Process-as-a-Service: Unifying Elastic and Stateful Clouds with Serverless Processes

Marcin Copik
ETH Zurich
Zurich, Switzerland
marcin.copik@inf.ethz.ch

Alexandru Calotoiu
ETH Zurich
Zurich, Switzerland
alexandru.calotoiu@.inf.ethz.ch

Gyorgy Rethy*
Oracle Labs
Zurich, Switzerland
gyorgy.rethy@oracle.com

Roman Böhringer*
OpenCore GmbH
Schindellegi, Switzerland
r.boehringer@opencore.ch

Rodrigo Bruno
INESC-ID, Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal
rodrigo.bruno@tecnico.ulisboa.pt

Torsten Hoefler
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

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 remote invocations up to 17× faster and reduces communication overhead by up to 99%.

*The work was done while at ETH Zurich.

ACM ISBN 979-8-4007-1286-9/24/11…$15.00
https://doi.org/10.1145/3698038.3698567

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Cloud computing**.

KEYWORDS

Serverless, Function-as-a-Service, Operating Systems

ACM Reference Format:
Marcin Copik, Alexandru Calotoiu, Gyorgy Rethy, Roman Böhringer, Rodrigo Bruno, and Torsten Hoefler. 2024. Process-as-a-Service: Unifying Elastic and Stateful Clouds with Serverless Processes. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3698038.3698567

PraaS Implementation: https://github.com/spcl/praas
PraaS Artifact: https://github.com/spcl/praas-artifact

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 [20, 66, 75]. 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, 34, 48, 51, 53]. 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 [19, 34, 35, 57].

As an example of a popular workload that is difficult to implement efficiently in FaaS, we take a cloud service decomposed into microservices. These include online map services, web-based editors such as LaTeX editors (Sec. 6.4), and

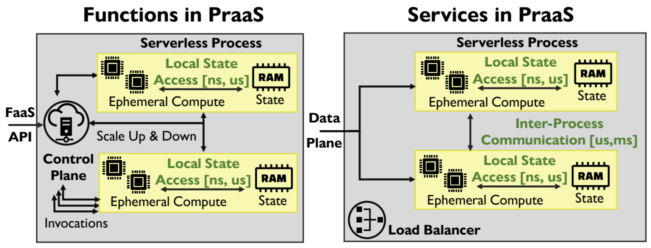


Figure 1: Cloud processes solve critical inefficiencies inherent to functions while retaining scalability and elasticity: stateful functions operate with same simplicity as FaaS, while services can use functions more efficiently.

social networks [29]. Functions are an attractive runtime even for complex services, as serverless can offer decreased costs and improved utilization for infrequent and variable workloads [18]. While each microservice component can be deployed as a function, a practical deployment requires an efficient coordination between many microservices, sticky sessions, and strong data locality when caching results of prior requests. While the stateless nature of FaaS simplifies scheduling and resource management, offloading the user data and service state to a remote storage incurs major performance and cost overheads. Furthermore, each request will require multiple FaaS invocations traversing the control plane many times.

Researchers have addressed the limitations of serverless with a variety of solutions: the lack of state management in functions [35, 57] was tackled with ephemeral storage [32, 36, 64, 73]. Repeated invocations over the same set of warmed-up containers are optimized with faster network protocols [20, 33], and the lack of efficient communication is being solved with direct communication and data locality [17, 43, 70, 72]. However, many of these solutions require *non-serverless* infrastructure with manual and non-elastic management, and they often solve one of the limiting factors in FaaS while not addressing the others. As a consequence, today's FaaS platforms still struggle to compete with the performance and efficiency of classical stateful infrastructure [18, 31, 35]. Instead of designing new solutions to fight FaaS limitations, we propose an enhanced programming model that combines the FaaS elasticity with the performance of containers and virtual machines.

The separation of data and computing in serverless is fundamentally inefficient and cannot be resolved by composing FaaS with additional remote cloud systems. Instead, we introduce a new abstraction: the **cloud process**. 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). The process provides a **persistent state** that functions can use to cache storage data, retain user sessions, memoize results, and keep invocation artifacts. When resources become scarce, *PraaS* follows traditional-OS design and transparently **swaps** out the state consisting of user-defined and durable objects and files, storing it in disks and cloud storage. Once the same instance of a process becomes active, the state is lazily loaded to memory. While stateless functions require users to access remote storage explicitly, adding performance and cost overhead to every invocation, the process loads a state only during the much less frequent restart.

The process strikes a new balance between serverless and traditional stateful and server-based applications: the lifetime of computing and storage is managed independently, but the state is retained close to compute resources, improving data locality and startup times. Traditional *functions* can now efficiently support stateful workloads with the same FaaS API, while retaining the simplicity of the model where the cloud provider is responsible for scaling resources (Figure 1). On the other hand, *services* implemented on top of FaaS consist of many functions connected with application-specific communication and scheduling logic. Services often benefit from having more control over resource provisioning than in the simple FaaS interface: serverless workflows have their own schedulers and executors [12, 15], parallel applications use customized launchers and communicators [17, 34, 48], and microservices have load balancers. A request that orchestrates many functions can be implemented much more efficiently than in a programming model of functions with coupled control and data paths. With processes, services can specify *what* resources are needed while leaving to the cloud provider the responsibility of deciding *how* they should be provisioned.

Process-as-a-Service (*PraaS*) is inspired by classical OS design and transfers concepts that have stood the test of time in the context of granular cloud computing. **Inter-Process Communication** defines a simple yet powerful messaging interface based only on two operations: **send** and **recv**, covering all types of function–to–function communication while hiding transport protocol: shared memory, TCP, or RDMA. 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 [34, 48]. Finally, instead of applying optimizations to decrease control overheads, we *bypass* the control plane overheads from the data path entirely [52] by exposing a **direct invocation channel** to the process that allows submitting the invocation payload as a

	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	Persistent	Persistent	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
Scaling Down To Zero	No	No	Yes	Yes

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

message – effectively separating control and data paths in the platform. By implementing responsibilities traditionally associated with operating systems, *PraaS* is a step towards a distributed cloud computing OS that provides a middle ground between the performance of persistent allocations and the elasticity of ephemeral workers (Figure 2).

We implemented *PraaS* atop AWS Fargate, a commercial black-box system of serverless containers, and a custom deployment of Docker containers. 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 with the same flexibility as serverless functions while reducing remote invocation latency by up to 17× and communication latency by up to 99%.

We make the following contributions:

- **Cloud processes** The new cloud abstraction unit that combines the elasticity and granularity of the function with the state and communication channels.
- ***PraaS*** A novel computing model that brings high-performance communication, increased data locality, and a fast data plane to serverless.
- **Experimental validation** We provide open-source implementations of PraaS on top of commercial and open-source FaaS platforms, demonstrating high scalability and efficiency in serverless benchmarks.

2 MOTIVATION

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 micro virtual machines [7]. A *cold* invocation requires allocation of a new sandbox that significantly increases invocation latency [19, 45]. Subsequent *warm* invocations achieve a lower latency by reusing existing sandboxes. Therefore, cloud systems employ complex and sophisticated retention and pre-warming strategies [21, 47, 62, 65], 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.

Storage Type	1B	100 kiB	10 MiB
Persistent storage (S3)	15.4 ± 4.3	29.3 ± 8.5	113.6 ± 15.9
Key-value storage (DynamoDB)	4.2 ± 0.5	6.2 ± 0.6	n/a
In-memory cache (Redis on EC2)	0.5 ± 0.06	1.17 ± 1.3	114.8 ± 46.1

Table 1: Access time [ms] to remote storage from AWS Lambda, mean with standard deviation. Python functions with 2 GiB RAM, with Redis 7.4 on *c3.xlarge*.

However, serverless has some prominent disadvantages: poor locality of data due to the non-existence of local state, high communication costs, and higher latency due to complex control planes [19, 25, 35, 69]. *PraaS* addresses the limitations while retaining the elasticity guarantees of FaaS.

2.1 Serverless State

The stateless nature of functions makes scalability and resource provisioning easier for the cloud provider, 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 [32, 64, 73]. 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 [32, 73]. In the FaaS model, removing remote storage access from the data path is impossible.

Serverless State In addition to user data placed in durable and replicated cloud storage, applications manage data that should be persisted for performance reasons. Caching is natural to many applications that manage user requests (sticky sessions) and retain results that are likely to be used in the future but are expensive to recompute, e.g., when fetching large data items in microservices or in incremental compilation. Finally, functions can produce ephemeral data, such as partitioned collections in Spark and results consumed by subsequent pipeline invocations [46]. This raises two important issues: how to retain data in the ephemeral serverless environment and how to match new invocations with the cached state?

Existing solutions address the first problem with automatically managed caches that retain storage data in functions [49, 55]. However, these focus on remote storage and do not support data produced by the function, and they cannot persist data across cold invocations. To solve the second problem, researchers proposed grouping connected invocations together through colors [6] and redesigning the programming model in a data-flow manner [72]. These solutions can optimize functions by redirecting new user requests to warm instances that previously hosted similar invocations. However, the locality is limited to warm containers as no data is persisted after downscaling. To fully benefit from the warm local state, the programming model must address both problems *simultaneously.*

2.2 Serverless Communication

Communication in FaaS has always been constrained as industrial offerings do not offer direct communication, forcing users to rely on storage or proxy-based communication - an expensive solution with high latency that lacks a portable API. State-of-the-art solutions implement communication through cloud storage, increasing latency, costs, and application complexity [34, 35, 43, 53]. Although direct network communication between functions could alleviate performance problems, functions do not offer the abstractions needed to communicate 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 [26, 27], but it is a complex process that applies only to two active functions simultaneously.

Furthermore, functions that want to receive messages need a full network stack, with a virtualized network device and a public IP address assigned. Network configuration increases startup latency and the cost of handling serverless workers. In practice, functions do not operate as servers that accept connections from many clients. Instead, functions need an interface to communicate efficiently with other functions.

Many serverless applications could benefit from efficient and direct communication: not only parallel applications in machine learning, HPC, and data analytics [20, 70], but also systems replacing traditional services that rely on direct and ordered TCP connections [18, 68].

2.3 Serverless Control and Data Planes

Modern platforms implement dynamic function placement through a centralized routing system [7]. It includes an abstraction of a REST API and a gateway with a persistent network address and uses an HTTP connection to hide the selection and allocation of function executors. The function input is forwarded to the central management 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 [58]. Finally, the input data is moved to a warm or cold sandbox, and the function returns the output through the gateway.

Expensive Requests The many steps of control logic add double–digit millisecond latency to each invocation [7, 16] 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 [19]. Serverless functions are predominantly short-running [59] and, as a consequence, the relative overhead on the multi-step function workflows and distributed applications is very high. Alternative approaches include manual provisioning, reusing function instances, decentralized scheduling, and direct invocations [8, 15, 20, 63, 72], but these optimizations do not offer a solution generalizable to all FaaS platforms and require manual scaling.

Request Alignment As shown before, functions benefit from accessing the state associated with prior invocations. However, FaaS control planes are oblivious to sticky sessions, and each upcoming request can be placed in any warm container. This leads to poor data locality, as functions cannot access each other's state unless it is placed in remote storage. While functions do not have a concept of state, processes do. There, the control plane could redirect invocation to an instance specified by the unique identifier returned from the previous request.

Request Multiplexing FaaS control planes tend to operate in a single-request mode: function container receives an invocation and becomes unavailable until it returns the result. While this setting is appropriate for compute-intensive workloads that consume all resources of a function, many workloads are I/O-bound [39], and a single server could process thousands of requests by employing event loops and multithreading. However, functions are limited to a single request by their invocation model. Serverless platforms allow batching requests with the help of queues, but in addition to increased costs of additional cloud services, batching introduces the trade-off between optimal batch size and latency, and complicates further the problem of poor state locality.

FaaS can be more than just a platform for irregular and lightweight workloads. However, to tap into massively parallel and granular computing [40, 42], FaaS platforms 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) Processes are equipped with persistent *state* which efficiently supports stateful applications (§3.1). (2) *Inter-process communication* defines data movement between processes and removes the dependency on external storage, enabling direct communication between ephemeral workers (§3.3). (3) Processes are scaled automatically by the cloud provider and use *data plane* that supports fast function invocations bypassing the control plane (§3.2). With the new process abstraction, we build *PraaS*, a platform taking advantage of cloud processes' new programming (§4).

A cloud 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 (Figure 3). When the cloud

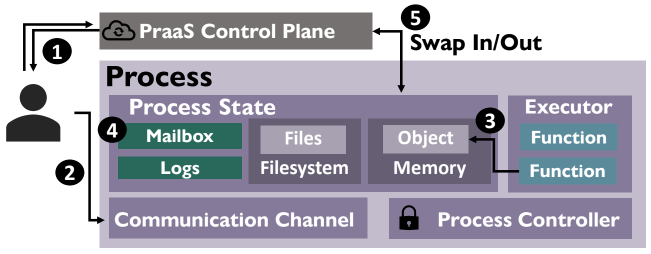


Figure 3: The process model in *PraaS*: ephemeral functions are executed in a *processs* with shared and persistent state. Communication channels provide quick user access and data exchange between functions.

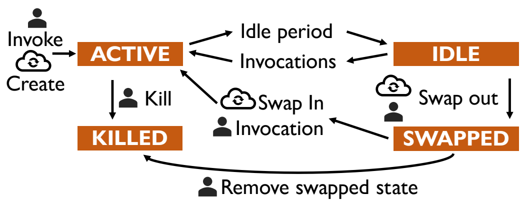


Figure 4: The life cycle of a cloud process. Process status changes in reaction to user operations (👤) or operator scaling actions (☁).

control plane allocates a process, it assigns it 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. Finally, the state is guaranteed to persist across container shutdowns, with the cloud responsible for swapping in and out the data (❺).

3.1 Locality with State

In FaaS, functions and microservices persist their state using remote cloud storage. In PraaS, we distinguish between the persistent and ephemeral state of an application. The persistent state cannot be lost and must be stored in a highly replicated storage to survive the crash of a virtual machine or a container; here, processes behave identically to IaaS and CaaS deployments. On the other hand, the ephemeral application state can be recreated after a crash. For example, in the case of the LaTeX microservice, the user files cached within the process can be served directly. Similarly, processed files and the generated PDF do not have to be recomputed, which could happen in FaaS when the container is evicted or invocations from the same user arrive at different warm instances. With a process state, we provide the opportunity for functions to distinguish between durable data, which must use replicated cloud storage in every type of deployment, and ephemeral state that is kept local but not lost upon eviction, as is the case in FaaS.

State Semantics 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 includes only a part of memory and the filesystem marked as storing the durable data of functions, and the remaining part of the working set is ephemeral (Figure 3). Thus, functions colocated within the same process can share state objects and improve data locality, e.g., by serving microservice requests in a process and using the state as a cache for requests and results. Functions within a single process can read and modify state, which operates as a reader-writer lock (Listing 1). This simple communication interface incorporates new state and communication features, requiring neither a complete redesign of serverless applications [72] nor dedicated compilers [30]. Depending on the underlying implementation and capabilities of the programming language, functions receive a data copy or obtain direct access to the shared memory.

PraaS processes are designed to be a single tenant only, as all functions in a process share the same state data. The isolation between functions handling different user data depends on the implementation of business logic in functions. If the function executes user code, then separate processes should be used for different tenants.

State Lifetime The new state cannot restrict cloud providers from scaling resources transparently, as in FaaS. Processes should be serverless; the cloud operator makes all allocation decisions, and users have no control over them. To that end, we extend the FaaS function model with a new state of being *swapped out* (Figure 4). A process is swapped out only when the sandbox is evicted, which in FaaS happens after several minutes of inactivity [19]. 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. Similarly to FaaS, the allocation is not persistent: the cloud provider controls the lifetime, and the process can be removed at any point. Process state enables the persistence of user data and execution of stateful functions without manual state management by users. Note that a process hosting a single function operates as a stateful FaaS (similar to a stateful entity in Durable Functions [13]), with the same ease of scaling and reclaiming resources.

```
# Access a copy or the original data over shared memory
data = state_write(key)
data_const = state_read(key)
# Change process state
state_commit(key, data)
# Send message over IPC
send(destination, key, data)
data = recv(source, key)
# Invoke function
invoke(destination, function, data)
```

Listing 1: New communication interface for functions.

3.2 Invocations with Control and Data Planes

Conceptually, a function invocation in a process 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. In the latter case, functions share the language runtime and can benefit from reduced memory overhead. The process can handle multiple function invocations simultaneously, and it is bounded by the resource limitations of the underlying sandbox. Users and cloud providers can limit the number of simultaneous invocations an individual process can start, for example, by setting the limit relative to the allocated memory. Larger workflows are supported by distributing the workload across multiple processes and communicating through the IPC interface (§4.2).

The FaaS simplicity relies on automatic scaling, and processes must support the same model for applications that do not have custom scheduling policies. Thus, functions can be invoked through the control plane, and invocation requests can supply the process ID to hint the system on which process to put the invocation. Additionally, orchestrators and load balancers can invoke functions more efficiently by sending the payload over the data plane (inter-process communication). The process controller receives invocation messages to start user functions. When the process reaches its resource limit, the invocation is rejected, notifying the control plane and orchestrators to scale up processes. When the control plane sends an eviction notice due to inactivity, the controller swaps the state to cloud storage.

Scaling Up New processes are allocated on-demand through FaaS executions or 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*). Since loading hundreds of megabytes of state data would significantly increase the cold startup time, and functions tend not to use the whole state in every invocation, processes can start by loading only the list of state contents. Actual data is loaded lazily: processes load the data in the background, and prioritize objects actually accessed by a function.

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, orchestrators and schedulers 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. Processes with an active connection to a migrating process could receive a packet with migration details and later 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, part of filesystem, and mailbox, as in Figure 3 — 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 can 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. 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.

3.3 Process Model with Communication

Compared to FaaS, functions executing in a cloud process have to use only six new primitives 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.

```
# Retrieve prior road computations
prior_route = praas.state_read(req["route_id"])
# Local computation
new_route = recompute_road(prior_route, req["destination"])
# Microservice updates route with new conditions
route = praas.invoke("traffic", "optimize", new_route)
# Store user data in the process state.
praas.state_commit(request["user"], route)
return route
```

Listing 2: Example of integrating process state into microservices for online maps.

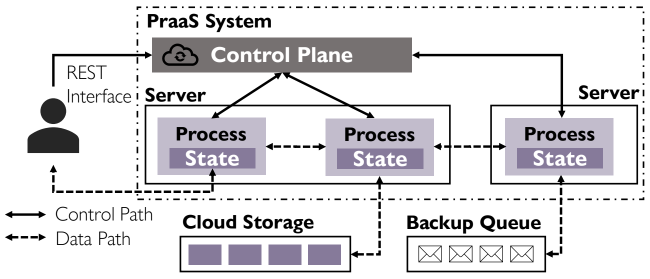


Figure 5: Platform architecture of *PraaS*.

Example We highlight crucial features of the process model with an example of an online map service (Listing 2). A microservice design will split across functions different tasks such as route finding, current traffic condition, or visualization. User applications will create many route update requests to change roads and receive the most recent traffic updates. Processing each such request requires warm data for a specific region. To avoid reinitializing the map cache from scratch after a period of inactivity, functions store it in the process state.

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 (Figure 5). Processes can be scaled automatically

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 [61].
Data Plane	Network data plane in Arrakis [52], kernel bypass in RDMA.

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

```
# Grouping processes and functions into a logical unit
app_id = create_application()
delete_application(app_id)
# FaaS Invocation on any process
result, pid = invoke(app_id, func, data)
# FaaS Sticky Invocation on a selected process
result = invoke(app_id, func, data, pid)
# PraaS Interface
pid, data_plane = create_process(app_id)
data_plane = swapin_process(app_id, pid)
status = swapout_process(app_id, pid)
delete_process(pid)
```

Listing 3: *PraaS* control plane REST interface.

by the cloud and are logically grouped into applications, enabling efficient communication and local invocations needed by serverless services.

4.1 Process Management

In *PraaS*, processes are grouped to create scalable applications spanning multiple server machines. 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 (Listing 3). 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. The system grows and shrinks with changes in the workload according to automatic scalability of functions (**FaaS Model**), and with user-driven scalability to acommodate custom orchestrators and schedulers (**PraaS Model**). However, process allocation is always 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 communication locality.

FaaS Model *PraaS* is backwards-compatible with the FaaS interface. Users can skip process management and directly invoke functions. There, the platform automatically manages a pool of processes and schedules function invocations over them. 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 the process identifier `pid` in request headers. Thus, processes can be used to implement *sticky sessions* [67] where requests from a single user are always handled by the same process. This type of invocation is *best-effort*: if the invocation request is rejected by the process, the control plane allocates a new clean process and sends the invocation there. Thus, users benefit from both automatic scalability and stateful functions.

PraaS Model New processes receive a clean state by default, but they can be initialized from a previously swapped state by providing the process identifier. 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 3 for simplicity). 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 accountability from the critical path of invocation to the control plane is essential to 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). 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.

```
# 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 4: Communication between functions A and B encodes data flow but does not expose the location nor the status of the recipient.

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 4). 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.

Asynchronous Communication The communication may not happen synchronously as the receiver might be swapped out. In such a scenario, messages can be delivered to a backup queue and will be 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.

Scaling Management 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*, we propose that 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.

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.

4.3 Function Invocations over 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 and schedulers 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.

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 propose 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.

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. We implement two distinct solutions: a custom *PraaS* control plane that manually deploys processes, and a second implementation extending Knative and Kubernetes. While the first implementation serves as the main prototype used in evaluation, we use the Kubernetes deployment to demonstrate the retrofitting of a serverless process model into state-of-the-art CaaS and FaaS platforms.

5.1 Main Prototype

We implement a *PraaS* control plane that deploys and processes running on top of AWS Fargate, a cloud service that outsources container management from the user, and manually scheduled Docker containers on virtual machines. 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 a 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.

The solution consists of roughly 13.5 thousand lines of code in C++, with an additional Python runtime for a process (400 lines). The control plane exposes an HTTP interface to end users (Listing 3), while the internal communication between processes, the control plane, and the data plane is done with binary serialized messages on top of TCP. We offer users a C++ SDK that encapsulates the data plane communication with a process and the REST requests needed to communicate with the control plane. 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 new transport protocols and network fabrics, e.g., QUIC and RDMA.

Process We propose that 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*. 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 support swapping process data, such as state, unread messages, and dedicated files, into S3 and Redis. Additionally, the direct deployment to Docker containers running on virtual machines can swap in and out to a locally mounted volume, demonstrating a full hierarchy of storage options for warm and cold process states.

Process State We define two modes of using the process state: copying objects with serialization and serialization-free object sharing. 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, UNIX domain sockets, or shared memory. 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 `state_*` operations, 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. In the current implementation, we use POSIX message queues instead of shared memory since serverless platforms such as Lambda and Fargate have limited support for shared memory devices.

5.2 Kubernetes

We demonstrate integration into existing serverless systems with the second prototype built as an extension to Kubernetes [3] and Knative [2].[1] There, we modify the control plane to manage processes as pods and store information on applications and processes in a Redis instance. 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.

We provide a custom process runtime implemented in Python, running a FastAPI server to manage control plane and data plane connections. The communication layer is implemented on top of WebSockets. The platform includes a function store, where users upload functions as Python wheels that can be dynamically installed on processes.

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.

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.

[1]An extended discussion of these platforms can be found in the Master thesis [54].

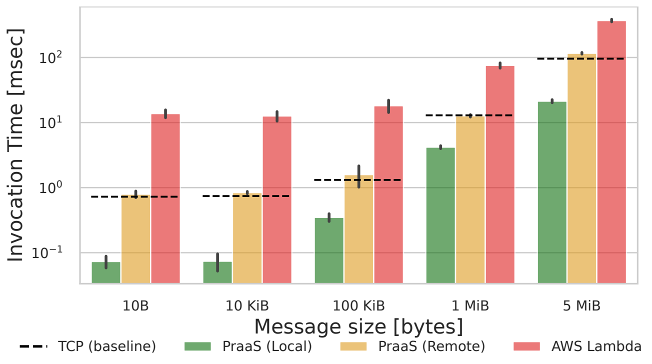


Figure 6: Invocation latency of a *no-op* function in *PraaS* on AWS Fargate, compared against TCP baseline (dashed horizontal line).

We place the benchmarker application in a virtual machine running Ubuntu 20.04 in the same cloud region, and invoke warm AWS Lambda and PraaS functions. We report the median time between start and end of each invocation from 100 repetitions. We invoke a remote *PraaS* function on a Fargate container with 1 CPU and 2 GB, which is equivalent to the Lambda configuration with 1792 MB and 1 vCPU. The version testing local follow-up invocations is using a Fargate container with 2 vCPU. Finally, we measure the baseline transfer over TCP by executing the `netperf` benchmark between a virtual machine and the Fargate container.

The results shown in Figure 6 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. *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. 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 (between 17 and 187 times faster than AWS Lambda), local invocations are limited in the current prototype by POSIX message queues. Queues are used within a single process to communicate between controller and OS processes executing functions, which is a bottleneck for communication.

The performance of message queues is limited by the maximum message size and number of messages in the queue; in our case, the hard limits imposed by the OS are ten messages of 8 kiB. Large payloads have to be split into smaller blocks, and the sender is required to push new messages iteratively once the receiver polls a message from the queue. Furthermore, the total memory allocated to message queues is limited to 800 kiB, restricting the total number of message queues with such parameters to ten. Thus, direct communication between two functions executing in the same PraaS process is not feasible, and all messages have to be relayed through the controller. As shown in Table 4, message queues can add 4x overhead than Unix Domain Sockets. Performance limitations of both IPC methods can be alleviated with zero-copy shared memory communication when available on the serverless platform.

		10 B	1 kiB	10 kiB	1 MiB	5 MiB
EC2	UDS	19.8	20.1	100.9	562.6	2027
	MQ	22.1	23.8	125	809.8	3629.7
Fargate	UDS	23.3	23.4	44	284.7	1398.5
	MQ	13.1	13.7	103.6	1204.2	5492.9

Table 4: Time of round-trip local communication [usec] over message queues and Unix Domain Sockets, between two OS processes running on Fargate (2 vCPU, 4 GiB) and EC2 virtual machine (c3.xlarge, 4 vCPUs).

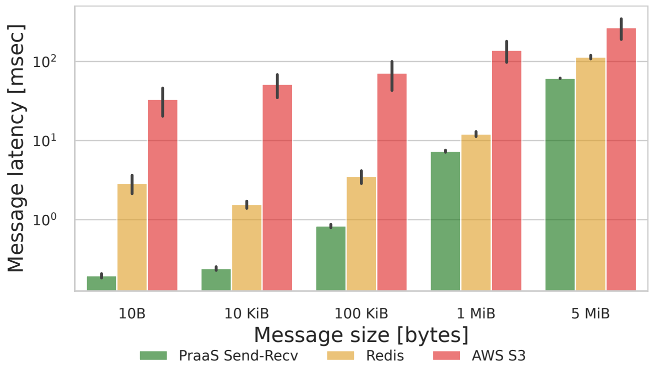
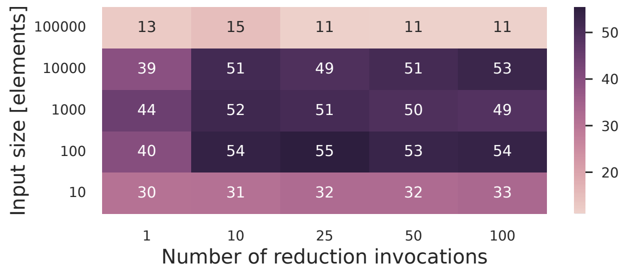


Figure 7: Communication latency of two *PraaS* processes running on Fargate with different communication channels.

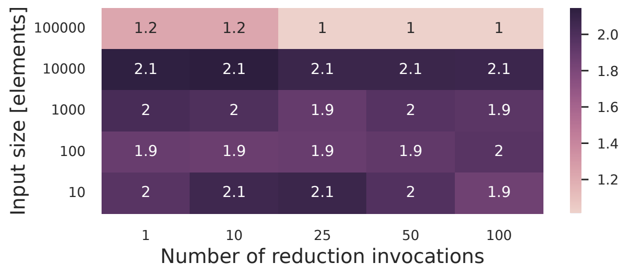
6.2 Inter-Function Communication

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 both storage systems, we use linear backoff to avoid extraneous charges (S3) and overloading the service (Redis), starting at 1 ms sleep and increasing by one with



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



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

Figure 8: The *reduction* benchmark storing state.

each failed communication. 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. All three benchmarks are executed as *PraaS* functions.

Figure 7 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 [55]. In addition to latency reductions, *PraaS* avoids significant costs and maintenance overheads associated with using S3 and setting up (and scaling) Redis instances.

In PraaS, the communication times grow with message size because of TCP performance and the dependency on internal IPC methods. At 5 MiB, the *PraaS* communication time reaches 61 ms, where the netperf reports almost 48 ms for remote data transfer. Then, message queues add 5.5 ms to transfer data between functions and their respective controllers.

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.

We evaluate a reduction that accumulates input data in a vector of 8 byte integers. We make an optimistic assumption that the result of the previous invocation is stored warm in the function's global memory, and only store the result in storage to avoid data loss. We evaluate the function with different input sizes and compute the time needed to invoke the reduction a varying number of times. We run Fargate with 1 vCPU and 2 GB memory in this benchmark and Redis on a c4.xlarge machine. We repeat the measurements 100 times, and we show in Figure 8 the speedup provided by *PraaS* using its persistent, swappable state compared to storing the state in S3 or Redis. *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 serverless microservice handling collaborative LaTeX editor, similar to the Overleaf project [4]. We implement four Python services that allow us to update project files, retrieve the newest file version, recompile the project, and retrieve the compiled PDF document. To support online editing, serverless functions must use 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 using the same process. In *PraaS*, functions place the contents of each file into the process memory. In both implementations, functions cache the last state of the project on the attached disk space and update locally stored files only when needed. Thus, when recompilation is requested, the function obtains keys and timestamps of project files from process state (*PraaS*) and from S3 (Lambda), compares timestamps with the locally cached files, and retrieves only files newer than the local cache version. Finally, since the process state is not replicated and user data could be lost in the event of a container failure, we use it only as a cache: `update-file` function stores the contents of the file both in process memory and S3.

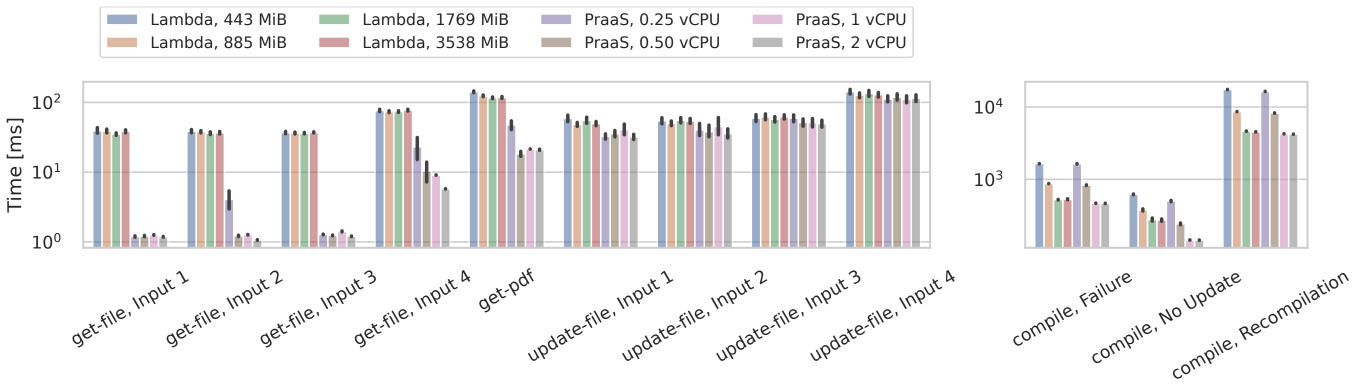


Figure 9: LaTeX microservice benchmark with serverless functions (AWS Lambda) and processes (AWS Fargate), serving LaTeX files from acmart 1.90a template [5]. Mean values with 95% confidence interval on a semi-log plot.

We evaluate each service with different inputs on AWS Lambda with S3 storage, and PraaS processes deployed on AWS Fargate. For compilation, we consider three scenarios representing realistic use cases: empty run triggered by reupload of an empty file, LaTeX failure caused by incorrect update erasing file's contents, and recompilation after updating multiple files. We change the Fargate container allocations and Lambda memory configurations to measure the impact of varying computing resources and I/O bandwidth availability, repeating each invocation 50 times. We measure the total client time of Lambda invocations and data plane PraaS executions. Lambda memory is tuned to use the same virtual CPU allocation as PraaS processes running 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 a an empty compilation run can be completed 1.23x - 1.91x faster than in a Lambda function. Thanks to the data plane and warm state, LaTeX files can be served to users up to 34.3x faster. When returning a larger PDF file (571 kiB), `get-file` is 3.3x - 13.4x faster in *PraaS* than on Lambda. *PraaS* has similar performance to Lambda on `update-file`, as we persist user files on S3 in both scenarios for durability. Nonetheless, *PraaS* can execute this function up to 1.82x faster, with Lambda matching our performance only on one 443 MB configuration that performs better than larger allocations.

6.5 Case Study - Machine Learning

To demonstrate the benefits of *PraaS*, we apply it to a workload from the distributed machine learning framework LambdaML [34]. We select the K-Means algorithm using the Higgs dataset and port it to the *PraaS* Kubernetes implementation. We compare *PraaS* against a version ported to Knative, which uses AWS S3 and Redis for communication. We use the original communication primitives for S3, and implement custom communication for Redis. In *PraaS*, we implement a naive *allreduce* in *PraaS* where each worker broadcasts new results to all other participants. Finally, we replace the original input reading procedure: instead of manually partitioning of input data uploaded to S3 buckets, we use a single input file and let functions read data at specified offset.

We execute a weak scaling benchmark with 8 workers on an AWS EKS cluster of consisting of 4 `t3a.large` EC2 nodes, each with 2 vCPUs and 8 GB of memory. We place two functions within a single *PraaS* process, and we deploy a 6-pod cluster of Redis 7.0.5 in the same cluster to provide the highest data locality. We execute the benchmark for 100 epochs, and repeated the measurement 10 times. Results shown in Figure 10 show that the IPC of *PraaS* performs better than an in-memory cluster, we speed up the runtime by 1.5 to 6 times against S3.

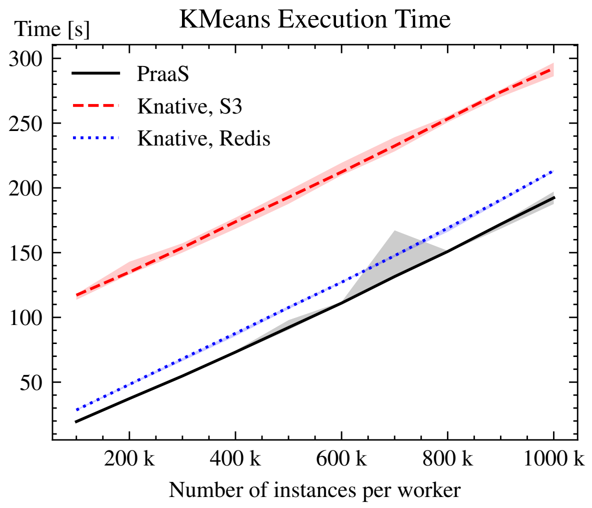


Figure 10: LambdaML with K-Means algorithm and Higgs dataset. *PraaS* against Knative with S3 and Redis.

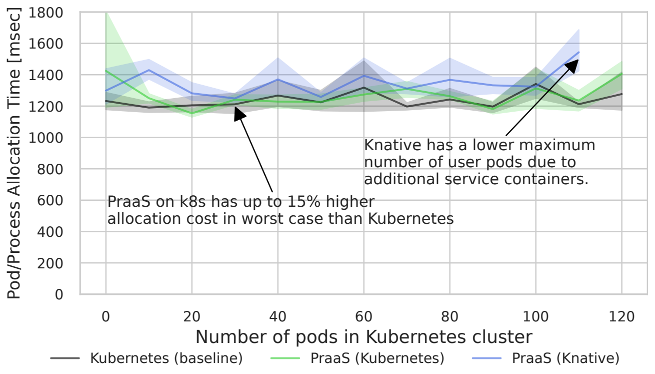


Figure 11: Allocating *PraaS* processes on managed Kubernetes services.

6.6 Trade-Offs

While the new process model requires minor adjustments to the lifecycle of serverless workers, these changes may introduce non-negligible overheads. Now we look into costs of process allocation, deallocation, and state swapping.

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. We run a benchmark that measures the time between user requesting a new process and receiving a message from it. We run this experiment on a four VM cluster using t3.medium EC2 instances, with each instance supporting up to 30 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 from five repetitions show that *PraaS* introduces a low overhead on top of Kubernetes and Knative (Figure 11). This overhead results from the acquiring lock on Redis and 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 the 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 the process reporting low data plane metrics that warrant deallocation and the moment the 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 evaluate the benchmark at a different system load by varying the number of active pods in the Kubernetes cluster, and repeat 5 times for each configuration. The time needed to delete a container in Kubernetes and PraaS (which builds on Kubernetes) does not differ significantly. Depending on workload and system noise, the median is about 1.7 to 1.8 seconds, and this latency comes primarily from Kubernetes logic to deallocate a pod. On the other hand, the Knative cannot directly delete pods but instead modifies deletion cost to guide the actual downscaling done by knative. There, we notice that this limitation of a purely serverless platform increases deletion time to over 60 seconds.

	1 MiB	5 MiB	10 MiB	50 MiB	100 MiB	200 MiB
Fargate	98.4	173	231.9	907.7	1719.3	3480.4
EC2	120.5	172.8	220.2	791.8	1525.7	2930.5

Table 5: Time of swapping [msec] in-memory state into AWS S3, from a process executing in a Fargate container and Docker container on EC2.

Swapping State *PraaS* retains containers in memory the same way as traditional serverless [19], using idle memory to increase warm startup frequency. The major difference is that we swap out the state once the process container is evicted. To understand the performance cost of this operation, we execute a function that transfers in-memory state to S3, and measure the time needed when executing on Fargate and in a Docker container on EC2. Table 5 presents the mean result from 20 repetitions. While swapping out 1 MB of data takes about 100 ms, this does not increase linearly with the size: swapping out 100x more data requires 12-17.5x more time.

Real-world Azure data demonstrates that 90% of the applications never consume more than 400MB of memory, and 50% of the applications allocate at most 170MB [59]. However, this estimate includes the entire memory of a function, which contains not only the durable state but also additional libraries, runtime, working memory, and temporary variables. In practice, only a fraction of data objects become state, limiting state-swapping overhead. The swapped states will also incur a storage cost proportional to the number of swap-in/swap-out operations and the size of the state. We estimate that this storage cost will not dominate the entire infrastructure cost and might even be compensated by reducing the initialization time that FaaS currently suffers.

6.7 Cost Analysis

Cost is another important trade-off of stateful serverless - we need to store more data in the memory of an instance, but at the same time, we can decrease costs by not making additional requests to the storage. While the state data can be often stored opportunistically in overprovisioned function memory [49], we make a pessimistic assumption that the process state requires additional memory. To evaluate *PraaS*, we estimate the hosting cloud's state in memory of a virtual machine (V) or a container (C), and use it as an estimation

	V_m (\$/hr)	C_m (\$/hr)	F_m (\$/hr)	G_V	G_C
AWS x86	$5.53 \cdot 10^{-3}$	$4.45 \cdot 10^{-3}$	$1.5 \cdot 10^{-2}$	63.13%	70.37%
AWS ARM	$3.53 \cdot 10^{-3}$	$3.56 \cdot 10^{-3}$	$1.2 \cdot 10^{-2}$	70.53%	70.33%
Azure	$4.95 \cdot 10^{-3}$	$4.45 \cdot 10^{-3}$	$1.23 \cdot 10^{-2}$	59.73%	63.82%

Table 6: Cost of *PraaS* state in comparison to FaaS provisioned instances, with V_m - cost of memory when changing from compute-optimized to memory-optimized VMs, C_m - memory cost in the managed container system, and F_m - fee for active serverless state. G_V and G_C are the cost decreases using VMs and containers rather than provisioned FaaS to store state.

of the cost of hosting cloud process state in memory. Then, we compare it against cost of provisioned FaaS instances (F). We select cloud platforms AWS (*us-east* region) and Azure (*East US* region).

FaaS 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.[2] There, in addition to paying for consumed computing resources, users are charged the active state fee F_s that depends on the memory size and the duration of provisioning.

PraaS To estimate the cost of retaining *PraaS* state alive in memory, we use the memory of other cloud services as a proxy for the cost to the cloud operator. We select virtual machines (V) and managed containers (C). First, we compute the added cost of changing from a *compute-optimized* to a *memory-optimized* virtual machine instances, needed to host process states. We divide the difference in hourly cost by the size of gained memory, which estimates the additional cost of adding one gigabyte of DRAM to a machine hosting *PraaS* (V_m). On AWS, we compare instances `c6i` (x86) and `c6g` (ARM) against `x2iedn` (x86) and `x2gd` (ARM). On Azure, we compare `Fs` and `Edsv5` series. We find that V_m is almost the same for each instance size, with minor variations on Azure. 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. There, compute and memory are billed separately, and we use cloud providers prices for each gigabyte of memory.

Summary 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 the state from provisioned FaaS to VMs and containers, respectively. The results presented in Table 6 prove that *PraaS* state can be offered at a lower cost, by up to 70.5%, 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 state at no additional cost.

[2]AWS provisioned concurrency instances can be recycled and reinitialized, making state persistence difficult, if not impossible, to implement in practice.

7 RELATED WORK

PraaS combines several features that have been explored in isolation in previous work. However, leveraging the new cloud process abstraction, *PraaS* is, to the best of our knowledge, the first platform to combine high-density sandboxes with a local durable state that can communicate and be invoked without the involvement of external storage and the control plane. Table 7 includes an overview of the main contributions of each research area that we detail below.

Ephemeral Storage augments 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 [10, 64], and elastic ephemeral caches [36, 49, 53, 55] 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 [32] to help developers build consistent and fault-tolerant systems on 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 on top of *PraaS* can be complemented with external databases and caches to facilitate consistency and fault tolerance.

Function Control Planes have also been extensively studied. Systems such as Speedo [22] and Nightcore [33] optimize function orchestration by either accelerating the control plane [22] or by completely skipping it [33] 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 [43], by avoiding moving data and allowing multiple functions to share the execution environment [38], or by offering direct network access to functions [70]. Pheromone [72] improves serverless workflows by binding control logic with ephemeral data objects, and Unum [41] proposes a decentralized orchestrator for FaaS workflows. Finally, Palette [6] adds color-based locality hints to serverless invocations.

Durable Functions [12, 13] (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

Areas of Research	Durable State	Invocation	Communication	Programming Model	Density
Ephemeral Storage	Remote	Control Plane	Proxy	Stateless Function	Low
Function Control Plane	Remote	**Direct**	Proxy/**Direct**	Stateless Function	**High**
Durable Functions	**Local**	Control Plane	Proxy	Stateful Entity	Low
Lightweight sandboxes	Remote	Control Plane	Proxy/**Direct**	Stateless Function	**High**
PraaS	**Local**	Control Plane and **Direct**	**Direct**	**Process**	**High**

Table 7: Comparison of different areas of related work with *PraaS*.

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 [7, 23] 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 [11, 24, 60] 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 [56] 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 [8, 14, 23, 50] and unikernels [37, 44, 74] to optimize process startup time 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 [71]. Similarly to the classical OS, a Cloud OS is expected to perform several tasks, such as resource allocation, 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 can be pushed directly to storage and enjoys the same guarantees as cloud queues.

Portability of PraaS Our process model makes no assumptions on the underlying virtualization technology, and is not restricted to any language, cloud, 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 storage.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement PSAP, No. 101002047), and from the Swiss State Secretariat for Education, Research and Innovation (SERI) under the SwissTwins project. This work was funded in part by Fundação para a Ciência e a Tecnologia, under the project UIDB/50021/2020. We thank Amazon Web Services for supporting this research with credits through the AWS Cloud Credit for Research. We also would like to thank anonymous reviewers and our shepherd for helping us improve the manuscript.

REFERENCES

[1] 2016. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed: 2024-10-19.

[2] 2021. Knative. https://knative.dev/. Accessed: 2024-10-19.

[3] 2021. Kubernetes. https://kubernetes.io/. Accessed: 2024-10-19.

[4] 2023. Overleaf: An open-source online real-time collaborative LaTeX editor. https://github.com/overleaf/overleaf. Accessed: 2024-10-19.

[5] 2024. acmart: ACM consolidated LaTeX styles. https://github.com/borisveytsman/acmart. Accessed: 2024-10-19.

[6] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 365–380. https://doi.org/10.1145/3552326.3567496

[7] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[8] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.

[9] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[10] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535

[11] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 645–650. https://www.usenix.org/conference/atc18/presentation/boucher

[12] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1591–1604. https://doi.org/10.14778/3529337.3529344

[13] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (oct 2021), 27 pages. https://doi.org/10.1145/3485510

[14] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[15] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3419111.3421286

[16] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. FuncX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/3369583.3392683

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

[18] Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, Konstantin Taranov, and Torsten Hoefler. 2024. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) *(HPDC '24)*. Association for Computing Machinery, New York, NY, USA, 94–108. https://doi.org/10.1145/3625549.3658661

[19] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. 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. https://doi.org/10.1145/3464298.3476133

[20] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 897–907. https://doi.org/10.1109/IPDPS54959.2023.00094

[21] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 356–370. https://doi.org/10.1145/3423211.3425690

[22] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2021. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 585–599. https://doi.org/10.1145/3472883.3486982

[23] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[24] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a Diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 45–59. https://doi.org/10.1145/3419111.3421297

[25] Adam Eivy and Joe Weinman. 2017. Be Wary of the Economics of "Serverless" Cloud Computing. *IEEE Cloud Computing* 4, 2 (2017), 6–12. https://doi.org/10.1109/MCC.2017.32

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

[27] Bryan Ford, Pyda Srisuresh, and Dan Kegel. 2005. Peer-to-Peer Communication across Network Address Translators. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, USA, 13.

[28] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[29] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, and et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[30] Zhiyuan Guo, Zachary Blanco, Mohammad Shahrad, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, Harry Xu, and Yiying Zhang. 2022. Resource-Centric Serverless Computing. *CoRR* abs/2206.13444 (2022). https://doi.org/10.48550/ARXIV.2206.13444 arXiv:2206.13444

[31] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *CoRR* abs/1812.03651 (2018). arXiv:1812.03651 http://arxiv.org/abs/1812.03651

[32] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[33] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings ofthe 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3445814.3446701

[34] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 857–871. https://doi.org/10.1145/3448016.3459240

[35] 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. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[36] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 427–444.

[37] Ricardo Koller and Dan Williams. 2017. Will Serverless End the Dominance of Linux in the Cloud?. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 169–173. https://doi.org/10.1145/3102980.3103008

[38] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[39] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3620678.3624648

[40] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 149–154. https://doi.org/10.1145/3317550.3321447

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

[42] Pedro García López, Aleksander Slominski, Michael Behrendt, and Bernard Metzler. 2021. Serverless Predictions: 2021-2030. *CoRR* abs/2104.03075 (2021). arXiv:2104.03075 https://arxiv.org/abs/2104.03075

[43] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[44] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[45] Johannes Manner, Martin EndreB, Tobias Heckel, and Guido Wirtz. 2018. Cold Start Influencing Factors in Function as a Service. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)* (2018), 181–188.

[46] Alex Merenstein, Vasily Tarasov, Ali Anwar, Scott Guthridge, and Erez Zadok. 2023. F3: Serving Files Efficiently in Serverless Computing. In *Proceedings of the 16th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '23)*. Association for Computing Machinery, New York, NY, USA, 8–21. https://doi.org/10.1145/3579370.3594771

[47] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[48] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International*

Conference on Management of Data (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 115–130. https://doi.org/10.1145/3318464.3389758

[49] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239

[50] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[51] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[52] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[53] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[54] Gyorgy Rethy. 2022. Process-as-a-Service Computing on Modern Serverless Platforms. https://www.research-collection.ethz.ch/handle/20.500.11850/599515. Master's Thesis.

[55] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[56] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan

[57] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (April 2021), 76–84. https://doi.org/10.1145/3406011

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

[59] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[60] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 419–433.

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

[62] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3423211.3425682

[63] Arjun Singhvi, Kevin Houck, Arjun Balasubramanian, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2019. Archipelago: A Scalable Low-Latency Serverless Platform. arXiv:1911.09849 [cs.DC]

[64] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[65] Kun Suo, Junggab Son, Dazhao Cheng, Wei Chen, and Sabur Baidya. 2021. Tackling Cold Start of Serverless Applications by Efficient and Adaptive Container Runtime Reusing. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 433–443. https://doi.org/10.1109/Cluster48925.2021.00018

[66] Amoghavarsha Suresh and Anshul Gandhi. 2021. ServerMore: Opportunistic Execution of Serverless Functions in the Cloud *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 570–584. https://doi.org/10.1145/3472883.3486979

[67] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[68] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 267–281. https://www.usenix.org/conference/fast20/presentation/wang-ao

[69] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 133–145.

[70] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*.

[71] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. 2010. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1807128.1807132

[72] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1489–1504. https://www.usenix.org/conference/nsdi23/presentation/yu

[73] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless

workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[74] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfei Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization *(USENIX ATC '18)*. USENIX Association, USA, 173–185.

[75] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. https://doi.org/10.1145/3477132.3483580