Process-as-a-Service: Elastic and Stateful Serverless with Cloud Processes.

Marcin Copik
ETH Zürich

Alexandru Calotoiu
ETH Zürich

Rodrigo Bruno
INESC-ID, IST, ULisboa

Gyorgy Rethy
ETH Zürich

Roman Böhringer
ETH Zürich

Torsten Hoefler
ETH Zürich

Abstract

Fine-grained serverless 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 the loss of data locality when accessing state, complex control plane interactions, and expensive inter-function communication. We revisit the foundations of FaaS and propose a new cloud abstraction, the ***cloud process****, that retains all the benefits of FaaS while significantly reducing the overheads that result from 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× 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 [21, 68, 78]. Serverless functions have been used in a wide spectrum of areas, ranging from web applications, media processing, data analytics, machine learning, to scientific computing [10, 30, 35, 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 [20, 35, 37, 59].

Take distributed machine learning training [16, 28, 35, 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 overheads, as *in each round of updates*, the output must be written and read from external cloud storage.

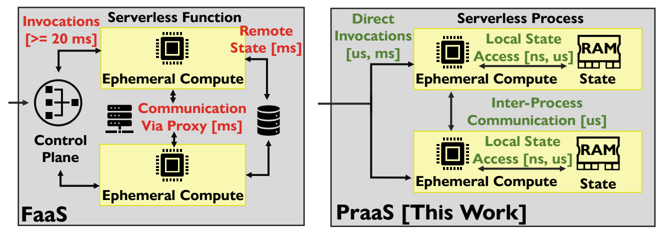


Figure 1: Cloud processes solve critical inefficiencies inherent to functions while retaining extreme scalability and elasticity.

Furthermore, each new round requires a FaaS invocation that goes through the entire cloud control plane.

Researchers have addressed this lack of state management in ephemeral functions [37, 59] by proposing ephemeral storage solutions [33, 38, 66, 77] 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 by nature. In addition to an increase of cost and complexity [33, 59], serverless applications that heavily depend on non-serverless infrastructure to keep state are only as elastic as the least elastic component. As a consequence, state management has become a burden and a limiting factor for FaaS developers. As a consequence today's platforms cannot compete with the performance and efficiency of classical multi-processing [19, 32, 37].

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**. A process extends the semantics of a function by introducing a durable **state** and by defining a universal **inter-process communication** API. We demonstrate how processes do not impose significant design challenges for providers while retaining the elasticity and scalability characteristics attributed to FaaS. **Process-as-a-Service (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. Similarly to OS processes using threads for concurrent computations, *cloud processes* run on a single machine and launch functions within a shared environment (here, a function invocation would be equivalent to a thread OS). When resources become scarce, PraaS follows traditional-OS design and transparently **swaps** state to persistent storage. By implementing responsibilities traditionally associated with operating systems, *PraaS* is a step towards a distributed cloud computing OS that provides

	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 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.**

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 persistent and low latency **state**.

Inter-Process Communication Current FaaS platforms restrict peer–to–peer communication, forcing users to rely on storage or proxy-based communication. This approach is expensive, leads to higher latency, and 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**.

A message is sent to a mailbox of the process hosting the destination function when the process is active, or is stored externally while the process is swapped out. Messages can be transferred between two concurrently executing functions (*message passing*) but can also be used as triggers for functions (*invocation*), effectively replacing storage-based communication [35, 50]. Orchestration patterns are also enabled with messages (e.g., if a function should only be invoked after a certain number of messages is received) through the use of *mailboxes* (Sec. 4.3). 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 [61] and, as a consequence, the relative overhead of the multi-step function workflows is high (§2.2). Slow control-plane invocations prevent wider adoption of parallel and granular computing [20, 42, 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 *bypass* the control plane overheads from the data path entirely [53] by exposing a direct communication channel to the process that allows submitting the invocation payload as a message – effectively separating control and data paths in the platform.

Durable State The stateless nature 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 [61]. However, functions cannot solely rely on resource caching 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** that is automatically swapped-in/swapped-out as the process becomes active/inactive. The lifetime of computing and storage is managed 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 invocation latency by up to 32× and communication latency by up to 99%.

2. Background

Function-as-a-Service (FaaS) has found its way to major cloud providers with a fine-grained and elastic programming model. 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 or lightweight virtual machines [7]. A *cold* invocation requires allocation of a new sandbox that significantly increases invocation latency [20, 48]. Subsequent *warm* invocations achieve a lower latency by reusing existing sandboxes. Therefore, cloud systems employ complex and sophisticated retention and pre-warming strategies [22, 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, it does have some prominent disadvantages: high communication costs, higher latency due to complex control planes, and poor locality of data due to the non-existence of local state [20, 26, 37, 72].

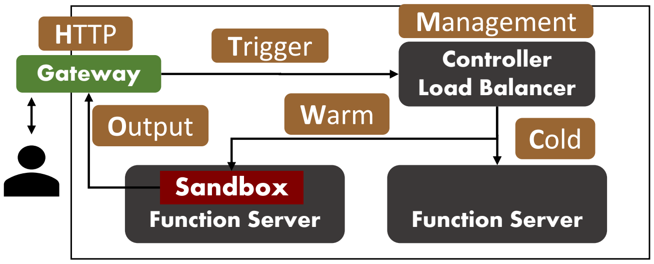


Figure 3: **FaaS control plane:** each invocation includes the management overhead (M).

2.1. Serverless Communication

Communication in FaaS has always been constrained as industrial offerings do not offer direct communication. State-of-the-art solutions implement communication through cloud storage, increasing latency, costs, and application complexity [35, 37, 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. I.e., the message–passing paradigm cannot be applied directly to ephemeral functions, as the worker lifetime is not well defined: new workers can be launched and terminated by the provider at any time according to internal scheduling and scaling policies that are completely unknown to users. Furthermore, in typical FaaS deployments, functions operate behind NAT and cannot accept incoming connections. Connections can be established with the help of hole punching [27, 29], but it is a complex process that applies only to two active functions simultaneously.

2.2. Serverless Control and Data Planes

Modern platforms implement dynamic function placement through a centralized routing system (Fig. 3) [7]. 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 [7]. 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 [60]. 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 [7, 18] 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 [20]. 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 [8, 17, 21, 65, 76], but these optimizations do not offer a solution generalizable to all FaaS platforms.

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.

2.3. 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 [33, 66, 77]. 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 [33, 77]. In the FaaS model, removing remote storage access from the data path is impossible.

FaaS can be much more than just a platform for irregular and lightweight workloads, and the serverless programming model aligns well with requirements for massively parallel and granular computing [42, 44]. Nevertheless, serverless systems must first overcome critical limitations: complex **control plane** involved in every invocation, lack of a fast and durable **state**, and expensive storage-based **communication**.

3. Cloud 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 traditional OS processes to the cloud. Processes are dynamically and automatically allocated on abstracted cloud resources, and like functions they execute in fine-grained and isolated environments. Each process contains a *controller* that controls the state, invokes functions, accumulates logs and metrics, and implements message passing between processes.

The new cloud process model overcomes the limitations of existing systems in three areas. (1) *Inter-process communication* defines data movement between processes and removes the dependency on external storage, enabling direct commu-

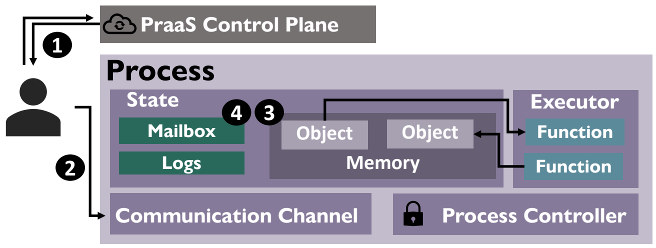


Figure 4: **The process model in *PraaS***: ephemeral functions are executed in a *processs* with shared and transient state and communication channels for quick user access.

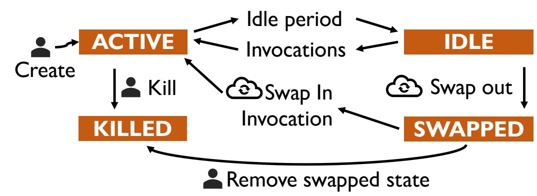


Figure 5: The life cycle of a cloud process. Process status changes in reaction to user operations ( ) or operator scaling actions ( ).

```
# Operate on process state
state(key, data)
data = state(key)
# Send message over IPC
send(destination, key, data)
data = recv(source, key)
# Invoke function
invoke(destination, function, data)
```

Listing 1: New communication interface available to functions.

nication between ephemeral workers (§3.1). (2) Processes have access to a *data plane* that supports fast function invocations bypassing the control plane (§3.2). (3) Processes are equipped with durable *state* with well-defined durability semantics which efficiently supports stateful applications (§3.3). With the new process abstraction, we build *PraaS*, a new programming and execution model (§4).

3.1. Process Model with Communication

Compared to FaaS functions, a process contains two new components to support inter-process communication: a data plane communication channel and a durable state, which includes all the memory storing user data and a mailbox (Fig. 4). When a process is allocated by the cloud control plane, it is assigned an identifier and a user-defined amount of resources. A communication channel is then 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 invocations in the state (❸). The mailbox (❹) handles invocations (§4.3) and communication between processes (§4.2). This component is allocated within the process and managed by the process controller to minimize access latency and provide reliability; messages may live longer than a single function invocation.

Compared to FaaS, functions executing in a cloud process have to use only five new functions to benefit from local state and fast communication (Listing 1). We define two messaging routines that implement all communication tasks handled by processes. A `recv` operation that 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. A `send` operation 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. We define two special identifiers `SELF` and `ANY` to support intra–process communication and receiving messages from an arbitrary sender. These routines are optimized to transmit binary data as efficiently as possible and hide all details of the underlying network transport and local communication. A simple interface with just five functions expedites porting applications, and function developers are not exposed to the complexity of managing the state and establishing communication.

3.2. Computing Using the Data Plane

Functions can also be invoked by sending the payload over the data plane (inter-process communication) instead of going over the control 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 in-memory state. We leave it up to the implementers to decide if a function executes in a dedicated OS process or a thread within the main OS process.

The process can handle multiple function invocations simultaneously, and it is bounded by the resource 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 allocated. Larger workflows are supported by distributing the workload across multiple processes and communicating through the IPC interface (§4.2).

The process controller receives invocation messages to start user functions. Functions queued beyond the upper threshold wait until resources become available. Furthermore, the process controller uploads data plane invocation metrics to the control plane. When the control plane sends an eviction notice for example due to inactivity, the controller swaps the state to cloud storage.

3.3. Scalability with State

Processes are serverless; the cloud operator makes all allocation decisions, and users have no control over them. Similarly to FaaS, the allocation is not persistent, the lifetime of a process is controlled by the cloud provider, and it can be removed at any point. Process state enables the persistence of user data and execution of stateful functions without the user needing manual state management. Note that a process hosting a single function operates as stateful FaaS (similar to a stateful entity in [14]), with the same ease of scaling and reclaiming resources.

Processes have a 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 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 reinitialization.

Scaling Up New processes are allocated on-demand via an explicit request to the cloud control plane. Processes are allocated with a clean state (*creation*), or retrieve a swapped state from the storage and continue execution (*swapping in*). The fundamental assumption behind our process is that it never scales beyond a single server, since such a design radically simplifies handling memory and state. A process spanning multiple machines requires a partitioned and distributed shared memory, which comes with non-trivial issues in cache coherency, synchronization, and performance. Instead of using processes larger than a single machine, users are encouraged to allocate more processes to handle the increased load. This decision does not affect the programming model, as the communication interface available to functions is the same for local and remote operations. When possible, this communication will be optimized to use local operations.

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 API, regardless of whether the communication is intra- or inter-process.

Scaling Down Upon a process eviction, the sandbox is terminated and the state — including memory, logs, and mailbox, as in Fig. 4 — is *swapped out* to persistent cloud storage. Processes are swapped implicitly by the control plane according to the provider's down-scaling policy (fixed period of inactivity for example). In practice, our API could be easily extended to allow users to explicitly swap out processes. When scaling down, processes do not accept any new invocation requests, and the state, together with unread messages, is written to the store. Messages sent to a swapped-out process are delivered to the backup queue (Fig. 6), and read once the process is swapped in. If a specific state is no longer required (for example, if the user deleted the process), the swapped state can be completely removed from the cloud storage. Swapping guarantees data persistence in the presence of *intentional failures*,

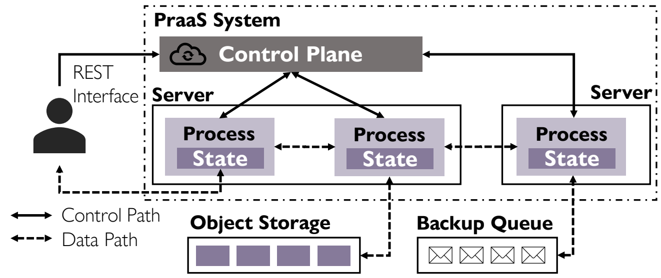


Figure 6: Platform architecture of *PraaS*.

PraaS Concept	Inspiration
Application	Operating system.
Process	POSIX process model.
Function	Thread in a process.
State	POSIX process memory.
State 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 cloud process model into a distributed and serverless space.

that is, when the resource is evicted by the cloud provider. For unintentional failures, when the machine crashes unexpectedly, we guarantee only the persistence of the state snapshot from the last swap-out operation.

4. PraaS: Process–as–a–Service

With the cloud 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).

4.1. Process Management

In *PraaS*, processes are grouped to create scalable applications spanning multiple server machines (§4.1). 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 the **control plane**, focused on function invocations, with process management operations.

A *PraaS* application provides group semantics for a set of processes, including processes that are active and that have been swapped out by the cloud provider. In *PraaS*, the system grows and shrinks automatically with changes in the workload, and the cloud operator decides where and how processes are allocated. Processes in an application can establish inter-process communication.

Process allocation is controlled by the cloud provider control plane, and we do not place any restrictions in this regard. Therefore, low-latency schedulers like the one in Lambda can be supported [7], and placement can be optimized to increase data and communication locality. The REST interface of the control plane allows for managing applications and processes

```
1 app_id = create_application()
2 status, pid = create_process(app_id, optional[pid])
3 result, pid = invoke(app_id, func, data, optional[pid])
4 delete_process(pid)
5 delete_application(app_id)
```

Listing 2: *PraaS* control plane REST interface.

```
# Function A: send data to the mailbox of process_id
def sender(process_id, message_id, data):
  praas.send(process_id, message_id, data)
# Function B: gather received results
def receiver(message_id):
  data = praas.recv(praas.ANY, message_id)
```

Listing 3: Communication between functions **A** and **B** encodes data flow but does not expose the location nor the status of the recipient.

(Listing 2). New processes receive a clean state by default (2), but they can be initialized from a previously swapped state by providing the `pid`. To start a new process, the user must specify the application, the container image used, and the resource configuration (we omit some details in Listing 2 for simplicity).

PraaS is backwards-compatible with the FaaS interface. Users can skip process creation and directly invoke functions. At this point, the platform automatically creates a process and invokes the function (3). 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 process identifier in request headers. Thus, processes can be used to implement *sticky sessions* [70] where requests from a single user are always handled by the same process.

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

PraaS offers efficient and disaggregated communication 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 cloud 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, co-located with the process in the same sandbox to minimize data copies.

Functions communicate by sending messages (`send`) into the recipient's mailbox. Recipient functions read messages (`recv`) and optionally specify the source to match the exact recipient. Since cloud processes communication target mailboxes, we establish message passing without enumerating ephemeral and unreliable functions, as demonstrated in the example of two functions exchanging data (Listing 3). 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. Message names encode focus on the data and its semantics, similar to storage-based communication in FaaS that requires a key for object and NoSQL storage, and multithreading in OS processes that uses variable names for that purpose.

Furthermore, the communication may not happen synchronously as the receiver might be swapped out. In such a scenario, messages are delivered to the backup queue and are processed once the process has been swapped in. 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.

Distributed applications need to control active workers and their location in the cloud. This is even more important when using serverless resources, as allocations are ephemeral and change often. However, FaaS systems offer little to no support for controlling the global state of an application. In *PraaS*, all processes are equipped with an up-to-date list of active and swapped-out processes, and with information needed to establish IPC-style connections. The control plane is involved in every scaling up and down operation and is responsible for distributing updates to active processes. Functions are notified of a change, allowing them to adjust communication operations and support collective communication patterns, even when they involve workers that can be swapped out.

```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 4: Integrating process state into Python functions.

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 [35]. *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 single-hop approach helps 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.

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

Serverless workflows may require complex function interactions such as function chaining, 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 basic 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. These messages are tracked by the process controller which accumulates provided invocation keys and checks if any of the function triggers are satisfied. More complex orchestrators can be implemented atop cloud processes.

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.

4.4. State

Process state includes memory and unread mailbox messages, which, on a process eviction, are swapped out to cloud object storage. Saving messages guarantees access from future function invocations while not relying on an external cloud service that comes with additional latency and cost. We implement a simple interface for accessing and modifying the state memory that internally uses the same communication interface as invocations and IPC (Listing 4). The simple communication interface incorporates new state and communication features, requiring neither a complete redesign of serverless applications [75] nor dedicated compilers [31].

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 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 process implementation that uses zero–copy-shared memory instead of traditional IPC methods to communicate between functions and 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.

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 solution consists of roughly 11.5 thousand lines of code in C++ and Python, with an additional Python runtime for a process (400 lines). We use dedicated libraries for event handling, I/O multiplexing, HTTP serving, and data serialization. *PraaS* 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.

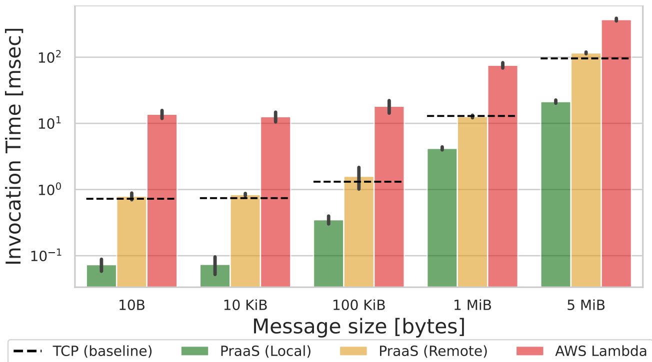


Figure 7: Invocation latency of a *no-op* function in *PraaS* on AWS Fargate.

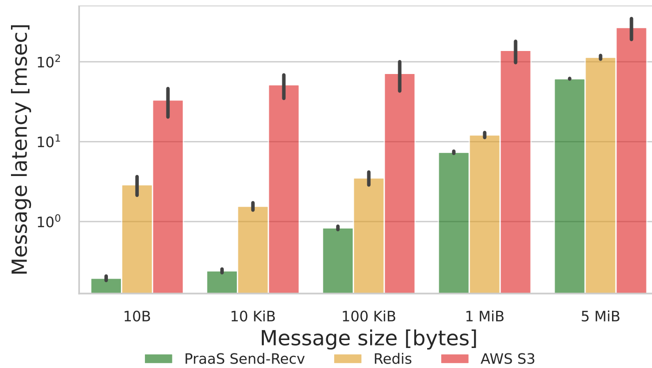


Figure 8: Communication latency of two *PraaS* processes running on Fargate with different communication channels.

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.

We demonstrate the execution of processes on top of the managed containers using Kubernetes [4] and Knative [3]. 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. Note that the IP is not exposed to the user code. 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.

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 Invocations 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.

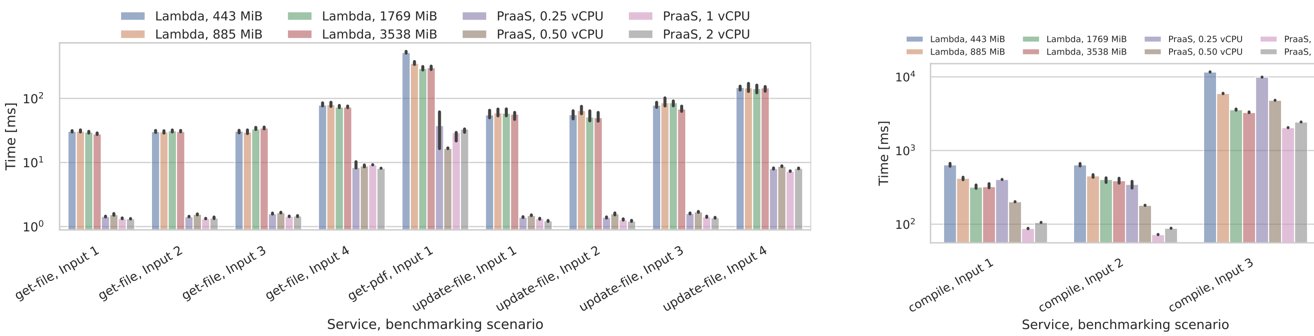
The results shown in Fig. 7 show a consistent, significant benefit for using *PraaS*, with remote invocations having virtually no overhead compared to the baseline of simply transferring the payload over TCP.

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 the process controller and function invoker. *PraaS* invocation have significantly less latency compared to Lambda: 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.

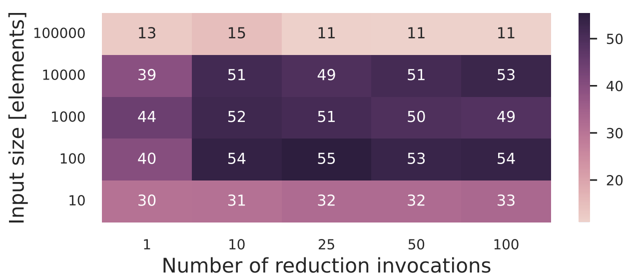
We design the experiment to send a single message between two *PraaS* processes across two Fargate containers with 1 vCPU and 2 GB RAM. As baselines, we use a Redis instance (allocated on a c4.xlarge EC2 VM) and AWS S3. Both Redis and S3 are used to replace point-to-point communication. The sender uploads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



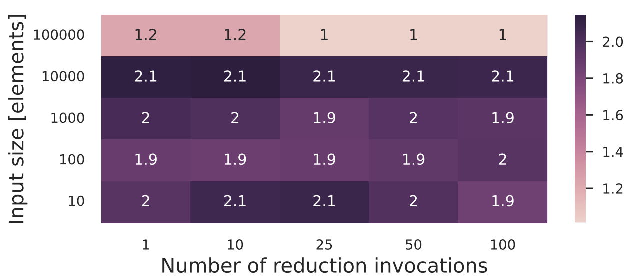
(a) Services for collaborative editing of LaTeX files.

(b) Compilation microservice.

Figure 9: LaTeX microservice with serverless functions (AWS Lambda) and cloud processes (AWS Fargate). Semi-log plot.



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



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

Figure 10: The *reduction* benchmark storing state.

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 higher 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. The benefits of Cloud 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 storage. 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.

The reduction shown in Fig. 10 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 approximately 2× faster than Redis except for the largest input sizes, 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 - LaTeX Service

We demonstrate the benefits of state and data plane invocations with a case study of a microservice handling collaborative LaTeX editor, similar to the Overleaf project [5]. We implement four services that allow us to update project files, retrieve the newest version, recompile, and retrieve the compiled document. To support online editing, serverless functions must use an external storage as two independent calls to a service might be placed in different containers. On the other hand, functions in PraaS offer a persistent state and data plane connection which guarantees that calls to a service for the same project are handled by the same process.

We evaluate each service with different inputs on AWS Lambda with S3 storage, and PraaS processes deployed on AWS Fargate. We vary the Fargate container allocations and Lambda memory configurations to measure the impact of varying availability of computing resources and I/O bandwidth, and repeat each invocation 50 times. Lambda memory is tuned to use the same virtual CPU allocation as PraaS processes run-

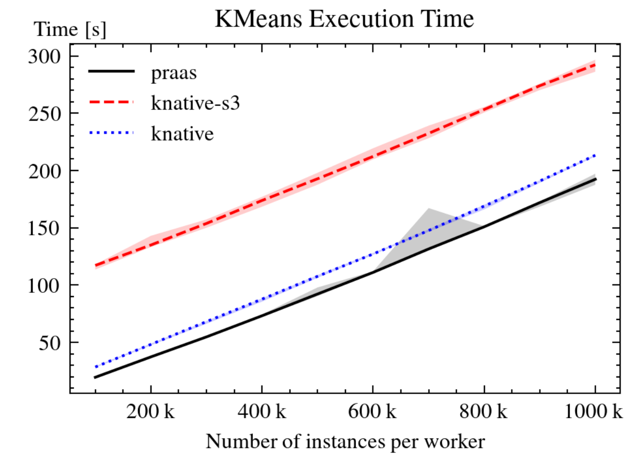


Figure 11: LambdaML with K-Means algorithm and Higgs dataset. *PraaS* against Knative with S3 and Redis.

ning in a Fargate container. Figure 9 shows that a persistent process state decreases the overhead of running microservices in a serverless setting. Even for a compute-intensive incremental LaTeX compilation, a local state guarantees that updated files can be served quickly.

6.5. Case Study - Machine Learning

To demonstrate the benefits of *PraaS*, we apply it to the distributed machine learning framework LambdaML [36]. 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. 11. Compared to the S3 version, we speed up the runtime by a factor of 1.5 to 6 times.

6.6. 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 12 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 12) show that *PraaS* introduces a low overhead on top of Kubernetes and knative. This overhead is

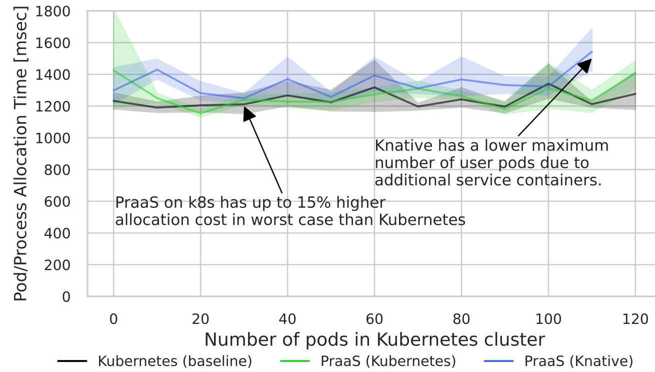


Figure 12: Allocating *PraaS* processes on managed services.

	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.

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.

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 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 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 state.

Shahrad et al. [61] 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 state, therefore, resulting in a low latency overhead for swapping state. Swapped state will also incur a storage cost that will be proportional to the number of swap-in/swap-out operations and size of the state. We estimate that this storage cost will not dominate the entire cost of the infrastructure and might even be compensated by reducing the initialization time that FaaS currently suffer.

7. Related Work

Stateful Functions augment the spectrum of applications that 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 [11, 66], and elastic ephemeral caches [39, 55, 57] which combine different placement strategies to manage cost and performance. Others have gone a step further and offered transaction support and fault tolerance atop FaaS [33] to help developers build consistent and fault tolerant systems atop ephemeral functions. Instead of relying on external cloud services to work around the limitations of FaaS, we propose rethinking and redesigning the underlying abstraction to support state and communication. Similarly to stateful cloud applications (such as microservices), applications built atop *PraaS* can naturally be complemented with external databases and caches to facilitate synchronization, fault tolerance, and consistency.

Function Orchestration and Data Locality are also being extensively studied. Systems such as Speedo [23] and Nightcore [34] optimize function orchestration by either accelerating the control plane [23] or by completely skipping it [34] 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], by avoiding moving data by allowing multiple functions to share the execution environment over time [41], or by offering direct network access to functions [73]. Pheromone [75] improves serverless workflows by binding control logic with ephemeral data objects, and Unum [43] proposes a decentralized orchestrator for FaaS workflows. Durable Functions [13, 14] (DF) extended FaaS's programming model by incorporating support for orchestration, stateful entities, and critical sections. DFs build on existing cloud services to offer consistency and synchronization across all entities. Palette [6] proposes locality hints that can be used to forward requests of a client to the same worker. *PraaS*, on the other hand, proposes a general-purpose execution environment that looks similar to the one available in an OS-level process. In fact, the process abstraction can be used to implement traditional FaaS applications and stateful entities. *PraaS* offers basic orchestration primitives that rely on message passing but more advanced orchestration frameworks such as Unum could be easily integrated at the application level. For communication, processes use mailboxes which can be implemented atop direct communication or indirect communication via proxy/storage. Mailboxes do not require the recipient to be alive upon sending nor the sender to be alive upon receiving.

Lightweight sandboxes utilize specialized virtualization engines [2, 24] that offer low startup times and memory footprint when compared to traditional virtual machine managers. However, to continue improving the scalability and elasticity of serverless applications, Software Fault Isolation-based systems [12, 25, 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 process (note that a *PraaS* process can be implemented different sandboxing technology as long as it allows multiple functions to share memory). By doing so, resource redundancy is reduced and new opportunities for local communication arise. Nu [58] proposes logical processes that span many proclets executing on a distributed execution environment. However, unlike *PraaS*, Nu is not designed for serverless platforms as it assumes always-on stateful instances with direct communication Finally, *PraaS*'s design does not preclude orthogonal optimization techniques such as image pre-initialization [9, 15, 24, 51] to optimize process startup time and memory footprint, or unikernels [40, 47, 79] 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. We envision the cloud process as one of the missing building blocks of a cloud OS.

Fault-tolerance Cloud processes 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*, applications benefit from a low-latency state, fast invocations that bypass the control plane, and fast communication between processes. *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] Overleaf: An open-source online real-time collaborative LaTeX editor. https://github.com/overleaf/overleaf, 2023. Accessed: 2023-08-10.

[6] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 365–380, New York, NY, USA, 2023. Association for Computing Machinery.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proc. VLDB Endow.*, 15(8):1591–1604, apr 2022.

[14] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[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 *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD

'21, page 857–871, New York, NY, USA, 2021. Association for Computing Machinery.

[37] 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.

[38] 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.

[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 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 427–444, USA, 2018. USENIX Association.

[40] 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.

[41] 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.

[42] 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.

[43] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. Doing more with less: Orchestrating serverless applications without an orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1505–1519, Boston, MA, April 2023. USENIX Association.

[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] 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.

[59] 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.

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

[61] 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.

[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] 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.

[78] 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.

[79] 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.