Accelerating cloud application cold-start with initscripts

Ariel Szekely, Hannah Gross, Robert T. Morris, Frans Kaashoek
MIT

Abstract

Serverless functions are a popular way of deploying cloud applications. Because many of these functions are short-running and experience frequent cold-starts, start latencies often dominate their execution latency. Start latency can be broken down into two components: *setup* and *intitialization*. 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.

This paper contributes *initscripts*, which provide a scriptable interface for developers to specify their application's initialization routine to the cloud platform. The platform can then run an application's initscript and reduce application start latency by *overlapping* setup and initialization steps. Once the application is up and running, the initscript bootstraps the application by transferring initialization results to it.

Using initscripts, we were able to speed up cold-starts of several of the ServerlessBench [19] Python applications by an average 1.67× with no modifications to the application. Initscripts also speed up start times for a serverless image recognition workload by 1.81×, and off-the-shelf microservices like `etcd` and `memcached`.

1 Introduction

The convenience and cost-effectiveness of serverless computing has driven developers to refactor many cloud applications into serverless functions, and to opt for a serverless architecture for new cloud applications. Latency-sensitive user-facing websites written purely as serverless applications are now common [4, 14, 16], 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, 15, 18].

Many of these serverless functions are short-running and experience frequent cold-starts. As a result, start latency often dominates their execution latency [2, 21, 34, 37, 50, 59]. 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 this paper is to accelerate end-to-end cold-start latency.

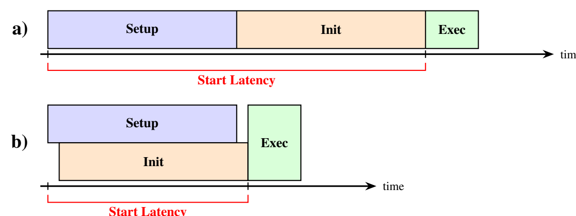

Figure 1: Cloud application start timeline, **a)** without initscripts, and **b)** with initscripts.

Start latency can be broken down into two components which we call *setup* and *initialization*, shown in part **a)** of Figure 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 [34, 53, 59] and initialization [12, 58] are comparable in length for many real-world applications: each takes O(100ms).

Prior work on reducing cloud application start time mostly focuses on reducing setup costs. Work on fast setup centers around lightweight isolation, specialized hardware and kernel modifications for fast binary downloads, and forking or snapshot-restore mechanisms [1, 7, 12, 33, 35, 51, 53, 59]. Some work has explored reducing initialization costs [20, 35, 52]. 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.

This paper proposes a missing programmatic interface for developers to specify application initialization to the cloud platform: *initscripts*. Initscripts allow developers to write initialization programs which the platform can run on their behalf. Platforms can run initscripts in parallel with the setup phase to *overlap* setup and initialization and reduce end-to-end start latency, as shown Figure 1 part **b)**. Once setup completes, the initscript bootstraps the application by transferring initialization results to it. Initscripts provide developers with a scriptable interface which allows developers to express a wide range of application initialization steps. For example, initscripts can establish connections to other services, issue RPCs and fetch state which can only be identified at runtime.

Initscripts must start with low latency in order to maximize the amount of overlap with setup. A key design challenge (challenge 1) in fast initscript start is keeping initscripts small. Small initscripts 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 initscript 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 initscript API to make it easy to speed up start times for existing cloud applications with few modifications.

Initscripts are implemented as WASM modules, allowing the platform to run them with strong isolation. To resolve challenge 1, initscript binaries are kept small, around 100KB, because the initscript host API is carefully designed to implement a small set of primitives which can be composed to run common initialization steps, namely establishing network connections and issuing RPCs. This high-level RPC interface allows the platform to implement an asynchronous RPC stack for the initscript and reduces initscript size.

To resolve challenge 2, the initscript 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 initscript and application container. The API's design enables applications to claim and use connections established by the initscript, and to read initialization state via shared memory.

Finally, to resolve challenge 3, the initscript API is designed to support off-the-shelf cloud applications with few-to-no modifications. RPC client libraries can support initscript-accelerated starts transparently to the applications which depend on them. Using this approach, we used initscripts to speed up start latency of several of the ServerlessBench [19] Python functions without any application code changes, and accelerate startup of two popular open-source microservices, `memcached` and `etcd`, with a thin compatibility layer.

We implement support for initscripts on σOS [53], a recent multi-tenant cloud Operating System which supports fast container creation and a uniform interface for a range of serverless and microservice tasks.

We use initscripts to accelerate cold-start latency of serveral unmodified serverless applications from ServerlessBench [19] by an average 1.67×, a WASM-based image recognition serverless function `imgrec-wasm` and its Python analogue `imgrec-py` by 1.81×, two unmodified real-world microservices, `memcached` and `etcd`, and two representative custom-built microservices, `cached` and `vecdb`. Initscripts cold-start onto new machines quickly, in 1-2ms including the cost of downloading and isolating the initscript. We also demonstrate that initscripts are complementary to existing techniques which reduce setup and initialization latency: initscripts speed up start latency for serverless functions which run on a state-of-the-art snapshot-restore system [32] by 1.48×.

The main contributions of this work are:

- Initscripts, a new interface for application developers to specify application initialization to cloud platforms, enabling overlap of setup and initialization (§3).
- The design of the initscript API, which keeps initscripts small and enables efficient transfer of initialization results to applications with few-to-no modifications (§3).
- An implementation of initscripts in σOS (§4).
- An evaluation of initscripts and how they reduce start latency for off-the-shelf and custom cloud applications, and in conjunction with state-of-the-art cold-start speedup techniques (§6).

2 Motivation

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 ServerlessBench [19]. We compare a WASM implementation (`imgrec-wasm`) and a Python implementation (`imgrec-py`) to understand how the application runtime and isolation affects setup costs.

In order to initialize, `imgrec-wasm` and `imgrec-py` connect to Amazon's Simple Queue Service (SQS) [6], 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 1 shows the order-of-magintude cost of each setup and initialization step of `imgrec-wasm` and `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

Phase	Step	WASM	Python
Setup	Binary 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 1: Setup and initialization steps of an Image Recognition serverless application `imgrec`, when implemented in WASM (`imgrec-wasm`) and Python (`imgrec-py`).

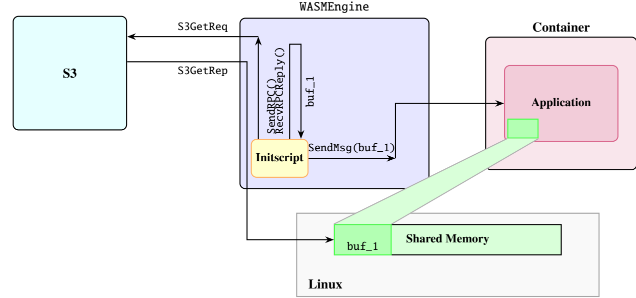

Figure 2: A initscript fetching initialization state and communicating it to its application via the `WASMEngine`. The initscript sends an RPC requesting the application's initialization state to S3 via `WASMEngine` using `SendRPC`, and blocks until the reply comes back using `RecvRPCReply`. `WASMEngine` writes the response from S3 directly into a shared memory region, and sends the buffer `buf_1` containing the reply to the initscript. The initscript forwards `buf_1` to the application, which fetches the state by mapping the shared memory region and reading `buf_1` directly.

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 it so separate initialization from the application so that the platform can overlap setup and initialization, and transfer the initialization to the application once it is 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 ouptut destination once it has successfully claimed a task from SQS. The following section describes how we achieve this goal using initscripts.

3 Design

This section describes the design of initscripts and their API. Initscripts are WASM modules run by `WASMEngine`, a per-machine daemon deployed by the cloud platform (§3.1). Developers write initscripts with the initscript API (§3.2), which is designed to keep initscripts small so they can be downloaded and start fast. The initscript result transfer API (§3.3) enables initscripts to efficiently bootstrap the application once it has started running. Its design enables existing applications to use initscripts with few modifications. This section also discusses the initscript developer workflow (§3.4), and changes to APIs used to start cloud applications (§3.5).

3.1 Initscripts: scriptable WASM init routines

Multi-tenant cloud platforms that support initscripts need to start them quickly and with strong isolation. In order to do so, initscripts are implemented as WASM modules executed by `WASMEngine`, a per-machine daemon run by the platform. Because initscripts are WASM modules, they don't have access to a full Linux environment with standard syscalls. Instead, they use the initscript API to communicate with the outside world and carry out initialization steps. Figure 2 illustrates how a initscript uses the initscript API to interact with `WASMEngine` and fetch initialization state from S3. The following section describes the design of the initscript API.

3.2 Initscript API

The primary design goal of the initscript API, shown in Table 2, is to keep initscripts small (challenge 1) by shifting as much functionality as possible into the platform. In service of this goal, the initscript API is high-level and provides a small set of functions for connection establishment and RPCs: `Connect`, `SendRPC`, and `RecvRPCReply`. The high-level design of the API reduces the compiled binary size of initscripts because `WASMEngine` can supply an implementation of functionality needed to interact with cloud services such as client-side RPC and networking stacks, support for long-lived sessions, and authentication.

`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 on the connection named by `conn_fd`. `RecvRPCReply` blocks until a response is received, and returns a buffer containing the response to the initscript.

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

`Exit` terminates the initscript and reports the exit `status` to the platform.

3.3 Initscript result transfer API

The initscript result transfer API, shown in Table 3, is designed to efficiently pass initialization results from the

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 2: API developers use to write initscripts.

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

Table 3: Initscript result transfer API that developers use to bootstrap a initscript's applications once the application is up and running.

initscript to the application (challenge 2), and to make it possible for existing applications to use initscripts with few modifications (challenge 3).

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

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

After the application starts, its initscript library maps the shared memory region `WASMEngine` set up for it. The application calls `RecvMsg` to receive buffers passed to it by the initscript, and `WASMEngine` returns corresponding buffers backed by the shared memory region. `WASMEngine` synchronizes access to RPC results, because long-running RPCs initiated by the initscript 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 initscript, 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 initscript-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 initscript. For example, the initscript 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 initscript via `WASMEngine`. `WASMEngine` uses the message and connection IDs to synchronize access to RPC results and ensure both the initscript and application see a consistent transcript of initialization RPCs and results. `WASMEngine` prevents the application from getting ahead of the initscript and issuing RPCs that the initscript has yet to send or re-issuing RPCs that are already in progress.

3.4 Developer workflow

Developers can write initscripts in any language that compiles to WASM. Developers upload their initscript 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 initscript. This can be used, for example, to communicate function inputs or credentials for the initscript to use when sending RPCs and establishing sessions to other services.

3.5 Cloud API changes

Cloud providers can support initscripts by augmenting existing APIs. For example, application launch and serverless function invocation APIs that specify a function or applica-

tion container ID can be augmented to specify a initscript to run.

4 Implementation

We implemented initscripts on σOS [53], a recent cloud operating system built on Linux which supports applications with fast container creation times. The remainder of this section discusses the details of the new σOS components which implement the initscript API (§4.1), modifications to σOS APIs that developers use to spawn tasks (§4.2), and implementation details of the initscripts we wrote to support our applications and evaluation (§4.3).

4.1 `WASMEngine`

The `WASMEngine` implements the initscript API (Table 2) and makes the results of initscript RPCs available to new application instances via the initscript result transfer API (Table 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 initscripts using the Wasmer WebAssembly runtime [57].

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

`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 initscript. `WASMEngine` implements a simple protocol over the pipe to block microservice `RecvMsg` requests in the event until the initscript sends the corresponding `SendMsg`, and vice versa. In the event that the initscript 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 initscript.

4.2 σOS API modifications

We augmented the σOS Spawn API include a serialized initscript binary for the `proc` and an input byte slice for the initscript. Additionally, we added support for resource reservations for initscripts.

In order to support unmodified Linux microservices like `memcached` and `etcd`, we added a GVisor isolation backend for σOS procs. We also added support for Python applications with dynamic imports in order to support the ServerlessBench applications and `imgrec-py`, and a new isolation backend to run WASM `procs` like `imgrec-wasm`. For the comparison to Spice [32], we added Junction [24] as an isolation backend as well.

Additionally, we added support for WASM `procs` and applications isolated by Spice [32] in order to run `imgrec-wasm` and the snapshot-restore benchmarks.

4.3 Initscript implementation details

We wrote initscripts for the applications in Rust. initscripts import a library which declares external functions corresponding to the initscript API (Table 2) and initscript result transfer API (Table 3).

```
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: ServerlessBench Python storage API.

5 Case-studies: applications using initscripts

We ported several off-the-shelf cloud applications to use initscripts, including all of the ServerlessBench [19] 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 initscript-native analogue to `memcached`. The remainder of this section discusses the experience of adapting existing applications (§5.1) to benefit from initscript-accelerated cold-starts and writing new (§5.2) initscript-accelerated applications from scratch.

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

Developers can use initscripts to accelerate cold-start for existing applications by writing a simple compatibility layer which communicates with the initscript, and makes the fetched state available to the application (§3.3).

For example, ServerlessBench'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. We added initscript support to all the ServerlessBench Python functions with no modifications to the applications themselves. Instead, we changed the client library implementation to intercept the initialization RPCs and communicate with the initscript rather than send the RPCs to the remote storage service. The initscript passes the stored results of the initialization RPCs back to the client library, which returns the results to the application.

Figure 4 shows the implementation of the download function in the storage library with initscript support. The initscript 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

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

Figure 4: ServerlessBench storage client library modified to use initscripts 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 initscript.

```python
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 5: The unmodified ServerlessBench ImageRecognition application benefits from initscript support in the storage client library.

and speed up their application's start time with a initscript can do so by using the same naming system defined by the storage library developer.

For example, the ServerlessBench ImageRecognition application shown in Figure 5 first downloads its input image and then downloads its model weights. The application developer can make use of initscript support in the storage library by writing a initscript which fetches the initialization state in the same order, as shown in Figure 6. Since the application developer followed the naming scheme defined by the storage library, the unmodified Image Recognition will transparently benefit from initscript-accelerated start time. The initscript will run while the platform carries out the application's setup, and will bootstrap the application once it is up and running.

One reason initscripts 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 retrieve data. By modifying the RPC libraries to take advantage of the initscript 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.

```go
imgReq := &S3GetReq{
  Bucket: "my-bucket",
  Key: initSArgs[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 6: The ImageRecognition application's initscript uses the storage library-defined result naming scheme to fetch and pass on the application's initialization state.

5.2 Fast cold-start for initscript-native applications

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

`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` [43].

`vecdb` uses initscripts to accelerate scaling up. When a new `vecdb` instance starts, its initscript 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 initscript to fetch its shards from existing peer `cached`s.

Figure 7 shows the implementation of the initscript which fetches `vecdb`'s shards and sends them to the new `vecdb` instance. Figure 8 shows how the newly running `vecdb` communicates with its initscript to receive its state and load the state into its internal data structures.

The message-passing API which the initscript and `vecdb` use to communicate initialization results, `SendMsg` and `RecvMsg`, are implemented using shared-memory set up by `WASMEngine`. This allows the initscript to issue RPCs to fetch

```
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 7: vecdb initscript fetches the shard assignment and sends it to the microservice application.

```
// Block until initscript 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 8: vecdb microservice application receives state from its initscript.

`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.

6 Evaluation

The goal of initscripts is to reduce end-to-end cold-start latency with few changes to applications. In order to determine whether initscripts meet their goal, this evaluation answers the following questions:

1. How much do initscripts speed up off-the-shelf applications' start latency? (§6.1)
2. How much more do initscript-native applications improve their start latency? (§6.2)
3. How small can initscripts be? (§6.3)
4. Are initscripts able to speed up start latency on platforms with fast setup? (§6.4)

Experimental setup. We evaluate initscripts 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. For the comparison to state-of-the-art snapshot-restore systems, we evaluate on a bare-metal server with an Intel Xeon Gold 5420+ CPU with 56 logical cores.

We evaluate several of the Python ServerlessBench [19] applications. We also evaluate a serverless image recognition application `imgrec-wasm` built in Rust and compiled to WASM, an equivalent version `imgrec-py` written in Python, and 4 microservice applications, `etcd`, `memcached`, `vecdb`, and `cached`. We use GVisor containers to isolate `etcd`, `memcached`, fast-starting σcontainers to isolate `cached` and `vecdb`, the Wasmer runtime to isolate `imgrec-wasm`, and σcontainers to isolate the Python applications.

6.1 Initscripts reduce off-the-shelf application start latency

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

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 initscripts decrease the start-latency of the applications, because they overlap setup and initialization.

Figure 9 shows the start latency of several ServerlessBench applications. Functions marked by a (*) download their inputs and model weights from S3. Initscripts speed up their start-latency by an average of 1.67×, because initscripts 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 initscripts 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 initscripts due to variability in download times of the packages they import.

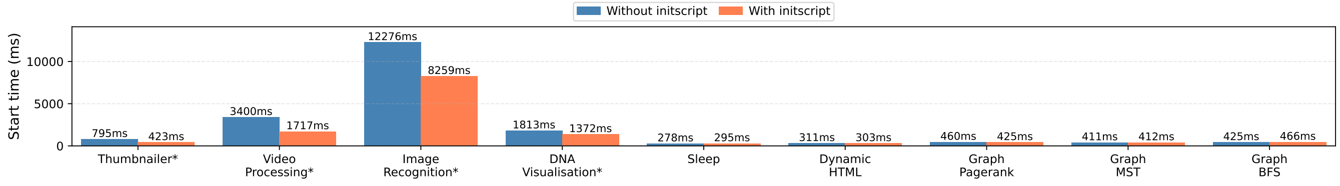

Figure 9: Start time of the Python ServerlessBench benchmarks in σOS without and with initscripts.

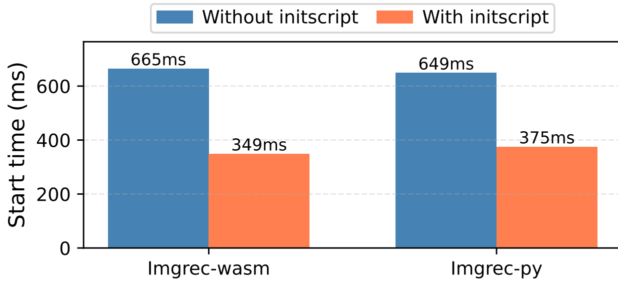

Figure 10: Start time of `imgrec-wasm` and `imgrec-py` in σOS without and with initscripts.

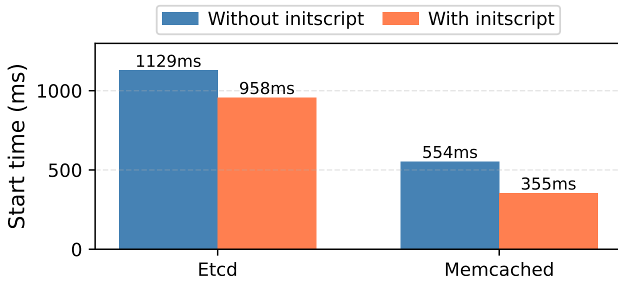

Figure 11: Start time of `etcd` and `memcached` in σOS without and with initscripts.

Figure 10 shows the start latency of `imgrec-wasm` and `imgrec-py` when running without and with initscripts. Both implementations benefit from initscript-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.

Initscripts 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 its large (45.6MB) WASM binary gives its initscript 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 initscript time to carry out initialization steps.

Figure 11 shows the start latency of `etcd` and `memcached` when running without and with initscripts. In this benchmark, `etcd` initializes by restoring a 14MB snapshot [22], and `memcached` initializes with a warm restart [42] from a 200MB snapshot. `etcd` and `memcached` fetch their state from a storage service. `etcd` starts 1.18× faster, and `memcached` starts 1.56× faster when using initscripts. The start time speedup initscripts 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 initscript, the initscript cannot carry out this initialization step for them.

In order to understand how initscripts speed up `memcached` and other existing applications, we break down `memcached`'s start latency without and with initscripts in Figure 12. The initscript downloads the `memcached` snapshot from the storage service and transfers it to a small 180 line compatibility shim via the initscript 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 [42] 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 initscripts provide to `memcached`: the shim and `memcached` spend around 85ms exchanging state via the filesystem.

Summary. Initscripts are able to reduce end-to-end start latency for several existing serverless and microservice applications. Some applications benefit from initscripts without any modifications, while others require modest changes.

6.2 Initscript-native applications achieve additional speedup with initscripts

Another goal of the initscript API is to support efficient transfer of initialization results to the application. The evaluation of `memcached` and `etcd` in §6.1 shows that unmodified applications which load initialization state into complex internal data structures pay a penalty in start latency. This section uses two initscript-native microservice applications, `vecdb` and `cached`, to explore how much of the lost start time can be recovered by restructuring applications around the initscript result transfer API.

`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 down-

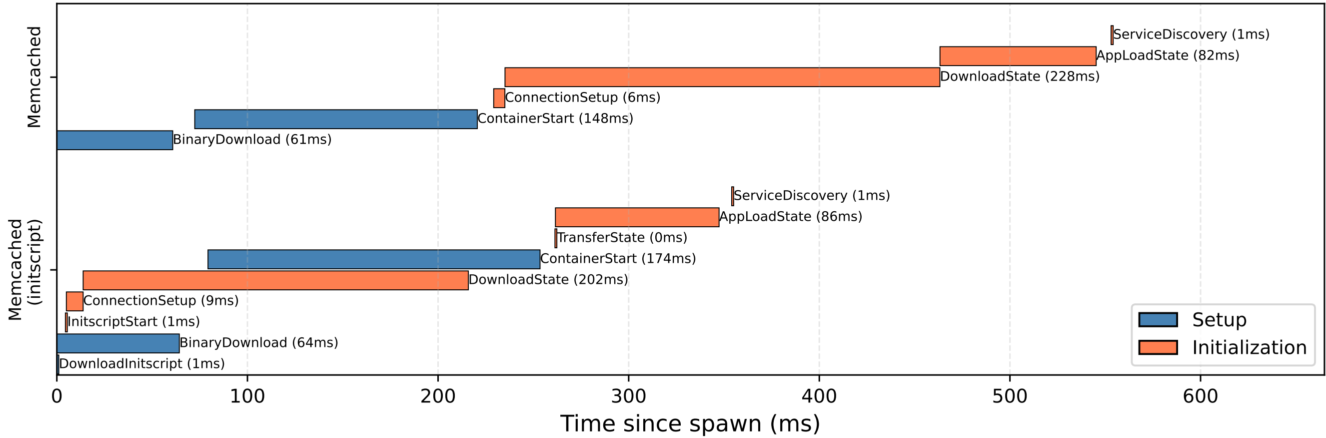

Figure 12: Start latency breakdown for `memcached` without and with initscripts.

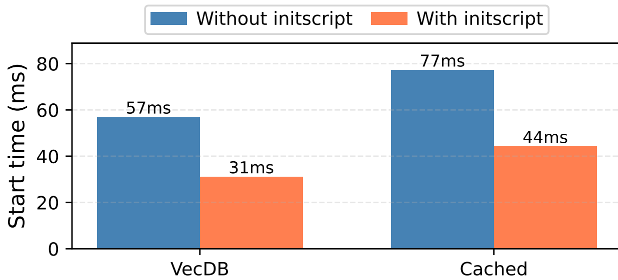

Figure 13: Start time of `vecdb` and `cached` in σOS without and with initscripts.

loading 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 initscript result transfer API in mind. Both internally assign pointers to the memory region they share with their initscript, enabling them to set up their state without copying their initialization data. Due to their optimized use of the initscript result transfer API, a good result would show `vecdb` and `cached` experiencing start latency acceleration closer to that experienced by the serverless functions.

Figure 13 shows the start latency of `vecdb` and `cached` without and with initscripts. `vecdb` starts 1.72× faster and `cached` starts 1.75× faster. Initscripts speed up both applications' starts because the initscript fetchees the application's state in parallel with the container setup, runtime initialization, and binary download.

In order to understand why these initscript-native applications achieve better speedup than `etcd` and `memcached`, we show a timeline which breaks down `cached`'s cold-start latency in Figure 14. The **`cached` (initscript, no shared memory)** breakdown copies data when loading the initialization state from the initscript into `cached`'s internal data structures. The additional memcopies slow down state transfer from the initscript to `cached` from 1 millisecond to 13 millseconds, eroding the speedup benefit of initscripts by 39%. This is concordant with the start latency penalty experienced by `memcached` and `etcd`. When utilizing the shared memory features of the initscript result transfer API, however, `cached` achieves much more speedup. This highlights the importance of efficiently transferring results with the initscript result transfer API.

Summary. Efficiency enabled by the initscript result transfer API design allows applications to derive additional start latency speedup when designed to use the API. This is particularly important for applications which reconstruct complex internal data structures during initialization.

6.3 Initscripts can be small

One reason initscripts can start quickly is that initscript binaries can be much smaller than application binaries, which makes downloading a initscript's binary fast. Figure 14 and Figure 12 show that `cached` and `memcached`'s 200KB initscript 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 [9, 56] in size. In the event of a cold-start, network bandwidth and binary download latency become a limiting factor for performance. Initscript 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 shows a breakdown of the components which contribute to the size of the `vecdb` binary, as reported by Bloaty [29]. `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

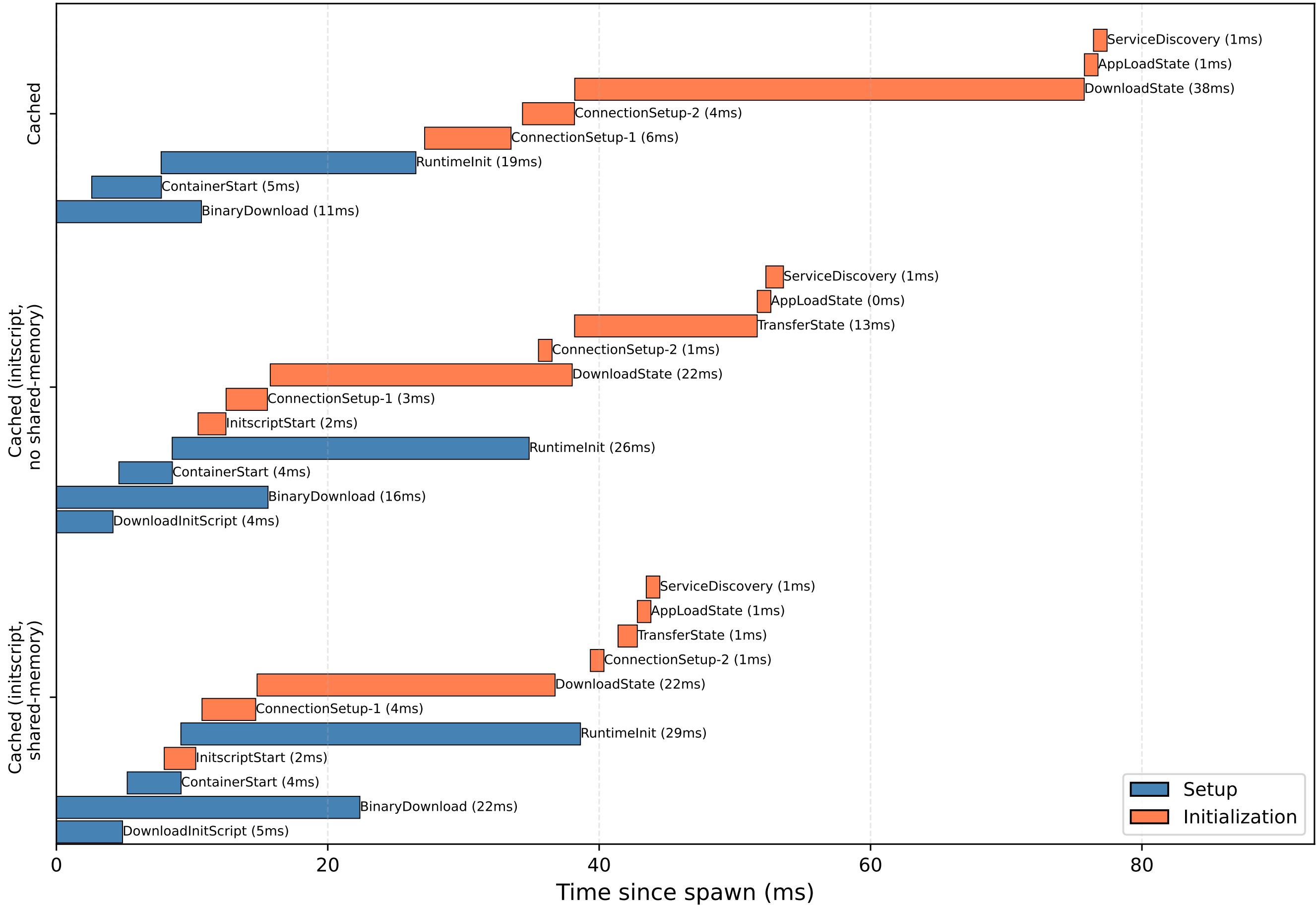

Figure 14: Start latency breakdown for `cached` without and with initscripts, without and with shared-memory transfer to pass results from the initscript to `cached`.

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: Contributors to the size of a stripped `vecdb` binary, as reported by Bloaty [29].

support for marshaling RPCs of several services it depends on and the associated client RPC stubs for each.

Unlike the full `vecdb` application, a initscript 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 initscript API places many general-purpose components in the `WASMEngine`, allowing the initscript binary to be small. Table 5 shows which components the initscript API shifts out of the initscript and into the `WASMEngine`.

Component	`WASMEngine` KB	Initscript 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 5: The initscript API design shifts general-purpose components out of the initscript and into the `WASMEngine`.

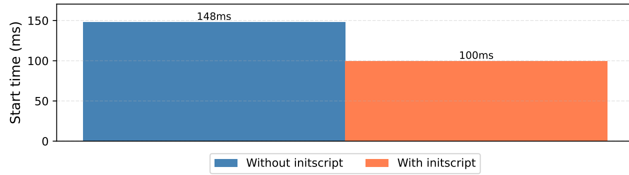

Figure 15: Start latency of `imgrec-py` restored from a Spice snapshot, without and with initscripts.

Summary Initscripts 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 initscript API allows the platform to supply much of the machinery required for initscripts to run and communicate.

6.4 Initscripts are complementary to platforms with fast setup

Initscripts are a complementary techinque to prior work which reduces setup and initialization costs. In order to demonstrate the benefit initscripts provide to applications on these platforms we run `imgrec-py` with Spice [32], a state-of-the-art serverless snapshot and restore system.

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 initscript version of `imgrec-py` uses a initscript to fetch the input while Spice restores the function. A good result would show lower start latency for the initscript version of `imgrec-py`, because the initscript can start quickly and begin initialization while Spice sets up and restores the `imgrec-py` snapshot.

Figure 15 shows the results. By using initscripts, `imgrec-py` is able to reduce its start latency by a factor of 1.48×. This is possible because its initscript 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 initscripts to perform additional initialization steps and to further reduce start latency for these systems.

Summary Initscripts are a complementary technique to prior work on fast setup and initialization. Using initscripts together with these systems further improves cloud application start latency.

7 Related Work

The main contribution of this work is initscripts, a new technique to reduce end-to-end cloud application cold-start time by overlapping setup and initialization. A great deal of prior work has sought to improve cold-start for applications by reducing setup costs, and some researchers have explored pushing initialization operations, like fetching input data, into the platform. Initscripts are complementary to this work, and allow platforms to further reduce application cold-start time. Developers can use initscripts' scriptable interface to specify application initialization to the platform separately from the rest of the application. In exchange, the platform can reduce start latency by overlapping setup and initialization.

This section describes prior techniques and how they relate to the goal of accelerating cloud application start latency.

Isolation. MicroVMs [60], lightweight virtualization techniques like LightVM [41], and serverless-oriented unikernels like Seuss [10] seek to provide VM-level isolation for multi-tenant workloads while keeping start times low.

Systems like GVisor [27], SigmaOS [53], Cntr [54], Particle [55], and RunD [37] streamline containers, eliminate OS-level bottlenecks, and offer constrained container environments to enable fast container creation.

Several systems improve startup performance by sharing a single process to run mutually distrustful workloads. Faasm [51], Cloudflare Workers [17], and Sledge [26] rely on WASM's language-level isolation, whereas Lightweight Contexts [40] propose OS primitives to enable independent units of execution to share a process while preserving isolation.

XFaaS [48] starts serverless functions fast with less isolation.

Faster isolated execution environment creation reduces setup costs and is a complementary approach to accelerating application cold-start with initscripts. By overlapping initialization and setup, initscripts enable developers to speed up cold-starts even more. Additionally, some lightweight isolation techniques sacrifice security guarantees in order to reduce setup costs. Initscripts make setup time useful by overlapping it with initialization, which reduces the pressure on minimizing setup time. This may enable platforms to recover strong isolation while preserving fast application start times.

Binary download. FaaSNet [56] accelerates container provisioning using a novel peer-to-peer download system to distribute container images, and AWS Lambda [9] uses multiple caching layers in the datacenter to accelerate container loading.

Mitosis [59] and SigmaOS [53] leverage the host OS' demand-paging to lazily download application binaries.

Poby [13] and Mitosis [59] use specialized network hardware to accelerate image downloads.

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

Runtime initialization. One approach to reducing setup costs is to avoid initializing application runtimes on repeated invocations. Several prior works, like Catalyzer [21] and Spice [32], take this approach using snapshot and restore techniques to cache pre-initialized runtimes. AFaaS [11] builds on Catalyzer by using trees of increasingly function-specific snapshot layers to skip some runtime initialization steps shared by multiple applications. GroundHog [3] enables secure reuse of initialized containers by returning applications to a clean state between invocations using snapshot and restore.

Some approaches use specialized hardware to share snapshots across machines. Sabre [36] uses hardware-accelerated compression to efficiently manage snapshot, while Mitosis [59] leverages RDMA hardware and a customized kernel to implement a remote-fork primitive.

FaasCache [25] avoids cold-starts using keep-alives to keep warm functions up and running at a scale commensurate to predicted load.

Initscripts are complementary to these techniques which reduce the setup cost of runtime initialization. As demonstrated in this paper's evaluation (§6.4), initscripts can speed up applications which rely on snapshot and restore techniques by using the restore time to initialize the application.

Multi-container application platforms. Kubernetes [28] supports applications structured as multiple containers, tied together into a Pod. Applications in a pod can communicate, are physically co-located and share file system state.

Kubernetes supports InitContainers [30], which start before an application's main container. InitContainers can be used to store secrets out of reach of the application container, include debugging tools not built into the application container, delay application start, and prepare the filesystem for the application. Kubernetes Sidecars [31] are similar to InitContainers, but do not have to run to completion before the main application container starts.

Initscripts use different isolation technology from the main application container, which allows initscripts to start quickly. The initscript API keeps their compiled binaries orders-of-magnitude smaller than most container images, which makes them fast to download. Additionally, the API which initscripts use to communicate with the application enables passing of initialization results, like established connections and downloaded application state, between the initscript and application.

State management. Faa$T [45] and Locus [44] manage a distributed cache for serverless functions. BlitzScale [62] uses datacenter GPU training network fabric to distribute application state when scaling up, and several systems [7, 8, 23, 35, 38, 39, 61] leverage the DAG structure of serverless workflows to optimize data movement and dynamically provision resources.

Nu [47], AIFM [46], PLASMA [49] enable providers to flexibly manage state by asking developers to write applications with new APIs.

Initscripts enable developers to specify how to load application state when applications start up.

8 Conclusion

This paper presented initscripts, a novel abstraction which can be used to accelerate cloud application cold-start. initscripts allow cloud platforms to overlap application setup and initialization, and transfer initialization results to the application efficiently using the initscript result transfer API.

Acknowledgments

References

[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 simple queue service. https://aws.amazon.com/sqs/, 2026.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

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

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

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

[19] 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.

[20] 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.

[21] 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`.

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

[23] 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.

[24] 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`.

[25] 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`.

[26] 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`.

[27] 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.

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

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

[30] Google. Init containers. `https://kubernetes.io/docs/concepts/workloads/pods/init-containers/`, 2025.

[31] Google. Sidecar containers. `https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/`, 2025.

[32] 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`.

[33] 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`.

[34] 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`.

[35] 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`.

[36] 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`.

[37] 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.

[38] 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.

[39] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. Dataflower: Exploiting the dataflow 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] Radu Weiss, Noah Meyerhans, James Turnbull, and Alexandra Iordache. Firecracker design document. https://github.com/firecracker-microvm/firecracker/blob/main/docs/design.md, May 2019.

[61] 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.

[62] 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.