Process-as-a-Service: FaaSt Stateful Computing with Optimized Data Planes

Marcin Copik
ETH Zürich

Alexandru Calotoiu
ETH Zürich

Rodrigo Bruno
INESC-ID, IST, ULisboa

Roman Böhringer
ETH Zürich

Torsten Hoefler
ETH Zürich

Abstract

Fine-grained and ephemeral functions power many new applications that benefit from elastic scaling and pay-as-you-use billing model with minimal infrastructure management overhead. To achieve these properties, Function-as-a-Service (FaaS) platforms disaggregate compute and state and, consequently, introduce non-trivial costs due to data locality loss, complex control plane interactions, and expensive communication to access state. We revisit the foundations of FaaS and propose a new cloud abstraction, **cloud process**, that retains all the benefits of FaaS while significantly reducing the overheads that result from the disaggregation. We show how established operating system abstractions can be adapted to provide powerful granular computing on dynamically provisioned cloud resources while building our **Process as a Service (PraaS)** platform. PraaS improves current FaaS by offering data locality, fast invocations, and efficient communication. PraaS delivers invocations up to 32 times faster and reduces communication overhead by up to 99%.

1 Introduction

In less than a decade, Function-as-a-Service (FaaS) has established itself as one of the fundamental cloud programming models. Users invoke stateless and short-running functions and benefit from pay–as–you–use billing while cloud providers gain more efficient resource usage and opportunities to reuse idle hardware [18, 68, 80]. Serverless functions have been used in a wide spectrum of areas, ranging from web applications, media processing, data analytics, machine learning, to scientific computing [9, 28, 36, 50, 52, 54]. Although FaaS has achieved remarkable success in reducing the costs of burstable stateless computations, its adoption to stateful applications such as data analytics and machine learning is currently hampered by the limitations of its execution model [17, 36, 38, 58].

Take distributed machine learning [13, 26, 36, 69, 71], a popular serverless workload as an example. In this workload, each invoked function must compute new gradients using a subset of data. While the stateless nature of FaaS simplifies deployment and resource management, combining the new weights and using them in the next round of invocations incurs major performance overhead, as *in each round of update*, the output must be written and read from external cloud storage. Furthermore, each new round requires an FaaS invocation that goes through the entire cloud control plane.

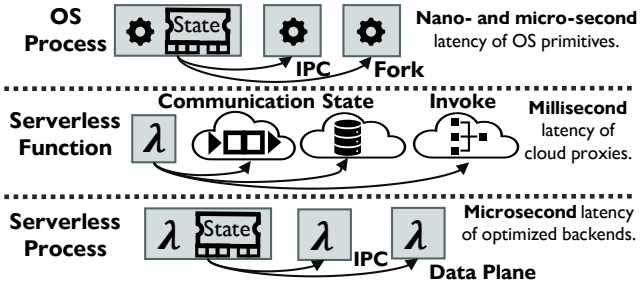


Figure 1: FaaS cannot offer the same versatility and efficiency of computing as operating system processes.

Researchers have addressed this lack of state management in ephemeral functions [38, 58] by proposing ephemeral storage solutions [34, 39, 66, 78] that functions can use the keep state. However, all of these are based on *non-serverless* infrastructure which requires manual infrastructure management and is not elastic. In addition to an increase of cost of serverless systems [34, 58], serverless applications that heavily depend on non-serverless infrastructure to keep state are only as elastic as the least elastic component. As a consequence, serverless *functions* running in today's platforms cannot achieve the performance and efficiency comparable to that of classical multi-processing [16, 33, 38].

The separation of data and computing in serverless is fundamentally inefficient and cannot be resolved by composing FaaS with additional remote cloud systems (Fig. 1). Instead, we introduce a new abstraction: the **cloud process**. The process extends the semantics of functions with a well-defined **session state** and more efficient **inter-process communica-**

	IaaS	CaaS	PraaS [This Paper]	FaaS
Computation Unit	Virtual machine	Container	Process	Function
External Interface	SSH, TCP, HTTP, RPC, RDMA	SSH, TCP, HTTP, RPC	HTTP, TCP	HTTP
Lifetime	Months	Days, hours	Minutes, hours	Seconds
State Duration	Persistent	Transient	Transient	Ephemeral
State Location	Local disk, memory	Memory, cloud storage	Memory, cloud storage	Cloud storage
Provisioning	Manual, minutes	Semi-automatic, secs	Automatic, msecs	Automatic, msecs
Compute Resources	Persistent	Persistent	Ephemeral	Ephemeral
Billing	Provisioned	Provisioned	Pay-as-you-go	Pay-as-you-go

Figure 2: Evolution of computing platforms in the cloud - *PraaS* enables **state persistence** for **ephemeral workers.**

tion while retaining the ephemeral and serverless nature of FaaS. Similarly to OS processes using threads for concurrent computations, *cloud processes* launch functions within a single shared environment (here, a function invocation would be equivalent to an thread OS). Unlike a FaaS function, the state of the process is not removed upon eviction of the sandbox, but instead **swapped** to persistent storage. **Process-as-a-Service (PraaS)** comes with a few simple extensions to FaaS, which allow us to serve scalable processes without the need to overhaul existing serverless architectures. *PraaS* is heavily inspired by classical OS design and transfers concepts that have stood the test of time into the context of granular cloud computing. By implementing responsibilities traditionally associated with operating systems (*resource sharing* between processes, *portable* communication interface, and *state* management), *PraaS* is a step towards a distributed cloud computing OS that provides a better balance between the performance of persistent allocations and the elasticity of ephemeral workers (Fig. 2). The new model allows us to introduce new solutions to three of the most significant limitations of FaaS: the lack of efficient and portable **communication**, inefficient **data plane** coupled with control logic, and the absence of consistent and low latency **state**.

Inter-Process Communication Current FaaS platforms restrict peer–to–peer communication forcing users to rely on storage-based communication. This approach is expensive, high latency, lacks a portable API. To encapsulate network transport between two ephemeral entities in serverless (§2.1), we take inspiration from the indirect IPC methods that use mailboxes to store message data [63]. In *PraaS*, we define a simple yet powerful messaging interface based only on two operations: **send** and **receive**. The message is *sent* by a function to a mailbox associated with another process, and then it is retrieved by using the **receive** operation. Data is transferred between two concurrently executing functions (*message passing*), to trigger functions (*invocation*), and to a function that will be executed at some point in the future (*mailboxes*), effectively replacing storage-based communication [36, 50]. A single interface across platforms covers all types of function–to–function communication while hiding transport protocol details: shared memory, TCP, QUIC, or RDMA.

Data Plane Serverless functions are predominantly short-running [60] and, as a consequence, the relative overhead of the multi-step routing logic is high (§2.2). Slow invocations prevent wider adoption of parallel and granular computing [17, 43, 44], and functions cannot fully benefit from the availability of fast network transport when using the control plane. Instead of applying optimizations to decrease control overheads, we *remove the control plane overheads from the data path entirely* [53] by exposing a direct communication channel to the process and submitting the invocation payload as a message over the **process data plane** – effectively creating separate control and data paths in the system.

Durable State The statelessness of FaaS enables automatic scalability and resource provisioning but significantly limits the efficiency of applications (§2.3). While cloud operators retain function containers to minimize cold startups, this uses limited memory resources to hold function states across invocations [60]. However, functions cannot rely on this resource because ephemeral containers can be removed anytime. Thus, users must resort to saving state in an external store after each invocation. *PraaS* strikes a new balance between serverless and traditional stateful and server-based applications by providing **durable state** attached to functions in a process. This disaggregation of computing and storage allows one to manage resources independently, but the state is retained close to compute resources, improving data locality and startup times.

We implemented *PraaS* atop AWS Fargate, a commercial black-box system of serverless containers, and Knative, an open-source Kubernetes-based serverless platform. The new system consists of a dedicated control plane, a client library, and a process runtime that can be deployed in any container or virtual machine. We demonstrate that *PraaS* can scale as fast as serverless functions while reducing the latency of invocation by up to 32 times and communication overhead by up to 99%.

2 Motivation

Function-as-a-Service (FaaS) enhanced clouds with a fine-grained and elastic programming model, and FaaS has found

its way to the major cloud systems. Functions are stateless, and invocations cannot rely on resources and data from previous executions. Instead of using persistent and user-controlled virtual machines and containers, function instances are dynamically placed in cloud-managed sandboxes, e.g., containers and lightweight virtual machines [5]. A *cold* invocation requires allocation of a new sandbox that significantly increases invocation latency [17, 48]. Subsequent *warm* invocations achieve a lower latency by reusing existing sandboxes. Therefore, cloud systems employ complex and sophisticated retention and pre-warming strategies [19, 49, 64, 67], trading off higher memory consumption for faster executions. In addition, flexible pay-as-you-go billing is another significant advantage of serverless systems: users are charged only for computation time and resources used. However, along with the benefits serverless computing provides, it does have some disadvantages, such as high communication costs, higher latency due to complex control planes, and poor locality of data due to the non-existence of local state [17, 23, 38, 72].

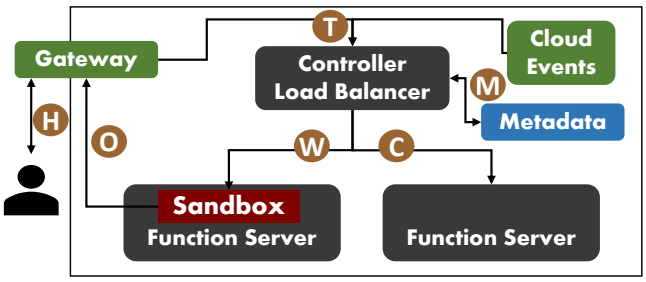


Figure 3: **FaaS control plane:** each invocation includes the management overhead (M).

2.1 Serverless Communication

Communication in FaaS has always been constrained. Serverless computing has been adopted to support distributed applications and workflows at the cost of complex and domain-specific optimizations in scheduling and communication [8, 16, 45, 77]. State-of-the-art solutions implement the communication with cloud storage, which increases latency, costs, and application complexity [36, 38, 45, 54]. Although direct network communication between functions could alleviate performance problems, functions do not offer the abstractions needed to implement communication between functions with a dynamic lifetime. The message–passing paradigm cannot be applied directly to ephemeral functions, as the worker lifetime is not defined and functions would require *mailboxes* to store messages. Furthermore, in typical FaaS deployments, functions operate behind NAT and cannot accept incoming connections. Connections can be established with the help of hole punching [25, 27], but it is a complex process that applies only to two active functions simultaneously. *Cheap and scalable communication is necessary to build parallel and distributed applications, but serverless programming models lack an efficient fabric and interface for communication.*

Storage Type	1B	1 MB	10 MB
Persistent storage (AWS S3)	10.3	20.4	102.4
Key-value storage (AWS DynamoDB)	34	n/a	n/a
In-memory cache (AWS ElastiCache)	0.9	24.6	84.6

Table 1: Access time [ms] to remote cloud storage from an AWS Lambda Function with 2 GiB RAM.

2.2 Serverless Data Plane

Modern FaaS systems implement the dynamic placement of function executors with a centralized routing system (Fig. 3) [5]. It includes an abstraction of a REST API and a gateway with a persistent network address and uses an HTTP connection (H) to hide the selection and allocation of function executors. Invocations are *triggered* by internal and external events (T). The input is forwarded to the central management (M) responsible for authorization, allocation of resources, and routing to the selected server. In AWS Lambda, the control logic is responsible for authorizing requests, managing sandbox instances, and placing execution on a cloud server [5]. In OpenWhisk [1], the critical path for the function execution is even longer. Each invocation includes a front-end web server, controller, database, load balancer, and a message queue [59]. Finally, the input data is moved to a warm (W) or cold (C) sandbox, and the function returns the output through the gateway (O).

The many steps of control logic add double–digit millisecond latency to each invocation [5, 15] and require copying user payload multiple times, even though subsequent invocations reuse the same warm sandbox when available. The overhead of the control plane can dominate the execution time and is much higher than the network transmission time needed to transfer the input arguments [17]. The long and complex invocation path is even more visible in distributed applications and serverless workflows that often invoke functions with large payloads. Alternative approaches include manual provisioning, reusing function instances, decentralized scheduling, and direct invocations [6, 14, 18, 65, 76], but these optimizations do not offer a solution generalizable to all FaaS platforms.

Every serverless invocation incurs overheads of management and data copying across cloud services, even though the placement of the function does not change in many warm invocations. Control logic should be removed from the critical path.

2.3 Serverless State

The stateless nature of functions makes them easy to schedule, but places significant constraints on the usability of FaaS systems. Computing resources are allocated with ephemeral memory storage that cannot be guaranteed to persist across invocations. Since many applications require the retention of state between invocations, *stateful* functions place their state in remote cloud storage [34, 66, 78]. While the function's state is located in storage far away from the compute resources, fetching and updating the state adds dozens of milliseconds of latency to the execution (Table 1), resulting in significant performance overhead [34, 78]. In the FaaS model, removing remote storage access from the data path is impossible.

Restricting serverless to stateless functions puts unnecessary constraints on user applications and reduces data locality. ***Stateful*** *functions help mitigate some of these challenges, but their efficient implementation is hindered by the current FaaS model.*

FaaS can be much more than just a platform for irregular and lightweight workloads, and the serverless programming model aligns well with requirements for massive parallel and granular computing [43, 44]. Nevertheless, serverless systems must first overcome critical limitations: complex **control plane** involved in every invocation, lack of a fast and transient **state**, and expensive storage-based **communication**.

3 Serverless Processes

The lack of state and data movement in functions made serverless simple, but resulted in major performance and usability limitations (§2). We address these limitations with the new concept of a cloud **process**, a natural extension of serverless functions. Conceptually, we lift the OS process to the cloud and enhance it with the pay-as-you-go billing model of serverless. Processes are dynamically allocated on abstracted cloud resources, like functions they execute in fine-grained and isolated environments, and their status is determined by the user.

The new process model overcomes the limitations of existing systems in three areas. (1) *Inter-process communication* defines data movement in terms of processes and removes the dependency on external storage, enabling direct communication between ephemeral workers (§3.1). (2) A process is accompanied by a *data plane* that supports fast function invocations from multiple users (§3.2). (3) Thanks to the disaggregation of computing and memory resources, the decoupled and transient process *state* now has well-defined durability semantics (§3.3). With the process, we build *PraaS*, a new programming and execution model (§4).

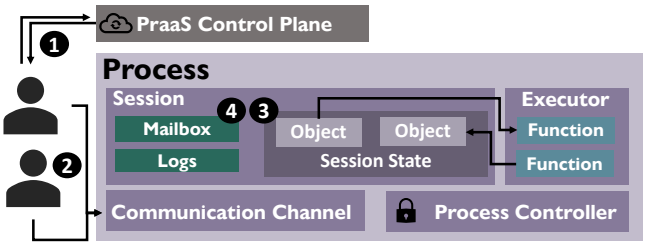


Figure 4: **The process model in *PraaS***: ephemeral functions are executed in a *processs* with shared and transient state and communication channels for quick user access.

3.1 Process Model with Communication

In direct comparison to a function, the process contains several new components to support the data movement: data plane communication channel, memory that stores the state, and a mailbox (Fig. 4). Each process instance is associated with a unique *session* to encapsulate the state of the process. When a process is allocated by the cloud control plane, it is assigned the identifier and a user-defined amount of memory. A communication channel to the user is opened with the first invocation to transmit the input and output data directly (❶). Subsequent invocations can bypass the control plane and use the *data plane* (❷). Functions use data stored by previous function invocations in the session state (❸). The mailbox (❹) handles invocations (§4.3) and communication between processes (§4.2). This storage is allocated within the session to provide reliability; messages persist until they are read by the intended recipient.

We define two messaging routines that implement all communication tasks handled by the process components. A `receive` has two required parameters only - the sender identifier and message name - and returns the contents of a message with the given name if such exists. We define two special identifiers `SELF` and `ANY` to support intra–process communication and receiving messages from an arbitrary sender. A `send` takes three standard arguments: the identifier of the target process, message name, and the content of the message. Both routines accept a set of optional flags to support copy and sharing semantics that vary between programming languages. These routines are optimized to transmit binary data as efficiently as possible and hide all details of the underlying network transport.

3.2 Computing Using the Data Plane

In the cloud process, a function is invoked every time a payload message arrives with a tag that describes the invocation request. In particular, functions are invoked by sending payload from the user over the data plane. Conceptually, a function invocation is similar to an allocation of a thread in a running program. A new thread starts working with a fresh stack but can still access the process memory. We leave it

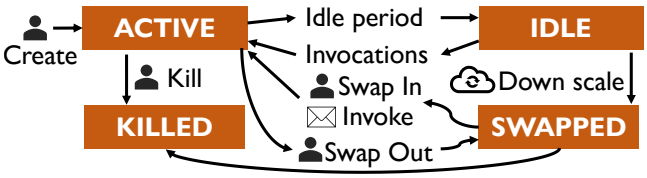


Figure 5: The life cycle of a cloud process in *PraaS*. Process status changes in reaction to user operations (), cloud operator actions (), and explicit invocation messages ().

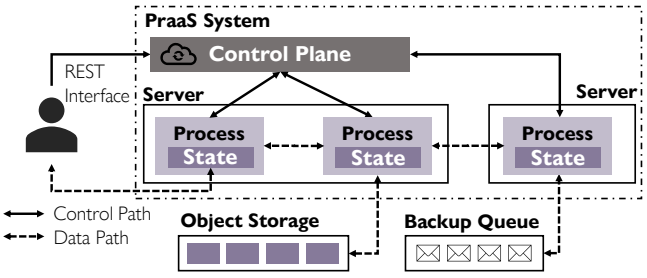


Figure 6: Platform architecture of *PraaS*.

up to the implementers to decide if a function executes in a dedicated OS process or a thread within the main OS process.

Sessions allow the process to handle workloads associated with different users and workflows in a single process instance. Functions can access memory only within their session, and the process can handle multiple invocations simultaneously but is bound by the memory and compute limitations of the underlying sandbox. Cloud providers can limit the number of simultaneous invocations an individual process can start, for example, by setting the limit relative to the memory the session has allocated. Larger workflows are supported by distributing the workload across multiple processes and implementing communication between session states (§4.2).

The controller receives invocation messages to start user functions. Functions queued beyond the upper threshold set by the cloud provider wait until resources become available. Furthermore, the controller uploads data plane invocation metrics to the control plane. When the control plane sends an eviction notice, or the user closes the data plane connection, the controller blocks pending invocations, waits for existing invocations and swaps the session state to cloud storage.

3.3 Scalability with State

Processes are serverless; all allocation decisions are made by the cloud operator, and users have no control over it. Like in FaaS, the allocation is not persistent, the lifetime of a process is controlled by the cloud, and it can be removed at any point. The session state in a process enables the persistence of user data and execution of stateful functions without the need for manual state management by the user. However, processes have a *transient* session state that must always be locally available to ensure minimal access latency, but it does not perish once the sandbox is removed. The state cannot restrict cloud providers from scaling resources transparently like in FaaS. To that end, we extend the FaaS function model with a new state of being *swapped out* (Fig. 5). By introducing a persistent swap, we can remove the statelessness restriction from FaaS while not adding any new limitations to the serverless allocation model. A swapped process can be reactivated later through a function invocation and an explicit request of the user.

Scaling Up New processes are allocated on-demand via an explicit allocation request to the cloud control plane. The fundamental assumption behind our process is that it never scales beyond a single server, simplifying management. Instead, users are encouraged to allocate more processes to handle the increased load. Active communication channels do not prohibit cloud operators from performing load balancing and consolidation, since processes can be migrated between machines. Clients with an active connection to a migrating process receive a packet with migration details and can establish a connection to the new communication channel. This design decision does not introduce additional complexity into writing serverless functions, since data exchange between two functions always uses the same interface, regardless of whether the communication is intra- or inter-process.

Scaling Down Upon eviction of a process, the container is terminated, server resources are released, and the session state is *swapped out* to persistent cloud storage. Processes are swapped out explicitly by users and implicitly by the control plane after a period of inactivity, according to a configured down-scaling policy. When scaling down, the process does not accept any new invocation requests, and the state, together with unread messages, is written to the store. If a session is no longer required, the user can completely remove the state information from the cloud storage. Swapping guarantees data persistence in the presence of *intentional failures*, that is, when the resource is removed by the cloud provider. For unintentional failures, when the machine crashes unexpectedly, we guarantee only the persistence of the last snapshot.

4 PraaS: Process–as–a–Service

Using the new process model introduced above, we now apply proven system design concepts (Table 2), and present **Process–as–a–Service**, a new execution model and serverless platform (Fig. 6). In *PraaS*, processes are grouped to create scalable applications spanning multiple server machines (§4.1) We show how a simple communication interface with only two functions allows for portable inter-process communication (§4.2), data plane invocations (§4.3), and incorporating state into functions (§4.4).

PraaS Concept	Inspiration
Application	Operating system.
Process	POSIX process model.
Function	Thread in a process.
Session State	POSIX process memory.
Session Persistence	Swapping memory pages.
Communication Channel	System-V message queues.
Communication Model	Indirect message passing with mailboxes [63].
Data plane	Network data plane in Arrakis [53], kernel bypass in RDMA.

Table 2: In *PraaS*, the concepts of systems design are used to lift the process model into a distributed and serverless space.

```
app_id = create_application()
status, pid = create_process(app_id)
status = retrieve_process(pid)
status = swap_process(pid)
result = invoke(app_id, func, data)
```

Listing 1: The REST interface of control plane in *PraaS*.

4.1 Process Management

Managing processes instead of functions requires two extensions to the FaaS model. First, processes are grouped into **applications** to create communication partners for functions, a feature missing in current serverless platforms. Then, we enhance the REST interface of **control plane**, focused on function invocations, with process management operations.

Application This logical unit provides group semantics for a set of processes, including processes that are active and that have been swapped out by the cloud provider. When an OS process is forked, other processes can always determine its location and status by using the `/proc` partition. In *PraaS*, the system grows and shrinks automatically with changes in the workload, and the cloud operator decides where and how processes are allocated. The *application* forms an equivalent of such partition and enables processes to locate their peers and establish inter-process communication.

Control Plane Process allocation is controlled by the cloud provider, and we do not place any restrictions in this regard, so low-latency schedulers like the one in Lambda can be supported [5], and placement can be optimized to increase data and communication locality. The REST interface of the control plane allows for managing applications and processes (Listing 1). Users allocate new processes with a clean state (2) and swap-in existing process state (3). To start a new process, the user must specify the application, the container image used, and the resource configuration. We expose explicit swap operations to allow users more control over the workflow(4).

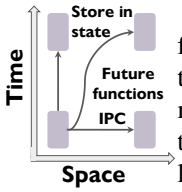
PraaS supports the standard FaaS interface of invoking functions via the control plane to support end users that cannot benefit from the data plane (5). Like in FaaS, the control plane implements standard container management techniques to increase the frequency of warm invocations. Unlike in FaaS, users can control the routing of invocations into selected process instances by providing a session state identification in request headers. Thus, processes implement *sticky sessions* [70] where requests from a single user are always handled by the same worker, in the same session.

In addition to scaling processes, the control plane is responsible for arranging them into applications. Processes directly connect to the control plane to receive group change notifications, invocations, and swap requests. Since not every invocation now uses the control plane, processes report data plane metrics back to accumulate billing data, drive the down-scaling policy, and update logs. Shifting the accountability from the invocation critical path to the control plane is essential in enabling fast serverless computing.

4.2 Inter-Process Communication

FaaS applications include a few communication patterns: message-passing, usually implemented as a propagation of a function's output to the input of a consecutive invocation [45]; an *aggregation* of the outputs from many functions [29, 30]; and using data cached in nearby instances to bypass storage [57, 76]. Most advances in serverless computing focus on optimizing *north-south* communication between functions and public services rather than targeting internal *east-west* communication between functions and processes.

On the contrary, *PraaS* improves communication efficiency by binding mailboxes and channels to the process instance. Our communication model does not concentrate on function invocations since they have a limited lifetime and might not execute simultaneously, but it is focused on data movement operations, allowing dynamically reshapable applications to benefit from peer–to–peer communication. In an application, processes know about each other's existence and can communicate directly. Instead of moving data from a function to a function via a cloud proxy, it is transmitted between processes hosting functions that want to communicate, increasing performance and decreasing network communication volume. Thus, communication services scale up automatically with the processing units, data is always locally available, and processes save the latency of reaching an external service.



Messaging routines provide an abstraction for all communication in space and time between processes and functions. When the message name indicates a function invocation, its contents are interpreted as input payload for a new invocation (§4.3). A message sent to itself becomes part of the process state (§4.4). All other messages are placed in the *mailbox* in a recipient process.

Message Passing Functions communicate by sending a message to another process using the `send` operation, transferring data into the recipient's mailbox. The recipient reads the message using `receive` and optionally specifies the source to match the exact recipient. Since now the process identi-

```python
# Function A
# Send data to the mailbox of process_id
def sender(process_id, message_id, data):
  praas.put(process_id, message_id, data)
# Function B
# Gather received results
def receiver(message_id):
  data = praas.get(praas.ANY, message_id)
```

Listing 2: Communication between functions **A** and **B** encodes data flow and neither location nor status of recipient.

fies communication targets, we establish message passing without enumerating ephemeral and unreliable functions, as we demonstrate in the example of two functions exchanging data (Listing 2). The communication interface stays the same, regardless of the actual location of both functions, as they can execute in the same process and in two different processes. Furthermore, the communication is not happening simultaneously: at the time of `send`, the cloud process with `process_id` can be swapped out. Furthermore, senders are always identified in the same way, which makes communication independent of the distribution of functions across processes. *PraaS* communication replaces pushing updates and polling for changes in a cloud proxy, allowing serverless programs to benefit from the higher bandwidth and lower costs of peer–to–peer communication.

Backup Queue When a message is sent to a swapped-out process, its contents must be retained until the process is swapped back in. To that end, we use a backup cloud queue. Once the process is active again, it reads all messages in the queue during initialization.

Group Membership In PraaS, the application state changes when the cloud provider scales the system down according to demand. Furthermore, users can always add and remove processes from the application. Functions need a consistent view of the application state to implement collective communication patterns, which requires each change to be applied atomically. We define this as *epoch change* applied in a two-step operation. The controller announces to all processes that a configuration change is necessary, and the processes wait until all outstanding communication of the last epoch is finished, notify functions, and acknowledge the change. Once all processes accept the change, the control plane distributes the confirmation. This notification allows applications to gracefully adjust messaging policies and avoid data failures in the presence of ephemeral workers.

In FaaS, functions use cloud storage to communicate data, and the lack of state requires that functions pay the penalty of reinitialization or functions are artificially kept alive and incur costs while waiting in the communication round.

```python
# Copy data from state and deserialize
model = praas.state("model").deserialize()
data = praas.state("data").deserialize()
# Example of applying new changes
new_data = compute(new_input, model, data)
# Store data in the process state.
praas.state("data", new_data)
```

Listing 3: *PraaS* programming model: integrating process state into Python functions.

We consider a global reduction operation applied by distributed workers before continuing with the next batch of work, as is the case in distributed machine learning training [36]. *PraaS* can be more affordable and efficient at the same time: functions store their local state in the process state and communicate reduction data directly to the destination, avoiding copying in cloud proxies. In the subsequent iteration, functions are guaranteed to access the warm environment through the process state.

4.3 Data Plane

PraaS helps to minimize FaaS latencies with fast and high-throughput invocations via the data plane. In FaaS, a serverless invocation includes authorizing the request, selecting and optionally allocating resources, and redirecting the payload to the executor function. Repeated control operations are redundant when many execution requests are redirected to the same warm container. Therefore, as long as the authorization remains valid, users can bypass control operations and move data directly to the process. The payload is sent from the user to the process mailbox, and this zero-copy approach helps to achieve high throughput on larger payloads. Thus, the invocation latency is bounded only by the network fabric and the performance of function executors in a process.

Type	Mechanism
Standard	Each message creates a new function invocation.
Multi-Source	Invocation waits for N messages with the same key.
Batch	Invocation waits for N messages with any key.
Pipeline	Blocks until previous instances complete.
Dependency	Blocks until all dependent functions complete.

Table 3: In *PraaS*, function invocation patterns are defined as conditions on messages arriving in the cloud process.

Invocations Serverless workflows and applications require function executions more complex than a simple request-response model, e.g., function chains, conditional invocation, and batching of input data. These often require external orchestrators and service-based triggers that increase costs and complexity even for small workflows, e.g., a function pipeline or an aggregation function taking more than one input. To facilitate serverless programming, we implement control and

data policies that allow users to support dynamic and configurable invocations (Table 3). Invocations are represented as regular messages whose names encode function and an invocation key. The key is used by *multi-source* functions to group messages into different invocations. Functions can be executed conditionally based on the status of prior invocations, allowing to implement *pipelines* and input *dependencies* of a task.

Multi-source invocations can be used by machine learning applications for reduction with the epoch number as an invocation key, while a MapReduce task would use keys to send intermediate data into different reducers. *Pipelines* provide a FIFO order in serverless processing [16]. Instead of using function fusion [24] to avoid the accumulation of invocation latencies on a function chain [24], developers can use *dependencies* to dispatch a sequence of functions.

4.4 State

Process state includes user-defined objects and unread mailbox messages, and all contents are swapped out to cloud object storage on process eviction. Saving messages guarantees access from future function invocations while not relying on an external cloud service that comes with additional latency and cost. Users implement *handlers* to serialize non-standard objects, compress state, skip datafields that do not need to be retained, and support custom initialization, e.g., creating a persistent connection to a database. We implement the simple interface for accessing and modifying the session state that internally uses the same communication interface as invocations and IPC (Listing 3). The simple communication interface incorporates new state and communication features, requiring neither a complete redesign of serverless applications [75] nor dedicated compilers [32].

Shared Memory The communication interface provides copy and sharing semantics for data exchange, depending on the language and platform support. Objects are stored in binary form in the former, and each call to `recv` returns a new copy. Functions receive the object data from the process session state by using standard local IPC methods, such as POSIX message queues or UNIX domain sockets. In the latter, objects are stored directly in a shared memory pool accessed by all functions in the cloud process, providing serialization–free and zero–copy access. For example, functions executing in C-based languages can receive a pointer to a shared object. On the other hand, Python functions require pickling data for each state operation, but they can still benefit from a a process implementation that uses zero–copy shared memory instead of traditional IPC methods to communicate between functions and session state. The state implementation is hidden from the user, who only sees the operations `send` and `recv`, allowing cloud operators to decide where and how objects should be stored and find a balance between access latency and the cost of in-memory storage of user data.

We consider the widely-used pattern where persistent applications offload computations to cheap and elastic serverless accelerators [18, 31]. A user can employ the same process session to invoke functions, and these functions will access the same memory pool. Consecutive and concurrent invocations will access a warm and initialized state. Furthermore, sessions enhance data sharing, e.g., reusing a large machine learning model across many inference functions.

5 *PraaS* in Practice

We implement *PraaS* as an extension to CaaS and FaaS platforms to facilitate wide adoption and demonstrate the compatibility of our process model with existing systems. The entire solution consists of roughly 11.5 thousand lines of code in C++ and Python, and can be extended with deployment to new serving platforms, execution in new container types, and support for new transport protocols and network fabrics, e.g., QUIC and RDMA.

Process As in other serverless platforms, users deploy containers with function code and dependencies, which are later extended with the *PraaS* process runtime. In addition to the OS processes that execute user code, we add *controller* running with superuser permissions. The controller handles invocation messages, accumulates data plane metrics, manages state, and implements swapping policies. Then, it uses TCP connections to propagate messages to other processes.

Serving We demonstrate the execution of processes on top of the managed containers using Kubernetes [4] and Knative [3]. Instead of running regular containers and functions, respectively, platforms now deploy containers holding the cloud process. Processes are allocated with standard control plane operations. Furthermore, we replace the default down–scaling policies that terminate a randomly selected or the least recently used container. Instead, we terminate process containers with data plane activity below a specified threshold. On Kubernetes, we manually modify the scaling set, while in Knative, we use the pod deletion cost mechanism to target selected containers.

Furthermore, we deploy *PraaS* on the AWS Fargate, a cloud service that outsources container management from the user. *Serverless containers* offered by Fargate are allocated on demand without resource provisioning for a Kubernetes cluster. We use a container instead of running a cloud process directly as a serverless function on AWS Lambda because Fargate allows us to attach public IP address to the container, a feature necessary for direct communication. Thanks to a resource configuration scheme similar to AWS Lambda, we can compare serverless containers with an equivalent resource allocation as Lambda functions. To use Fargate as the service backend of *PraaS*, we only need to implement the down–scaling policy in the *PraaS* control plane.

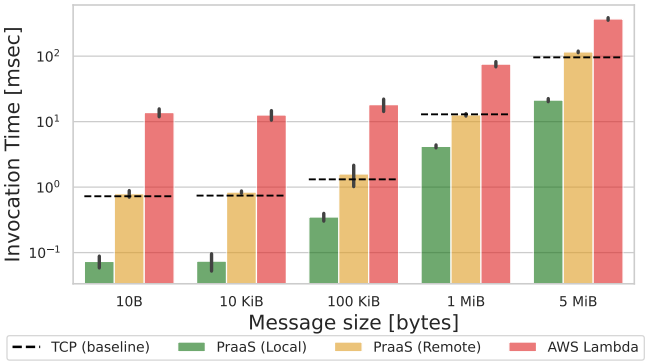


Figure 7: Invocation latency of a *no-op* function in *PraaS* on AWS Fargate.

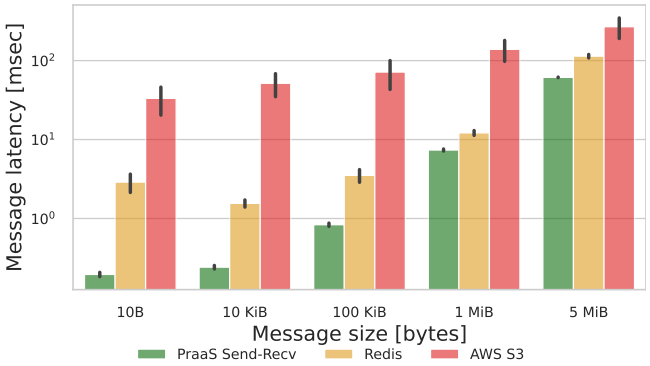


Figure 8: Communication latency of two *PraaS* processes running on Fargate with different communication channels.

6 Evaluation

In this Section, we focus on showing the practical benefits of *PraaS* with respect to improving invocation latency, reducing the overhead of communication between functions, and avoiding the need to rely on slower cloud storage by using the local process state. We then evaluate the trade-offs of *PraaS* and its cost compared to FaaS systems.

6.1 Lower Latency via the Data Plane

We start our evaluation of *PraaS* by comparing the latency of function invocation over the data plane compared to using AWS Fargate.

For this purpose, we invoke a function that accepts a payload of a given size and returns it immediately - this is the serverless invocation equivalent of a no-op.

We invoke warm AWS Lambda and PraaS functions. PraaS is running on AWS Fargate, and invokes a remote function on a Fargate container with 1 CPU and 2 GB which is equivalent to the Lambda configuration that uses 1792 MB and 1 vCPU. The version testing local follow-up invocations is using a Fargate container with 2 vCPU.

The results shown in Fig. 7 show a consistent, significant benefit for using *PraaS*, with remote invocations having virtually no overhead compared to the baseline of simply transferring the payload over TCP.

Local invocations measure invocations of a follow-up function scheduled on the same process rather than going through the control plane. While much faster than alternatives (up to 32 times faster than AWS Lambda), local invocations are limited by the delay introduced by the POSIX message queues used to move invocation data between process controller and function invoker. *PraaS* invocation have significantly less latency compared to Fargate: remote invocations are between 68% and 94% faster while local invocations are between 94% and 99% faster.

6.2 Inter-Function Communication

An important concept in serverless workflows is chaining functions to pass the output of one as input to the other one. We now evaluate the impact of direct message passing between processes compared to communication through Redis and S3 for different payload sizes.

We design the experiment to send a single message between two *PraaS* processes across two Fargate containers with 1 vCPU and 2 GB RAM. As baselines, we use a Redis instance (allocated on a c4.xlarge EC2 VM) and AWS S3. Both Redis and S3 are used to replace point-to-point communication. The sender uploads an object/key and the receiver reads it.

For all cases, we measure the round-trip latency of sending and receiving between the two processes and divide the time by two. Results represent the median out of 100 runs.

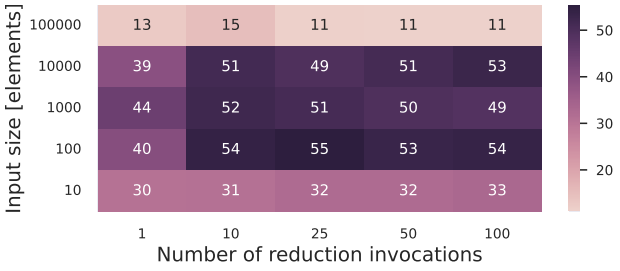
Figure 8 presents the results of this experiment. *PraaS* improves the latency against S3 from 77% to 99% (smallest message) and against Redis from 39% to 93%. The benefits are the largest for small messages, which is particularly important when considering deploying large stateful functions or services [56]. In addition to latency reductions, *PraaS* avoids significant costs and maintenance overheads associated with using S3 and setting up (and scaling) Redis instances.

6.3 Speed-up by Use of Process State

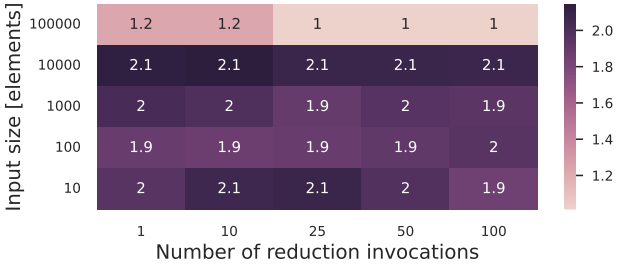
We now evaluate how much time can be saved by using the local process state *PraaS* provides instead of saving partial results in cloud store.

The scenario where many workers aggregate results using reduction functions is common in many cloud applications, especially in distributed machine learning. The reduction function needs to update the state resulting from previous invocations whenever it is invoked with new data.

Instead of loading data from cloud storage, updating and storing it again, serverless functions can skip the first and last steps by keeping the data in the memory of a warm container.



(a) Speedup of reduction in PraaS over state in S3.



(b) Speedup of reduction in PraaS over state Redis.

Figure 9: The *reduction* benchmark storing state.

The reduction shown in Fig. 9 accumulates data in a vector of 8 byte integers. We invoke the function with different input sizes, and we compute the time needed to invoke the reduction N times and display the speedup provided by *PraaS* using its persistent, swappable state compared to storing the state in S3 or Redis.

We run Fargate with 1 vCPU and 2 GB memory in this benchmark and Redis on a c4.xlarge machine. *PraaS* does not incur the additional costs of running a separate in-memory cache that Redis does.

PraaS is usually a factor 2 faster than Redis except for the largest input size, where there is enough computation to effectively hide the time needed to load and store partial results. The speed-up compared to S3 is overwhelming - at least 11 times faster, with some scenarios being over 50 times faster.

6.4 Case Study

To demonstrate the benefits of *PraaS*, we apply it to the distributed machine learning framework LambdaML [37]. We select the K-Means algorithm using the Higgs dataset and execute it on the *PraaS* Kubernetes implementation, comparing against execution on Knative, which uses AWS S3 and Redis for communication. We execute the benchmark with 8 workers on a cluster of consisting of 4 `t3a.large` EC2 nodes and present the results in Fig. 10. Compared to the S3 version, we speed up the runtime by a factor of 1.5 to 6 times.

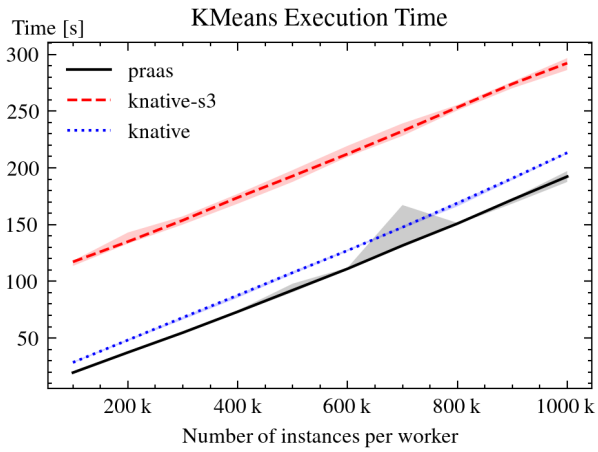


Figure 10: LambdaML with K-Means algorithm and Higgs dataset. *PraaS* against Knative with S3 and Redis.

6.5 Trade-Offs

While the new process model requires minor adjustments to the lifecycle of a serverless workers, these changes may introduce non-negligible overheads. In this section we look into the process allocation, process deallocation, state swapping costs in *PraaS*.

Process Allocation Allocating a process requires accessing a shared control plane state in Redis and deciding whether process can be allocated, and to which application it belongs. Figure 11 shows the process allocation time on Kubernetes (our baseline), *PraaS* directly on top of Kubernetes, and *PraaS* on Knative. We run this experiment on a VM cluster using t3.medium EC2 VM instances. Each instance supports up to 30 pods. In total, we use four 4 VMs with a maximum of 120 pods. We also ran this experiment on a larger deployment (cluster of 6 VMs with up to 160 pods) and the results are very similar therefore we do not show both experiments.

Results (Figure 11) show that *PraaS* introduces a low overhead on top of Kubernetes and knative. This overhead is the result of the extra access to storage to check if there is a swapped state that should be brought back from storage.

Process Deallocation Deallocating a process differs from deallocating a FaaS function. When scaling down FaaS functions, cloud operators only need to reduce some arbitrary ephemeral workers to adjust the scale to the current workload. On the other hand, in *PraaS*, each process can have a different activity on data plane and we should deallocate processes that are idle instead of the active ones.

We evaluate the added overhead of deallocation by comparing the time between process reporting low data plane metrics that warrant deallocation and the moment process receives a termination signal. An external benchmark triggers the deletion of a specific process and waits until the process reports that it started the termination process.

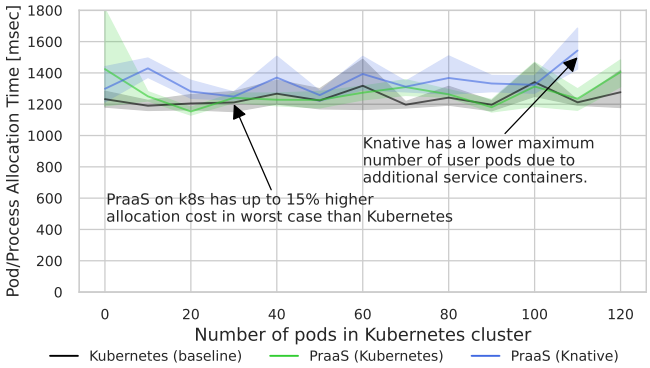


Figure 11: Allocating *PraaS* processes on managed services.

	1 MiB	5 MiB	10 MiB	50 MiB	100 MiB	200 MiB
Fargate	98.4	173	232	907.7	1719.3	3480.4
EC2	120.5	172.8	220.2	791.8	1525.7	2930.5

Table 4: Time of swapping [msec] in-memory state into AWS S3, from a process executing in a Fargate container and Docker container on EC2.

We find that time needed to delete a container in pure Kubernetes and in PraaS (which builds on Kubernetes) does not differ significantly and depending on the workload and system noise, the median is approximately 1.7 to 1.8 seconds. Most of this latency comes from Kubernetes logic to deallocate a pod.

Swapping State *PraaS* swaps state in and out of storage. To measure the performance cost of this operation, we measure the time needed to transfer the session state from a process to S3 from both Fargate and EC2. We run each experiment 20 times in repetition. As can be seen in Table 4, swapping session state takes 100 ms to write of 1 MB to S3 from Fargate takes approximately 100ms. This latency increases proportionally with the size of the session state.

Shahrad et al. [60] show (using real-world Azure data) that 90% of the applications never consume more than 400MB, and 50% of the applications allocate at most 170MB. However, the actual swappable state will be much lower - this includes the entire memory of a function, including additional libraries, runtime, local and temporary variables. In practice, only a fraction of data objects will become session state, therefore, resulting in a low latency overhead for swapping state.

6.6 Cost Analysis

To evaluate *PraaS* we must compare the cost of running a cloud process to using FaaS and relying on cloud storage.

PraaS users must be charged for keeping session state alive and the cost should depend linearly on the lifetime of an active session. The only comparable feature on modern commercial FaaS systems is a provisioned function instance, known as *provisioned concurrency* on AWS Lambda and *premium plan* for Azure Functions. Cloud providers guarantee ready function instances to decrease cold startup frequency. While arguably such functions are not *serverless*, such instances can be treated as a limited substitute of warm and low-latency state [1]. In addition to paying for using computing resources, users are charged the active state fee F_s that depends on the memory size and the duration of resource provisioning.

	V_m	C_m	F_m	G_V	G_C
AWS	$3.53 \cdot 10^{-3}$	$4.45 \cdot 10^{-3}$	$1.5 \cdot 10^{-2}$	76.43%	70.37%
Azure	$3.87 \cdot 10^{-3} \pm 0.005$	$4.45 \cdot 10^{-3}$	$1.14 \cdot 10^{-2}$	$66.05\% \pm 0.05$	60.96%

Table 5: Decreased costs of *PraaS* sessions in comparison to FaaS provisioned instances, with V_m - cost of changing from compute-optimized to memory-optimized VMs, F_m - the fee for active state, C_m - the cost of the managed container system. G_V and G_C are the cost decreases for moving session state from provisioned Faas to VMs and containers.

To estimate the cost of keeping *PraaS* sessions alive in memory, we use the memory of other cloud services as a proxy for the cost to the cloud operator. First, we compare the cost V_m of changing from a *compute-optimized* to a *memory-optimized* virtual machine instances, and divide it by the size of gained memory, which estimates the additional cost of gaining one gigabyte of DRAM. We compare `c6g` and `x2gd` instances on AWS and `Fs` and `Eadsv5` series on Azure, and find that V_m is almost the same for each instance size. Then, we select the cost of allocating additional memory when deploying *PraaS* on managed container systems C_m, and we consider AWS Fargate and Azure Container Instances. By comparing the memory costs V_m, C_m of *PraaS* deployment to the cost of provisioned storage F_m on FaaS, we estimate the cost decrease G_V and G_C of moving session state from provisioned FaaS to VMs and containers, respectively. The results presented in Table 5 prove that *PraaS* sessions can be offered at a lower cost, by up to 76%, and the estimation covers the cloud provider costs and profit included in the price of a VM instance. Furthermore, the memory-optimized instances come with additional SSD storage, which could be used to implement a low-latency tier for swapped sessions at no additional cost.

7 Related Work

Stateful Functions open the spectrum of applications that can benefit from FaaS by allowing functions to keep state, even if disaggregated. Researchers have built stateful functions on top of key-value stores specialized to Serverless [10, 66], and elastic ephemeral caches [40, 55, 57] which combine different placement strategies to manage cost and performance. For many applications however, statefulness is not enough

[1] Provisioned concurrency instances on AWS can be recycled and reinitialized, which makes state persistence difficult, if not impossible, to implement in practice.

as functions can fail at any time. Hence, systems such as Beldi [78], and Boki [34] have been proposed to help developers build consistent and fault tolerant systems atop ephemeral functions. Instead of offering yet another storage option to persist function state or propose a new technique to handle faults, *PraaS* strikes a new balance between IaaS and FaaS by proposing the concept of sessions that retain state. *PraaS* eliminates the need for explicit state synchronization with automatic function state management by swapping it out if the function becomes inactive and swapping it in when the function becomes active again..

Function Orchestration and Data Locality are also being extensively studied. Systems such as Speedo [20] and Nightcore [35] optimize function orchestration by either accelerating the control plane [20] or by completely skipping it [35] for internal function invocations. Other systems have looked into how to optimize the data path by comparing different function communication strategies and automatically adapting deployment decisions [46], or by avoiding moving data by allowing multiple functions to share the execution environment over time [42]. Pheromone [75] improves serverless workflows by binding control logic with ephemeral data objects.

PraaS, on the other hand, proposes that users should be able to directly connect to sessions and thus completely avoid the control plane after the first initialization step. In addition, by partitioning state among sessions, process functions can exchange data directly among them instead of relying on external storage proposed by previous systems [50, 52]. By taking advantage of direct communication between functions whenever possible [73], *PraaS* also moves computation to the data by defining a durable and swappable state and exposing messaging.

Serverless Sandboxes utilize specialized virtualization engines [2, 21] that drastically reduce the startup time, and memory footprint when compared to traditional virtual machine managers. However, to continue improving scalability and elasticity of serverless applications, soft-isolation systems [11, 22, 62] have been proposed to co-execute multiple invocations inside the same OS process. *PraaS* is, in part, inspired by such systems by allowing multiple functions of the same user to execute concurrently inside a single session. By doing so, resource redundancy is reduced and new opportunities for local communication arise. However, *PraaS*'s design does not preclude orthogonal optimization techniques such as image pre-initialization [7, 12, 21, 51] to optimize session startup time and memory footprint, or unikernels [41, 47, 81] to optimize process startup and memory footprint.

8 Discussion

A step towards a Cloud Operating System Distributed operating systems have been an active research topic for a long time, but, despite the efforts, researchers have not converged on a scalable system that transparently distributes the load and manages resources across multiple cloud machines communicating via a shared messaging service [74]. Similarly to the classical OS, a Cloud OS is expected to perform several tasks, such as resource allocation/management, scheduling, and file system management. For example, LegOS is a recently proposed distributed OS for hardware resource disaggregation [61]. LegOS manages a set of distributed resources using a messaging bus. On the other hand, a slightly different direction has been proposed with a Single System Image (SSI), which abstracts a cluster of machines to appear as one single system. SSI has been proposed as a way to hide the complexity of distributed systems [79].

We envision the cloud OS as a collection of cloud services that offer different OS-related functionality such as scheduling and file system. The cloud process proposed in this work is one of the missing building blocks of a cloud OS. Processes offer a durable state, inter-process communication, and an efficient data plane. To demonstrate the usefulness of this new abstraction, we implemented *PraaS*, a platform that resembles a cloud OS by offering scheduling and resource allocation services. *PraaS* allows applications to easily scale out with a simple programming model (similar to FaaS) that hides the complexity of distributed resources [79].

Fault-tolerance in *PraaS* *PraaS* users should enjoy a level of fault-tolerance comparable to using the non-serverless infrastructure. By providing a swappable state, *PraaS* handles intentional/planned failures (such as evictions) by removing the ephemeral, on-spot executor but persisting state data. If more data is generated than previously allocated to state memory, the overflow is pushed directly to cloud storage and enjoys the same guarantees as cloud queues. A sudden failure of the cloud process is still possible (as a sudden failure of a VM instance is possible). In the case of such unintentional failures, users can retrieve the last state snapshot from cloud storage. For long-lived processes, users can periodically swap/checkpoint the state to ensure that a recent snapshot is available.

Portability of PraaS Our cloud process model makes no assumptions on the underlying virtualization technology (container, VM, microVM, etc), and is not restricted to language, cloud platform, or serverless system. In sum, *PraaS* can be used in all major cloud providers and even allows platforms to offer specialized back-ends tailored to the systems themselves, as long as the required operations are supported.

9 Conclusions

PraaS is the next step towards a cloud computing OS. By taking advantage of *processes* and *sessions*, applications benefit from a low-latency state, fast invocations that bypass the control plane, and fast communication between sessions. *PraaS* brings persistent state to ephemeral workers and offers a speed-up of up to 55 times over using cloud storage, while providing a 76 % reduction in cost.

References

[1] Apache OpenWhisk. https://openwhisk.apache.org/, 2016. Accessed: 2020-01-20.

[2] Firecracker. https://github.com/firecracker-microvm/firecracker, 2018. Accessed: 2020-01-20.

[3] Knative. https://knative.dev/, 2021. Accessed: 2021-11-21.

[4] Kubernetes. https://kubernetes.io/, 2021. Accessed: 2021-11-29.

[5] 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 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[6] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 923–935, USA, 2018. USENIX Association.

[7] 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.

[8] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[9] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[12] 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.

[13] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[14] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, page 65–76, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Marcin Copik, Alexandru Calotoiu, Konstantin Taranov, and Torsten Hoefler. Faaskeeper: a blueprint for serverless services, 2022.

[17] 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. Association for Computing Machinery, 2021.

[18] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. RFaaS: RDMA-Enabled FaaS Platform for Serverless High-Performance Computing, 2021.

[19] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 356–370, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. *Speedo: Fast Dispatch and Orchestration of Serverless Workflows*, page 585–599. Association for Computing Machinery, New York, NY, USA, 2021.

[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.

[22] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Adam Eivy and Joe Weinman. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[24] Tarek Elgamal, Atul Sandur, Klara Nahrstedt, and Gul Agha. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312, 2018.

[25] J. L. Eppinger. TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem. *Carnegie Mellon University, Technical Report*, ISRI-05-104, January 2005.

[26] L. Feng, P. Kudva, D. Da Silva, and J. Hu. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341, July 2018.

[27] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 13, USA, April 2005. USENIX Association.

[28] 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.

[29] 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 *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 363–376, USA, 2017. USENIX Association.

[30] V. Giménez-Alventosa, Germán Moltó, and Miguel Caballer. A framework and a performance assessment for serverless mapreduce on aws lambda. *Future Generation Computer Systems*, 97:259–274, 2019.

[31] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 199–208, 2019.

[32] Zhiyuan Guo, Zachary Blanco, Mohammad Shahrad, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, Harry Xu, and Yiying Zhang. Resource-centric serverless computing, 2022.

[33] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[34] 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.

[35] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing forlatency-sensitive, interactive microservices. In *Proceedings ofthe 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*, June 2021.

[37] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021*

International Conference on Management of Data, SIGMOD '21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[39] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 427–444, USA, 2018. USENIX Association.

[40] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 427–444, USA, 2018. USENIX Association.

[41] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 169–173, New York, NY, USA, 2017. Association for Computing Machinery.

[42] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021.

[43] Collin Lee and John Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 149–154, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Pedro Garcia Lopez, Aleksander Slominski, Michael Behrendt, and Bernard Metzler. Serverless predictions: 2021-2030, 2021.

[45] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. {SONIC}: Application-aware data passing for chained serverless applications. In *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, pages 285–301, 2021.

[46] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[47] 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.

[48] Johannes Manner, Martin EndreB, Tobias Heckel, and Guido Wirtz. Cold start influencing factors in function as a service. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, 2018.

[49] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[50] Ingo Müller, Renato Marroquin, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. *ArXiv*, abs/1912.00937, 2019.

[51] 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.

[52] 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.

[53] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[54] 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.

[55] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 193–206, USA, 2019. USENIX Association.

[56] 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.

[57] 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, 2021.

[58] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5):76–84, April 2021.

[59] M. Sciabarrà. *Learning Apache OpenWhisk: Developing Open Serverless Solutions*. O'Reilly Media, Incorporated, 2019.

[60] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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 20)*, pages 205–218. USENIX Association, July 2020.

[61] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 69–87, USA, 2018. USENIX Association.

[62] 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, 2020.

[63] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating System Concepts, 10e Abridged Print Companion*. John Wiley & Sons, 2018.

[64] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[65] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Archipelago: A scalable low-latency serverless platform, 2019.

[66] 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, July 2020.

[67] Kun Suo, Junggab Son, Dazhao Cheng, Wei Chen, and Sabur Baidya. Tackling cold start of serverless applications by efficient and adaptive container runtime reusing. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–443, 2021.

[68] Amoghavarsha Suresh and Anshul Gandhi. Servermore: Opportunistic execution of serverless functions in the cloud. SoCC '21, page 570–584, New York, NY, USA, 2021. Association for Computing Machinery.

[69] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514, 2021.

[70] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[71] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1288–1296. IEEE, 2019.

[72] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 133–145, USA, 2018. USENIX Association.

[73] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. Boxer: Data analytics on network-enabled serverless platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*, 2021.

[74] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 3–14, New York, NY, USA, 2010. Association for Computing Machinery.

[75] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing, 2021.

[76] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Restructuring serverless computing with data-centric function orchestration. *arXiv preprint arXiv:2109.13492*, 2021.

[77] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 138–148, 2021.

[78] 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.

[79] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. Giantvm: A type-ii hypervisor implementing many-to-one virtualization. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[80] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 724–739, New York, NY, USA, 2021. Association for Computing Machinery.

[81] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfei Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization. USENIX ATC '18, page 173–185, USA, 2018. USENIX Association.