SigmaOS: a Cloud OS with a Unified API for Serverless and Microservice Applications

by

Ariel Szekely

S.B.H., University of Texas at Austin (2020)
S.M., Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2026

Authored by:	Ariel Szekely Department of Electrical Engineering and Computer Science August 14, 2026
Certified by:	M. Frans Kaashoek Professor of Computer Science and Engineering, Thesis Supervisor
Certified by:	Robert Morris Professor of Computer Science and Engineering, Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Thesis Committee

Thesis Supervisors

M. Frans Kaashoek
Professor of Computer Science and Engineering
Department of Electrical Engineering and Computer Science

Robert Morris
Professor of Computer Science and Engineering
Department of Electrical Engineering and Computer Science

Thesis Readers

Christina Delimitrou
Associate Professor of Computer Science and Engineering
Department of Electrical Engineering and Computer Science

SigmaOS: a Cloud OS with a Unified API for Serverless and Microservice Applications

by

Ariel Szekely

Submitted to the Department of Electrical Engineering and Computer Science
on August 14, 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Many cloud applications use both serverless functions, for bursts of stateless parallel computation, and container orchestration, for long-running microservices and tasks that need to interact. Ideally a single platform would offer the union of these systems' capabilities, but current platforms do not meet this goal: serverless functions are lightweight but cannot easily communicate and act as servers with long-term state, while container orchestration offers general-purpose computation but instance start-up takes too long to support burst-parallelism.

σOS is a new multi-tenant cloud operating system that combines the best of container orchestration and serverless in one platform with one API. σOS computations, called *procs*, can be long-running, stateful, and interact with each other, making them a good match for both serverless and microservice tasks. A key aspect of the σOS design is its cloud-centric API, which provides flexible management of computation, a novel abstraction for communication endpoints, *σEPs*, which allow `proc`s of a tenant to communicate efficiently but prohibits `proc`s from sending packets to other tenants, and a flexible naming and coordination system.

One challenge in realizing σOS is quick `proc` start-up. Start latency can be broken down into two components: *setup* and *initialization*. Setup involves steps the cloud platform takes when starting an application, such as downloading its binary and creating an isolated execution environment. Initialization involves steps the application takes after it has started running but before it can do useful work, such as connecting to other services, coordinating to claim work, and downloading inputs. σOS addresses both the setup and initialization components of start latency.

σOS delivers fast setup to `proc`s with its design of *σcontainers*, a per-proc execution environment which provides lightweight-but-strong isolation. A key enabling observation is that both serverless and microservice applications rely on cloud services for much of the work traditionally done by the local OS (e.g., access to durable storage and additional compute resources). σcontainers exploit this observation to provide each proc with a slimmed-down local OS interface which can be isolated quickly. σOS enables fast initialization for `proc`s with *co-sandboxes*, a scriptable interface which allows developers to move much of an application's initialization work into the platform.

An evaluation with a few cloud applications shows that σOS provides good performance to serverless and microservice applications, multiplexes resources between tenants, and improves `proc` start latency with techniques complementary to state-of-the-art fast-starting serverless systems.

Thesis supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Thesis supervisor: Robert Morris
Title: Professor of Computer Science and Engineering

Acknowledgments

Earning a Ph.D. from MIT has been a personal dream of mine for many, many years. The road to this point has had its share of dead ends, twists, turns, disappointments, failures, moments of doubt, and a great deal of uncertainty as to whether I would ever get here. However, it could have been much harder still.

In my life I have had the great fortune of being surrounded by a loving family, supportive friends, and caring mentors who guided me through the difficult moments. These people made me the person I am today. They made the past few years' good moments far outshine the challenging ones. They brought joy, camaraderie, love, and support into my life in a way that I feel and appreciate each day. Words cannot fully express my gratitude. This chapter of my thesis is dedicated to these people, without whom this thesis would not have been possible.

Thank you to Emmett Witchel and Chris Rossbach, for starting my research career, getting me excited about systems, and giving me the opportunity to come to MIT and study with Frans and Robert.

Thank you to Christina Delimitrou for serving on my PhD thesis committee, and now sitting through TWO defenses of mine.

Thank you PDOS: Lily Tsai, Tej Chajed, Jonathan Behrens, Anish Athalye, Derek Leung, Baltasar Dinis, Ben Holmes, Dongjae Lee, Gohar Irfan Chaudhry, Ryan Lemkuhl, Sanjib Bhat, Yun-Sheng Chang, Ankit Bhardwaj, Atalay Mert Ileri, Assel Ismoldayeva, Ben Kettle, Cel Skeggs, Inho Cho, Josh Fried, Ralf Jung, Zain Ruan, Nickolai Zeldovich, Henry Corrigan-Gibbs, and Adam Belay. A special thank you to the 2020 PDOS cohort, who made all the milestones along the way that much more fun and who I feel lucky to call friends: Alexandra Henzinger, Upamanyu Sharma, Kevin Liao. A special thank you Hannah Gross for being such a supportive collaborator, to Benny Rubin and Akshay Srivatsan for helping me refine my ideas and for all of the fun research discussions and ideas. Thank you PDOS for all of the great conversations, the camaraderie, for pushing me out of my comfort zone and helping me expand my thinking with new ideas and challenges to my own. Thank you for all the feedback on my work and the encouragement to keep going when things were tough.

Thank you to all the great M.Eng. and undergraduate students who chose to work with me and contribute to σOS: Ivy Wu, Nicolas Camenisch, Nathan Mustafa, Ryan Chang, Frederick Tang, Katie Liu, Kevin S. Chen, Yizheng He, Gideon Witchel, Maximillian Smith, Alan Zhu, Nour Massri. You have taught me a tremendous amount, helped improve the σOS project no-end through your ideas and all the tricky bugs you found (apologies for those...). Most importantly, you have given me some of the most fun moments of my Ph.D. by shining with your creativity, passion, and excitement to learn.

Thank you Robert, for your incredible attention to detail. Thank you for your inquisitiveness, for instilling in me your desire to deeply understand every result. Thank you for teaching me that sometimes the simplest questions are the deepest and hardest to answer, and for teaching me to be brave and ask questions regardless of whether or not they would make me look silly. Thank you for teaching me to relentlessly chase the *right* solution, and passing on your incredible ability to re-examine past work with fresh eyes and start from scratch if that is the right thing to do.

Thank you Frans, for your patience and for your direct but kind feedback without which I would not have grown into the researcher and thinker I am today. I can't think of any other mentor who has shaped my thinking as meaningfully as you have. You helped me develop my own taste in research and systems design. You taught me how to dissect a problem. Your undying optimism helped me push through the difficult moments of my Ph.D. when it felt like the resubmission cycle would never end. Your positivity made the bright moments of the Ph.D. that much brighter. But most importantly, thank you for truly caring about me as a person, for looking out for me and for doing what is best for me. Thank you for your willingness to give me a second chance, and for fighting and advocating for me and my work. I am forever grateful.

Thank you to my friends for always being there to support me, especially when the going got tough, and for lending me your ears and enduring my rants. Thank you for always being willing to crack a silly joke and laugh, and for all the hangouts and adventures we shared.

Thank you to my extended family for loving me, caring for me, and being there for me even when I was present less than I would have liked to be. You have been a source of comfort and joy which I hope to return to all my life. Los quiero con todo mi corazon.

Thank you to my new Van Cauwenberge-Rakic family, Natacha Rakic, Francoise Van Cauwenberge, and Jean-Marie Rakic, as well as the rest of the Van Cauwenberge clan, for welcoming me with open arms and making me feel like one of you. I deeply feel and appreciate the kindness and warmth you have shown me as I join your family.

Mama and Papa, thank you for giving me the greatest gift a son could ask for: a loving family, and a beautiful example of what it means to be a good person. Thank you for celebrating every good moment and showing me what it means to live a good life. Thank you for loving me, no matter what, because of who I am and not just because of what I achieve. Thank you for teaching me how to learn, how to grow, and how to get back up and play the next point after a tough loss. Thank you for always being on my side, cheering me on, and believing in me more than I believed in myself. I would not be the man I am today without you.

My dearest Marianne, thank you for always knowing the right words to say, when to hold me close in the warmest hug. Thank you for being the first to celebrate every win, the first to cry with me for every loss, and the first hand helping me stand back up. Thank you for inspiring me to be the best person I can be. Thank you for going on countless adventures and cooking experiments, and laughing about all the minor disasters along the way. Every day I am in disbelief with how lucky I am to have you as my wife. I love you.

This thesis is dedicated to all of you. Thank you, always.

Contents

Chapter 1

Introduction

The last decade has seen dramatic growth in the popularity of public clouds as a means for developers to deploy their applications. Cloud platforms relieve developers, tenants in this context, of the burden of procuring and administering hardware resources themselves. Tenants lease compute resources on-demand and scale their allocations up and down on short notice. Cloud providers multiplex tenants across shared hardware in order to efficiently use hardware resources, amortize the cost of ownership, and pass some of the savings to their tenants.

Tenants deploy a wide range of applications on public clouds, which has led providers like Amazon Web Services (AWS), Google Cloud, and Microsoft Azure to offer a correspondingly diverse choice of platforms to support these applications. These include on-demand Virtual Machines (VMs), Functions as a Service (FaaS), container systems, and managed Kubernetes clusters. Platforms differ in their fault-tolerance guarantees, performance characteristics, backwards-compatibility, and limitations they place on applications. For example, VMs start slowly but offer hardware-enforced isolation and allow applications to be stateful and communicate. FaaS platforms start computations quickly, but restrict communication, time-limit applications, and guarantee at-least-once execution semantics.

The diverse platform offerings support many applications well, and have been profitable for cloud providers. However, this arrangement has several downsides, both for tenants and for providers. Tenants are forced to choose a platform on which to deploy each application component, with the accompanying restrictions of that platform. Applications which need the fast-starting capabilities of FaaS systems can't communicate, and applications which need the generality and communication offered by VMs or containers can't start quickly. Supporting multiple platforms is also a burden for providers. Providers must decide how to distribute hardware resources among their platforms, and redundantly add new hardware or new features to each platform.

This thesis envisions a new multi-tenant cloud compute platform, σOS, which unifies support for long-running stateful applications and FaaS applications with a single API. The goal of σOS is to bring the best characteristics of cloud platforms which support these applications, namely fast starts and flexible communication, into a single system, and allow all tasks to benefit from these features. The remainder of this section discusses the motivation behind σOS, its goals, approach, our contributions, and the roadmap of this thesis.

1.1 Motivation

A typical cloud tenant executes a variety of tasks on many machines: long-running analytics, short background job queues, fleets of web API servers, sharded database servers, and more. An important axis on which these workloads vary is the rapidity with which tasks come and go. Some are long-running stateful services, like key-value caches and API servers, most recently typified by microservices. A contrasting class of workloads consists of serverless functions, like ETL-style image processing applications [6, 30, 39, 55, 83], which are often invoked in large parallel bursts and tend to be short-lived. For convenience, this thesis will refer to these two workload classes as "microservices" and "serverless."

Cloud applications require infrastructure to deploy software, manage execution and communication, and enforce isolation. For microservice-style workloads, container orchestration [14, 38, 56] works well: each instance provides the full facilities of a traditional operating system, and is thus quite general-purpose. These instances, however, are ill suited to serverless workloads. The core problem is that container orchestration instances start slowly, on the order of hundreds of milliseconds to seconds. Slow start is reasonable in static situations but problematic if functions are short-running [2, 41, 67, 74, 105, 124], on the order of tens of milliseconds, or need burst-parallelism [46–48].

One source of slowness is that each instance involves a user-level Linux installation: an isolated read/write file system populated from an application-specific Linux container image [88]. Another is that, in order for cooperating instances to communicate, the setup process must configure each with an IP address and a connection to an isolated overlay network [116]. Isolation is important when a provider has many independent tenants. Finally, the mechanism for finding a machine with enough resources to execute a new instance is typically not fast enough for rapidly-initiated serverless functions [35].

Serverless platforms work well for many short-running and burst-parallel applications. Serverless platforms enable tasks to start quickly, on the order of tens of milliseconds [6], but are not a good fit for many microservices. Platforms like AWS Lambda, Google Cloud Functions, and Azure Functions don't provide the facilities for functions to accept incoming connections and communicate directly. The result is that functions which wish to exchange state, such as MapReduce mappers streaming output to reducers, need to proxy their communication via another service [46, 47, 70, 97].

1.2 Goals

An ideal platform would support microservices and serverless applications with a single API and give all applications the best of both worlds. The platform would be able to initiate new work rapidly enough for serverless uses, but also provide enough flexibility to satisfy microservice applications. This would simplify applications that need both kinds of support. Serverless functions which need to communicate [46, 47] could do so without a proxy, and could keep state [70, 94]. Microservices could quickly scale up their fleet size in response to spikes in load.

Specifically, a platform which supports both microservice and serverless applications must enable fast start, powerful facilities for tasks to communicate, uniform coordination and

naming, and isolation for both performance and security.

1.3 Approach

σOS is a provider-hosted multi-tenant distributed operating system that addresses the above challenges. A key enabling insight is that cloud applications depend chiefly on cloud infrastructure, such as remote storage services and cloud schedulers, for much of the work traditionally done by a local Operating System. The trend towards "cloud-native" applications [53] illustrates this arrangement. Such applications can be designed to require little from the local machine beyond CPU, memory, and network access to cloud services. This simplifying assumption allows σOS to provide applications with a single cloud-centric API which enables both the creation speeds that serverless workloads need and the communication services that microservice workloads require.

A unit of work in σOS is called a `proc`. σOS supports fast `proc` creation with two techniques: σcontainers and co-sandboxes. An σcontainer is a lightweight secure execution environment which σOS uses to isolate `proc`s. σcontainers start in 2.0 milliseconds because they provide only the facilities needed to implement the σOS API, and no more. For example, σcontainers assume that `proc`s will use cloud storage and so avoid the cost of creating per-`proc` local read/write file systems. Instead, σcontainers mount a generic read-only file system shared among multiple `proc`s. Similarly, σcontainers disallow Linux networking system calls and so avoid the cost of creating isolated network namespaces. Instead, the σOS API provides a network addressing scheme (σEPs) for `proc`s that is more efficient than per-instance IP addresses, but allows communication among the `proc`s of the same tenant.

σOS enables `proc`s to initialize quickly with co-sandboxes, a scriptable interface to declare application initialization steps and allow the platform to execute these steps on behalf of the application. In addition to a `proc`'s binary, developers can optionally upload an accompanying co-sandbox which specifies the `proc`'s initialization routine. When starting the `proc`, σOS then executes the co-sandbox in parallel with platform setup steps like downloading the application binary, creating the σcontainer, and starting the application's runtime. The co-sandbox can exchange RPCs to, for example, claim a task from a durable queue service or download application state or inputs from storage. Once the application is up and running inside its σcontainer, the co-sandbox bootstraps it by transferring initialization state and connections to the application.

To help a tenant's `proc`s cooperate, σOS provides a per-tenant naming service (`named`) inspired by `etcd` [42] and Plan 9 [93]. `proc`s use `named` to register σEPs as well as to store configuration information and small items of shared and/or fault-tolerant state. As an example, σOS has a fast and scalable placement service that chooses a machine to execute each newly spawned `proc`; this service organizes itself via `named`.

Following Borg [118], σOS asks developers to mark each `proc` as either latency-critical (LC), with reserved CPU time and RAM, or best effort (BE). σOS schedules BE `proc`s as CPU and memory become available, either on the completion of previous BE `proc`s, or because LC `proc`s are temporarily under-using their reserved resources. The two classes capture a common distinction in cloud tasks, between services that are some combination of long-running, stateful, and in need of performance guarantees (LC); and tasks that are some

combination of short-running, non-latency-critical, and in need of burst parallelism (BE).

Because σOS can create `procs` rapidly, it is suitable for serverless tasks. Because `procs` can communicate with each other, coordinate through the σOS name system, and can be long running, σOS is also suitable for stateful microservices, as well as offering this set of facilities to serverless tasks.

1.4 Envisioning an API for cloud applications

Cloud applications are different from traditional Linux applications in several important ways. They rely on cloud services instead of the local OS to spawn new computations, to access durable storage, and to coordinate. They need to communicate over the network with other applications under tight latency bounds, are collocated with mutually distrustful workloads, and some are ephemeral and need to start quickly.

Our goal when conceiving of σOS was to design a new API for cloud-native applications from scratch, without the baggage of backwards compatibility for existing Linux applications. Our view is that such an API should contain a small set of primitives which can be composed to support a wide range of applications with differing performance, fault-tolerance, and communication requirements. We take an end-to-end [101] approach, which requires applications to explicitly opt-in to fault-tolerance protocols so that only applications which need strong fault-tolerance guarantees from the platform infrastructure pay this cost.

The design of the σOS API reflects this view. The σOS API provides a single, generic unit of compute, `procs`, a high-performance communication technique which allows the platform to isolate tenants, σEPs, and per-tenant namespaces, realms, which allow the platform to provide `procs` with service discovery, access to durable storage, and fault-tolerant coordination primitives. `procs` interact with and configure the realm namespace with a Unix-like API to centralize or decentralize state in order to meet their performance and fault-tolerance needs.

1.5 Contributions

The main contributions of this thesis are:

- The design of σOS, a new multi-tenant distributed operating system that supports both long-running stateful executions and short-lived serverless tasks (§2.1).
- A σcontainer implementation of `procs` that provides strong isolation with quick start (§2.2.1).
- σEPs, a novel abstraction which provides low-overhead network communication between `procs` with strong security isolation (§2.1.2).
- Co-sandboxes, a new interface for application developers to specify application initialization to cloud platforms, enabling fast end-to-end cold-start (Chapter 3).
- An implementation of co-sandboxes in σOS (§3.4).

- An evaluation showing that σOS provides high throughput for both serverless and microservice workloads, that co-sandboxes reduce start latency for off-the-shelf and custom cloud applications, and that co-sandbox speedup is complementary to state-of-the-art cold-start speedup techniques (Chapter 4).

The σOS and co-sandbox source code is open-source and available at https://github.com/mit-pdos/sigmaos.

1.6 Related Publications

This work builds on work published in SOSP '24 [111], and in an ArXiv pre-print [112].

1.7 Roadmap

The remainder of this thesis is organized as follows. Chapter 2 describes the σOS API, `procs`, σEPs, and realms, as well as some implementation details of σOS. Chapter 3 describes the co-sandbox abstraction and design, and explores several case-studies for how cloud applications can speed up their end-to-end start times using co-sandboxes. Chapter 4 shows an evaluation of σOS. Chapter 5 discusses some take-aways and lessons learned in the design of σOS. Chapter 6 describes several extensions to σOS explored in a series of Master's theses written at MIT. Chapter 7 describes related work on which σOS builds and compares it to σOS, and Chapter 8 concludes.

Chapter 2

σOS: a cloud OS with a unified API

2.1 σOS: a cloud-centric OS

σOS's goal is to provide tenants with a single platform that supports both serverless and microservice tasks and to multiplex the workloads from different tenants on the provider's servers. σOS's API caters to cloud applications' need to launch computations, to communicate among those computations, and to share storage; it aims to provide interfaces well-enough tailored to the cloud that traditional local OS interfaces are not needed.

Figure 2.1 shows a simplified diagram of a σOS deployment, in which `procs` run on a machine and interact with σOS' global and per-machine services. The remainder of this section describes σOS and its interfaces from the perspective of the tenant, while the next section §2.2 describes how σOS is implemented on the provider's hardware.

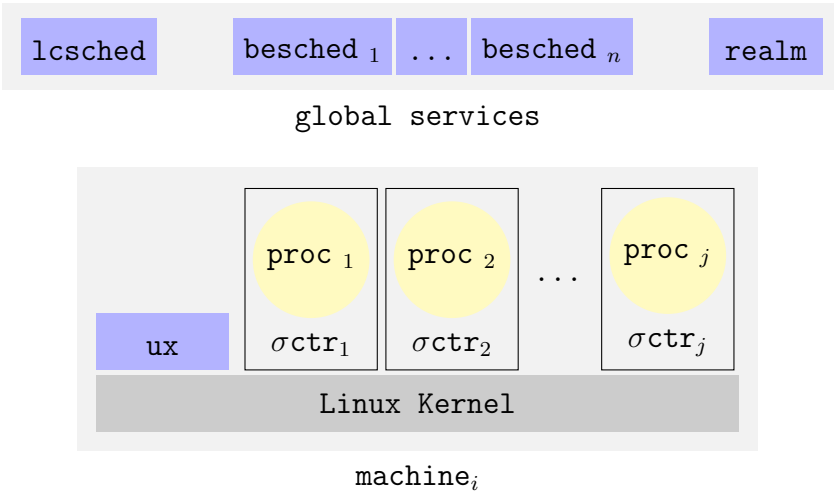

Figure 2.1: A σOS deployment runs several global services including a scheduler for LC `procs` (`lcsched`) and a sharded scheduler for BE `procs` (`besched`) (§2.2.3), and a realm for each tenant. On each machine, σOS runs each `proc` in a σcontainer and some per-machine services like `ux`, the σOS local file server which `procs` use for on-disk scratch space.

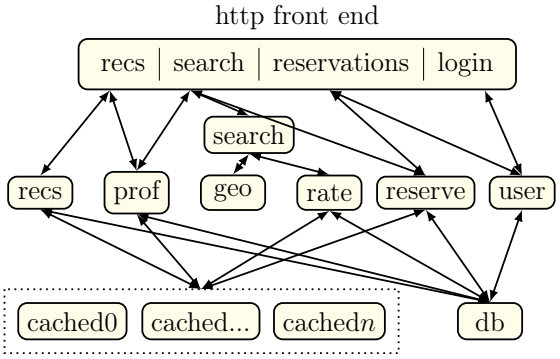

Figure 2.2: Each yellow box is a `proc` in the Hotel Web site. The cache service consists of one `proc` per shard.

2.1.1 `procs`: application execution units

Developers write their applications in terms of `procs`, which are executed by σOS. To illustrate how developers use `procs`, consider the following two running examples: `mr`, a MapReduce library, and `hotel`, a microservice application from DeathStarBench [52]. Both `mr` and `hotel` combine tasks that are best implemented as microservices with tasks that fit the serverless model.

The `mr` library creates a long-lived `proc` that coordinates the job. This coordinator spawns a serverless-style mapper `proc` for each shard of the input files and a reducer `proc` for each reducer bin. Current cloud offerings require developers to either use two platforms, a microservice platform to run the coordinator and a serverless platform to run the mappers and reducers, or run a coordinator together with a cluster of long-lived worker processes, which fails to take advantage of the elastic structure of MapReduce.

The `hotel` application creates a `proc` for the web front-end, a `proc` for each microservice, and a `proc` for each cache service shard. Figure 2.2 illustrates the `proc`-level organization of the `hotel` application. Some of the `hotel` microservices, like the sharded cache service and compute-intensive reservation service, benefit from the elasticity of serverless. Additionally, the `hotel` application may run periodic background jobs, such as image resizing or data analytics, which are a good fit for the serverless model. A developer who wishes to implement `hotel` on current cloud offerings would have to compose several cloud platforms to implement the different application tasks, and would be unable to make microservice tasks as elastic as serverless tasks.

In σOS, the tenant can use a single cloud platform and API to implement all tasks of `mr` and `hotel`. Long-lived microservices and short-running functions can both be implemented as `procs`. The tenant does not provision worker machines when implementing `mr` and `hotel`; σOS is in charge of choosing which of the provider's machines should run each `proc`.

Table 2.1 shows the interface with which σOS applications create and control `procs`. An application creates a new `proc` using `Spawn`, which returns a process identifier. The descriptor argument describes the desired attributes of the `proc`:

- the σOS name-system pathname of the binary to execute (§2.1.3).
- arguments to be passed to the `proc`.

Methods	Description
`Spawn(descriptor)`	Queue `proc`, return `pid`
`Kill(pid)`	Kill `proc` `pid`
`WaitStart(pid)`	Wait until `pid` starts
`WaitExit(pid)`	Wait until `pid` exits
`Started(pid)`	`pid` marks itself started
`Exited(pid, status)`	`pid` marks itself exited
`WaitKill(pid)`	`pid` wait for kill signal
`NewSigmaEP()` → `(Listener, SigmaEP)`	Create σEP
`Accept(Listener)` → `(Conn, SigmaEP)`	Accept connection
`Dial(SigmaEP)` → `Conn`	Connect σEP
`CloseEP(SigmaEP)` → `nil`	Close σEP
`Create(...)`, `Open(...)`, `Close(...)`, `Remove(...)`, `Rename(...)`, `Stat(...)`, `Read(...)`, `Write(...)`, `Lseek(...)`, `Watch(...)`	Access files in realm
`AbsPath(path)`	Resolve `~local`
`OpenWatch(dir, func)`	Watch for changes in dir

Table 2.1: Summary of the σOS interface.

- Optionally, the `proc`'s co-sandbox (§3.2) and its arguments.
- whether the `proc` is to be latency-critical (LC) or best-effort (BE), following Borg [118]. For example, a developer would likely specify BE for a map or reduce worker, and LC for a microservice `proc`.
- for LC `procs`, the amount of CPU power to reserve (typically chosen for peak expected load).
- for all `procs`, the amount of RAM to reserve.
- optionally, a failure domain for situations where `procs` should execute on independently-failing machines.

`Spawn` adds the `proc` request to a queue managed by the σOS scheduler (§2.2). This queue gives the scheduler a view of demand, allows it to limit active load by deferring the start of some `procs`, and allows it to make informed decisions about how to distribute `procs` over machines. The caller of `Spawn` can wait until the scheduler starts its child by calling `WaitStart`. For example, `hotel` spawns microservices as LC `procs` and waits until they are started before accepting client requests. A `proc` signals to its parent, using `Started`, that it has started running.

```
func (c *Coord) runProc(p *Proc) {
  for {
    SigmaOS.Spawn(p)
    exitStatus := SigmaOS.WaitExit(p.GetPid())
    if exitStatus = SUCCESS {
      break
    } else if exitStatus = ERROR {
      // Need to retry, so continue in loop
    }
  }
  c.procDone(p)
}

func (c *Coord) runMR(procs []*Proc) {
  for _, p := range procs {
    go c.runProc(ch, p)
  }
}
```

Figure 2.3: Simplified version of the code a MapReduce coordinator might use to start map or reduce `proc`s, wait for them, and re-start them on failure.

A parent `proc` can wait until its child finishes using `WaitExit`, which returns an exit status provided by the child's call to `Exited`. σOS itself may also call `Exited` on behalf of a child `proc` if a machine crashes or a partition makes the `proc` unreachable; `WaitExit` returns an error in this case. For example, the `mr` coordinator waits for its mappers and reducers, as illustrated in Figure 2.3, and uses the returned exit status to decide whether any `proc`s must be re-run due to failures.

2.1.2 σEPs: `proc` communication

σOS must provide `proc`s with high-performance network communication and must prohibit different tenants' `proc`s from interfering with each other. σOS does so using a novel abstraction: σEPs. The σEP API, shown in Table 2.1, allows σOS to mediate connection setup, so that it can ensure that communication only occurs within each tenant.

When a server `proc` wishes to create an endpoint for incoming network traffic from other `proc`s it calls `NewSigmaEP`, which requests σOS to create an σEP and a `Listener`. The σEP is an opaque shareable token identifying the new endpoint. The server `proc` calls `Accept` with the `Listener` to wait for incoming connections. Any `proc` of the same tenant can use the σEP to connect to the server by calling `Dial`. `Dial` returns a connection which can be used to exchange messages directly with the server `proc`.

An σEP is usable by any `proc` in the same realm. σEPs are assigned by σOS, however, so server `proc`s must use the facilities in §2.1.3 to publish σEPs under well-known names.

```
lis, ep := SigmaOS.NewSigmaEP()
SigmaOS.Write("/s3/s3srv_3", ep.Marshal())
for true {
  conn := SigmaOS.Accept(lis)
  go HandleClientConn(conn)
}
```

Figure 2.4: Server code to create a σEP and register it under a name that clients can connect to.

```
b := SigmaOS.Read("/s3/s3srv_3")
ep := UnmarshalSigmaEP(b)
conn := SigmaOS.Dial(ep)
SendMsgToServer(conn)
```

Figure 2.5: Client code to retrieve a σEP given its name, and connect to the listening server.

2.1.3 Realms: per-tenant global name spaces

σEPs provide `procs` a low-level mechanism to create servers and establish connections. However, `procs` need a naming system in order to exchange σEPs, interact with each other, and share data. σOS supports this interaction with a per-tenant name space called a *realm*. A `proc` uses the σOS API to discover and access resources in a realm using pathnames (inspired by Plan 9 [93]). A `proc` in one realm cannot name or access σOS resources in other realms.

The root of a realm's file system is hosted by a name server called `named`, implemented using `etcd` [42], a Raft-replicated [89] persistent key/value store. σOS-provided services and `procs` extend the name space by hosting subtrees. As shown in Figure 2.4, a server which wishes to register a subtree creates a file in the name space containing its σEP object. When a `proc` traverses the name space and reaches the σEP link file, it will transparently connect to the hosting server using the σEP API and continue its traversal by communicating directly with the host of the subtree.

In this way, the logically global realm name space enables transparent access to a distributed set of resources, including filesystem-like services, named RPC services, proxies to storage systems, and storage for small configuration files. The namespace acts as a rendezvous so that `procs`, which are not directly aware of where they or other `procs` are physically executing, can find each other when needed. Furthermore, since a given pathname has the same meaning to all of a realm's `procs`, `procs` can directly exchange pathnames.

An example is σOS's `s3` proxy service, which exposes a tenant's Amazon S3 buckets and keys in the tenant's name space. σOS stores binaries for procs in the σOS S3 bucket and reads/writes them using pathnames (e.g., `/s3/sigmaos/mr-mapper`).

To help applications spread load, a directory can list σEP link files for multiple proxy servers: `/s3/` can contain `s3srv_0`, `s3srv_1`, etc. Then a pathname with `~local` asks σOS to find a proxy on the same machine. Now, `mr` mappers can read input files (e.g., books from the S3 bucket `gutenberg`) using `/s3/~local/gutenberg/pg-being_ernest.txt` in parallel

through local s3 proxies and avoid copying each input file twice over the network. The `~local` pathname component resolves a fundamental tension in the realm namespace between locality and location-obliviousness. `procs` which want to access local resources without having to know which machine they are physically running on can include `~local` in the pathnames they access.

As another example of the use of `~local`, σOS exposes per-realm scratch space on individual machines' local file systems under the pathname `/ux/<machine-name>/`; a `proc` can read and write scratch files on both its own machine and other machines with such pathnames. A `proc` that wishes to share a file in its local scratch space (e.g., `/ux/~local/a.txt`) with other `procs` first resolves the pathname to an absolute realm-global pathname with the `AbsPath(pathname)` API call, which translates `/ux/~local/...` to a global `/ux/<machine-name>` name. This ability to transparently access remote storage helps some data-intensive applications such as MapReduce [70, 80, 94, 108]. Uniform access via pathnames makes it straightforward for developers to choose between local and remote (e.g., S3) storage.

2.1.4 `procs` and failures

Many virtual machine and container systems re-start instances after a failure, which requires time-consuming persisting of configuration information at creation time. σOS does not restart `procs` in response to failure, and the σOS scheduling layer does not persist `proc` descriptors, which helps decrease creation cost. If a σOS scheduling component crashes while involved in spawning a `proc`, the spawn request may be lost. σOS guarantees that if such a failure prevented a `proc` from being spawned, `WaitStart`/`WaitExit` will return an error; the requesting `proc` can then re-issue the `Spawn` if appropriate.

σOS provides mechanisms to support fault-tolerant applications. A realm's `named` provides fault-tolerant storage, backed by `etcd`. `procs` can achieve fault-tolerance by storing critical state in `named`; if such a `proc` fails, another can be started (or already be waiting) to take over, and read the latest state from `named`. `named` uses `etcd`'s support for leader election to help fault-tolerant `procs` avoid split-brain (more than one `proc` thinking it is the active leader).

As an example, the `mr` library creates three coordinator `procs`. One is elected as the leader, and the other two are standbys. The leader stores its progress (e.g., which mappers have completed) in the directory `/mr/coord/`. If the leader crashes, one of the standbys takes over and picks up from where the crashed coordinator left off.

To survive a system-wide failure (e.g., all machines crash), σOS provides `initd`. An application can register a `proc` with `initd`; when σOS reboots, it restarts registered `procs`. For example, the `mr` library registers the coordinator `proc` with `initd`. `initd` stores its state persistently in `named`.

2.1.5 Reflecting on the σOS API design

This section describes why we think `procs`, σEPs, and realms, make a good API to support modern cloud applications.

`procs`. Applications which need access to additional compute resources and burst-parallelism need an API to request resources from the platform, and specify computations to run on those resources. Additionally, the application needs to monitor these computations to wait for their completion, handle failures, and kill them when they are no longer needed.

`procs` are different from Linux processes in several ways which makes their API different from Linux's process API. `procs` exist in a multi-machine context, so their process identifiers (PIDs) need to have cluster-wide significance. `procs` also need to be scheduled to a machine, and may need to be queued if no machine has sufficient resources to run them. This requires that parent `procs` specify the resource reservations their child `procs` need so that the σOS infrastructure can run admission control when scheduling a newly `Spawn`-ed `proc`.

The σOS infrastructure needs to provide fair allocation of resources to `procs` and realms across machines, and maintaining fairness requires that σOS dynamically adjust `procs` reservations or evict `procs` as new realms start their applications.

`procs` do not get a full Linux environment, because providing a full Linux image with strong isolation triggers several scalability bottlenecks in the Linux kernel. For some applications, this may be a non-trivial limitation.

`procs` of the same tenant may be scheduled to different machines, which gives the provider the flexibility to best use their cluster's resources. As such, `procs` need a way to communicate and to coordinate irrespective of whether or not they are running on the same machine, and need to handle failures and network partitions. σOS supports this via σEPs and realms, respectively.

σEPs. In order to communicate, client `procs` need to establish connections to servers, and servers need to accept them. It is the platform's responsibility to isolate communication between tenants.

We opt for logical endpoint names, σEPs, over IP addresses in σOS. σEPs allow the platform software more flexibility in assigning physical ports to applications, simplify migrating computations across machines, and allow the platform to enforce more fine-grained and flexible security policies. For example, σEPs can serve as an exchangeable capability-style handle. `procs` can delegate the ability to establish connections to server `procs` by passing them the σEP corresponding to a server `proc`'s endpoint (for example, via the realm).

Realms. Our view is that service discovery, fault-tolerant coordination primitives like leader election and leases, and access to external services like durable storage, should be services offered by the provider. σOS's per-tenant realms provide these facilities.

The tree structure of the realm namespace makes it a good fit for a cloud setting. Pathnames can encode hierarchy which matches the realities of physical hardware. For example, directories can contain services hosted on the same host or in the same rack. Directories are also an intuitive way to represent logically sharded services: each shard server can post its σEP in a directory, and clients can read the directory to select and connect to a shard. Unlike traditional DNS, which uses TTLs, the realm namespace also provides strong consistency using leases and leader election. Clients benefit from this consistency to quickly learn of updates to the set of available shard servers. This enables clients to reshuffle their load on short notice when a sharded service scales up or down.

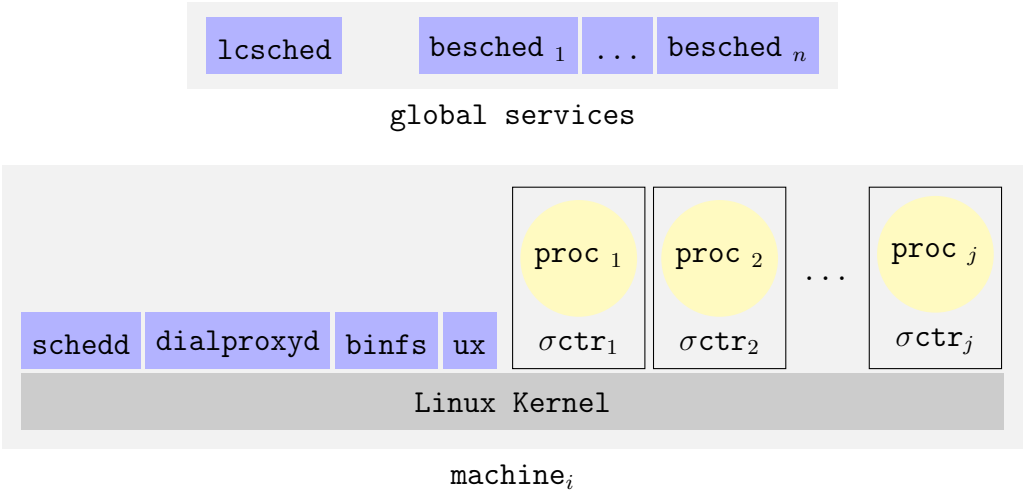

Figure 2.6: On each of a provider's machines, σOS runs a `schedd` (for creating σcontainers), `dialproxyd` (for mediating network connection setup), a `binfs` (for demand-paging `proc` binaries), and a `ux` (for exposing local storage). For distributed scheduling, each machine's `schedd` cooperates with one global `lcsched` for scheduling LC `procs` and several `besched`s for scheduling BE `procs`. Not shown are provider `procs` that run in each realm (e.g., `named`, `s3`).

The tree structure also provides a natural way to decentralize the namespace: subtrees can be hosted by different servers. This feature is important to provide good performance when some services or application components are placed across high-latency boundaries (e.g., in another datacenter), where centralized consistent state would make performance intractable. Using pathnames, σOS applications can choose to make these boundaries transparent by recursively resolving a full path, or opaque, by only traversing decentralized subtrees explicitly.

2.2 Implementing the σOS API

Implementing the σOS API poses three challenges: how to both isolate `procs` strongly and start them quickly; how to communicate efficiently with σEPs; and how to schedule BE and LC procs. σOS addresses these challenges using the components shown in Figure 2.6. `lcsched` and `besched` are global schedulers which place `procs` on machines. Each machine runs a local scheduler agent (`schedd`), a network isolation agent (`dialproxyd`), a `proc` binary server (`binfs`), and a server providing temporary local storage (`ux`).

σOS is implemented on Linux, which allows σOS to benefit from Linux's debugging tools and well-tested isolation primitives. This section describes how σOS uses Linux primitives to implement the σOS components.

2.2.1 Isolating `procs` with light-weight σcontainers

Each `proc` must be isolated to prevent it from disturbing provider infrastructure, other realms' `procs`, and (except as allowed by the σOS API) `procs` in the same realm. σOS isolates each

`proc` with a σcontainer: a dedicated computing environment with low initialization overhead. What allows σcontainers to be light-weight is the fact that `proc`s use only a small subset of the underlying operating system's facilities, relying instead on σOS' cloud-centric API. This allows σOS to avoid many expensive steps that traditional container systems must take to provide a full, private Linux environment. For example, σOS does not set up a network namespace, which would take around a hundred milliseconds, nor does it create an overlay file system, which would take around 5 milliseconds (plus the time to install files in the overlay file system). Instead, `proc`s use the σEP API to access the network with σOS-enforced isolation, and all `proc`s share a read-only generic file system image. σcontainers provide `proc`s with isolation as good as traditional containers but with faster start times.

Creating `proc`s. An executing `proc` consists of a Linux process within an isolating σcontainer started by `schedd` with the following steps:

- **Namespaces:** `schedd` gives the `proc` private Linux namespaces for UTS, IPC, and PIDs. Because a `proc` doesn't need its own IP address (it uses σEPs and connections established via `dialproxyd`), the σcontainer doesn't include a network namespace and overlay network. Further, because `proc`s that need temporary local storage talk to the local `ux` server rather than making direct filesystem system calls, the σcontainer doesn't include an overlay file system.
- **Jail:** `schedd` jails the `proc`'s Linux process in a file system with just a few read-only configuration files and a few `/proc` pseudo-files. It mounts σOS's `binfs` read-only on `/mnt/binfs`. `binfs` is a FUSE [11] server through which the Linux kernel demand-pages the `proc`'s binary. `proc`s' file system jail is generic, so `proc`s need to be deployed as statically-compiled binaries or dynamically load their dependencies from `binfs`.
- **Seccomp:** `schedd` uses a seccomp filter to allow only system calls that allocate memory, create and manage threads, handle a few signals, access randomness, and manage timers. These calls are needed by the Go and Rust runtimes. The filter forbids networking system calls such as `socket`, `connect`, `bind`, `accept`, and `listen`, since connection setup occurs via `dialproxyd`.
- **AppArmor:** `schedd` drops all Linux capabilities and further restricts file system access using an AppArmor profile. This profile denies many uses of signals, and forbids access to directories in `/proc` other than the `proc`'s own `/proc` directory.
- **`cgroups`:** `schedd` provides performance isolation by assigning the σcontainer to a `cgroup`. `schedd` pre-creates and manages a pool of `cgroups` to move `cgroup` creation off the `proc` start path. `schedd` uses one `cgroup` to isolate each realm's BE `proc`s, and another for each realm's LC `proc`s. §2.2.3 describes how σOS configures `cgroups` to enforce resource reservations and utilize idle resources.

After this setup, the σcontainer calls `exec` with the path `/mnt/binfs/<binary-name>`. Linux demand-pages the binary via FUSE and `binfs`, allowing the `proc` to start quickly. The first time a `proc` binary runs in a σOS cluster, `binfs` reads its pages from S3. `binfs` caches these pages on the local disk, to speed future invocations of the same binary. The former situation is "cold start"; the latter "warm start". σOS tracks where binaries have recently run so that `binfs` can take advantage of cached binaries on other machines in the σOS cluster.

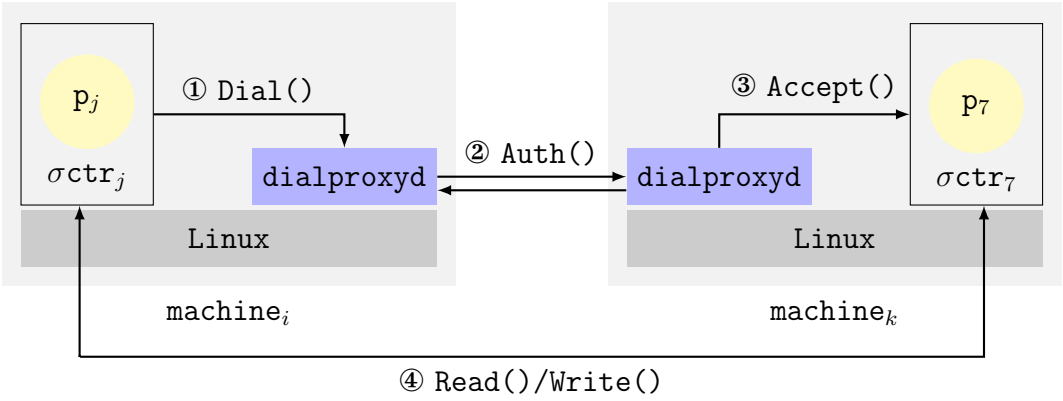

Figure 2.7: **1)** When a client `proc` `Dial`s a server `proc`, **2)** `dialproxyd`s on each machine collaborate to enforce isolation and set up the inter-`proc` connection (`Auth`). If the connection is allowed, **3)** the server `proc`'s `Accept` call returns, and the connection is established. **4)** From this point on, the `proc`s communicate directly using `Read` and `Write`, as they would with any other Linux socket.

σcontainer isolation. σOS restricts a `proc` to 69 system calls; the other 307 [77] are forbidden. For comparison, Docker's seccomp filter allows containers to use 352 system calls [40]; Kubernetes allows 340 [33]. Gvisor, which reimplements much of Linux in user space to reduce the number of host system calls a container requires, allows 55 system calls [126].

σcontainer startup time. A σcontainer is quicker to start than a traditional container because it doesn't unpack a container image, set up a network namespace (σOS uses σEPs), or create an overlay file system (`proc`s cannot directly create files on the local host). Additionally, σOS performs some expensive operations in advance of running any `proc` (e.g., creating a pool of `cgroups`). Finally, σOS leverages Linux's demand paging via `binfs` to allow the `proc` to start without fetching the entire binary in advance.

2.2.2 σOS endpoints

σEPs allow σOS to control the setup of network connections among `proc`s without the expense of setting up a virtual IP namespace for each `proc`. This allows σOS to preserve two important uses of network namespaces, namely port-collision avoidance and network isolation, without encountering the scalability bottleneck in Linux's network namespace creation path [88]. In order to avoid port collisions, the σEP API shifts port allocation into σOS instead of letting the server `proc`s choose the ports themselves. In order to implement fine-grained network isolation policies, the σEP API ensures connections are established via a per-machine σOS daemon: `dialproxyd`.

As shown in Figure 2.7, `dialproxyd` mediates all connection setup, both server-side and client-side, and verifies that each leads to a `proc` in the same realm. Once `dialproxyd` establishes and verifies a connection, it passes it to the `proc` with UNIX-domain message passing, and the `proc` directly reads and writes bytes on the socket. A σEP functions as a

network address with global significance that procs can use without needing to know about host names, IP addresses, or port numbers.

Communication between procs. A typical scenario is that a proc offering a service creates a new σEP by calling NewSigmaEP, writes that σEP into a file in the named namespace, and waits for incoming connections with Accept. Clients find the relevant σEP from named and call Dial.

Both Accept and Dial are IPCs to the local dialproxyd, which makes the required kernel TCP socket calls. Initially the new connection connects the two local dialproxyds, which interact to verify that the two procs' realms are the same, and then passes the connection file descriptors to the procs via UNIX-domain message passing. After that, the two procs can communicate directly over the TCP connection without further involvement by dialproxyd.

Outgoing communication with external services. dialproxyd allows procs to connect to entities outside of σOS by creating and connecting to special *external* σEPs.

A malicious proc may try to create an external σEP that refers to a server proc in another realm. To prevent this, we expect the provider to deploy σOS in a private network with a known range of IP addresses. dialproxyd inspects external σEPs passed to it during connection setup to ensure that the IP addresses they contain lie outside its private range[1].

Incoming communication from external services. procs may need to accept connections from entities outside of σOS, such as HTTP clients. As is standard practice in Kubernetes and other container systems, we expect the client to configure a provider-managed IP endpoint and load-balancer to proxy connections to procs. The load-balancer would speak the dialproxyd identity verification protocol to ensure that external connections are delivered to the intended realm.

2.2.3 Scheduling procs

From the provider's perspective, σOS's job is to decide the placement of spawned procs onto the provider's machines and, within each machine, to decide how to allocate CPU time and memory to running procs.

Placement. LC and BE procs have different placement requirements. LC procs must receive their guaranteed CPU and RAM, so σOS must take care that the sum of LC reservations on each machine is less than capacity. BE procs should use unreserved CPU and RAM, as well as reserved LC CPU currently left idle, though BE procs cannot be allowed to use RAM reserved by LC procs even if currently idle. Realms that have BE work should get roughly equal shares of total unreserved provider CPU, though a real deployment would likely use a different policy (e.g., based on resource pricing).

[1] An alternate design implemented in σOS, which does not rely on IP address verification, has the client's and server's dialproxyd exchange a cryptographically authenticated setup message to verify that connections internal to σOS are initiated via dialproxyd.

σOS places LC and BE procs with different mechanisms. When called for an LC proc, Spawn sends the descriptor to the provider's lcsched, a single provider-wide service. lcsched tracks, for each of the provider's machines, how much CPU power and RAM are currently reserved for existing LC procs on that machine. When a Spawn request arrives, lcsched chooses a machine with sufficient unreserved resources (if one exists) and forwards the request to the schedd on that machine; otherwise lcsched queues the proc request until a machine becomes available.

Because BE procs may be created at a high rate, σOS shards the work of BE placement over a set of besched servers running on a subset of the provider's machines. For a BE proc, Spawn sends the descriptor to a randomly selected besched server. That besched adds the descriptor to a private queue of procs waiting to run. Meanwhile, the schedd on each of the provider's machines monitors the machine's idle CPU time and idle RAM (less RAM reserved by LC procs), and if there are significant spare resources, sends a request to a randomly selected besched. That besched looks for a queued proc descriptor with a compatible memory request, giving each realm an equal chance. If the besched has nothing relevant queued, the machine's schedd asks a different besched.

Enforcement. schedd uses Linux cgroups to reserve CPU and memory for LC procs, and a combination of the cgroups network classifier and Linux's Traffic Control tc to prioritize LC procs' network traffic. schedd configures cgroups so that BE procs receive equal fractions of CPU reservations left idle by LC procs.

2.2.4 σOS prototype details

Most of σOS is written in Go; Table 2.2 shows each component's lines of code. schedd uses a trampoline program written in Rust, exec-uproc, which sets up an isolated σcontainer before exec-ing a new proc.

proc executables are statically-linked ELF files. Static linking results in larger binaries than dynamic linking, but allows the root file system to be generic, since it doesn't have to provide proc-specific files such as application shared libraries.

procs use the σOS client libraries to interact with the realm, other procs, and σOS services, as well as for common fault-tolerance primitives like leases and leader election.

2.2.5 σOS language support

σOS supports procs written in several languages in addition to Go, including C++, Rust, and languages which compile to WASM. In order to support these procs, σOS runs a sigmaproxyd daemon, written in Go, on each σOS node. The sigmaproxyd puts a pipe in each proc's mount namespace, and forwards σOS API calls sent over the pipe to other services in the proc's realm. This allows reuse of the σOS Golang client libraries, including the RPC and networking stack, across other languages.

σOS also supports interpreted languages like Python, which import modules dynamically at runtime, using the σOS API to download their dependencies. The developer provides a statically-linked version of the interpreter of their choice (e.g., CPython) as the proc binary,

	Component	LOC
Core	proc/σcontainer	5,335
	`lcsched besched schedd`	5,036
	net/σEP	2,117
	file API	5,694
	realm: `named realmd`	4,042
	`s3 ux db`	8738
	boot	1,114
Libraries	`client`	15,669
	`server`	7,370
Applications	`cached`	2,444
	`vecdb`	662
	`etcd`	582
	`hotel`	3,333
	`imgrec`	693
	`imgprocess`	1,085
	`memcached`	678
	`mr`	3,179
	SeBS	929
	`socialnet`	3,332
Total		72,032

Table 2.2: Lines of Go code for σOS's components (excluding `etcd` itself, `etcd`'s Raft [43], and `protoc`-generated files for protocol buffers).

and σOS provides a shim that intercepts the interpreter's accesses to module files and loads them via the σOS interface.

2.2.6 Support for additional isolation backends

In order to run some experiments, we added several additional `proc` isolation backends to σOS. In particular, σOS can optionally isolate each `proc` in a GVisor [54] container, a Junction [49] VM, or a snapshotted/restored Spice [61] VM. Together with a small compatibility layer which registers σEPs in the namespace and executes the `proc` binary, this allows σOS to run some microservices which use more of Linux than σcontainers provide, such as `etcd` and `memcached`, unmodified as `proc`s.

2.3 σOS applications

Table 2.2 lists the lines of code for each application. `hotel` (the running microservice example) and `socialnet` are ports of the corresponding DeathStarBench applications originally written

for Kubernetes. `cache` is a sharded in-memory cache, much like `memcached`, used in `hotel` and `socialnet`. `mr` is the running example of the MapReduce library. `imgprocess` is an image processing service that spawns an LC coordinator `proc` to manage resizing images; for each image, it spawns a BE `proc` that stores its results in S3. The coordinator handles failures of `imgprocess` `proc`s.

To demonstrate that σOS is complete enough to build fault-tolerant microservices, we built `kv`; it implements a linearizable, sharded, fault-tolerant in-memory key-value service. `kv` has groups of three `proc`s; each group replicates its shards using `etcd`'s Raft library [43]. `kv`'s balancer can add a new group in response to changes in load; it moves shards between groups by spawning a "mover" `proc` for each shard with keys that must be moved. The balancer uses a realm's `named` to store the configuration for each epoch so that client `proc`s can look up which group they should contact for a given shard. Clients set a watch on this configuration file to be alerted of a new configuration. The balancer itself has two standby `proc`s, which will elect one as the new balancer if the current balancer fails.

Porting applications to σOS. σOS does not provide out-of-the-box backwards compatibility for existing serverless or microservice applications. However, in our experience, porting an application to σOS does not require major application rewriting; most σOS ports involve switching some function calls in the existing codebase with analogous functions from the σOS API. Furthermore, many applications use higher-level libraries to, for example, establish network connections. These libraries already provide a degree of portability between operating systems by hiding the underlying syscalls required to achieve their functionality. Applications which use these libraries can often be ported to σOS by changing these libraries' implementations without changing the libraries' application-visible API.

σOS supports cloud APIs and access to databases like MongoDB well via proxies (e.g., `s3`, `db`) and σEPs. Additionally, container- and serverless-structured applications are already broken up into execution units which map naturally onto `proc`s.

σOS and σEP APIs (Table 2.1) provide replacements for most APIs that cloud applications use. σOS applications Read/Write files in the realm instead of using the local file system or the S3 Get/Put API, spawn `proc`s instead of forking Linux processes or Lambdas, and access the network using σEP APIs like `NewSigmaEP` (analogous to `listen`) and `Dial`. The `NewSigmaEP` and `Dial` APIs are similar to those used to start servers and initiate network connections in Go and other high-level languages.

σOS developers can port web server front-ends like Nginx to σOS with σEPs (§2.2.2). σOS enables patterns similar to those in Kubernetes, in which web servers register public services with cloud provider load-balancers [57].

We ported `hotel` and `socialnet` from DeathStarBench, and most changes were "find-and-replace-all" operations. σOS versions use the realm for service discovery instead of a service registry and call `NewSigmaEP` instead of `listen` to start a server and accept connections. σOS's RPC library, implemented with σEPs, replicates the core of gRPC.

Applications that use a wide range of Linux system calls (e.g., shared memory) are harder to port to σOS. In our experience, most cloud applications don't do this. They rely on cloud APIs and are organized into containers and serverless functions.

Discussion. There are two observations from the experience of implementing this set of applications. First, the applications can be implemented using just the σOS API. One reason is that the applications are cloud-centric and use few local OS services. Another is that σOS allows a provider to export services that applications need through proxies (e.g., `ux`, `s3`, `db`, etc.), which can be accessed using the σOS API. A final reason is that today's programming languages like Go come with a large ecosystem of portable packages that allow application code to avoid much reliance on the local operating system.

The second observation is that σOS can support both microservice and serverless-style applications in a single framework. Furthermore, some applications combine both LC and BE `procs`. For example, in `kv`, the caching `procs` are LC while the mover `procs`, which copy shards between groups, are BE.

Chapter 3

Co-sandboxes: a `Spawn` interface for fast end-to-end cold-start

Latency-sensitive user-facing websites written as serverless applications are now common [4, 27, 29], and new user-facing applications like Large Language Models (LLMs) and coding agents often invoke serverless function tools and Model Context Protocol (MCP) servers to construct their responses [5, 28, 31]. Many of these serverless functions are short-running and experience frequent cold-starts because tool call frameworks create fresh instances of each function in order to isolate different users' requests. As a result, start latency often dominates their execution latency [2, 41, 67, 74, 105, 124].

Definition of cold-start latency. Traditionally, cold-start latency is defined as the time it takes a fresh application instance to reach line 1 of `main` on a new machine. However, applications must execute several additional steps before they can start doing useful work. A more *end-to-end* definition of cold-start latency, which spans from the point an application is spawned until it begins processing work, better captures the application developer's concerns. Ultimately, this broader definition of start latency more accurately reflects the serverless infrastructure's overhead on the application's Service Level Objectives (SLOs). The goal of co-sandboxes is to accelerate end-to-end cold-start latency.

Start latency can be broken down into two components which we call *setup* and *initialization*, shown in part **a)** of Figure 3.1. Setup involves steps the platform takes before the application starts running. Setup is application-agnostic. Examples include downloading the application binary, creating an isolated execution environment, and starting the application runtime. Initialization involves steps the application takes after it is running, but which must complete before it can do useful work. Initialization is application-specific. For example, a serverless function may connect to a message queueing service to claim a task to process, or a new microservice instance may query a load-balancer for its shard assignment and download its shards from remote storage. Real-world traces indicate that setup [67, 111, 124] and initialization [21, 123] are comparable in length for many real-world applications: each takes O(100ms).

State-of-the-art. Prior work on reducing cloud application start time mostly focuses on reducing setup costs. Work on fast setup centers around lightweight isolation, specialized

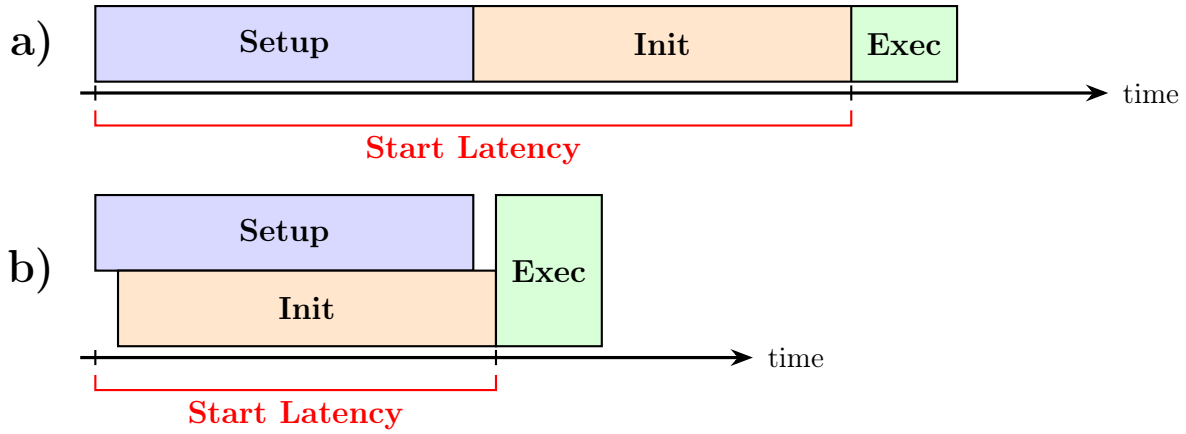

Figure 3.1: Cloud application start timeline, **a)** without co-sandboxes, and **b)** with co-sandboxes.

hardware and kernel modifications for fast binary downloads, and forking or snapshot-restore mechanisms [1, 8, 21, 65, 72, 106, 111, 124]. σOS reduces some components of setup time, namely container creation, through the design of the unified σOS API which can be isolated in lightweight σcontainers.

Some work has explored reducing initialization costs [37, 72, 109]. These systems push some simple operations into the platform with APIs that allow developers to statically declare application inputs and allow the platform to prefetch state on behalf of the application. Pushing initialization work into the platform in this way reduces initialization latency, but prior approaches are insufficiently expressive to capture the full gamut of application initialization behavior. For example, APIs which require developers to statically declare function inputs do not enable input prefetching when the input can only be determined at runtime, or fetching the input requires exchanging RPCs with specialized storage services.

Co-sandboxes. This thesis proposes a missing programmatic interface for developers to specify application initialization to the cloud platform: *co-sandboxes*. Co-sandboxes allow developers to write initialization programs which the platform can run on their behalf. Platforms can run co-sandboxes in parallel with the setup phase to *overlap* setup and initialization and reduce end-to-end start latency, as shown Figure 3.1 part **b)**. Once setup completes, the co-sandbox bootstraps the application by transferring initialization results to it. Co-sandboxes provide developers with a scriptable interface which allows developers to express a wide range of application initialization steps. For example, co-sandboxes can establish connections to other services, issue RPCs and fetch state which can only be identified at runtime.

Challenges. Co-sandboxes must start with low latency in order to maximize the amount of overlap with setup. A key design challenge (challenge 1) in fast co-sandbox start is keeping co-sandboxes small. Small co-sandboxes can be downloaded quickly onto the new application instance's machine, and can overlap with a larger fraction of the application container's setup to do more useful work. Another challenge (challenge 2) is efficiently transferring

initialization results to the application. The co-sandbox needs to transfer established network connections to the application. Additionally, applications which load a significant amount of state need to access the state with low overhead. A final challenge (challenge 3) is designing the co-sandbox API to make it easy to speed up start times for existing cloud applications with few modifications.

Approach. Co-sandboxes are implemented as WASM modules, which start in tens of microseconds, allowing the platform to run them quickly with strong isolation. To resolve challenge 1, co-sandbox binaries are kept small, around 100KB, because the co-sandbox host API is carefully designed to implement primitives which can be composed to run common initialization steps, namely establishing network connections and issuing RPCs. The high-level RPC interface allows the platform to implement an asynchronous RPC stack for the co-sandbox. This moves the RPC stack code segments out of each co-sandbox and into the shared platform infrastructure in order to reduce co-sandbox size.

To resolve challenge 2, the co-sandbox API is designed to enable efficient transfer of initialization results to the application with a socket transfer and message-passing API. Application developers use this API to hand off established connections to the application, and to move data between the co-sandbox and application container. The API's design enables applications to claim and use connections established by the co-sandbox, and to read initialization state via shared memory.

Finally, to resolve challenge 3, the co-sandbox API is designed to support off-the-shelf cloud applications with few-to-no modifications. RPC client libraries can support co-sandbox-accelerated starts transparently to the applications which depend on them. Using this approach, we used co-sandboxes to speed up start latency of several of the SeBS [34] Python functions without any application code changes, and accelerate startup of two popular open-source microservices, `memcached` and `etcd`, with a thin compatibility layer.

3.1 Overlapping setup and initialization

In order to concretize the steps involved in setup and initialization we examine `imgrec`, a serverless image recognition application similar to the image recognition application in SeBS [34]. We compare a WASM implementation (`imgrec-wasm`) which can be isolated quickly but has a large code package, and a Python implementation (`imgrec-py`), which has a small code package but suffers from slow isolation.

In order to initialize, `imgrec-wasm` and `imgrec-py` connect to Amazon's Simple Queue Service (SQS) [7], a durable message queue built for serverless applications, and claim a task. They then download their model weights and the task's input, perform inference, and store the result in S3. Table 3.1 shows the order-of-magnitude cost of each setup and initialization step of `imgrec-wasm` and `imgrec-py`. Despite having very different runtime and isolation costs, both `imgrec-py` and `imgrec-wasm` spend O(100ms) in both setup and initialization. This suggests an opportunity to overlap setup and initialization steps in order to speed up both implementations' end-to-end start times.

Phase	Step	WASM	Python
Setup	Function download	O(100ms)	O(1ms)
	Isolation	O(1ms)	O(100ms)
	Lang runtime init	O(1ms)	O(100ms)
	Fetch Dependencies	—	O(100ms)
Init	SQS GetTask RPC	O(10ms)	O(10ms)
	Connect to S3	O(1ms)	O(1ms)
	Download inputs	O(100ms)	O(100ms)
	Load state	O(10ms)	O(10ms)

Table 3.1: Setup and initialization steps of an Image Recognition serverless application `imgrec`, when implemented in WASM (`imgrec-wasm`) and Python (`imgrec-py`).

Problem. Modern cloud applications combine initialization with the remainder of the application in a single package. As such, cloud platforms must run application setup and initialization sequentially: initialization can only begin once the full application has started running. During cold-starts, unavoidable steps like binary downloads and starting the application runtime push back the initialization phase by hundreds of milliseconds.

Goal. Our goal is to separate initialization from the application so that the platform can overlap setup and initialization, and transfer the initialization to the application once it has started running. In order to enable separation of initialization from the rest of the application, the platform must provide a *scriptable* API which is sufficiently expressive to capture a common set of cloud application initialization steps like establishing connections and executing RPCs. For example, in `imgrec` later RPCs (e.g., the input download) depend on the results of earlier ones (e.g., the claimed task). Furthermore, `imgrec` knows which connections to establish only at runtime: a new `imgrec` instance only discovers the output destination once it has successfully claimed a task from SQS. The following section describes how we achieve this goal using co-sandboxes.

3.2 Co-sandbox Design

Co-sandboxes are WASM modules run by `WASMEngine`, a per-machine daemon deployed by the cloud platform (§3.2.1). Developers write co-sandboxes with the co-sandbox API (§3.2.2), which is designed to keep co-sandboxes small so they can be downloaded and start fast. The co-sandbox result transfer API (§3.2.3) enables co-sandboxes to efficiently bootstrap the application once it has started running. Its design enables existing applications to use co-sandboxes with few modifications. This section also discusses the co-sandbox developer workflow (§3.2.4), and changes to APIs used to start cloud applications (§3.2.5).

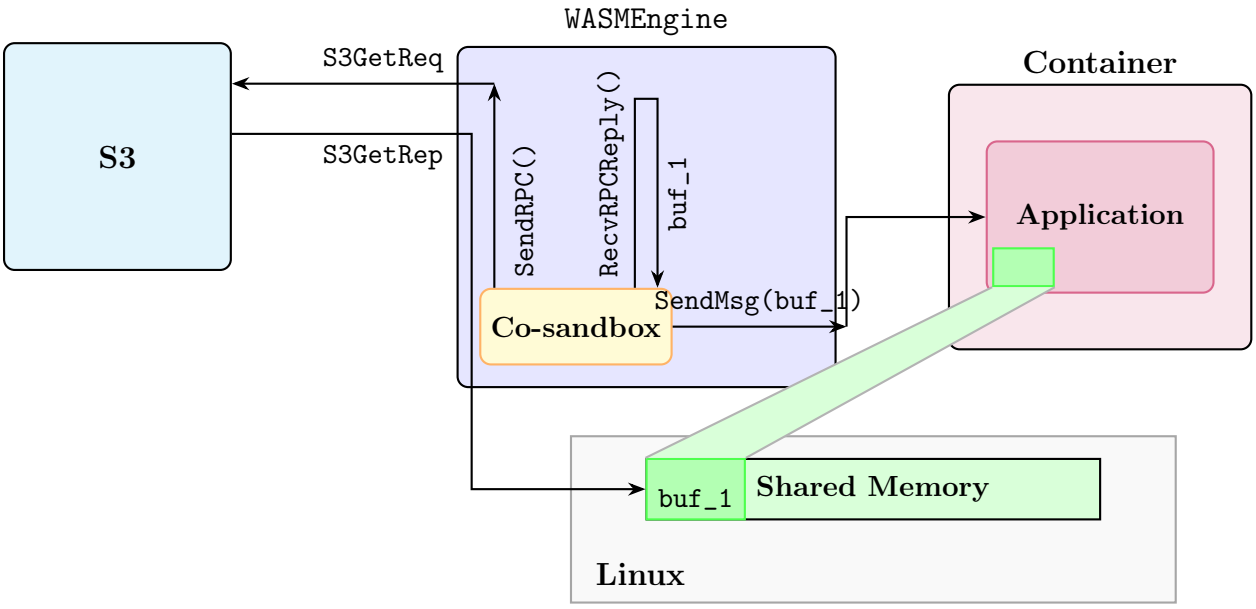

Figure 3.2: A co-sandbox fetches initialization state and communicates it to the application via the `WASMEngine`. The co-sandbox sends an RPC requesting the application's initialization state to S3 via `WASMEngine` using `SendRPC`, and blocks until the reply comes back using `RecvRPCReply`. `WASMEngine` writes the response from S3 directly into a shared memory region, and sends the buffer `buf_1` containing the reply to the co-sandbox. The co-sandbox forwards `buf_1` to the application, which fetches the state by mapping the shared memory region and reading `buf_1` directly.

3.2.1 Co-sandboxes: scriptable WASM init routines

Multi-tenant cloud platforms which support co-sandboxes need to start them quickly and with strong isolation. In order to make this possible co-sandboxes are implemented as WASM modules executed by `WASMEngine`, a per-machine daemon run by the platform. WASM modules, and by extension co-sandboxes, don't have access to a full Linux environment and have no standard syscalls by default. It is up to the platform, in this case `WASMEngine`, to define and implement an interface for co-sandboxes to communicate with the outside world and carry out initialization steps. Figure 3.2 illustrates how a co-sandbox uses the co-sandbox API to interact with `WASMEngine` and fetch initialization state from S3. The following section describes the design of the co-sandbox API.

3.2.2 Co-sandbox API

The primary design goal of the co-sandbox API, shown in Table 3.2, is to keep co-sandboxes small (challenge 1) by shifting as much functionality as possible into the platform. In service of this goal, the co-sandbox API is high-level and provides functions for connection establishment and RPCs: `Connect`, `SendRPC`, and `RecvRPCReply`. The high-level design of the API reduces the compiled binary size of co-sandboxes because `WASMEngine` can supply an implementation of functionality needed to interact with cloud services such as client-side RPC and networking

Methods	Description
`Connect(addr)` → `conn_fd`	Establish network connection to destination `addr` (IP or DNS name)
`SendRPC(conn_fd, rpc_id, buf)`	Asynchronously send `rpc_id`'s `buf` over `conn_fd`
`RecvRPCReply(conn_fd, rpc_id)` → `buf`	Block until `rpc_id` is complete and return its reply `buf`
`Exit(status)`	Exit with `status`

Table 3.2: API developers use to write co-sandboxes. The implementation of the API is supplied by `WASMEngine`, which moves the complexity and code size of the RPC stack implementation out of the co-sandbox and into the shared platform infrastructure.

stacks, support for long-lived sessions, and authentication. `WASMEngine` relies on the fact that all σOS `procs` interact with a uniform RPC protocol, the σOS API, and thus can implement the client-side of this protocol on behalf of the co-sandbox.

`Connect` establishes a network connection to a service with DNS name or IP address `addr`, and returns a file descriptor `conn_fd`. `SendRPC` asynchronously dispatches a marshaled RPC, labeled by an `rpc_id` of the co-sandbox's choice (e.g., a sequence number), on the connection named by `conn_fd`. `RecvRPCReply` blocks until a response for the `rpc_id` is received, and returns a buffer containing the response to the co-sandbox.

Asynchronous `SendRPC` is an important feature of the RPC API since WASM modules are single-threaded. `SendRPC` allows the co-sandbox to extract parallelism by delegating asynchronous communication to the host-side of the RPC API. This allows the co-sandbox to, for example, construct and send other RPCs while the host runs RPCs on its behalf.

`Exit` terminates the co-sandbox and reports the exit `status` to the platform.

3.2.3 Co-sandbox result transfer API

The co-sandbox result transfer API, shown in Table 3.3, is designed to efficiently pass initialization results from the co-sandbox to the application (challenge 2), and to make it possible for existing applications to use co-sandboxes with few modifications (challenge 3).

`SendMsg` and `RecvMsg` exchange buffers of data, such as application state or serverless inputs downloaded by the co-sandbox. `WASMEngine` uses shared memory to enable zero-copy transfer of data from the co-sandbox to the application, an important feature for applications that load a significant amount of state.

`WASMEngine` sets up a shared memory region to store co-sandbox's RPC results when a new application instance is assigned to a machine. `WASMEngine` dispatches the co-sandbox's RPCs and writes the replies directly into the shared memory region. When the co-sandbox calls `RecvRPCReply`, `WASMEngine` returns a buffer backed by this shared memory region to allow the co-sandbox to access the reply without copying it. The co-sandbox passes the buffer on to the application using `SendMsg`.

Methods	Description
`SendMsg(msg_id, buf)`	Send `buf` between co-sandbox/container
`RecvMsg(msg_id)` → `buf`	Receive a `buf` message from co-sandbox/container
`TransferConn(conn_id, addr, conn_fd)`	Transfer ownership of cached connection to co-sandbox/container
`GetConn(conn_id, addr)` → `conn_fd`	Receive ownership of connection from co-sandbox/container

Table 3.3: Co-sandbox result transfer API that developers use to bootstrap a co-sandbox's applications once the application is up and running. The co-sandbox calls `SendMsg` and `TransferConn` to transfer initialization results to the application, and the application calls `RecvMsg` and `GetConn` to block until the co-sandbox has sent them. `WASMEngine` synchronizes communication between the co-sandbox and application, and makes the RPC results available to the application via shared memory.

After the application starts, its co-sandbox library maps the shared memory region `WASMEngine` set up for it. The application calls `RecvMsg` to receive buffers passed to it by the co-sandbox, and `WASMEngine` returns corresponding buffers backed by the shared memory region. `WASMEngine` synchronizes access to RPC results, because long-running RPCs initiated by the co-sandbox may not complete until after the application has started running and called `RecvMsg`.

`TransferConn` and `GetConn` transfer ownership of a network connection, allowing the application to use the connection directly. By claiming ownership of a connection from its co-sandbox, the application avoids re-establishing and re-authenticating the connection if it needs to issue follow-up RPCs to the remote service. `WASMEngine` implements `TransferConn` and `GetConn` by passing the connection's Linux file-descriptor to the application over a socket.

The result transfer API employs a flexible naming system in which developers define the identifiers for connections and messages. RPC client library developers can use this naming system to support co-sandbox-assisted initialization below the RPC library's API and transparently to the application. For example, the RPC library can identify RPCs with a counter or hash of API function arguments.

The application developer then follows the naming scheme when writing their co-sandbox. For example, the co-sandbox may use a counter to line up the RPCs it issues with the order of initialization RPCs in the application. When the application starts up and tries to issue its initialization RPCs, the RPC client library intercepts the RPCs and serves the RPC results from the co-sandbox via `WASMEngine`. `WASMEngine` uses the message and connection IDs to synchronize access to RPC results and ensure both the co-sandbox and application see a consistent transcript of initialization RPCs and results. `WASMEngine` prevents the application from getting ahead of the co-sandbox and issuing RPCs that the co-sandbox has yet to send or re-issuing RPCs that are already in progress.

```
class storage:
   def download(self, bucket, key)
   def download_stream(self, bucket, key)
   def upload(self, bucket, key)
   def upload_stream(self, bucket, key, stream)
```

Figure 3.3: SeBS Python storage API.

3.2.4 Developer workflow

Developers can write co-sandboxes in any language that compiles to WASM. Developers upload their co-sandbox to the cloud provider separately from the application binary and declare resource reservations for it as they do for application containers and serverless functions today.

Developers can optionally specify a byte slice as an input argument to a co-sandbox. This can be used, for example, to communicate function inputs or credentials for the co-sandbox to use when sending RPCs and establishing sessions to other services.

3.2.5 Cloud API changes

Cloud providers can support co-sandboxes by augmenting existing APIs. For example, application launch and serverless function invocation APIs that specify a function or application container ID can be augmented to specify a co-sandbox to run. In the σOS implementation of co-sandboxes, the compiled co-sandbox and its input bytes are included in the `proc` descriptor.

3.3 Case-studies: applications using co-sandboxes

We ported several off-the-shelf cloud applications to use co-sandboxes, including all of the SeBS [34] Python applications and two microservice applications, `etcd` and `memcached`. We also wrote some applications from scratch, including serverless applications like `imgrec-wasm` and `imgrec-py`, as well as microservices like `vecdb`, an in-memory vector database, and `cached`, a co-sandbox-native analogue to `memcached`. The remainder of this section discusses the experience of adapting existing applications (§3.3.1) to benefit from co-sandbox-accelerated cold-starts and writing new (§3.3.2) co-sandbox-accelerated applications from scratch.

3.3.1 Fast cold-start for off-the-shelf applications

Developers can use co-sandboxes to accelerate cold-start for existing applications by writing a simple compatibility layer which communicates with the co-sandbox, and makes the fetched state available to the application (§3.2.3).

For example, SeBS's Python applications use a `get`/`put`-style API typical of client libraries used to access cloud storage services like S3, shown in Figure 3.3. We added co-sandbox support to all the SeBS Python functions with no modifications to the applications themselves.

```
def download(self, bucket, key):
  if self.use_cosandboxes:
    # Set the ID of the message to retrieve
    msg_id = self.msg_ctr
    # Increment the message counter
    self.msg_ctr += 1
    # Receive state from co-sandbox
    off, nbyte = self.csbox_clnt.recv_msg(msg_id)
    # Retrieve shmem region pointer
    shm_ptr = self.csbox_clnt.get_shm_ptr()
    # Index into shmem to access reply
    return shm_ptr[off:off+nbyte]
  else:
    # Normal download implementation...
```

Figure 3.4: SeBS storage client library modified to use co-sandboxes for initialization results. The library developer defines the initialization result naming scheme as a simple counter so that the library can fetch results from the co-sandbox.

```
import storage as s
import torch
def handle_request(req):
  bkt = "my-bucket"
  img = s.download(bkt, req.input)
  weights = s.download(bkt, "resnet50.pth")
  model = torch.load(weights)
  # ...
```

Figure 3.5: The unmodified SeBS ImageRecognition application benefits from co-sandbox support in the storage client library.

Instead, we changed the client library implementation to intercept the initialization RPCs and communicate with the co-sandbox rather than send the RPCs to the remote storage service. The co-sandbox passes the stored results of the initialization RPCs back to the client library, which returns the results to the application.

Figure 3.4 shows the implementation of the download function in the storage library with co-sandbox support. The co-sandbox result transfer API allows the storage library developer to choose a naming system to communicate results: in this case, a counter. Developers who wish to use the library and speed up their application's start time with a co-sandbox can do so by using the same naming system defined by the storage library developer.

For example, the SeBS ImageRecognition application shown in Figure 3.5 first downloads its input image and then downloads its model weights. The application developer can make use of co-sandbox support in the storage library by writing a co-sandbox which fetches the initialization state in the same order, as shown in Figure 3.6. Since the application developer followed the naming scheme defined by the storage library, the unmodified Image Recognition

```
imgReq := &S3GetReq{
  Bucket: "my-bucket",
  Key: cSboxArgs[0],
}
modelReq := &S3GetReq{
  Bucket: "my-bucket",
  Key: "resnet50.pth",
}
connFD := Connect("s3.region.amazonaws.com")
// Send get requests to S3
SendRPC(connFD, 0, imgReq.Marshal())
SendRPC(connFD, 1, modelReq.Marshal())
// Receive input image and model weights
imgBuf := RecvRPCReply(connFD, 0)
modelBuf := RecvRPCReply(connFD, 1)
// Send reply buffers to application
SendMsg(0, imgBuf)
SendMsg(1, modelBuf)
Exit(0)
```

Figure 3.6: The ImageRecognition application's co-sandbox uses the storage library-defined result naming scheme to fetch and pass on the application's initialization state.

will transparently benefit from co-sandbox-accelerated start time. The co-sandbox will run while the platform carries out the application's setup, and will bootstrap the application once it is up and running.

One reason co-sandboxes are able to benefit existing applications with few-to-no modifications is that modern cloud applications rely on RPC client libraries to communicate with external services. These libraries already abstract many of the details of this communication: applications are relatively oblivious to connection establishment, network transport, marshalling and unmarshalling, management and interpretation of the retrieved data. Whether the library actually fetches state from a remote service via RPC or locally from its co-sandbox is an implementation detail which does not affect the library's application-facing API. By modifying the RPC libraries to take advantage of the co-sandbox API's efficient initialization result transfer and defining a clear naming scheme, library developers enable applications which depend on those libraries to benefit as well.

3.3.2 Fast cold-start for co-sandbox-native applications

Although co-sandboxes can benefit off-the-shelf applications, co-sandbox-native applications can achieve even greater cold-start latency reduction by using the co-sandbox API's support for shared memory and connection passing. These applications can build their internal data structures from shared memory regions populated by the co-sandbox to avoid any `memcopying` and can reuse connections established by the co-sandbox. Two examples are `vecdb` and `cached`.

```
getShardsReq := &GetMyShardsReq{ ID: myID }
connFD := Connect("lb.svc")
// Send request to LB
SendRPC(connFD, 0, getShardReq.Marshal())
// Receive shard assignment
repBuf := RecvRPCReply(connFD, 0)
getShardsRep := &GetMyShardsRep{}
getShardsRep.Unmarshal(repBuf)
cachedAddr := "cached.svc"
cachedConnFD := Connect(cachedAddr)
dlReq := &DownloadShardReq{ Shards: rep.Shards }
// Request shard download from cached
SendRPC(cachedConnFD, 1, dlReq.Marshal())
// Wait for download to complete
dlBuf := RecvRPCReply(connFD, 1)
// Send state to container
SendMsg(0, dlBuf)
Exit(0)
```

Figure 3.7: `vecdb` co-sandbox fetches the shard assignment and sends it to the microservice application.

```
// Block until co-sandbox sends state
stateBuf := RecvMsg(0)
rep := &DownloadShardReq{}
rep.Unmarshal(stateBuf)
for _, shard := range rep.Shards {
  start := shard.Off
  end := shard.Off + shard.Len
  // Index into the shared memory buffer
  shardBytes := stateBuf[start:end]
  vecdb.LoadShard(shard.ShardID, shardBytes)
}
```

Figure 3.8: `vecdb` microservice application receives state from its co-sandbox.

`vecdb` is a sharded in-memory vector database implemented in C++. Clients send `vecdb` a request including an input vector as well as a range of vectors to search. `vecdb` performs a cosine-similarity search over its vector database, and returns the IDs of the vectors closest to the input vector. `cached` is an in-memory sharded key-value cache implemented in C++, similar to `memcached` [87].

`vecdb` uses co-sandboxes to accelerate scaling up. When a new `vecdb` instance starts, its co-sandbox downloads its shard of the vector space, which may be stored in an in-memory cache service (like `cached`), in a database, or in provider-managed storage (like S3). `cached` operates similarly to `vecdb` by using its co-sandbox to fetch its shards from existing peer `cached`s.

Figure 3.7 shows the implementation of the co-sandbox which fetches `vecdb`'s shards and sends them to the new `vecdb` instance. Figure 3.8 shows how the newly running `vecdb` communicates with its co-sandbox to receive its state and load the state into its internal data structures.

The message-passing API which the co-sandbox and `vecdb` use to communicate initialization results, `SendMsg` and `RecvMsg`, are implemented using shared-memory set up by `WASMEngine`. This allows the co-sandbox to issue RPCs to fetch `vecdb`'s shards and pass the results back to `vecdb` without additional memcopies. `vecdb` builds its internal data structures when loading its shards by simply setting references to the shared memory region containing them.

3.4 Co-sandbox implementation

This section discusses the details of the σOS components which implement the co-sandbox API (§3.4.1) and implementation details of the co-sandboxes we wrote to support our applications and evaluation (§3.4.2).

3.4.1 `WASMEngine` implementation

The `WASMEngine` implements the co-sandbox API (Table 3.2) and makes the results of co-sandbox RPCs available to new application instances via the co-sandbox result transfer API (Table 3.3). `WASMEngine` runs as a per-machine daemon on every node managed by the cloud platform. It collaborates with the platform to manage the application instance lifecycle. `WASMEngine` is written in Go and runs co-sandboxes using the Wasmer WebAssembly runtime [122].

When a new application instance starts, `WASMEngine` quickly downloads its co-sandbox and maps a region of shared memory using the Linux POSIX shared-memory APIs to store RPC replies. It then runs the co-sandbox in a WASM sandbox, establishing and caching connections to downstream services, and forwarding marshaled RPCs to them as requested by the co-sandbox.

`WASMEngine` communicates with an application instance via a Unix named pipe mounted into the container's filesystem in order to coordinate access to the initialization results sent by the co-sandbox. `WASMEngine` implements a simple protocol over the pipe to block microservice `RecvMsg` requests in the event until the co-sandbox sends the corresponding `SendMsg`, and vice versa. In the event that the co-sandbox or application crashes with an error, `RecvMsg` will return an error to the caller.

When it starts up, the application maps the shared memory region created by `WASMEngine` using POSIX shared-memory APIs. `SendMsg` and `RecvMsg` exchange pointers into this region, allowing the application to directly read data passed to it by the co-sandbox.

3.4.2 Implementing co-sandboxes for σOS applications

We wrote co-sandboxes for the applications in Rust. co-sandboxes import a library which declares external functions corresponding to the co-sandbox API (Table 3.2) and co-sandbox

result transfer API (Table 3.3).

Chapter 4

Evaluation

The primary goal of σOS is to provide a single API which seamlessly supports both serverless and microservice applications. To support short-running and burst-parallel serverless applications, σOS must start `proc`s with low latency and high scheduling throughput. To support microservices, σOS must enable low-overhead communication between `proc`s and allocate CPU to guarantee resource requests and achieve high utilization.

This section evaluates whether σOS achieves this goal by answering the following questions:

1. Do σcontainers allow fast `proc` setup, and can σOS schedule `proc`s at high throughput? (§4.1)
2. Do σEPs provide high-performance networking? (§4.2)
3. Do σOS applications perform as well as their serverless and container-based counterparts? (§4.3)
4. Can the σOS scheduler achieve good utilization with BE serverless tasks and LC microservice tasks, and provide fairness between tenants? (§4.4)
5. How much do co-sandboxes speed up off-the-shelf applications' end-to-end start latency? (§4.5)
6. How much more do co-sandbox-native applications improve their start latency? (§4.6)
7. How small can co-sandboxes be? (§4.7)
8. Are co-sandboxes able to speed up start latency on platforms with fast setup? (§4.8)

4.1 Fast `proc` setup with σcontainers

We measure the start latency of σOS `proc`s and compare it to several state-of-the-art platforms. For each platform, we time from the moment spawn is invoked until the first line of `main` executes. This captures the platform's setup cost and excludes the application initialization cost. All but the AWS Lambda and Mitosis measurements use two AWS EC2 m6i.4xlarge VMs with 16 vCPUs backed by 2.9GHz Intel Ice Lake 8375C CPUs, 64 GiB of

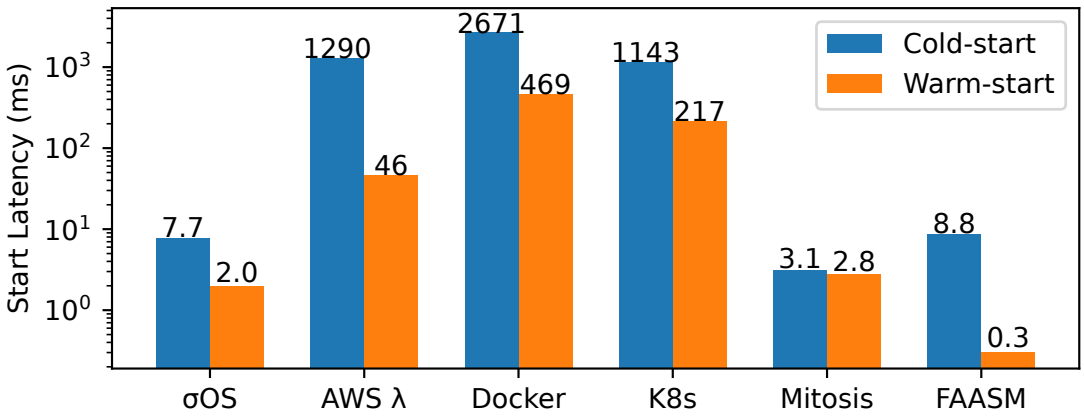

Figure 4.1: Start latency for a Hello World function on several platforms. Latency includes the time from when the `proc` is spawned until it reaches line 1 of `main`. This measure therefore represents the setup cost of each platform, and excludes the application's initialization cost.

memory, up to 12.5Gbps network bandwidth, and up to 10 Gbps EBS burst bandwidth. The client invoking the function runs on one machine, and the platform runs on the other. We do not know what hardware AWS uses to run the Lambda functions. The Mitosis numbers are as reported in the Mitosis paper [124], which uses similar speed CPUs to the above configuration.

The `proc` being started is a simple BE Hello World program written in Rust to be able to measure σcontainer start times while excluding the 15 milliseconds that the Go runtime takes to start. σOS uses co-sandboxes to minimize the start-latency impact of language runtime and application initialization, as shown in (§4.6).

Start latencies. Figure 4.1 shows the results. First we consider cold- and warm-start times for AWS Lambda, Docker containers, and Kubernetes (K8s) pods. All three systems isolate execution units using containers, and AWS Lambda additionally isolates functions with a MicroVM. Cold-starts in these systems require downloading and unpacking the container image (and, in AWS Lambda, additionally involve starting a MicroVM), whereas warm-starts exclude these costs.

σOS `procs` warm-start significantly faster than the other three systems' containers, because the other three systems start full containers with overlay file systems [120]. σOS' σcontainers (§2.2.1) avoid this cost because σOS offers neither custom file system contents nor direct file system write access. Instead, `procs` access remote or (via the `ux` server) local storage using the σOS API. Additionally, σOS' `dialproxyd` (§2.2.2) design avoids expensive network isolation operations, and σOS moves expensive `cgroup` creation off the critical path. σOS' cold-start latency benefits from `binfs`, which demand-pages `proc` binaries over the network.

Mitosis [124] is a platform that provides fast start of containers for serverless functions—its containers and scheduler are not suited to stateful, long-running microservices—using a new remote-fork primitive, implemented using RDMA and a modified Linux kernel [124]. Mitosis

		Time (ms)
Placement		0.42
σcontainer	Linux NS	0.28
	FS jail	0.42
	`seccomp`	0.46
	AppArmor	0.02
	`exec`	0.37
Total		1.97

Table 4.1: Breakdown of warm-start time for a Hello World BE `proc` written in Rust. Operations above the line are costs of `proc` placement, which includes time spent in the `besched` queue, whereas operations below the line are machine-local costs of σcontainer creation.

thus sets a high bar for how fast serverless functions can be started. To compare with Mitosis, we benchmark σOS on an AWS EC2 instance with the same CPU clock rate as reported in the Mitosis paper[1]. Cold-starts in Mitosis involve remote-forking a running serverless function from a remote machine and creating a Mitosis "lean container" for the function. As expected, cold starts in σOS are slower than Mitosis (7.7 ms vs. 3.1ms), because σOS provides network isolation, σOS' `proc` binary demand-paging does not make use of specialized networking hardware (e.g., RDMA-capable NICs), and σOS runs atop an unmodified Linux kernel, which adds latency as the paged `proc` binary data moves back and forth across the user-kernel boundary.

Faasm [106], a serverless platform for WASM programs, achieves low start time by relying on the WASM runtime for isolation and for restricting system calls. Faasm cold-starts include the cost of downloading the WASM program from a local Redis instance, whereas warm-starts exclude this cost. σOS has slower warm-starts than Faaslets, because σOS pays the cost for separate address spaces and other OS isolation techniques such as namespace isolation. σOS has faster cold-starts than Faasm because `binfs` demand-pages `proc` binaries, whereas Faasm downloads the entire WASM program before executing the function. We took the Faasm measurements without network isolation enabled; with network isolation, the cost of starting Faasm functions would likely be much higher.

Breakdown of `proc` start latency. Table 4.1 breaks down BE `proc`s' warm-start latency, including the time required for `Spawn` to send the descriptor to `besched`, place the `proc` onto a `schedd` instance, create a σcontainer, and start the `proc`. The cost of a warm-start in σOS is dominated by σcontainer setup time, which consumes nearly 80% of the total.

Cold-starts, which occur when a realm runs a given `proc` binary on a machine for the first time, include the additional time for `binfs` to page the `proc` binary's first few pages

[1]We were unable to access the RDMA hardware and software setup required to run Mitosis and so report the Hello World benchmark numbers from the Mitosis paper.

σOS component	Max Throughput (procs/sec)
lcsched	50,144
besched shard	53,306
proc start (1 machine)	1,590
proc start (24 machines)	36,650

Table 4.2: Maximum throughput of some components of proc creation on Cloudlab c220g5 machines.

over the network.

Spawn throughput. There are two main tasks that might limit the throughput at which σOS can spawn new BE procs: the sharded besched placement service (§2.2.3), and the time required to create the proc on the selected machine (§2.2.1). Both throughputs can be scaled up (by running more besched shard servers, and installing more machines, respectively). The following experiments examine how quickly a single besched can process spawn requests, how quickly a single machine can create procs, and the end-to-end achievable proc start throughput on a cluster of 24 machines. Table 4.2 summarizes the results.

To find how many BE Spawn requests per second a single besched can handle, we run a σOS cluster consisting of 24 CloudLab c220g5 machines, scheduled by a single besched. An open-loop client on a remote machine Spawns "dummy" BE procs at a fixed rate for 10 seconds. The BE procs never actually run, but they traverse the full BE proc scheduling path; they are spawned onto the besched, which places them onto a machine's schedd, which ignores them. The maximum client Spawn rate at which the single besched can keep up is 53,306 per second.

Once besched has placed a proc on a machine, that machine's schedd needs to create a σcontainer and start the proc. To evaluate the throughput limits of this task on a single machine, an open-loop client Spawns BE Rust Hello World procs at a set rate, regardless of how quickly σOS responds. For this benchmark, a single besched, running on a dedicated machine, schedules the Hello World procs onto a single schedd running on a different machine.

proc start throughput saturates and queues begin to build up once the Spawn rate exceeds 1,590 procs per second. The bottleneck is Linux mount namespace creation which occurs every time a σcontainer is created and involves taking a global lock in Linux.

We evaluate the end-to-end sustainable proc start throughput on a cluster of 24 CloudLab c220g5 machines by running an open-loop client which Spawns BE Rust Hello World procs at a set rate, independently of how quickly σOS starts them. A single besched places the procs across the cluster. σOS is able to start up to 36,650 procs per second before the Spawn rate exceeds the start rate.

Even as the Spawn rate approaches the maximum start rate, σOS' median and 90% scheduling latencies remain low. σOS achieves 36,650 proc starts per second with p50 start latency of 5.8 ms and a 90% start latency of 11.6 ms. To put σOS' proc start throughput in perspective, we compare to Mitosis, which can fork 10,000 containers across multiple

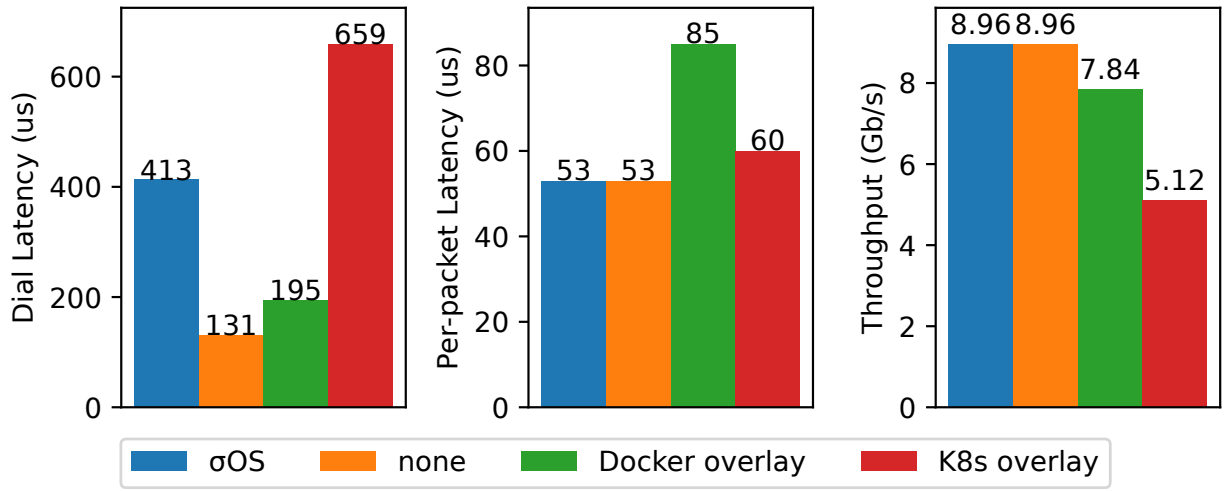

Figure 4.2: Different network isolation strategies' network latency and throughput between containers communicating across Cloudlab c220g5 machines. Measurements are taken from Go server and client programs which communicate via TCP. The implementations are identical across all isolation strategies, with the exception that the σOS client and server communicate via the σEP API.

machines in one second. Mitosis' container fork rate translates to 97 container starts per core per second, and σOS' peak `proc` start rate translates to 76 `proc` starts per core second. σOS achieves this performance without kernel changes or RDMA.

Summary. σOS starts `proc`s quickly and with high throughput. As a result σOS is a good fit for applications structured as many short-running `proc`s. Fine-grained `proc`s give providers more frequent opportunities to rebalance resource allocations, and to smooth out short drops in resource utilization as realms' application load varies.

4.2 σEP performance

To determine the performance of σEPs, we benchmark network throughput and latency between two Cloudlab c220g5 nodes using different network isolation techniques. The measurements use a simple Go client and server that communicate with TCP; the σOS version does this via the σEP API. Dial latency is the time to establish a new TCP connection. Per-packet latency is half a TCP round trip. Throughput is the bit rate at which the client can write 1MB buffers to the server. All measurements are the average of 1000 trials.

Figure 4.2 compares σOS' σEP-isolated network performance to no network isolation, isolation using Docker overlays, and isolation using Kubernetes (K8s) overlays. The σOS `dialproxyd`-mediated `Dial` protocol makes connecting more expensive than without network isolation. Once connected, σEPs provide the same latency and throughput as no network isolation, since `dialproxyd` sends the TCP socket to the `proc`, which reads and writes directly. Docker and K8s, in contrast, forward all connection data through a local proxy process.

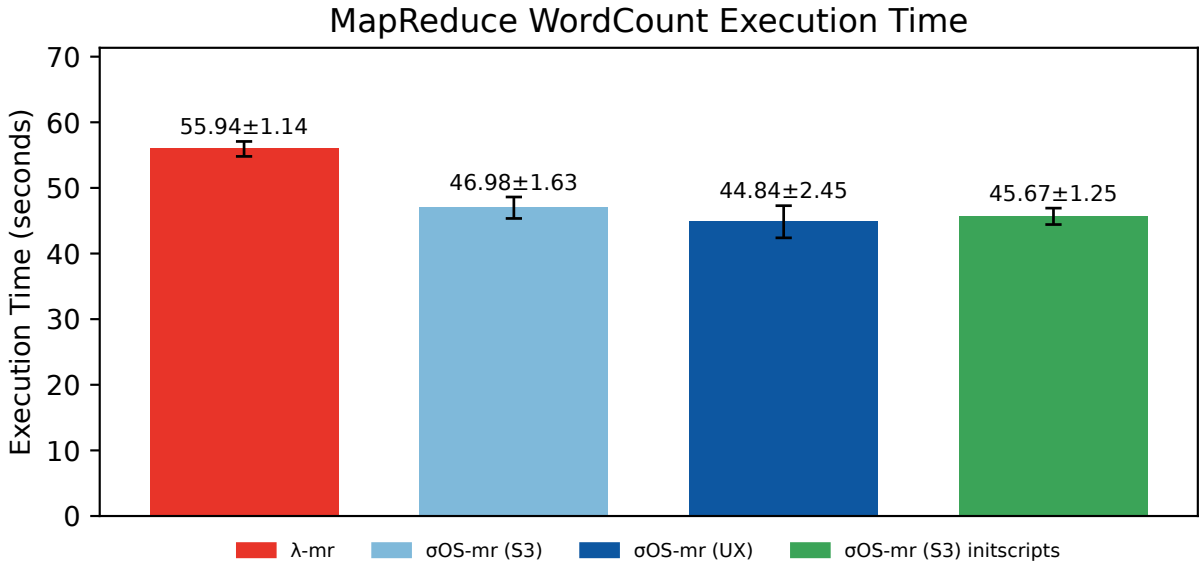

Figure 4.3: End-to-end execution time of MapReduce WordCount on σOS and on λ-`mr` from a warm-start, with 10GB of input. Both σOS-`mr` and λ-`mr` store input and output files in S3. σOS (`ux`) stores intermediate output files in `ux`, σOS's local file server. σOS (`s3`) co-sandbox pre-fetches mapper and reducer input with co-sandboxes.

4.3 σOS application performance

To evaluate application-level performance with σOS, we compare σOS-`mr` to Corral [32] (labeled λ-`mr`), a serverless MapReduce framework for AWS Lambda, σOS-`hotel` to K8s-`hotel`, and σOS-`socialnet` to K8s-`socialnet` .

σOS-`mr` vs. λ-`mr`. We run σOS-`mr` on an AWS Virtual Private Cloud (VPC) composed of 16 EC2 `t3.xlarge` VM instances. Each instance has 4 vCPUs, 16GiB of memory, and a 20GiB EBS volume. Each instance has up to 5Gbps network burst bandwidth, and 2,085Mbps EBS burst bandwidth. To make the comparison fair, we provision λ-`mr`'s lambdas with 1760MB of memory, which gives them the equivalent of 1 vCPU, and disable 2 vCPUs on each of σOS's 4-core VMs. Since λ-`mr` starts 32 lambdas in parallel for both the map and reduce phase, this gives λ-`mr` and σOS-`mr` the same resources when σOS-`mr` runs on 16 machines. Both σOS-`mr` and λ-`mr` are written in Golang, and pay for the cost of the Go runtime (e.g., garbage collection). The input is 10GB from a snapshot of English HTML Wikipedia pages.

The input and final output files for both versions of MapReduce are stored in S3. For λ-`mr`, the intermediate files are also stored in S3. To evaluate the benefit of σOS-`mr`'s transparent access to σOS-exposed local storage, we run σOS-`mr` in two configurations: σOS (`s3`) and σOS-`mr` (`ux`); the latter writes and then remotely reads intermediate files via `ux`, σOS' local file server. In order to evaluate the benefit of co-sandboxes, we run the σOS-`mr` (`s3`) configuration with co-sandboxes which prefetch the mappers' and reducers' inputs.

Figure 4.3 shows the results, averaged over 5 runs for each configuration. σOS-`mr` performs

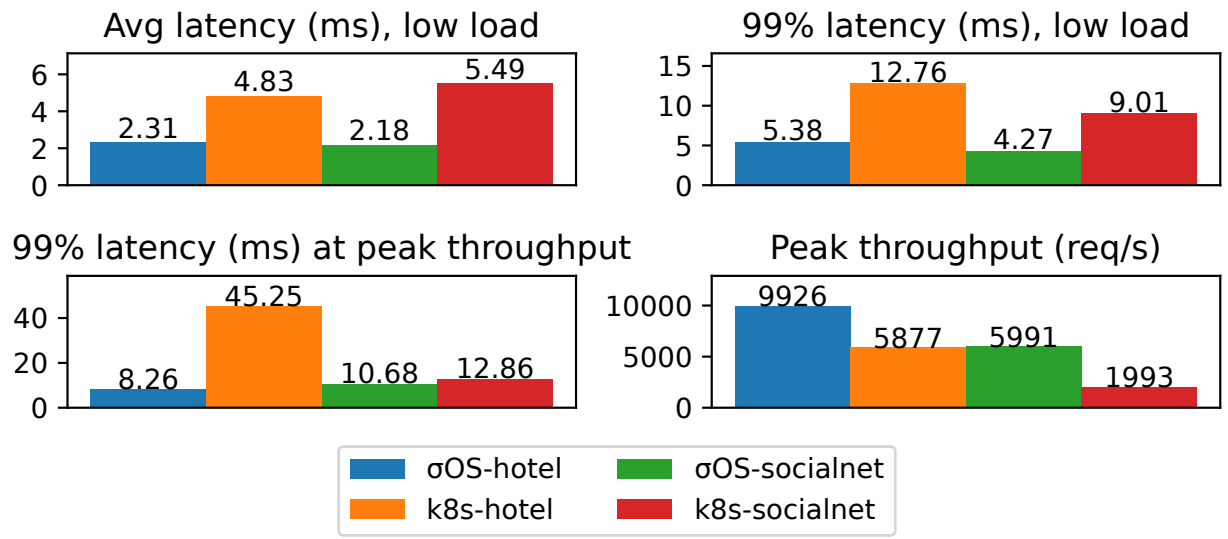

Figure 4.4: Latency and throughput of σOS-`hotel` and K8s-`hotel`, and σOS-`socialnet` and K8s-`socialnet` . The low-load 99% latency is measured at a request rate of 1000 Req/s. Peak throughput is the maximum request throughput with median request latency below 10ms.

comparably to λ-`mr` with intermediate files stored in S3, and 1.17× faster with intermediate files stored on local disk (really EBS) via `ux`. `ux` helps because it has higher throughput than S3. σOS-`mr` can transparently take advantage of `ux`'s local scratch space exposed by just changing the application's intermediate file pathnames.

Co-sandbox-accelerated input fetching allows σOS-`mr` to achieve similar performance when storing intermediate files in `s3` as when storing them in `ux`. This is because the co-sandbox pre-fetches mapper and reducer inputs while each mapper or reducer `proc`'s Go runtime initializes (which takes 15-20ms). When the `proc` tries to actually read its input, it is served from local shared memory.

σOS-`hotel` vs. K8s-`hotel` and σOS-`socialnet` vs. K8s-`socialnet` . We compare σOS-`hotel` to K8s-`hotel` and σOS-`socialnet` to K8s-`socialnet` on 8 Cloudlab `c220g5` machines. Each machine has two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, 192GB of memory, and a 10Gb Intel X520-DA2 NIC. To induce contention for resources and force σOS and Kubernetes to place microservices on multiple hosts, all but 4 CPUs are disabled on each machine.

Figure 4.4 shows the latency (average and 99%) and the peak throughput using the workload generator from DeathStarBench, running the search workload. Peak throughput is the maximum client request throughput for which median latency stays below 10ms. σOS yields 42% lower 99% latency under low load for σOS-`hotel` and 47% lower tail latency for σOS-`socialnet` under low load. σOS enables `hotel` and `socialnet` to achieve 1.68× and 3.01× higher peak throughput than Kubernetes.

The σOS implementations of `hotel` and `socialnet` outperform the Kubernetes implementations because σEP communication is more efficient than Kubernetes overlay networking,

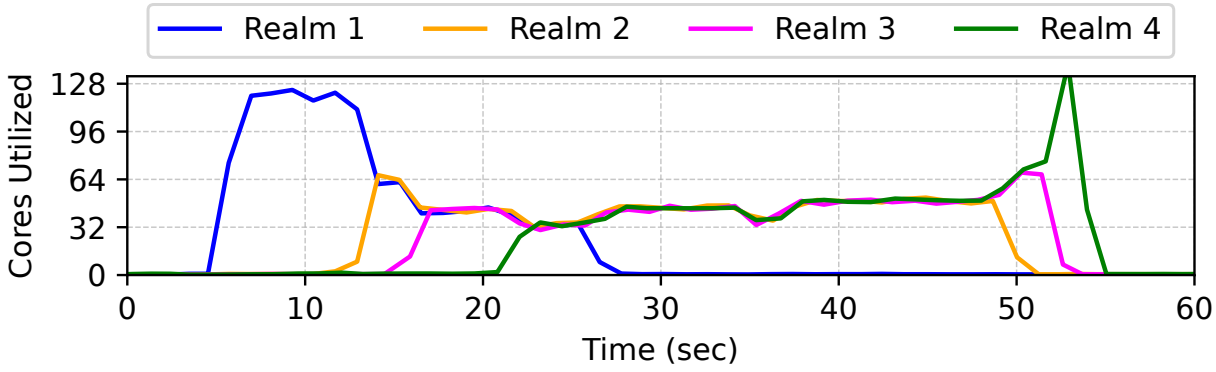

Figure 4.5: CPU utilization of 4 realms running BE fine-grained `mr` jobs starting at different times, on a 32 AWS `t3.xlarge` VM VPC scheduled by 2 `besched` shards. σOS dynamically resizes each realm's CPU allocation in response to load in order to give each realm an equal share of the cores. When one job's load decreases, σOS quickly reallocates its idle cores to the realms with remaining work.

as shown in §4.2.

Summary. σOS's abstractions perform well and allow stateless- and microservices-style applications to achieve high performance.

4.4 Scheduling `procs`

σOS's scheduler goals include sharing resources fairly among realms with BE work and achieving high utilization by allowing BE `procs` to use resources reserved but left idle by LC `procs` (§2.2.3). This section evaluates whether σOS meets its scheduling objectives through case studies in which σOS multiplexes multiple realms' BE workloads and realms with BE and LC workloads.

4.4.1 Fair sharing between realms with BE `procs`.

We evaluate σOS' ability to divide resources equally between realms with BE work using an `mr` implementation of `grep` on a 10GB snapshot of Wikipedia. We make `mr`'s mappers fine-grained, which is beneficial for the provider. Fine-grained mappers terminate quickly, and give the platform frequent opportunities to rebalance resources and resolve scheduling mistakes. Each mapper consumes 1MB of input, and the median mapper execution time is 123ms.

Fine-grained mappers also present a challenge for the platform infrastructure, because the cost of starting a mapper can dominate the mapper's compute time. The σOS `mr` implementation addresses this with co-sandboxes. Each mapper uses a co-sandbox to prefetch its inputs while σOS starts its σcontainer and the Go runtime initializes. In total, each `mr`

job runs 10,240 mappers. `mr` input and output files are stored in `ux` on a dedicated set of machines, since `ux` provides high-throughput access to storage.

Figure 4.5 shows four realms, each running a job consisting of many `mr procs`, and illustrates how σOS shifts resources between them. The graph shows the CPU utilization of each realm as a function of time. There are 32 AWS `t3.xlarge` machines, each with four cores, and 2 `besched` shards which distribute BE `procs` among them. Realm 1's job starts at 5 seconds, Realm 2 starts 5 seconds later, Realm 3 starts at 15 seconds, and Realm 4 starts at 20 seconds.

Each realm `Spawn`s all of its mapper `procs` as soon as the job starts, and then stops and waits for all its `procs` to complete before starting its reducers. `mr procs` are spawned with a 2GB memory request, and `procs` which cannot fit on any machine once spawned are queued at `besched`.

σOS gives all 128 cores' worth of CPU time to Realm 1. The graph shows that Realm 1 uses almost all of the CPU time, with the shortfall lost to I/O. As Realm 2 starts to enqueue `procs`, σOS begins dequeuing `procs` from each realm in equal shares. The CPU utilizations of Realm 1 and Realm 2 quickly equalize because `mr`'s fine-grained `procs` are short-running, giving many resource rebalancing opportunities to the `besched` cluster-level scheduler. Together with the `cgroups` policies set up by `schedd`, this results in a rapid redistribution of cores between the realms as `procs` exit and release their memory requests. Similarly, σOS redistributes cluster CPU time when Realms 3 and 4 start, giving each an equal share. As each Realm's work ends, σOS gives more CPU time to the remaining realms.

Because σOS provides fast starts for fine-grained `mr procs`, and the `mr procs` can use co-sandboxes to hide the Go runtime initialization by pre-fetching inputs, `mr` is able to utilize almost all 128 CPU cores in the cluster to full capacity.

Summary. σOS automatically manages the allocation of compute resources to realms. `procs`' fast start and co-sandbox acceleration enables fine-grained `procs` to effectively utilize the cluster's CPU resources.

4.4.2 Guaranteeing LC reservations.

Figure 4.6 shows a situation in which one realm runs a sequence of `imgprocess procs` while another runs the σOS-`hotel` site. The σOS-`hotel procs` are all LC, and the `imgprocess` procs are all BE. The x-axes show time; σOS-`hotel` receives requests from time 20 until time 100, but receives a burst from times 60 to 80. The upper graph shows the latency with which the σOS-`hotel` answers web requests; the second graph shows the σOS-`hotel` request rate; the third shows `imgprocess` throughput; and the bottom graph shows how σOS divides cores between the two realms.

When σOS-`hotel` is under low load, `imgprocess` receives the majority of the cores in the cluster and can achieve high processing throughput. In fact, those cores are reserved for the σOS-`hotel procs` (since they are LC), and merely borrowed for `imgprocess` when σOS-`hotel procs` underutilize their reservations. When σOS-`hotel` input load increases, σOS' `cgroups` configuration causes Linux to shift reserved CPU time back from `imgprocess` to σOS-`hotel`. Although `cgroups` are able to preferentially give σOS-`hotel` CPU time when it is under high load, the Linux `cgroup` API limits the degree to which one process can be

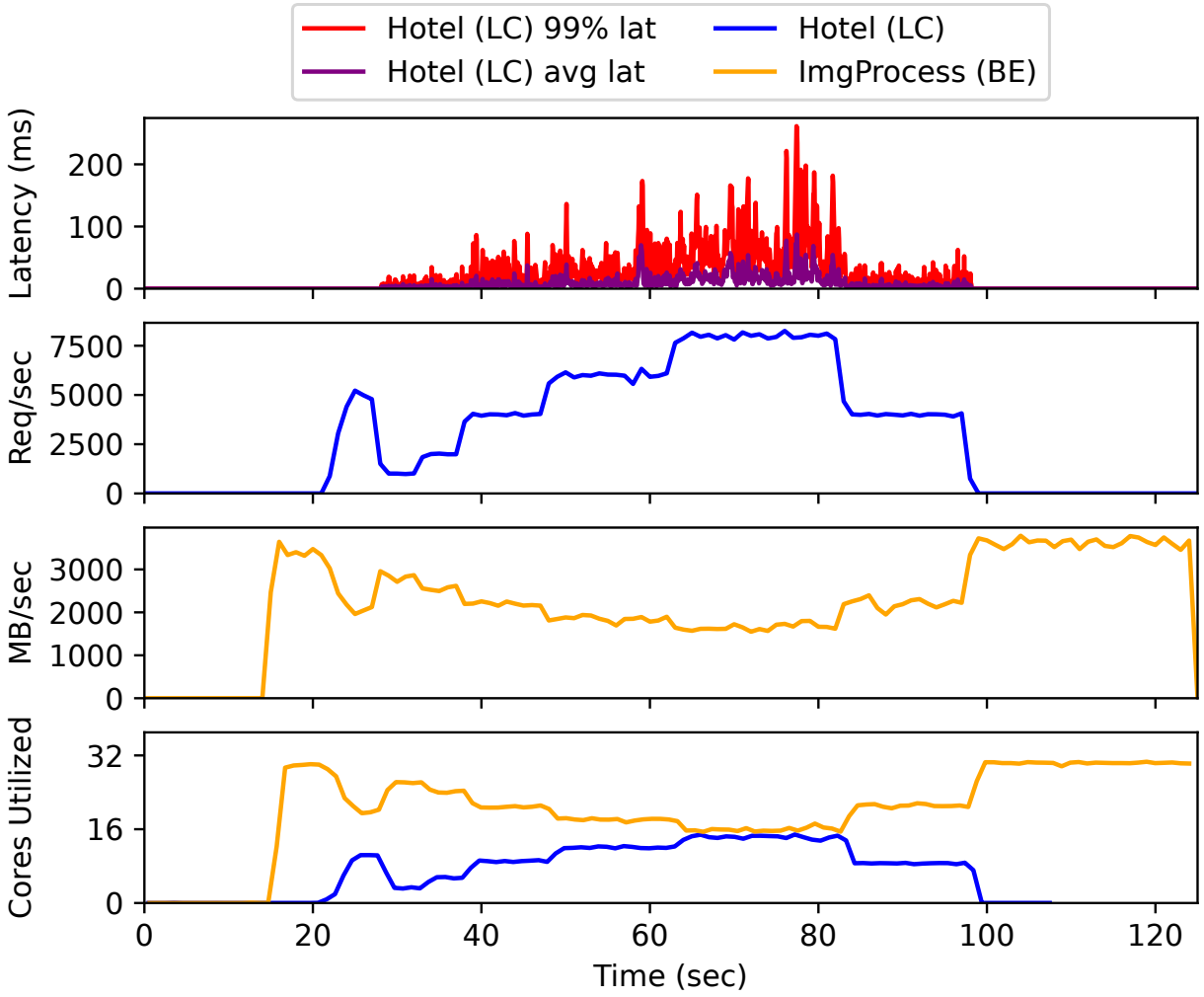

Figure 4.6: σOS multiplexing two realms across a cluster of 8 Cloudlab c220g5 machines each with 4 cores available, under varying load. One realm runs σOS-`hotel`, and the other runs `imgprocess`. As σOS-`hotel`'s load increases, it regains its reserved CPU time from the BE `imgprocess` realm. When LC load drops, the BE realm is allowed to use the idle CPU time.

prioritized over another. This causes σOS-`hotel` to experience occasional latency spikes under periods of particularly high load.

Summary. σOS's resource management helps utilization: the σOS-`hotel` application's reserved resources are available when needed, without preventing other tasks from using those resources when the σOS-`hotel` load is low.

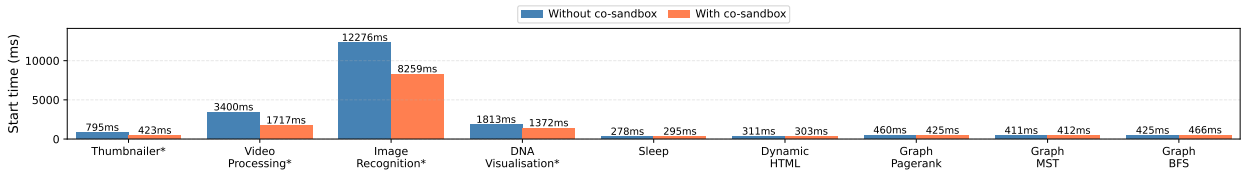

Figure 4.7: Start time of the Python SeBS benchmarks in σOS without and with co-sandboxes.

4.5 Co-sandboxes reduce off-the-shelf application start latency

One goal of the co-sandbox design is to enable developers to reduce existing applications' start latency with few application modifications. In order to evaluate whether co-sandboxes achieve this goal, we compare the start latency without and with co-sandboxes of several unmodified Python SeBS applications, `imgrec-wasm` and `imgrec-py`, and two popular microservice applications, `etcd` and `memcached`.

We evaluate co-sandboxes using 8 AWS EC2 m6i.4xlarge VMs with 16 vCPUs, backed by 2.9GHz Intel Ice Lake 8375C CPUs, 64 GiB of memory, up to 12.5Gbps network bandwidth, and up to 10 Gbps EBS burst bandwidth. The client program which spawns microservices and invokes serverless functions runs on one machine, and communicates with a σOS cluster running on the other machines to run the experimental workloads. We define start latency as the time from which a new application instance is spawned, and the time at which it starts to handle the first request. The following benchmarks report the start latency of each application during a cold-start, defined as the first time a machine runs a given application. In particular, this means that the costs of binary download and container creation are included in the setup latency, and the costs of fetching inputs, model weights, and soft-state are included in initialization latency. A good result would show that co-sandboxes decrease the start-latency of the applications, because they overlap setup and initialization.

Figure 4.7 shows the start latency of several SeBS applications. Functions marked by a (*) download their inputs and model weights from S3. Co-sandboxes speed up their start-latency by an average of 1.67×, because co-sandboxes pre-establish connections to S3 and fetch the inputs and model weights during setup. The remaining serverless applications do not experience a reduction in start-latency when run with co-sandboxes because they are purely functional. They do not fetch any input or write any output. Instead, they return their output as a string response to the invocation request. Some of these functions exhibit slightly higher start latency when running with co-sandboxes due to variability in download times of the packages they import.

Figure 4.8 shows the start latency of `imgrec-wasm` and `imgrec-py` when running without and with co-sandboxes. Both implementations benefit from co-sandbox-accelerated start time without any code changes. In this benchmark, each `imgrec` application downloads its 13.3MB model weights and the 4.9MB input image from S3 during initialization. Together, the downloads take 312ms on average.

Co-sandboxes speed up the `imgrec` applications' start time by 1.81× across `imgrec-wasm` and `imgrec-py`. Even though `imgrec-wasm` benefits from fast WASM isolation, downloading

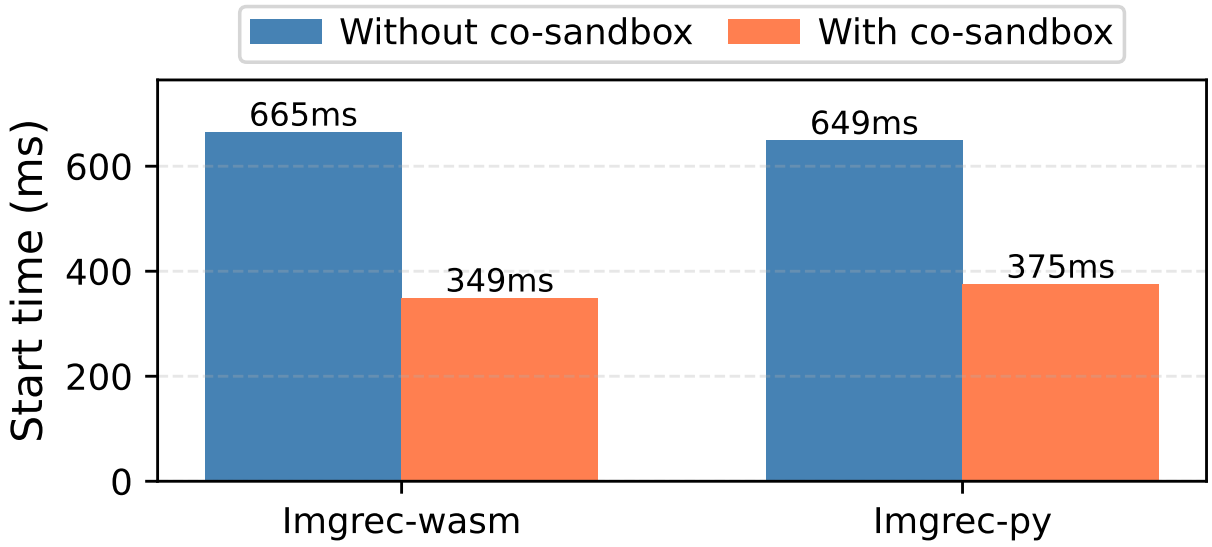

Figure 4.8: Start time of `imgrec-wasm` and `imgrec-py` in σOS without and with co-sandboxes.

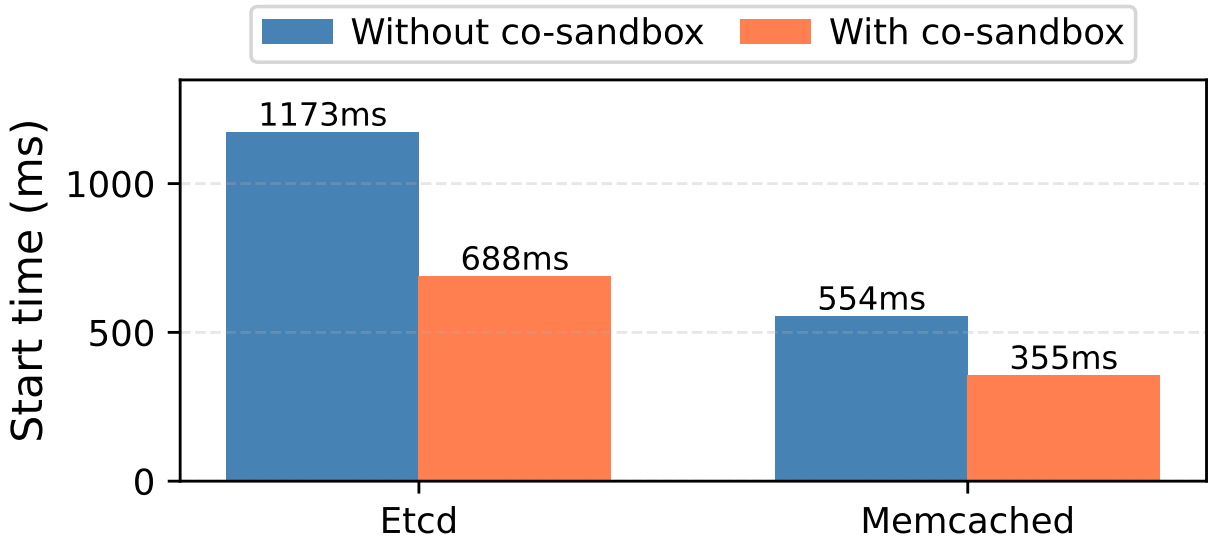

Figure 4.9: Start time of `etcd` and `memcached` in σOS without and with co-sandboxes.

its large (45.6MB) WASM binary gives its co-sandbox ample time to execute initialization RPCs and fetch its input and model weights. Conversely, `imgrec-py`'s binary is small (3.4KB) because it runs with a platform-supplied Python interpreter. However, the cost of starting an isolated Python interpreter and dynamically loading the function's libraries also gives the co-sandbox time to carry out initialization steps.

Figure 4.9 shows the start latency of `etcd` and `memcached` when running without and with co-sandboxes. In this benchmark, `etcd` initializes by restoring a 14MB snapshot [44], and `memcached` initializes with a warm restart [82] from a 200MB snapshot. `etcd` and `memcached` fetch their state from a σOS storage service. `etcd` starts 1.7× faster, and `memcached` starts

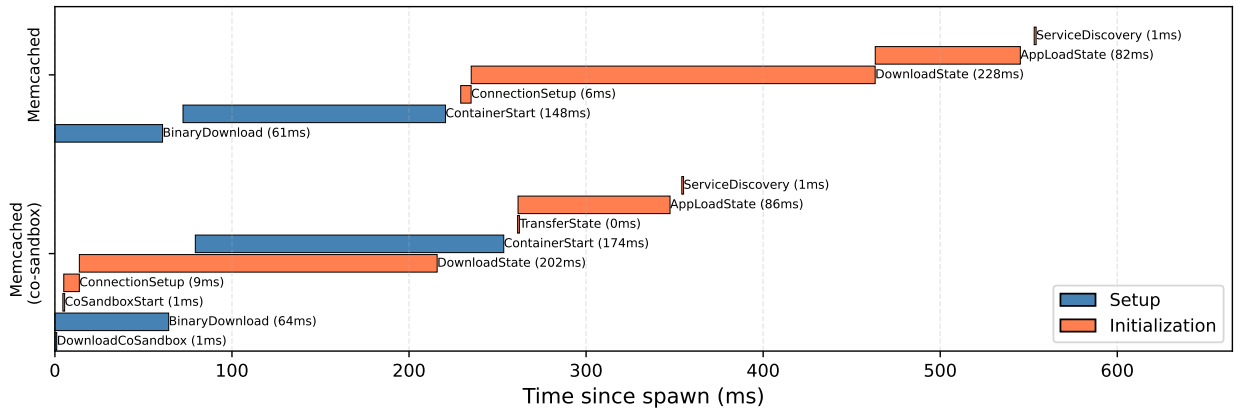

Figure 4.10: Start latency breakdown for `memcached` without and with co-sandboxes.

1.56× faster when using co-sandboxes. The start time speedup co-sandboxes provide to `etcd` and `memcached` is less than what they provide the previous serverless examples, because initializing these microservices requires reconstructing complex internal data structures from the downloaded state. Since `etcd`'s and `memcached`'s unmodified application binaries are not set up to share memory with the co-sandbox, the co-sandbox cannot carry out this initialization step for them.

In order to understand how co-sandboxes speed up `memcached` and other existing applications, we break down `memcached`'s start latency without and with co-sandboxes in Figure 4.10. The co-sandbox downloads the `memcached` snapshot from the storage service and transfers it to a small 180 line compatibility shim via the co-sandbox communication API. The shim writes the snapshot to the `memcached` container's local filesystem, and directs `memcached` to it. `memcached` then uses its warm restart [82] feature to parse the snapshot file and reconstruct its internal data structures. The extra copies induced by writing the snapshot to the container filesystem reduces the speedup co-sandboxes provide to `memcached`: the shim and `memcached` spend around 85ms exchanging state via the filesystem.

Summary. Co-sandboxes enable σOS to reduce end-to-end start latency for several existing serverless and microservice applications. Some applications benefit from co-sandboxes without any modifications, while others require modest changes.

4.6 Co-sandbox-native applications achieve additional speedup with co-sandboxes

Another goal of the co-sandbox API is to support efficient transfer of initialization results to the application. The evaluation of `memcached` and `etcd` in §4.5 shows that unmodified applications which load initialization state into complex internal data structures pay a penalty in start latency. This section uses two co-sandbox-native microservice applications, `vecdb` and `cached`, to explore how much of the lost start time can be recovered by restructuring applications around the co-sandbox result transfer API.

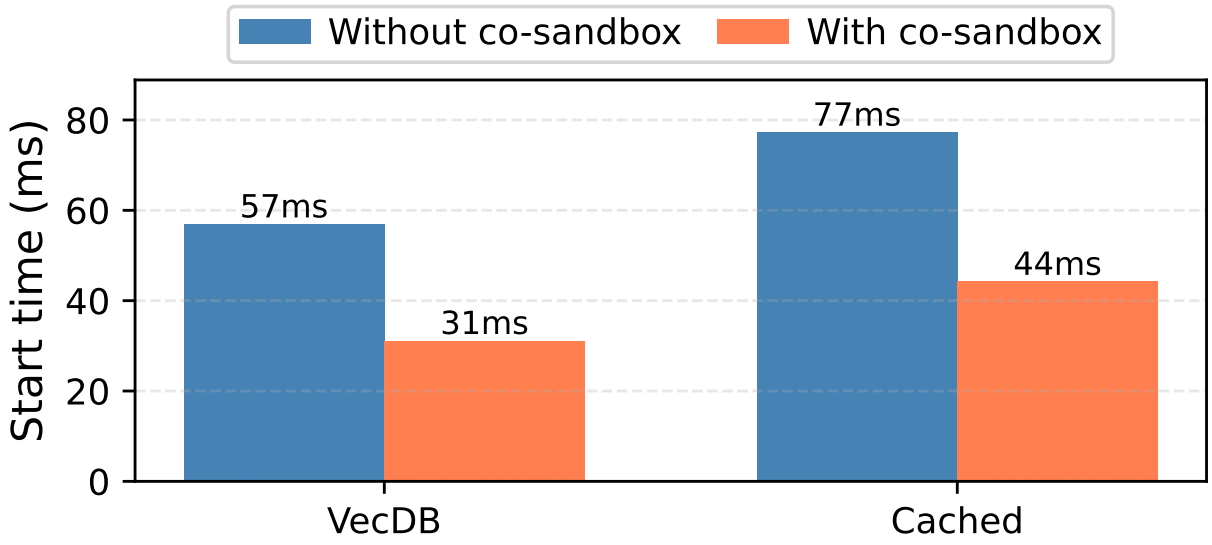

Figure 4.11: Start time of `vecdb` and `cached` in σOS without and with co-sandboxes.

`vecdb` and `cached` are two representative sharded soft-state microservices. In this benchmark, we cold-start a new `vecdb` instance and a new `cached` instance on a new machine in the σOS cluster. `vecdb` and `cached` initialize by downloading shards of their service's data assigned to them, which amount to 9.2MB and 15MB respectively. `vecdb` fetches its state from an in-memory key-value cache, and `cached` fetches its state from existing peer `cached`s.

`vecdb` and `cached` are written from scratch and designed to reconstruct their internal data structures with the co-sandbox result transfer API in mind. Both internally assign pointers to the memory region they share with their co-sandbox, enabling them to set up their state without copying their initialization data. Due to their optimized use of the co-sandbox result transfer API, a good result would show `vecdb` and `cached` experiencing start latency acceleration closer to that experienced by the serverless functions.

Figure 4.11 shows the start latency of `vecdb` and `cached` without and with co-sandboxes. `vecdb` starts 1.72× faster and `cached` starts 1.75× faster. Co-sandboxes speed up both applications' starts because the co-sandbox fetches the application's state in parallel with the container setup, runtime initialization, and binary download.

In order to understand why these co-sandbox-native applications achieve better speedup than `etcd` and `memcached`, we show a timeline which breaks down `cached`'s cold-start latency in Figure 4.12. The `cached` **(co-sandbox, no shared memory)** breakdown copies data when loading the initialization state from the co-sandbox into `cached`'s internal data structures. The additional memcopies slow down state transfer from the co-sandbox to `cached` from 1 millisecond to 13 milliseconds, eroding the speedup benefit of co-sandboxes by 39%. This is concordant with the start latency penalty experienced by `memcached` and `etcd`. When utilizing the shared memory features of the co-sandbox result transfer API, however, `cached` achieves much more speedup. This highlights the importance of efficiently transferring results with the co-sandbox result transfer API.

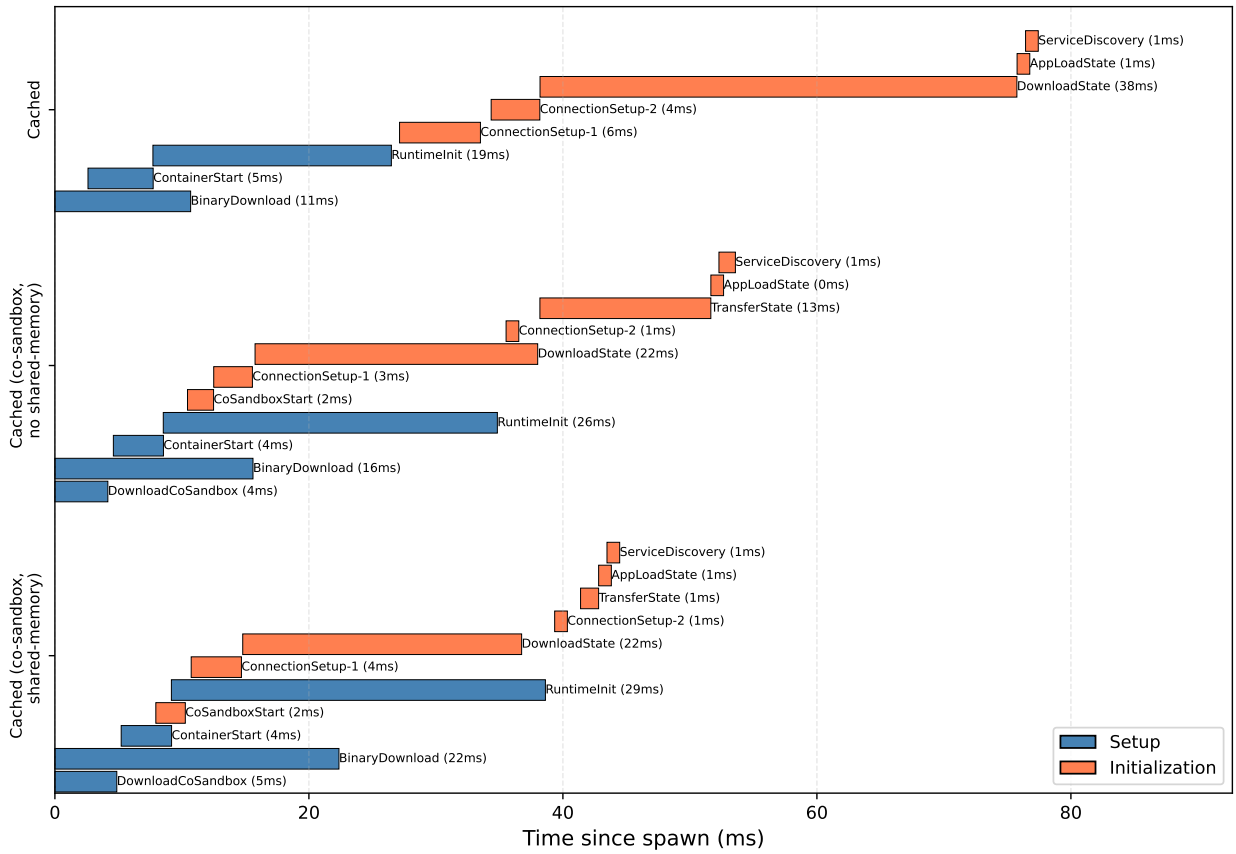

Figure 4.12: Start latency breakdown for `cached` without and with co-sandboxes, without and with shared-memory transfer to pass results from the co-sandbox to `cached`.

Summary. The co-sandbox result transfer API's design enables additional start latency speedup for σOS applications rewritten to benefit from its zero-copy implementation. This is particularly important for applications which reconstruct complex internal data structures during initialization.

4.7 Co-sandboxes can be small

One reason co-sandboxes can start quickly is that co-sandbox binaries can be much smaller than application binaries, which makes downloading a co-sandbox's binary fast. Figure 4.12 and Figure 4.10 show that `cached` and `memcached`'s 200KB co-sandbox binary downloads in 2-3ms, whereas downloading their multi-MB binaries takes 10s to 100s of milliseconds. Prior work shows that real-world cloud application binaries and container images range from several megabytes to hundreds of megabytes [13, 119] in size. In the event of a cold-start, network bandwidth and binary download latency become a limiting factor for performance. Co-sandbox binaries can be much smaller, on the order of a hundred kilobytes, because initialization is a small fraction of a full application's functionality.

Take `vecdb`, for example, a typical C++ microservice built on σOS. Table 4.3 shows a

Component	KB
Init RPC marshaling	13
Clnt RPC stubs	1,330
RPC stack	469
Logging	23
Perf monitoring	53
Srv RPC marshaling	23
Application impl	432
Exceptions	54
Misc	88
Total	2,485

Table 4.3: Contributors to the size of a stripped `vecdb` binary, as reported by Bloaty [58].

Component	`WASMEngine` KB	Co-sandbox KB
Init RPC marshaling		13
Clnt RPC stubs	1,330	
RPC stack	469	
Logging	23	
Perf monitoring	53	
Exceptions		54
Misc		88
Total	1875	155

Table 4.4: The co-sandbox API design shifts general-purpose components out of the co-sandbox and into the `WASMEngine`.

breakdown of the components which contribute to the size of the `vecdb` binary, as reported by Bloaty [58]. `vecdb` needs several client RPC stubs to operate, telemetry and performance monitoring, a full server-side RPC stack and support for marshaling client requests, and support for marshaling RPCs of several services it depends on and the associated client RPC stubs for each.

Unlike the full `vecdb` application, a co-sandbox written for `vecdb` need only include support for marshaling a small set of RPCs to fetch its soft-state from a single service. More importantly, the design of the co-sandbox API places many general-purpose components in the `WASMEngine`, allowing the co-sandbox binary to be small. Table 4.4 shows which components the co-sandbox API shifts out of the co-sandbox and into the `WASMEngine`.

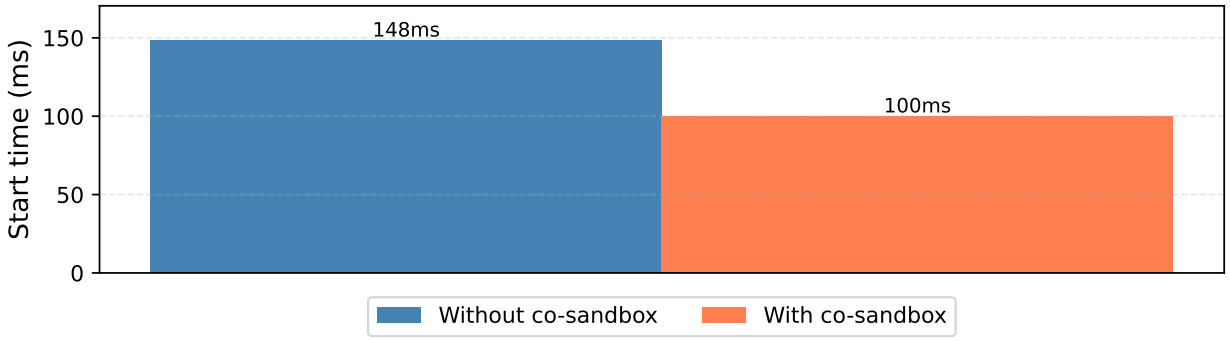

Figure 4.13: Start latency of `imgrec-py` restored from a Spice snapshot, without and with co-sandboxes.

Summary. Co-sandboxes can be small because they contain only a small subset of the functionality of a full microservice, namely initialization, and because the design of the high-level co-sandbox API allows the platform to supply much of the machinery required for co-sandboxes to run and communicate.

4.8 Co-sandboxes are complementary to platforms with fast setup

Co-sandboxes are a complementary technique to prior work which reduces setup and initialization costs. In order to demonstrate the benefit co-sandboxes provide to applications on these platforms we run `imgrec-py` with Spice [61], a state-of-the-art serverless snapshot and restore system. We evaluate on a bare-metal server with an Intel Xeon Gold 5420+ CPU with 56 logical cores.

We use Spice to generate a snapshot of an initialized `imgrec-py` function which has loaded all necessary Python libraries to perform inference. Then, we invoke `imgrec-py` on the same machine, and measure start latency from the point the function is invoked until it begins performing inference. The input is a 10MB input image downloaded from a σOS storage service. The co-sandbox version of `imgrec-py` uses a co-sandbox to fetch the input while Spice restores the function. A good result would show lower start latency for the co-sandbox version of `imgrec-py`, because the co-sandbox can start quickly and begin initialization while Spice sets up and restores the `imgrec-py` snapshot.

Figure 4.13 shows the results. By using co-sandboxes, `imgrec-py` is able to reduce its start latency by a factor of 1.48×. This is possible because its co-sandbox establishes connections to the storage service and fetches the input image in parallel with the Spice restore procedure. The connection setup and download, which take approximately 60ms, overlap with much of the 50ms Spice setup phase required to get to the first line of function code, and with some of the 36ms required to load the model weights from the snapshot into the inference library.

Spice represents a lower-bound on the start latency modern platforms can provide for serverless applications. Other systems, like Mitosis, incur additional network latency when starting functions, because they fetch their contents from a remote snapshot or Zygote. This

additional latency could be used by co-sandboxes to perform additional initialization steps and to further reduce start latency for these systems.

Summary. σOS' co-sandboxes are a complementary technique to prior work on fast setup and initialization. Using co-sandboxes together with these systems further improves cloud application start latency.

Chapter 5

Discussion

Chapter 4 demonstrates that σOS provides microservice and serverless applications with good performance, enables tenants to run fine-grained tasks, isolates and multiplexes resources between BE and LC applications, and accelerates `proc` startup with co-sandboxes. This section discusses some of the lessons we learned while working on σOS.

Expressiveness of the σOS API. In our experience, the σOS API supports high-performance implementations of many serverless and microservice applications. However, σOS and the σOS API are not a good fit for some applications. Databases and applications which need extremely low latency to durable storage are not a good fit to run on top of σOS. Our view is that durable storage should be a service offered by the provider infrastructure rather than an application built by the tenant.

The current σOS implementation doesn't support hardware accelerators like GPUs, because some GPU libraries require syscalls which are blocked by σcontainers' syscall filter. In principle, nothing prevents `proc`s from using accelerator API remoting techniques [45, 128] to enable accelerator access, but these techniques add some latency to accelerator computations which may be unacceptable for some applications.

Finally, the σEP API enables TCP connections between a tenant's `proc`s, because TCP's connection-oriented abstraction enables the σOS infrastructure to perform authentication once at connection setup and then hand off direct connection to the client and server `proc`s. Supporting isolation for connectionless protocols like UDP and QUIC may require modifications to the σOS API, or may require σOS to proxy communication and thereby introduce latency delays which may be too high for some applications.

Uniform RPC protocol. The σOS API establishes a uniform RPC protocol, including a session protocol, serialization format, and wire format for data transfer, between `proc`s and σOS services. This system-wide lingua franca enables several interesting optimizations between applications and the platform. Most notable among these is co-sandboxes. Co-sandboxes send RPCs to carry out initialization using a high-level RPC API which is implemented by `WASMEngine`. `WASMEngine` can implement the RPC stack because all σOS procs and services connect, authenticate, and exchange RPCs with a uniform protocol.

Other σOS extensions have leveraged the uniform RPC protocol to enable platform-driven transparent autoscaling of `proc`s, `proc` snapshotting, and transparent `proc` migration which

preserves open connections.

Dependency on Linux. σOS is built on Linux, which enables it to use Linux's well-tested isolation mechanisms and offer its rich ecosystem of debugging tools. However, dependency on Linux also comes at a cost. σOS inherits several scalability bottlenecks from Linux. For example, forking a Linux process can take hundreds of microseconds up to a millisecond. Since `procs` are implemented as Linux processes, they inherit this cost on every `proc` start. A bare-metal σOS implementation would enable a more fundamental redesign of the implementation of `procs`, and could leverage their restricted interface to provide even faster `proc` starts.

WebAssembly `procs`. The WebAssembly (WASM) ecosystem was still in its infancy when the first σOS prototype was implemented. The ecosystem has since evolved, and WASM is becoming a popular way to write serverless functions and even some microservices. σOS now supports `procs` implemented in WASM, which begs the question: why use a co-sandbox if you can just write the whole `proc` in WASM in the first place? §4.5 demonstrates that WASM `procs` benefit from co-sandbox-accelerated start, because realistic WASM `procs` have large binaries which limit cold-start performance. Co-sandboxes represent only a small subset of an application's functionality, and can thus be much smaller, cold-start faster than the WASM `proc`, and parallelize WASM `proc` initialization with setup.

General vs bespoke APIs. A key design goal of the σOS API is to provide a small set of primitives which supports a wide variety of cloud applications (microservices and serverless). However, some applications may require bespoke APIs to achieve good performance. For example, some applications need to express additional scheduling constraints such as gang scheduling, collocation with other computations, or specialized hardware requirements, to the infrastructure they run on.

One approach to support these applications is to write a layer on top of σOS which provides the functionality they need. §6.6 took this approach and implemented an additional layer on top of the σOS scheduling infrastructure to assign `procs` to machines based on the machine learning models they depended on. This layer avoids re-loading a model when it was already cached in-memory elsewhere in the σOS cluster.

Some applications cannot be easily supported in this way, and may need to run on a platform tailored to their specific needs in order to meet their performance requirements.

Fault-tolerance of the σOS API. Early versions of σOS [110] took inspiration from fault-tolerant serverless systems like Beldi [131] to provide strong guarantees at the API level. In particular, these prototypes enabled σOS servers to run in a Raft-replicated [89] mode in which failed RPCs were transparently retried against other replicas and the infrastructure guaranteed a total order over RPCs and exactly-once semantics. The original σOS implementation also made all `proc`'s scheduling and lifecycle state fault-tolerant in each realm's `etcd`.

These early prototypes offered poor performance, limited the granularity of `procs` and scalability of realms, and ultimately provided limited utility to most applications we built. Many cloud applications handle fault-tolerance at the application level, because application developers can design efficient failure recovery schemes using knowledge of the application's

behavior. For example, many MapReduce applications can be written such that mappers and reducers are idempotent and can be safely retried in the event of a suspected failure. These applications pay a heavy cost for redundant fault-tolerance mechanisms baked into the infrastructure. These costs are even more burdensome when failures and `proc` kills are rare, which aligns with our experience.

The design of σOS described in this thesis takes a more end-to-end [102] approach to API guarantees. Failed RPCs return errors to the client, servers and applications are responsible for ordering and replicating RPCs, and `proc`s which were `Spawn`-ed but had not started may be lost, so parent `proc`s need to handle `Spawn` errors with care. σOS provides building blocks and primitives to allow applications to achieve strong fault-tolerance guarantees, but makes them explicitly opt-in so only applications which need them pay the performance cost.

Reliance on open-source software. When we first prototyped σOS, we built everything from scratch. We configured Linux's IP tables and forwarding rules to build our own overlay networks, created overlay filesystems from scratch and effectively managed our own Docker-like "image" format, and wrote our own libraries to interact with Linux's `procfs` by hand. We ended up solving a *LOT* of boring problems that other people had already solved. However, building everything from the ground up *was* very instructive. We solved some interesting problems, and built a good sense of what issues were fundamental.

Finding the right balance between reusing existing software and writing it ourselves was important to get right. Later prototypes of σOS grew to depend on the open-source software ecosystem wherever possible, surgically cutting out minimal pieces which needed to be replaced with σOS-specific approaches.

Chapter 6

σOS extensions

I have had the great privilege to work with some exceptional Master's of Engineering and Master's of Science students who have extended σOS in several interesting ways. This chapter gives an overview of these extensions.

6.1 Support for Python `procs`

σOS initially supported only statically-compiled `procs`. Ivy Wu sought to add support for Python `procs` in her thesis [125]. In order to do so, she wrote a shim library in C which intercepted syscalls made by the Python interpreter when searching for dynamically imported libraries, translated them into downloads over the σOS API, and supplied the downloaded files via a per-realm overlay filesystem mounted into the `proc`'s σcontainer.

6.2 Accelerating dynamic language runtimes with zygote-based forking

Languages with dynamic runtimes, like Python, present two main challenges for the σOS `proc` model: 1) dynamic language runtimes are often slow to start, and 2) applications written in these runtimes require support for dynamic package imports. Nicolas Camenisch addressed these challenges in his thesis [19]. In order to speed up start times for dynamic language runtimes, Nicolas added a Zygote-based fork primitive to σOS and a zygote-aware cluster scheduler which optimistically tries to steer `Spawn`-ed `procs` to nodes with existing zygotes. In order to support dynamic imports, Nicolas built `pyenvd`, a per-machine daemon which creates virtual environments for `procs` and manages a package cache to lazily materialize Python `procs`' dependencies. Using these two techniques Nicolas enabled Python `procs` to start with single-digit millisecond latencies and reduce memory overhead by exploiting fork-based page sharing.

6.3 RDMA support

A major source of latency during a `proc`'s cold-start is due to the `proc`'s binary download. Typical Go `procs` we built range from 15-30MBs in size, and C++ `procs` are often 5-10MBs. Nathan Mustafa explored using RDMA to speed up `proc` binary downloads in σOS in his thesis [85]. By modifying the σOS peer-to-peer binary distribution infrastructure to operate over RDMA, he was able to reduce cold-start latency of `procs` by 25% and reduce the CPU usage per byte transferred by a factor of 3.

6.4 Burst-parallel fault-tolerant workloads

σOS provides each tenant with a realm: a fault-tolerant namespace backed by `etcd`. The realm provides several primitives which `procs` can use to build fault-tolerant applications, such as leader election, leases, watches, and some replicated storage for configuration files. However, burst-parallel workloads which run very fine-grained computations, such as some implementations of MapReduce, can stress the realm sufficiently to make it a performance bottleneck. Ryan Chang addressed this in his thesis [22] by re-implementing the σOS Watch API, used to monitor directory contents in the realm namespace, to improve its scalability and efficiency. He further optimized a fine-grained burst parallel version of MapReduce by building a high-throughput fault tolerant task server, which provides the abstraction of a fault-tolerant "bag of tasks." A workload orchestrator, like the MapReduce coordinator, can then use this to track the state of fine-grained mapper and reducer `procs` it `Spawns`, and recover incomplete tasks in the event of a crash.

6.5 Snapshot-restore in σOS

σOS `procs` written in languages with a runtime, like Golang, pay the runtime initialization cost every time they run. In Golang, this can add 15-20ms of latency to each `proc` invocation. In his thesis [114], Frederick Tang explored using Linux's Checkpoint/Restore in Userspace (CRIU) to snapshot σOS microservices immediately after they start, and restoring them on-demand from the snapshot. This allows `procs` to avoid the cost of initializing the Go runtime every time they are `Spawn`-ed. The σCRIU implementation leverages several optimizations to further-accelerate `proc` starts: snapshot compression, caching kernel metadata to improve CRIU's restore latency, and demand-paging of snapshots over the network during cold-start with working-set based page prefetching.

6.6 Scheduling ML inference tasks in σOS

σOS was originally written to support CPU workloads, and was not designed to run `procs` which need GPU access, such as ML workloads. In her thesis [79], Katie Liu explored supporting ML serving applications in σOS by exposing RayServe in the σOS namespace via an infrastructure-run proxy server. This thesis also explores adding a user-space scheduling

layer on top of the σOS cluster scheduler in order to colocate `procs` with proxies with cached models needed by the `proc`, in order to avoid redundantly loading models into cluster memory, and avoid paying the cost of loading a model every time a `proc` runs.

6.7 Multi-cloud computing in σOS

σOS was originally built in a single-cloud setting. However, the realm namespace's decentralized design makes it a natural fit to support workloads that span multiple clouds. In his thesis [24], Kevin S. Chen explored using σOS as a multi-cloud compute platform. This thesis extends σOS' realms to span multiple clouds, leveraging the sharded namespace design to enable high-performance coordination within a cloud, and cloud-aware `proc` scheduling. It also leverages σOS' API to allow uniform access to functionally equivalent services across clouds, which may differ in their application-facing APIs.

6.8 Evaluating σOS and Kubernetes for microservice and serverless workloads

A key point of comparison for σOS is Kubernetes, which is the industry-standard cluster management system for microservices. In order to understand whether σOS meets its goal, we needed to build equivalent apps on both σOS and Kubernetes and compare their performance. In his thesis [59], Yizheng He built the SocialNet DeathStarBench application on σOS as well as on Kubernetes. His thesis compares the experience of writing applications on both systems, as well as the performance isolation and absolute performance of applications running on each.

Chapter 7

Related Work

Compared to prior work, σOS's main contribution is unifying support for serverless and microservice tasks in a single, cloud-centric platform with unique support for network isolation through σEPs and for isolated `proc` fast starts through σcontainers and co-sandboxes. This section discusses related work that σOS builds upon along the dimensions below.

Fast isolation. Many techniques have been explored for fast start, since it is critical for burst-parallelism and transparent scaling of tenants' workloads. SAND [2] shares a single container among the serverless functions of a given application to ease local communication, and relies on process boundaries to isolate functions within the application. For many applications SAND's isolation is too weak. Consider an application which processes end-users' images with a serverless function invocation per image. In SAND, each image would be handled by a different Linux process in a shared container. If a malicious user could craft an image to exploit a bug in the application, they could take control of the Linux process handling the application's function invocation and corrupt or steal images of other users being handled by other Linux processes in the application's container. σOS makes this attack more challenging, as each image processing `proc` would run in a separate σcontainer with strong isolation.

SOCK [88] introduces "lean" containers specialized for serverless functions; they cannot communicate directly, which allows SOCK to avoid the cost of network isolation. σOS's σEPs allow `proc`s to communicate directly with low overhead and strong isolation.

Gvisor [54] reimplements much of Linux in user space while restricting the set of system calls made available by the underlying host. σOS σcontainers also restrict the set of system calls, to those needed to implement the σOS API. Other container systems like Cntr [115] and RunD [74] streamline containers, eliminate OS-level bottlenecks, and offer constrained container environments to enable fast container creation. XFaaS [100] starts serverless functions fast with less isolation.

Particle [116] amortizes costs of configuring network namespaces and overlay networks when launching batches of containers on a single node. σOS's σEPs don't use network namespaces or overlay networks, and allow quick start on cold and warm nodes with strong network isolation between tenants.

AWS Lambda uses MicroVMs [1] to provide stronger isolation than containers with less overhead than a full VM; functions of the same tenant may share a microVM [120]. AWS

Lambda targets serverless workloads and isn't suitable for microservice-style workloads, because Lambdas cannot communicate directly and cannot maintain long-term state (they may be terminated after 15 minutes).

Other lightweight virtualization techniques like LightVM [81], and serverless-oriented unikernels like Seuss [17] seek to provide VM-level isolation for multi-tenant workloads while keeping start times low. LightVM [81] can start a VM with a unikernel and devices in 4ms. However, LightVM's reported start time is not comparable to σOS's 1.8ms, because while σOS provides strong network isolation, LightVM's start time does not include the cost of establishing network isolation (i.e., creating and configuring veth devices, network namespaces, and overlays).

Several systems improve startup performance by sharing a single process to run mutually distrustful workloads. Faasm [106], Cloudflare Workers [30], and Sledge [51] rely on WASM's language-level isolation, whereas Lightweight Contexts [78] propose OS primitives to enable independent units of execution to share a process while preserving isolation.

Sidecar-less service meshes [26, 63] provide network isolation without per-container user-space proxies. However, they rely on overlay networks and network namespaces. σOS applications use σEPs, which provide networking isolation without these costs.

Fast application start. Application initialization, which traditionally begins after creating an isolated execution context (e.g., container or VM) and downloading the application's binary, is often another major bottleneck in fast start.

Catalyzer [41] and Spice [61] avoid re-initializing application runtimes on repeated invocations by snapshotting an initialized serverless function and restoring future invocations from the checkpointed application image. Mitosis [124] improves on Catalyzer's `sfork` by introducing `rfork`, which provides fast application start using remote fork over RDMA, reducing the need for caching. AFaaS [20] builds on Catalyzer by using trees of increasingly function-specific snapshot layers to skip some runtime initialization steps shared by multiple applications. Sabre [73] uses hardware-accelerated compression to efficiently manage snapshots. AWS Lambda [13] and FaasNET [119] reduce image fetch time with caching and efficient distribution.

Faasm and SEUSS [17] also speed up application-start by checkpointing a WASM runtime or a unikernel respectively . REAP [117] records and replays the working sets of functions to make starting from a snapshot faster. GroundHog [3] enables secure reuse of initialized containers by returning applications to a clean state between invocations using snapshot and restore. RainbowCake [127] reduces the cost of cold-starts by caching serverless functions at several setup steps: environment setup, language runtime initialization, and user deployment package loading. RainbowCake must still load the function's code package during a cold-start, and co-sandboxes can use this time to carry out application initialization steps.

The above systems target serverless applications and aren't suitable for microservice-style workloads because these systems don't provide direct communication or network isolation. σOS's co-sandboxes are complementary to these techniques. co-sandboxes can speed up applications which rely on snapshot and restore techniques by using the restore time to initialize the application.

Binary download. FaaSNet [119] accelerates container provisioning using a novel peer-to-peer download system to distribute container images, and AWS Lambda [13] uses multiple caching layers in the datacenter to accelerate container loading.

Poby [23] and Mitosis [124] use specialized network hardware to accelerate image downloads.

FaasCache [50] avoids cold-starts using keep-alives to keep warm functions up and running at a scale commensurate to predicted load. However, FaasCache does not eliminate cold-starts or binary downloads entirely.

Downloading application binaries is a fundamental cost which adds latency to application cold-starts. σOS's co-sandboxes allow the developer to use this time to overlap initialization costs before the application starts running.

State management. Faa$T [97] and Locus [94] manage a distributed cache for serverless functions. BlitzScale [130] uses datacenter GPU training network fabric to distribute application state when scaling up, and several systems [8, 10, 47, 72, 75, 76, 129] leverage the DAG structure of serverless workflows to optimize data movement and dynamically provision resources.

Nu [99], AIFM [98], PLASMA [103] enable providers to flexibly manage state by asking developers to write applications with new APIs.

σOS enables developers to specify how to load application state when applications start up using the co-sandbox API.

Single system image. σOS's naming system inherits the idea of transparent access across a cluster from single-system image distributed systems [25, 36, 86, 90, 93, 113] but σOS targets cloud computing and provides a single-system image per tenant. σOS extends Plan9's 9P protocol [60], which is widely supported[1], and which allows σOS to be administered from the Linux command-line. σOS extends 9P with σEPs, watches to wait for a file to be created or removed (inspired by Chubby [15], `etcd` [42], and ZooKeeper [62]), and RPCs for services that don't fit a file system interface. σOS uses `etcd` to implement the root of a tenant's name space.

Schedulers. σOS takes inspiration from prior work on schedulers to allow the σOS scheduler to quickly schedule best-effort `procs` and guarantee resources to latency-critical `procs`. Like the Kubernetes scheduler [14, 104, 118], σOS uses a centralized scheduler and resource requests to schedule long-running computations carefully. σOS takes ideas from the distributed schedulers of Ray [121] and Sparrow [91] to schedule BE `procs` quickly and scalably. σOS's two-scheduler design resembles Mercury's hybrid scheduler [69]. Like Apollo [12], σOS uses real-time resource utilization to inform scheduling decisions.

Improving serverless. AWS Step Functions [8] and Azure Durable functions [9] simplify coordination and sharing among serverless functions. Recent research has eased restrictions on serverless functions by proxying communication [46, 47], by shuffling data efficiently [2, 66, 92,

[1] Linux, Windows Subsystem for Linux (WSL), QEMU, and Gvisor support or use 9P.

94, 106], by resuming terminated functions [132], by allowing functions to communicate and maintain transient state [96], by providing exactly-once semantics despite re-execution [64, 68, 95, 131], by making function invocation fast [16, 30, 65, 84, 106], and by running functions close to the data and providing transactions [18, 71, 107].

σOS is suitable for both serverless and microservice tasks, since it provides fast start, communication among `proc`s, and support for stateful services. Like container orchestration systems, the σOS scheduler allows `proc`s to reserve resources for latency-critical microservice tasks, which serverless systems don't support. For example, although Faasm can run serverless functions from different tenants, developers cannot reserve resources for a faaslet that is long-running and latency critical and the Faasm scheduler runs all faaslets in round-robin, making it inappropriate for long-running, latency-critical microservices.

Chapter 8

Conclusion

This thesis presented a multi-tenant cloud operating system, σOS, that supports both microservices and serverless functions by combining the best features of container orchestration systems and serverless frameworks. σOS provides developers with a unified cloud-centric API that allows developers to structure their applications using fast-starting `proc`s and that provides a shared namespace per realm, which hides machine boundaries and allows `proc`s to easily communicate and coordinate. σOS can efficiently multiplex the `proc`s of different tenants on the provider's hardware and responds automatically to shifts in load. By restricting `proc`s to the σOS interface, σOS can use σcontainers to cold-start `proc`s rapidly (in 7.7 ms) with strong isolation. Finally, σOS' co-sandbox API allows application developers to push initialization steps into the σOS platform. σOS can use co-sandboxes to overlap application setup and initialization in order to accelerate end-to-end cloud application cold-starts.

Bibliography

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 419–434, 2020.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/akkus.

[3] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 398–415, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi:10.1145/3552326.3567503. URL https://doi.org/10.1145/3552326.3567503.

[4] Amazon. Serverless architectures with AWS lambda: Overview and best practices. https://aws.amazon.com/blogs/architecture/serverless-architectures-with-aws-lambda-overview-and-best-practices/, 2018.

[5] Amazon. Effectively building ai agents on aws serverless. https://aws.amazon.com/blogs/compute/effectively-building-ai-agents-on-aws-serverless/, 2025.

[6] Amazon. AWS lambda. https://aws.amazon.com/lambda/, 2025.

[7] Amazon. AWS simple queue service. https://aws.amazon.com/sqs/, 2026.

[8] AWS. Aws step functions. https://aws.amazon.com/tutorials/create-a-serverless-workflow-step-functions-lambda/, 2024.

[9] Microsoft Azure. Azure durable functions. https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=in-process%2Cnodejs-v3%2Cv1-model&pivots=csharp, 2024.

[10] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container

provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 153–167, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi:10.1145/3472883.3486992. URL https://doi.org/10.1145/3472883.3486992.

[11] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 121–134, Renton, WA, July 2019.

[12] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 285–300, USA, 2014. USENIX Association. ISBN 9781931971164.

[13] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9. URL https://www.usenix.org/conference/atc23/presentation/brooker.

[14] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14(1), 2016.

[15] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.

[16] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309769. doi:10.1145/2038916.2038932. URL https://doi.org/10.1145/2038916.2038932.

[17] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi:10.1145/3342195.3392698. URL https://doi.org/10.1145/3342195.3392698.

[18] Michael J. Cafarella, David J. DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. A polystore based database operating system (DBOS). In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB Workshops, Poly 2020 and DMAH 2020, Virtual Event, August 31 and September 4, 2020, Revised Selected Papers*, volume 12633 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2020.

[19] Nicolas Camenisch. Fork around and find out: Accelerating dynamic language runtimes in sigmaos via lightweight container forking, May 2026. URL https://pdos.csail.mit.edu/papers/nicolascamenisch-meng-eecs-2026-thesis.pdf.

[20] Xiaohu Chai, Tianyu Zhou, Keyang Hu, Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, Le Gao, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. Fork in the road: reflections and optimizations for cold start latency in production serverless systems. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-47-2.

[21] Xiaohu Chai, Tianyu Zhou, Keyang Hu, Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, Le Gao, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. Fork in the road: reflections and optimizations for cold start latency in production serverless systems. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-47-2.

[22] Ryan Chang. Optimizing sigmaos for efficient orchestration of fault-tolerant, burst-parallel workloads, May 2025. URL https://hdl.handle.net/1721.1/162546.

[23] Zihao Chang, Jiaqi Zhu, Haifeng Sun, Yunlong Xie, Kan Shi, Ninghui Sun, Yungang Bao, and Sa Wang. Poby: Smartnic-accelerated image provisioning for coldstart in clouds. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association. ISBN 978-1-939133-48-9.

[24] Kevin S. Chen. Icσos: σos for intercloud environments, May 2024. URL https://hdl.handle.net/1721.1/156569.

[25] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3): 314–333, March 1988.

[26] Cilium. Cilium service mesh. https://cilium.io/use-cases/service-mesh/.

[27] Google Cloud. What is function as a service (faas)? https://cloud.google.com/discover/what-is-function-as-a-service-faas?hl=en, 2026.

[28] Google Cloud. feedbackoverview of vertex ai. https://docs.cloud.google.com/vertex-ai/docs/start/introduction-unified-platform, 2026.

[29] Cloudflare. Workers sites: Extending the workers platform with our own serverless building blocks. https://blog.cloudflare.com/extending-the-workers-platform/, 2019.

[30] Cloudflare. Cloudflare workers. https://workers.cloudflare.com/, 2025.

[31] Cloudflare. Sandboxing ai agents, 100x faster. https://blog.cloudflare.com/dynamic-workers/, 2026.

[32] Ben Congdon. Corral. https://github.com/bcongdon/corral.

[33] Containerd. Kubernetes/containerd default seccomp filter. https://github.com/containerd/containerd/blob/36cc874494a56a253cd181a1a685b44b58a2e34a/contrib/seccomp/seccomp_default.go, 2024.

[34] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3464298.3476133. URL https://doi.org/10.1145/3464298.3476133.

[35] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. Dirigent: Lightweight serverless orchestration. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 369–384, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi:10.1145/3694715.3695966. URL https://doi.org/10.1145/3694715.3695966.

[36] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *Computer*, 24(11):34–44, November 1991. ISSN 0018-9162. doi:10.1109/2.116849. URL https://doi.org/10.1109/2.116849.

[37] Yuhan Deng, Akshay Srivatsan, Sebastian Ingino, Francis Chua, Yasmine Mitchell, Matthew Vilaysack, and Keith Winstein. Fix: externalizing network i/o in serverless computing. In *Proceedings of the 21st European Conference on Computer Systems*, EUROSYS '26, page 2260–2275, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722127. doi:10.1145/3767295.3769387. URL https://doi.org/10.1145/3767295.3769387.

[38] Docker. Docker. https://www.docker.com/, 2023.

[39] Docker. Docker swarms. https://docs.docker.com/engine/swarm/, 2023.

[40] Docker. Docker default seccomp filter. https://github.com/moby/moby/blob/master/profiles/seccomp/default.json, 2024.

[41] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi:10.1145/3373376.3378512. URL https://doi.org/10.1145/3373376.3378512.

[42] etcd.io. Etcd. https://github.com/etcd-io/, 2023.

[43] etcd.io. Etcd Raft. https://github.com/etcd-io/etcd/, 2023.

[44] etcd.io. Disaster recovery. https://etcd.io/docs/v3.6/op-guide/recovery/, 2025.

[45] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. Towards a machine learning-assisted kernel with lake. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 846–861, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi:10.1145/3575693.3575697. URL https://doi.org/10.1145/3575693.3575697.

[46] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[47] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 475–488, USA, 2019. USENIX Association. ISBN 9781939133038.

[48] Sadjad Fouladi, Brennan Shacklett, Fait Poms, Arjun Arora, Alex Ozdemir, Deepti Raghavan, Pat Hanrahan, Kayvon Fatahalian, and Keith Winstein. R2e2: low-latency path tracing of terabyte-scale scenes using thousands of cloud cpus. *ACM Trans. Graph.*, 41(4), July 2022. ISSN 0730-0301. doi:10.1145/3528223.3530171. URL https://doi.org/10.1145/3528223.3530171.

[49] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/fried.

[50] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi:10.1145/3445814.3446757. URL https://doi.org/10.1145/3445814.3446757.

[51] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: a serverless-first, light-weight wasm runtime for the edge. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery. ISBN

9781450381536. doi:10.1145/3423211.3425680. URL https://doi.org/10.1145/3423211.3425680.

[52] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. URL https://github.com/delimitrou/DeathStarBench.

[53] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017. doi:10.1109/MCC.2017.4250939.

[54] Google. Open-sourcing gvisor, a sandboxed container runtime. https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime, May 2018.

[55] Google. Cloud functions. https://cloud.google.com/functions, 2023.

[56] Google. Kubernetes. http://kubernetes.io/, 2023.

[57] Google. Kubernetes. https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/, 2023.

[58] Google. Bloaty: a size profiler for binaries. https://github.com/google/bloaty, 2025.

[59] Yizheng He. Evaluating sigmaos with kubernetes for orchestrating microservice and serverless applications, September 2023. URL https://hdl.handle.net/1721.1/152879.

[60] Eric Van Hensbergen. Grave robbers from outer space: Using 9p2000 under Linux. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, page 83–94, 2005.

[61] Ben Holmes, Baltasar Dinis, Lana Honcharuk, Joshua Fried, and Adam Belay. Rethinking process snapshots for near-warm serverless cold starts. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, Seattle, WA, July 2026. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/holmes.

[62] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.

[63] Istio. Introducing ambient mesh. https://istio.io/latest/blog/2022/introducing-ambient-mesh/.

[64] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi:10.1145/3477132.3483541. URL https://doi.org/10.1145/3477132.3483541.

[65] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi:10.1145/3445814.3446701. URL https://doi.org/10.1145/3445814.3446701.

[66] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi:10.1145/3127479.3128601. URL https://doi.org/10.1145/3127479.3128601.

[67] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. Serverless cold starts and where to find them. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 938–953, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi:10.1145/3689031.3696073. URL https://doi.org/10.1145/3689031.3696073.

[68] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. Executing microservice applications on serverless, correctly. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2023.

[69] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 485–497, USA, 2015. USENIX Association. ISBN 9781931971225.

[70] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/klimovic.

[71] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, Peter Bailis, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. Apiary: A dbms-integrated transactional function-as-a-service framework, 2023.

[72] Tom Kuchler, Pinghe Li, Yazhuo Zhang, Lazar Cvetković, Boris Goranov, Tobias Stocker, Leon Thomm, Simone Kalbermatter, Tim Notter, Andrea Lattuada, and Ana Klimovic. Unlocking true elasticity for the cloud-native era with dandelion. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 944–961, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400718700. doi:10.1145/3731569.3764803. URL https://doi.org/10.1145/3731569.3764803.

[73] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. Sabre: Hardware-Accelerated snapshot compression for serverless MicroVMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 1–18, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lazarev.

[74] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 53–68, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-27. URL https://www.usenix.org/conference/atc22/presentation/li-zijun-rund.

[75] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 782–796, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi:10.1145/3503222.3507717. URL https://doi.org/10.1145/3503222.3507717.

[76] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 57–72, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703942. doi:10.1145/3623278.3624755. URL https://doi.org/10.1145/3623278.3624755.

[77] Linux. Linux syscall table. https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl, 2024.

[78] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton.

[79] Katie Liu. Enabling efficient ml inference in sigmaos with model-aware scheduling, May 2025. URL https://hdl.handle.net/1721.1/162971.

[80] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An extensible shuffle architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 564–577, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365. doi:10.1145/3603269.3604848. URL https://doi.org/10.1145/3603269.3604848.

[81] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi:10.1145/3132747.3132763. URL https://doi.org/10.1145/3132747.3132763.

[82] Memcached. Warm restart. https://etcd.io/docs/v3.6/op-guide/recovery://docs.memcached.org/features/restart/, 2025.

[83] Microsoft. Azure functions. https://azure.microsoft.com/en-us/blog/introducing-azure-functions/, 2025.

[84] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

[85] Nathan Mustafa. Efficient rdma for sigmaos, May 2026. URL https://pdos.csail.mit.edu/papers/nathanmustafa-meng-eecs-2026-thesis.pdf.

[86] Roger Michael Needham and Andrew J. Herbert. *The Cambridge Distributed Computing System*. Addison Wesley, 1983.

[87] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[88] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/oakes.

[89] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[90] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *Computer*, 21(2):23–36, February 1988. ISSN 0018-9162. doi:10.1109/2.16. URL https://doi.org/10.1109/2.16.

[91] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi:10.1145/2517349.2522716. URL https://doi.org/10.1145/2517349.2522716.

[92] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi:10.1145/3318464.3380609. URL https://doi.org/10.1145/3318464.3380609.

[93] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3): 221–254, Summer 1995.

[94] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/pu.

[95] Sheng Qi, Xuanzhe Liu, and Xin Jin. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 314–330, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi:10.1145/3600006.3613154. URL https://doi.org/10.1145/3600006.3613154.

[96] Gyorgy Rethy. Process-as-a-service computing on modern serverless platforms. Master thesis, ETH Zurich, Zurich, 2022-11-23.

[97] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 122–137, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi:10.1145/3472883.3486974. URL https://doi.org/10.1145/3472883.3486974.

[98] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated far memory. In *14th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 20), pages 315–332. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ruan.

[99] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. Nu: Achieving Microsecond-Scale resource fungibility with logical processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1409–1427, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/ruan.

[100] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, Andrii Golovei, Pradeep Venkat, Andrew Mcfague, Dimitrios Skarlatos, Vipul Patel, Ravinder Thind, Ernesto Gonzalez, Yun Jin, and Chunqiang Tang. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 231–246, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi:10.1145/3600006.3613155. URL https://doi.org/10.1145/3600006.3613155.

[101] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984. ISSN 0734-2071. doi:10.1145/357401.357402. URL https://doi.org/10.1145/357401.357402.

[102] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in systems design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[103] Bo Sang, Pierre-Louis Roman, Patrick Eugster, Hui Lu, Srivatsan Ravi, and Gustavo Petri. Plasma: programmable elasticity for stateful cloud computing applications. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi:10.1145/3342195.3387553. URL https://doi.org/10.1145/3342195.3387553.

[104] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013. URL http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[105] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference, USENIX ATC*, pages 205–218, 2020.

[106] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shillaker.

[107] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. DBOS: A dbms-oriented operating system. *Proc. VLDB Endow.*, 15 (1):21–30, jan 2022. ISSN 2150-8097. doi:10.14778/3485450.3485454. URL https://doi.org/10.14778/3485450.3485454.

[108] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020. ISSN 2150-8097. doi:10.14778/3407790.3407836. URL https://doi.org/10.14778/3407790.3407836.

[109] Akshay Srivatsan, Yuhan Deng, Katherine Mohr, Emma Sudo, Sebastian Ingino, Francis Chua, and Keith Winstein. Continuation-centric computing with arca. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, Seattle, WA, July 2026. USENIX Association. URL https://www.usenix.org/conference/osdi26/presentation/srivatsan.

[110] Ariel Szekely. σos: Elastic realms for multi-tenant cloud computing, September 2022. URL https://hdl.handle.net/1721.1/147373.

[111] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 385–402, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi:10.1145/3694715.3695947. URL https://doi.org/10.1145/3694715.3695947.

[112] Ariel Szekely, Hannah Gross, Robert Morris, and M. Frans Kaashoek. Fast end-to-end cloud application cold-start with initscripts, 2026. URL https://arxiv.org/abs/2608.07358.

[113] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the Amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, December 1990. ISSN 0001-0782. doi:10.1145/96267.96281. URL https://doi.org/10.1145/96267.96281.

[114] Frederick Tang. Accelerating burst parallelism of sigmaos processes with criu, September 2025. URL https://hdl.handle.net/1721.1/164651.

[115] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, July 2018. USENIX Association. ISBN ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/thalheim.

[116] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 16–29, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi:10.1145/3419111.3421275. URL https://doi.org/10.1145/3419111.3421275.

[117] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi:10.1145/3445814.3446714. URL https://doi.org/10.1145/3445814.3446714.

[118] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM EuroSys Conference*, pages 18:1–18:17, Bordeaux, France, April 2015.

[119] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/wang-ao.

[120] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association. ISBN ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/wang-liang.

[121] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/cheng.

[122] Wasmer. Wasmer. https://wasmer.io/, 2025.

[123] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 121–136, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-42. URL https://www.usenix.org/conference/atc22/presentation/wei.

[124] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/wei-rdma.

[125] Ivy Wu. Interposing the syscall boundary: Transparent python execution in sigmaos, May 2025. URL https://pdos.csail.mit.edu/papers/wu-ivywu-meng-eecs-2025-thesis.pdf.

[126] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The true cost of containing: A gVisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association. URL https://www.usenix.org/conference/hotcloud19/presentation/young.

[127] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 335–350, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703720. doi:10.1145/3617232.3624871. URL https://doi.org/10.1145/3617232.3624871.

[128] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. Ava: Accelerated virtualization of accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 807–825, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi:10.1145/3373376.3378466. URL https://doi.org/10.1145/3373376.3378466.

[129] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL https://www.usenix.org/conference/nsdi23/presentation/yu.

[130] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Blitzscale: fast and live large model autoscaling with o(1) host caching. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation*, OSDI '25, USA, 2025. USENIX Association. ISBN 978-1-939133-47-2.

[131] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

[132] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 328–343, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi:10.1145/3419111.3421277. URL https://doi.org/10.1145/3419111.3421277.