechanisms we provide Second, we assume 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. Third, we assume 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. Lastly, our current system prototype does not aim to provide complete transparency (e.g., applications can find out they are running in Boxer) however, we do not aim to cover unusual corner cases for now as the functionality of typical applications is not affected.

4 Boxer System Design

We provide an overview of the Boxer system and the reasoning that guided our design of such an interposition layer.

4.1 Design Overview

Boxer must emulate the required *network-of-hosts* execution model to applications on top of the *event-triggered-functions* model of the serverless platform. Since neither applications nor platforms can be modified, we design Boxer as an interposition layer between cloud applications and platforms, i.e., a form of *cloud overlay* (Figure 4). To achieve this, Boxer *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.

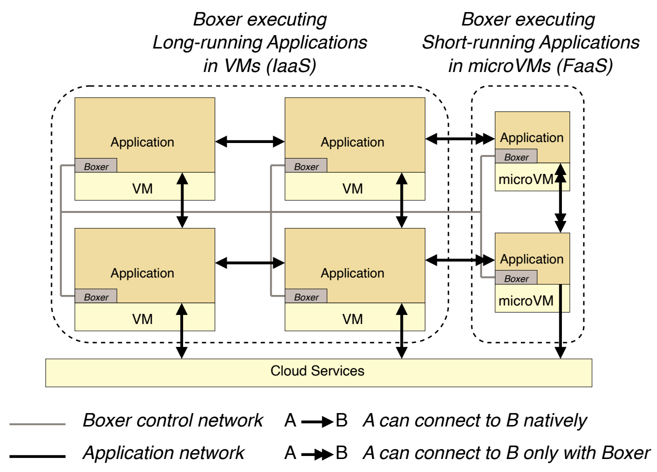

Boxer executing Long-running Applications in VMs (IaaS)
Boxer executing Short-running Applications in microVMs (FaaS)
Application
Boxer
VM
microVM
Cloud Services
Boxer control network
Application network
A can connect to B natively
A can connect to B only with Boxer

Figure 4. An unmodified long-running networked datacenter application running in VMs (temporarily) augmented with FaaS microVM based ephemeral elasticity using Boxer.

4.2 Intercepting the guest application execution

Interposition must be enforced dynamically at runtime and not by modifying the application code or binaries. At the same time, our target platforms include microVMs of publicly available FaaS services (AWS Lambda) that provide a restricted environment. These restrictions eliminate approaches to trap application executions that rely on hardware-accelerated nested virtualization, modifying the kernel, or loading kernel modules. On the other hand, unprivileged userspace virtualization techniques based on full dynamic translation (e.g., QEMU) add too much overhead to be viable. We also cannot rely on other standard methods to intercept system calls of Linux processes, such as those based on `ptrace` or `seccomp` system calls, because their use is restricted in the unprivileged environment.

Given this combination of constraints, we choose to intercept the application execution at the system C Library function call level. We implement the interception of the calls by controlling the dynamic linking of application processes as they started (described in §5). Hence, our 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, supporting our requirement for low system overhead. To minimize the overall performance overhead of the emulation, Boxer aims to limit the intercepted surface area to a minimum. We designed the system 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 are limited to the control path operations only. Most significantly, we avoided intercepting data path calls (e.g. `send`, `write`, `recv`, `read`, `sendfile`) and I/O notification calls (e.g. `epoll`, `select`) providing no-overhead performance for those operations.

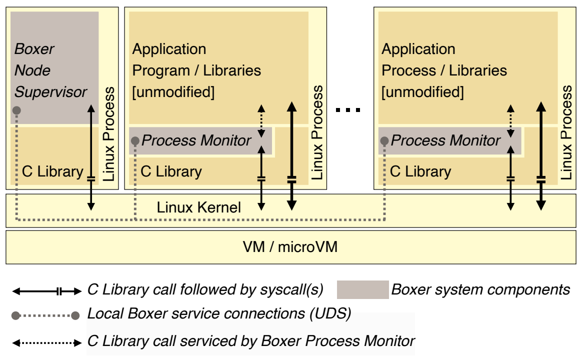

Boxer Node Supervisor
C Library
Linux Process
Application Program / Libraries [unmodified]
Process Monitor
C Library
Linux Process
...
Application Process / Libraries [unmodified]
Process Monitor
C Library
Linux Process
Linux Kernel
VM / microVM
C Library call followed by syscall(s)
Boxer system components
Local Boxer service connections (UDS)
C Library call serviced by Boxer Process Monitor

Figure 5. Boxer node (VM, microVM, or a container).

Figure 5 shows the components running on each node in a distributed application cluster with Boxer. The Boxer Process Monitor (§5) is the system component that is responsible for intercepting the necessary system C Library calls. It is loaded by the Node Supervisor (described further in §5) 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.

4.3 Emulating the network-of-host model

Emulating the *network-of-hosts* model that spans all nodes participating in a Boxer setup (VMs, containers, microVMs of FaaS) requires providing additional services to the guest applications. Boxer exposes all nodes, including FaaS microVMs as networked hosts to the guest application. To provide network connectivity between different hosts, Boxer must provide network transport between the hosts, manage network addresses, and provide hostname resolution, all these services are provided by the Node Supervisor.

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 without interfering with the guest application. For the former two, we chose to use Unix domain sockets because they can be used to send file descriptors between processes. For the later, we developed a technique of delivering the required signals to Process Monitors using marked local stream sockets (we later refer to them as *signal sockets*) that requires minimal mechanism in the Process Monitor and is compatible with the guest using blocking and non-blocking I/O.

5 System Implementation

In the following sections we discuss some implementation aspects of the Process Monitor (PM), the Node Supervisor (NS), and how they interact to provide the environment emulation, what services are provided and how the system can be transparently used with container orchestration tools (§5.1).

Process Monitor (PM). Boxer PM is responsible for selectively intercepting the execution of guest application processes to emulate the desired 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 that are normally exported by the system C Library, and it is linked between the platform system C Library and the application program and application libraries (Figure 5). 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, a total of 24 functions are intercepted. 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 - there is no additional overhead once network connections are established or files are opened.

The functionality in the PM is kept to a minimum, with no state persisted between intercepted calls, and with most of the functionality accessed through the NS services. PMs access their local NS by exchanging messages on a named Unix domain socket, referred to as *service connections*. For some of the intercepted calls, e.g., `getaddrinfo`, the functionality of PM is 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 service connections is more involved. For example, the protocol requires that when handing an intercepted `accept` function, a non-blocking native accept call must be made before potentially proceeding with sending an accept request to Boxer. This is because if Boxer has a ready connection to be accepted by the guest program, and no process is blocked to accept it, the program may be waiting for the I/O event to be delivered before calling accept (e.g. via epoll). To address such a scenario, Boxer NS will create a *signal connection*, a local connection from a reserved address

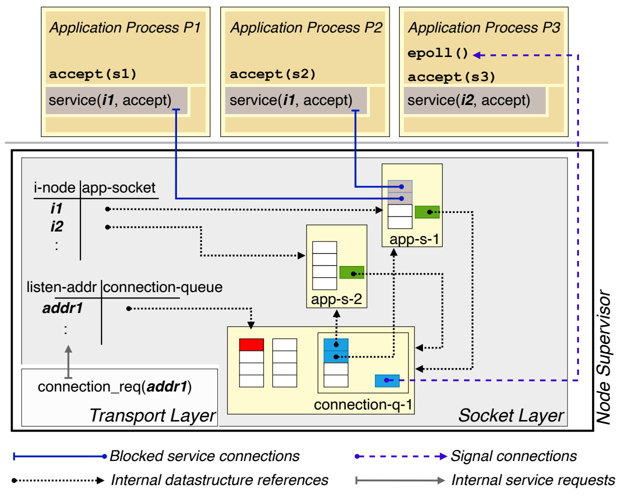


Figure 6. A configuration of core internal data structures of the stream socket layer for listening sockets (§5).

only to trigger a matching I/O event delivery to the guest program. If the guest process then calls accept, the PM will first accept the signal connection, discard it, and then proceed to send the service request to the NS to receive the new 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 substantial setup before issuing service requests, but the main mechanisms ara implemented as services, leaving the PM stateless and relatively simple.

Node Supervisor (NS). The NS is an unprivileged process that runs in every node (VM, container, or microVM) participating in the Boxer network (Figure 5). The NS is responsible for managing the local guest application processes, servicing the requests of the local PMs, and maintaining the control network with NSs of other nodes in the network.

The NS starts the specified guest application programs (with the specified arguments and additional configuration environment variables) and preloads all of their processes with the PM library (§5). It then listens for the PMs to open local service connections and start issuing requests.

The secondary role of the NS is to bootstrap and maintain a 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 TCP connectivity that supervisors establish between nodes when they start. 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.

Network Service. Conceptually, the network service is composed of two layers: the socket layer and the transport layer. The socket layer provides the mechanism for creating and setting up guest application sockets. The network transport layer (not meant to be interpreted with OSI model) provides network data delivery between Boxer nodes that can back the sockets created by the socket layer.

Socket Layer The socket layer interacts with the PMs from above and with the transport layer from below. When a PM intercepts a relevant socket call, e.g., `connect` call to establish a stream connection to a destination in Boxer network, it will send a request to the network service to initiate the connection to the remote host. 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. To support unmodified datacenter applications, Boxer socket layer must implement a mechanism to support the complete (stream) socket interface with correct error handling, including non-blocking I/O, and interactions with other system features, such as the ability to share sockets between different processes.

Figure 6, shows a subset of internal data structures on the passive side of the socket layer. It shows the state configured for 2 listening sockets. One of the sockets is shared between two guest application processes (P1 and P2), which are both waiting in blocking `accept` to receive new connected sockets. Process P3 has a different socket and uses non-blocking I/O to accept new connections. However, both of the listening sockets are bound to the same local address. This example is not an uncommon interaction of features used by datacenter applications, and Boxer must be able to handle it.

The socket layer keeps track of sockets used by the guest processes. First, Boxer maintains the mapping between inodes and 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, first, it may need to look up the inode associated with the relevant socket and use that in the request. The socket layer can then map it to the unique socket data structure. In Figure 6, both processes P1 and P2 request `accept` service on the same inode value `i1` which maps their requests to the same listening socket entry. Because these two requests are blocking, the PMs will block waiting on the responses. 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 will be drained once there are matching new connections that can be passed back to the blocked PMs, 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 6 example, all sockets point to the same connection-queue `connection-q-1`. Connection-queues are created when guest processes create listening sockets bound to a new address. They are added to the connect-queue-table that is indexed by the listening address. In the 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 makes a connection request to the socket layer, it will use the connection destination address in the request. 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 transport setup may continue. If there is no match, 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, references to to the matching listening sockets are used to return the new sockets to the blocked processes on the accept queues (e.g., Process 2 on the accept queue of socket app-s-1). The PMs are unblocked returning the new sockets to the application.

To handle non-blocking accept requests by the guest processes, when a new connection is available, the network service will create a new signal-connection to the local address that is bound to the real socket that the guest process is listening on. This is also the socket that the guest process (oblivious to what is actually happening) will add to its I/O notifications (e.g., epoll_ctl) to be notified by the kernel if there are new connections to accept. The signal-connection is configured to trigger this event, and if the guest process chooses to accept, the PM will hide (and discard) the signal-connection and make a request to the network service to retrieve the new connection from the appropriate connection-queue, or return immediately if none are left.

Transport Layer This layer is responsible for the setup of data delivery for the sockets managed by the socket layer. Currently, Boxer has implementations of direct TCP, NAT-hole-punching TCP transport, and IP-forwarding-proxy TCP transport. Other transports, such as those based on S3, DynamoDB, or other intermediary services or overlays, could be implemented in the future. The transport layer implementations use the control network managed by the NSs to exchange necessary messages to configure connectivity. For example, the NAT-hole-punching TCP transport that Boxer uses in AWS Lambda, exchanges messages with remote Boxer nodes to agree on the addresses to be used for NAT-hole-punching. Once agreed, direct TCP connections are established and passed up to the socket layer and then to the guest processes, transparently to the application.

Coordination Service. As Boxer nodes join the network, they first contact a node that is the seed coordinator to be assigned a unique node ID, bootstrap their network membership set, and register their name. All Boxer nodes run a coordinator service that listens for membership updates to maintain its local membership set and to propagate updates to other nodes connected to it. The membership set contains records for node-ids, their addressable IP addresses, and the (optionally) assigned names.

The NS 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.

Name Resolution. 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, e.g., 'node-ID' name will always resolve to the IP address of the Boxer node with the named ID.

Utilities. Boxer provides additional useful utilities, one of which is the ability to transparently remap file system names visible to guest applications. This is useful when applications expect hard-coded pathnames that are not available or are restricted in FaaS. Boxer also 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.)

5.1 Container Orchestration with Boxer

Container-based orchestration systems, such as Kubernetes [2], Docker Swarm [1], or Docker Compose [5], 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. To enable this we produce Boxer versions of commonly used base container images that can be used to transparently define application containers as if Boxer was not present, and use *trampoline container* technique to invoke Boxer functions or containers via the same container orchestration systems (Figure 7). Boxer trampoline containers are context-sensitive containers that start the container execution differently depending on their environment. When

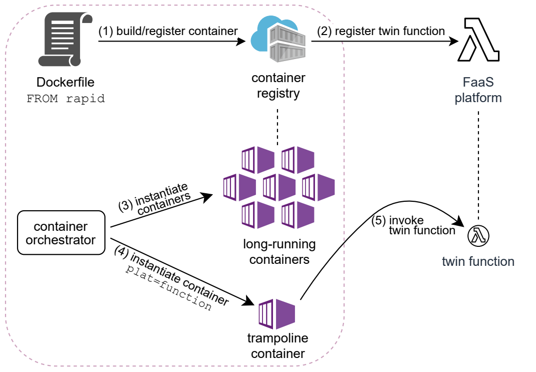


Figure 7. Container Orchestration with Boxer.

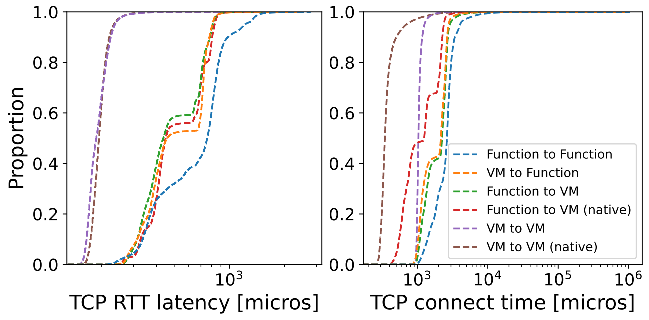


Figure 8. Empirical CDF of RTT latencies (left) and TTFB TCP connection establishment times (right) between different types of connecting and accepting hosts running Boxer or without (native).

the orchestrator starts a container but specifies target platform to be a 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 function NS to join the overlay and start the appropriate container entrypoint running in FaaS. 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 phantom-container, receiving logs, waiting for the function container to terminate, signaling termination to the container orchestrator that stays under the illusion that the application is running locally.

6 Evaluation

To evaluate Boxer we run, unmodified, DeathStarBench [23], a suite of cloud microservice benchmarks deployed using container networks that mimic how large scale, complex distributed applications are often deployed in the cloud. We use the Zookeeper benchmark to demonstrate how Boxer helps to quickly recover from node failures.

6.1 Microbenchmarks

To confirm that the properties of Boxer provided networking are compatible with the ephemeral elasticity use case, we measured connection establishment times and latency of Boxer provided TCP-hole-punching network transport for different combinations of endpoints. We used EC2 m4.large VMs and Lambda Functions with 3007MB of memory as endpoints. To measure connection establishment times, we measured time-to-first-byte (TTFB) observed by the client guest process running in Boxer connecting to a remote server guest process also running in Boxer. For comparison, we measured the native times without using Boxer for the two possible combinations of functions connecting to VM and VM-to-VM directly. To measure latency, the client program measured 128 rounds of 1024 byte ping-pong exchanges on an established connection. For each endpoint combination, the experiments were repeated 1024 times on 32 distinct endpoint pairs. Figure 8 shows the empirical CDF of the measurements. As expected, the connection establishment times when using Boxer TCP-hole-punching network transport compared to native times show the overhead of the connection setup and extra message round. Mean TTFB of VM-to-VM connections without Boxer (native) is 408μs while with Boxer is 1067μs. The latency results confirm that Boxer adds no data path overhead once connections are established. The mean RTT for VM-to-VM connections with and without Boxer are very similar at 198μs and 194μs, respectively, and have similar distributions. Boxer provided Function-to-Function connections have mean TTFB connection establishment of 2735μs, and RTT latency of 694μs with an increased dispersion. We find these acceptable for our use cases, especially considering that without Boxer the application processes cannot establish Function-to-Function connections at all.

6.2 Running DeathStarBench on Boxer

Next, we focus on DeathStarBench's *socialNetwork*, which offers a social network service to users and is organized using three microservice layers: i) front-end layer (implemented using an NGINX webserver); ii) logic layer (implemented using stateless Thrift services that communicate through RPCs); iii) caching and storage layer (implemented with MongoDB and Memcached 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 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.

We did not have to make *any modifications* to the application code to deploy DeathStarBench on AWS Lambda with Boxer. The benchmark was only modified to i) use hostnames

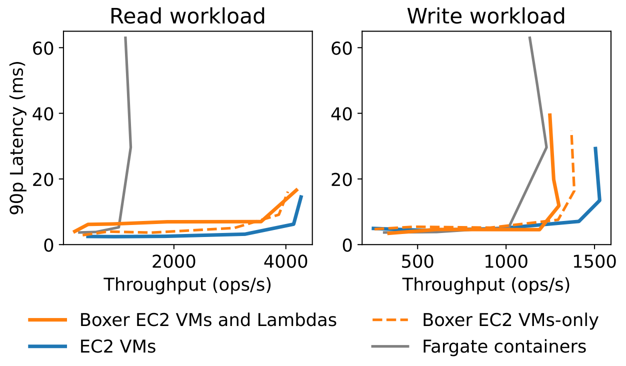


Figure 9. DeathStarBench results in a static deployment.

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.

Methodology. To evaluate Boxer, we use four deployments (refered to as *EC2-VMs*, *Boxer-EC2-VMs-only*, *Boxer-EC2-VMs-and-Lambdas*, *Fargate-containers*), one baseline and three exercising Boxer in different ways. (1) All components deployed as EC2 VMs(baseline, *EC2-VMs*). (2) All components deployed as VMs in EC2 but the components the front-end and logic layers use Boxer (*Boxer-EC2-VMs-only*). This deployment measures the performance overhead of using Boxer. (3) A mixed deployment with front-end, and caching and storage layers are deployed as VMs, and logic layer deployed using Lambdas (*Boxer-EC2-VMs-and-Lambdas*). (4) A mixed deployment with the logic layer using AWS Fargate container service (*Fargate-containers*).

To measure the throughput and latency of the end-to-end system we use two workloads included in the DeathStarBench suite. A 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` [10] 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 mostly 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 in AWS Ohio (us-east-2) region. All VMs use a base Amazon Linux 2 [3]. For front-end, and caching and storage layers, we use `t3a.micro` instances due to the memory requirements of the services included in these layers. For the logic layer, when deployed in VMs, we use `t3a.nano` instances. Each Lambda is configured with 2048MB of memory (we experimentally determined that in us-east-2, the performance of a 2048MB Lambda is similar to a t3a.nano VM instance). For Fargate, we deploy containers also with 2048MB of memory and 1.0

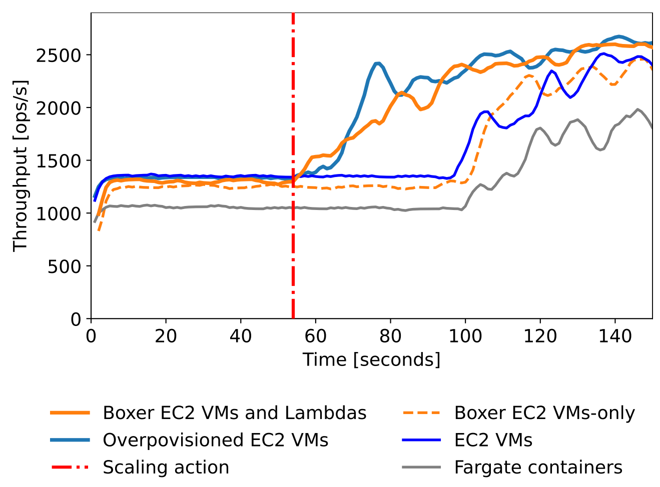


Figure 10. DeathStarBench write workload comparing elastic deployments (Section 6.2).

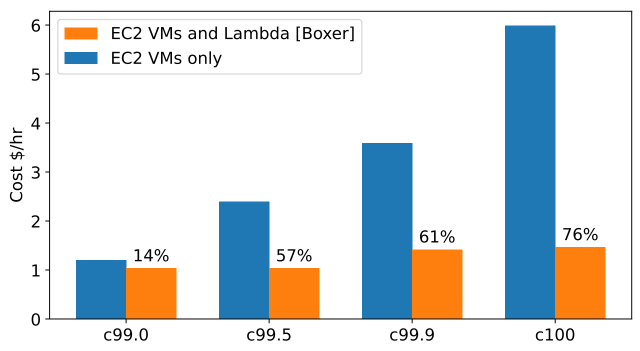


Figure 11. DeathStarBench logic layer absolute cost and cost reduction based on 1-day Reddit trace sample (Section 6.2).

vCPU unit (this configuration is also the one that yields faster container startup time, see Figure 2).

Overhead of using Boxer. Figure 9 shows the results for both read and write workloads across the four different types of deployments. For each workload, we collect the average throughput and 90th percentile latency with an increasing load in the system. 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.

We use a similar analysis to measure the overhead of launching the logic layer services in AWS Lambda by comparing the Boxer-EC2-only and Boxer deployments. Figure 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.

We conclude that using Boxer incurs a small performance overhead. 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. We now show how Boxer can provide *ephemeral elasticity* to increase the elasticity of microservices running on VMs and containers by leveraging serverless platforms. We start by deploying all logic layer services on VMs. When the load increases, additional logic layer services are allocated to handle the increased load either on VMs, containers, or Lambdas. In addition, we also include an overprovisioned VM deployment (*Overp. EC2*) in which already allocated resources are added to the pool of workers. Our goal is to compare the elasticity of different deployment types.

Figure 10 presents a throughput trace for different deployments. Throughput is measured using wrk [10] 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). EC2 and Fargate take approximately 45 seconds to fully deploy all new workers (t=100s), while Lambda and overprovisioned EC2 scale almost immediately (approximately 1 second, t=55s). 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 11 shows comparison of cost for using EC2 VM-based overprovisioning and Boxer elasticity (using EC2 and Lambda). Based on a 1-day Reddit trace sample and the DeathStarBench throughput benchmarks (Figure 9), we calculated 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). We then compared it to the cost of allocating a single VM instance for each logic layer service in VMs and on-demand dynamically scaling up using Boxer to Lambdas to absorb load bursts. We observe that the cost reduction for using Boxer provided elasticity ranges from 14% to 76% depending on the capacity levels.

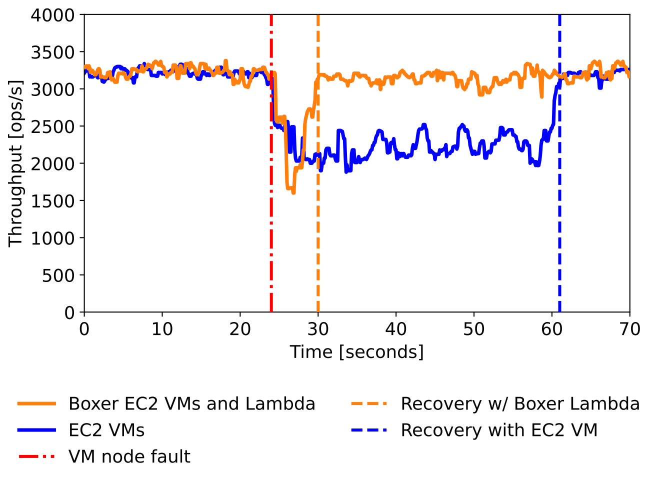


Figure 12. Recovering from node crash in a 3-node EC2 Zookeeper cluster using EC2 and Lambda using Boxer.

6.3 Elastic Fault Tolerance in Zookeeper

Boxer provided elasticity can also be used to reduce down time due to recovery from node crashes. Minimizing node down time 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.

For this scenario, we setup a 3-node Zookeeper [25] cluster deployed on EC2. Using this cluster, we forcibly shutdown one of the nodes and recover from the fault using either a newly allocated EC2 VM or using a Lambda with *Boxer*. We use `t3a.micro` VMs as Zookeeper nodes and Lambdas with 2048 MBs. We configure Zookeeper to allow dynamic reconfiguration, i.e., 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. We use a read-only workload based on the Zookeeper Benchmark[1]. Figure 12 shows an execution trace of the workload throughput through time. After approximately 25 seconds, one of the Zookeeper VMs is shutdown and a new instance (EC2 or Lambda) is started to replace the failed node. Using Boxer, the fault recovered in under 6.5 seconds compared to 37.0 seconds with VMs (EC2), a 5.7x improvement in Zookeeper node recovery time.

7 Discussion

Opportunities. The elasticity bottleneck shifts from resource allocation to the application. By leveraging the ephemeral elasticity, datacenter applications can 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 an interval that is too high.

Current limitations. Several limitations in the current prototype will be addressed as part of future work: *(1)* The current implementation of the process monitor relies on the lightweight dynamic linking mechanism for interposition, which cannot handle applications issuing system calls directly. We are investigating techniques based on lightweight selective binary rewriting [50] to extend the generality of the process monitor to arbitrary processes. *(2)* Boxer has only been tested on AWS, we plan to support other cloud platforms, but there may be provider-specific logic to adapt to each platform. *(3)* The interfaces that Boxer emulates have complex semantics and, combined with the additional constraints, make handling all of the cases challenging. We have successfully tested Boxer with several complex systems, but we are aware of corner cases that we have not handled yet, and we are working to make the system more complete.

8 Related Work

Although initially designed for event-triggered stateless functions, FaaS is now being presented as the next generation of cloud computing [41]. Fouladi et al. demonstrated using FaaS as a supercomputer on-demand to run highly parallel jobs like video encoding and software compilation [20, 21]. The gg framework enables users to run applications on FaaS by providing an intermediate representation and SDKs with which users can express their application as a composition of lightweight, functional containers [20]. Boxer pursues a similar vision but rather than just accelerating trivially parallel jobs, it aims to leverage FaaS for elasticity acceleration of off-the-shelf cloud applications. Boxer also helps generic applications adapt to node failures and load spikes without resorting to overprovisioning, similar to how MArk [53] spills to FaaS to accommodate ML inference load spikes, Beehive [56] utilizes FaaS nodes to offload load spikes for JVM-based applications, and Pixels-Turbo [15] accelerates query processing of unpredictable workload spikes with FaaS.

Improving elasticity by making compute and memory resources more fungible is an active area of research [12, 37, 38]. However, we argue that cloud users can benefit from high elasticity and greatly reduce overprovisioning for their applications without waiting for cloud providers to evolve and optimize their underlying infrastructure.

Complementary to Boxer, others have explored enabling general computation on FaaS by providing GPU support [8, 27, 39], familiar concurrency APIs [55], transactional workflows [42, 54], atomicity guarantees over shared storage [44], and handling timeouts by checkpointing and generating

[1]https://github.com/brownsys/zookeeper-benchmark

continuation functions [55]. Other optimizations such as locality-oriented scheduling [22], cold-start reduction [33], and memory footprint optimizations [40] are orthogonal to Boxer as our system is implemented on top of such serverless architectures and benefits from such optimizations.

Prior work has addressed function networking limitations by using intermediaries to relay messages between functions. *mu* [21] proposed a framework for parallel computation and communication across buffers and relaying messages between functions. Others [28, 32, 34, 35, 51] leveraged external storage to exchange data between functions. Projects such as InfiniCache [47] and gg [20] use a proxy-based approach. Nat-hole-punching in AWS Lambda has been previously leveraged for data analytics and for function communication primitives by [17, 48], but not to transparently provide network-of-hosts model to datacenter applications.

9 Conclusion

We presented Boxer, a system that transparently improves cloud application elasticity. We demonstrated that it is possible to unbundle the *event-triggered functions* programming model of FaaS from its underlying microVM resources and to provide the *network-of-hosts* programming model on top of them. We showed that this enables running cloud applications using publicly available FaaS infrastructure, which can be used to temporarily augment long-running unmodified cloud applications with fast ephemeral elasticity. We showed that the availability of such fast ephemeral elasticity provides elasticity-fill that can significantly reduce the level of overprovisioning required to react to dynamic load and failure recovery.

References

[1] [n. d.]. Docker Swarm overview. https://docs.docker.com/engine/swarm/

[2] [n. d.]. Kubernetes. https://kubernetes.io/

[3] 2023. Amazon Linux 2. https://aws.amazon.com/amazon-linux-2/

[4] 2023. AWS Lambda. https://aws.amazon.com/lambda

[5] 2023. Docker Compose. Retrieved 2023-07-27 from https://docs.docker.com/compose/

[6] 2023. May 2015 Reddit Comments. https://www.kaggle.com/datasets/kaggle/reddit-comments-may-2015

[7] 2023. Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions

[8] 2023. Nuclio Serverless Platform. https://nuclio.io/

[9] 2023. Prime Video Switched from Serverless to EC2 and ECS to Save Costs. https://www.infoq.com/news/2023/05/prime-ec2-ecs-saves-costs/

[10] 2023. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk

[11] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.

[12] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 735–751. https://www.usenix.org/conference/osdi20/presentation/ambati

[13] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (apr 2010), 50–58.

[14] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take It to the Limit: Peak Prediction-Driven Resource Overcommitment in Datacenters. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 556–573. https://doi.org/10.1145/3447786.3456259

[15] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proc. ACM Manag. Data* 1, 2, Article 161 (jun 2023), 27 pages. https://doi.org/10.1145/3589306

[16] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (2019), 44–54.

[17] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. FMI: Fast and Cheap Message Passing for Serverless Functions. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 373–385. https://doi.org/10.1145/3577193.3593718

[18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. 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)*. 153–167.

[19] Christina Delimitrou and Christos Kozyrakis. 2014. 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)*.

[20] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *USENIX ATC*.

[21] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI*.

[22] Alexander Fuerst and Prateek Sharma. 2022. Locality-Aware Load-Balancing For Serverless Clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)*.

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS*.

[24] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR*.

[25] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems *(USENIX ATC'10)*.

[26] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data*. 857–871.

[27] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. 2018. GPU Enabled Serverless Computing Framework. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 533–540.

[28] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI*. 427–444.

[29] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1505–1519. https://www.usenix.org/conference/nsdi23/presentation/liu-david

[30] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. 2884–2892. https://doi.org/10.1109/BigData.2017.8258257

[31] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. 2017. Imbalance in the cloud: An analysis on Alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*. 2884–2892. https://doi.org/10.1109/BigData.2017.8258257

[32] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD*.

[33] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *USENIX ATC*.

[34] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD*.

[35] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI 19*.

[36] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 7, 13 pages. https://doi.org/10.1145/2391229.2391236

[37] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K. Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. 2023. Unleashing True Utility Computing with Quicksand. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 196–205. https://doi.org/10.1145/3593856.3595893

[38] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA, 1409–1427.

[39] Klaus Satzke, Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Andre Beck, Paarijaat Aditya, Manohar Vanga, and Volker Hilt. 2020. Efficient GPU Sharing for Serverless Workflows. In *Proceedings of the 1st Workshop on High Performance Serverless Computing (HiPS '21)*. 17–24.

[40] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. *Proceedings of the Seventeenth European Conference on Computer Systems* (2022).

[41] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (April 2021), 76–84.

[42] Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[43] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. 2023. Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 301–314. https://www.usenix.org/conference/atc23/presentation/song

[44] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[45] Xiaoyang Sun, Chunming Hu, Renyu Yang, Peter Garraghan, Tianyu Wo, Jie Xu, Jianyong Zhu, and Chao Li. 2018. ROSE: Cluster Resource Scheduling via Speculative Over-Subscription. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 949–960. https://doi.org/10.1109/ICDCS.2018.00096

[46] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Article 30, 14 pages.

[47] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *USENIX FAST*.

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

[49] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. 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)*. 945–960.

[50] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From Hack to Elaborate Technique—A Survey on Binary Rewriting. *ACM Comput. Surv.* 52, 3, Article 49 (jun 2019), 37 pages. https://doi.org/10.1145/3316415

[51] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for Any Scale. In *ICDE*.

[52] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2019. Autoscaling Tiered Cloud Storage in Anna. *PVLDB* (2019).

[53] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.

[54] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1187–1204.

[55] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. 328–343.

[56] Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, and Haibo Chen. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 74–87. https://doi.org/10.1145/3575693.3575752