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

Marcin Copik  
ETH Züri

Alexandru Calotoiu  
ETH Züri

Rodrigo Bruno  
INESC-ID, IST, ULisboa

Roman Böhringer  
ETH Züri

Torsten Hoefler  
ETH Züri

Abstract

Fine-grained and ephemeral functions power many new applications that benefit from elastic scaling and lower computing costs of serverless platforms. However, they are hampered by expensive and limited communication, and high invocation latency. Functions cannot keep track of state across invocations and must rely on remote storage, making workflows and applications with dependencies between tasks, or relying on communication between workers, difficult to implement using Function-as-a-Service (FaaS) computing.

To continue the serverless revolution, we introduce the concept of Process as a Service (PraaS) and show how established operating system abstractions can be adapted to model and implement dynamically provisioned cloud computing workers. We present the new serverless data plane that improves invocation performance by up to 32 times while preserving the ephemeral and elastic nature of serverless workers. Finally, we build a unified and portable communication interface for serverless, enabling optimized peer–to–peer communication allowing workers to solve problems in parallel. to save costs.

1 Introduction

In less than a decade, Function-as-a-Service (FaaS), has established itself as one of the fundamental programming models in the cloud. In FaaS, users invoke short-running and stateless functions and benefit from the pay–as–you–go billing and lower costs, while cloud providers gain more efficient resource usage and opportunities to reuse idle hardware [1,2]. Serverless functions, the computing model FaaS is based on, have been used in web applications, media processing, data analytics, machine learning, and scientific computing [3,4]. Even though FaaS has achieved remarkable success in decreasing the costs of stateless computations and microservices, the wide adoption of serverless computing is hindered by the limitations of its execution model [4,10,12].

As an example of a trending workload in serverless systems, we consider distributed machine learning solutions [4,13–16]. There, each invoked function must compute new gradients using a subset of data. While the stateless nature of FaaS simplifies deployment and resource management, combining the new weights and using them in a next round of invocations incurs major performance overheads as in each round of updates the output must be written and read from external cloud storage, and each new round requires an invocation that goes through the entire cloud control plane.

Without direct communication channels, a service like serverless federated machine learning is difficult to implement and inefficient. The lack of state management in ephemeral functions [10,12] can partially be addressed with workarounds such as storage-based shared states, transactions, and logs [17,20]. However, these solutions rely on manually managed non-serverless storage instances, and they negatively impact the execution times, throughput, and cost of serverless systems [12,20]. The performance of such systems could be vastly improved by taking advantage of local available memory rather than constantly needing to access remote, slow storage and by having the capability to bypass the control plane for repeated invocations.

We propose a new paradigm in the serverless world: Process-as-a-Service (PraaS) — we incorporate state seamlessly, preserve the ephemeral nature of FaaS computations, and overhaul the serverless system architecture. PraaS is heavily inspired by classical Operating System (OS) design and transfers concepts that stood the test of time into the context of granular cloud computing. Specifically, this new model allows us to address three of the most significant constraints of current FaaS systems: lack of consistent and low-latency state, a complex control plane involved in each invocation, and a lack of unified and portable communication interface.

PraaS is a new step in the evolution of cloud com-
A further benefit of PraaS is the ability to move data directly between sessions within a single process. This allows users to deploy large-scale applications such as distributed ML that can communicate directly by accessing state memory. Until now, inter-function communication has been constrained, forcing users to implement standard communication patterns on top of cloud storage [4, 8] and introducing additional costs, non-negligible latencies, or requiring non-serverless or even manually managed storage services. Inspired by partitioned global address space (PGAS), we propose a global memory model for serverless functions backed by reliable and persistent storage. We provide a communication model for serverless that adjusts the requirements of direct messaging to ephemeral functions and enables fast communication in a portable and abstract interface effectively replacing the functions traditionally provided by the control plane: direct invocations using PraaS are 32 times faster than using AWS Lambda, for both small and large payload sizes.

Figure 1: Evolution of computing platforms in the cloud - PraaS enables state persistence for ephemeral workers.

To summarize, we make the following contributions:

- The new concept of serverless processes that provides an efficient model for stateful FaaS functions.
- An optimized serverless data plane with fast and direct invocations for scalable workflows and high-performance FaaS.
- A global memory model for serverless functions that permits optimized function–to–function communication and provides an interface portable across systems and cloud providers.

2 Motivation

A further benefit of PraaS is the ability to move data directly between sessions within a single process. This allows users to deploy large-scale applications such as distributed ML that can communicate directly by accessing state memory. Until now, inter-function communication has been constrained, forcing users to implement standard communication patterns on top of cloud storage [4, 8] and introducing additional costs, non-negligible latencies, or requiring non-serverless or even manually managed storage services. Inspired by partitioned global address space (PGAS), we propose a global memory model for serverless functions backed by reliable and persistent storage. We provide a communication model for serverless that adjusts the requirements of direct messaging to ephemeral functions and enables fast communication in a portable and abstract interface effectively replacing the functions traditionally provided by the control plane: direct invocations using PraaS are 32 times faster than using AWS Lambda, for both small and large payload sizes.

To summarize, we make the following contributions:

- The new concept of serverless processes that provides an efficient model for stateful FaaS functions.
- An optimized serverless data plane with fast and direct invocations for scalable workflows and high-performance FaaS.
- A global memory model for serverless functions that permits optimized function–to–function communication and provides an interface portable across systems and cloud providers.

Figure 2: FaaS control plane: each invocation includes the management overhead (M).
ers and lightweight virtual machines [29]. A cold invocation requires allocation of a new sandbox adding significant overhead to the invocation latency [11][30]. The consecutive warm invocations can benefit from reusing sandboxes by a preceding cold invocation, achieving lower invocation latency. Therefore, cloud systems attempt to retain virtual machines and containers after the invocation, employing complex and sophisticated retain and prewarming strategies [31][34], and trading off higher memory consumption for faster executions. In addition, the flexible pay-as-you-go billing is another significant advantage of serverless systems: users are charged only for computation time and resources used. However, along with the benefits serverless computing provides, it does have some disadvantages such as higher latencies due to complex control planes, poor locality of data due to the nonexistence of local state, and high communication costs. [10][11][35][36].

2.1 Serverless Control Plane

The serverless execution model requires that the function executors are allocated dynamically, and users do not provision cloud resources. In practice, most modern FaaS systems implement this as a centralized management and routing system that moves user data to a function instance. Figure 2 presents a high-level overview of the serverless control plane. It includes an abstraction of a REST API and a gateway with a persistent network address, and uses an HTTP connection (11) to hide the on-the-fly selection and allocation of function executors. Cloud events and gateway requests trigger (1) function invocations. The function’s input is forwarded to the central management (11) responsible for authorization, allocation of resources, and routing execution to the selected server. In AWS Lambda, the control logic is responsible for authorizing requests, managing sandbox instances, and placing the execution on a cloud server [29]. In OpenWhisk [57], the critical path of function execution is even longer. Each invocation includes a frontend webserver, controller, database, load balancer, and a message queue [38]. Finally, the function’s data moves to a warm (4) or cold (5) sandbox. When the execution is finished, the function’s output returns to the user via the gateway (1).

The many steps of control logic add double-digit milliseconds latency to each invocation and require copying user payload multiple times, even though serverless systems optimize cold startups, and subsequent invocations reuse the same warm sandbox when available. Overheads of the control plane can dominate the execution time, and are much higher than the network transmission time needed to transfer input arguments [11]. The long and complex invocation path is even more visible in distributed applications and serverless workflows that invoke functions many times with large payloads. The optimization effort of serverless workflows is guided towards reusing function instances and provisioning dedicated stateful executors [39][42], but does not offer a solution can be generalized. Alternative approaches include decentralized scheduling and direct allocations [3][39], but they are limited to systems optimized for serverless workflows and RDMA acceleration.

Observation: Every serverless invocation incurs overheads for authentication, resource management, and data copying across cloud services. However, the function placement does not change in many warm invocations, making the control logic unnecessary.

2.2 Serverless State

The stateless nature of functions makes them easy to schedule for cloud providers, but puts significant constraint on the programmability FaaS systems. Computing resources are allocated with ephemeral memory storage that cannot be guaranteed to persist across invocations. Since many applications require retaining state between invocations, stateful functions place their state in remote cloud storage [18][20]. While function’s state is located in storage far away from the compute resources, fetching and updating state adds dozens of milliseconds of latency to the execution (Table 1), resulting in significant performance overheads [19][20].

Relying on remote storage to implement stateful functions requires every invocation to incur into storage overheads, even if function sandboxes are kept warm. With the current FaaS model, it is not possible to overcome the remote storage access from the data path.

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

2.3 Serverless Communication

Communication in FaaS has always been constrained. The message-passing paradigm cannot be applied directly to the ephemeral functions, as workers’ lifetime is not defined, and functions require mailboxes to store and
access messages. Unfortunately, the widely-used communication pattern with persistent cloud storage adds non-trivial latencies and costs [10]. While direct network communication between functions could alleviate the performance issues (Fig. [3]), functions do not offer the abstractions needed to implement communication on groups of functions with dynamic group membership. Furthermore, in typical FaaS deployments, functions operate behind NAT and cannot accept incoming connections. UDP and TCP hole punching [43, 44] must be applied to establish communication between function instances. Not only is it a complex process, but the connections can only be created between two active functions - the programming model of FaaS does not provide mailboxes needed to send messages reliably.

**Observation** Cheap and scalable communication is necessary to build parallel and distributed applications, but serverless programming models lack an efficient fabric and interface for communication.

**Summary** FaaS can be much more than just a platform for irregular and lightweight workloads, and the serverless programming model aligns well with requirements for massive parallel and granular computing [23, 24]. Nevertheless, serverless systems must first overcome critical limitations: complex control plane involved in every invocation, lack of a fast and cheap function state, and expensive, storage-based communication.

3 PraaS: Process as a Service

In this section, we present the process as a service model starting with the core design overview and then providing details for its features and components. A process (Figure 4) is a natural extension of ephemeral functions with a decoupled and transient state. A process can support multiple users, isolated from each other if necessary, associated with one tenant. Multiple sessions within PraaS can cooperate using globally accessible process memory, using methods described in Sec. 4.1.

![Figure 3: Latency and throughput for peer-to-peer TCP communication between Lambda functions.](image)

![Figure 4: The process model in PraaS: ephemeral functions are executed in the context of sessions with shared and transient state and communication channels for quick user access.](image)

<table>
<thead>
<tr>
<th>PraaS Concept</th>
<th>Inspiration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>POSIX process model.</td>
</tr>
<tr>
<td>Function</td>
<td>Thread in a process.</td>
</tr>
<tr>
<td>State</td>
<td>POSIX process memory.</td>
</tr>
<tr>
<td>Session Persistence</td>
<td>Swapping memory pages.</td>
</tr>
<tr>
<td>Communication channel</td>
<td>Socket in a process.</td>
</tr>
<tr>
<td>Communication model</td>
<td>Remote memory access, TCP</td>
</tr>
<tr>
<td>Data plane</td>
<td>Network data plane in Arrakis [22], kernel bypass in RDMA.</td>
</tr>
</tbody>
</table>

Table 2: In PraaS, we show how the major concepts of systems design can be used to lift the process model into a distributed and serverless space.
Sessions encompass a unique identifier, the communication channel used to route invocations to already warm containers, and the transient state memory that is available to functions. This memory is guaranteed to be locally accessible to functions invoked in the context of the session, ensuring fast access. For this reason, session memory is statically allocated at creation. Sessions can dynamically allocate additional process memory on demand (Sec. 4.1).

**Isolation and Sharing** Sessions allow PraaS to handle workloads associated with different users and workflows in a single process instance. Each session has its own private memory that is isolated from other sessions (each session has its own page table). When a function is invoked, it can only access memory within its session, as shown in Figure 4 - thus functions within a session can share memory and cooperate. The user is responsible for avoiding data races (e.g. locking). Functions cannot access the memory of a different session directly, but can cooperate with functions within other sessions by using global process memory, which we detail in Sec. 4.1.

Sessions can handle multiple invocations simultaneously, but are bound by the memory and compute limitations of the system they run on. Cloud providers can limit the number of simultaneous invocations an individual session can start, for example in relation to the memory the session has allocated. To allow larger workflows, we discuss scalable management and communication of multiple sessions within a process in Sec. 4.1.

**Example** We consider the widely-used pattern of using FaaS where persistent and IaaS-based applications offload computational workload to cheap and elastic serverless accelerators. A user can employ the same session to invoke functions, and these functions will access the same shared memory pool. The session model in PraaS allows consecutive or concurrent invocations to always access the warm and initialized applications’ state. Furthermore, sessions enable enhanced data sharing, e.g., reusing a large machine learning model across many inference functions, allowing for local communication and decreasing management overhead.

**Swapping.** Sessions are not persistent: their lifetime is limited and they can be removed at any point by the cloud provider. However, sessions are virtualized: their memory is shared across invocations within a single session and it does not perish once the sandbox - container or a virtual machine - is removed.

Instead, sessions are swapped out to persistent cloud storage - and the standard policy is to swap sessions after a period of inactivity. Within a process, selecting different sessions allows users to isolate their private data from other users even if belonging to the same tenant.

Furthermore, a process is considered active when it has active sessions, and we will not evict a process until all of its sessions are swapped out. Sessions provide a natural way of defining process lifetime and selecting process containers for eviction. By default, when a session is swapped out to storage by the session controller, all current state information is written to persistent cloud storage under a unique session ID. If a session is no longer required, the user can also completely remove a session along with any associated state information.

Users have the option to create user-defined checkpointing handlers that define how a sessions’ state should be stored into persistent and reliable cloud storage with the session ID to minimize the storage required.

When a function is invoked, the user can include the session identification to recover from a saved state. When a new process is allocated, function invocations can be queued along with their session IDs, such that the sandbox initialization can be overlapped with downloading the state from storage, helping to hide the startup latency for stateful functions.

### 3.2 Function Invocation Handling

PraaS functions exist in the context of sessions, and it is the session they are associated with that determines how they are handled. If the session is active, the user connects directly to the correct communication channel, bypassing the control plane altogether and ensuring a zero-copy path from the user to function memory.

If the process is not running that session, the control plane is involved: the system then checks whether state information matching that session already exists in storage. If yes, the session is swapped in and the invocation can use all information stored in the session. Otherwise, a new session is allocated. In both of these instances, the communication channel information that allows fast subsequent invocations is forwarded to the user. We multiple functions can share a communication channel within a session via multiplexing.

**Local dispatch** We allow users to specify a direct dependency on the result of one or many previous invocations in each execution request. The execution request is stalled when there is a match between specified dependencies and active invocations. Co-locating invocations is an important optimization technique in serverless workflows, and the local encoding of dependencies avoids interactions with workflow orchestrators. Furthermore, dependencies allow for offloading function invocations while preserving a first-in, first-out order of invocations. To implement FIFO executions in the current serverless model, users must manually wait for
Multi-source functions  Serverless functions can be seen as unary operations, but the practice has shown that many aggregation and workflow functions receive input from more than one function predecessor. Thus, PraaS can start a multi-source function as soon as all dependencies are fulfilled. The multi-source functions can be used in serverless workflows to efficiently implement data flow between functions.

Example  Aggregation functions are used to implement the reduce and allreduce functionalities in serverless workloads [4, 8, 46]. Each aggregation function receives input from more than one predecessor function. In FaaS, users must use workflow orchestrators and cloud storage triggers to handle aggregation, as functions cannot be invoked simultaneously. FaaS lacks support for more complex logic, i.e., functions are invoked on each new file in the storage, and they must manually check if all arguments have already been placed in the storage. In PraaS, users define an n-ary reduce function instead. When clients invoke the reduce function through a single session, they attach the same invocation identification, and the process controller counts the delivered arguments. The function is invoked as soon as all input arguments arrive. The reduction is computed within a single process, saving the costs and latencies introduced by cloud storage.

3.3 PraaS Data Plane

PraaS helps to minimize FaaS latencies with the new concept of a serverless data plane optimized for low-latency and high-throughput invocations (Fig. 5). A full serverless invocation needs to authorize requests, select and optionally allocate resources, and redirect users’ payload to the function executor. When many execution requests are redirected to the same warm container, the repeated control operations are redundant and could then be removed from consecutive invocations. Therefore, for as long as the authorization stays valid, the invocation can bypass the control operations entirely and move data directly from the user to the function executor in a process.

In PraaS, the first time a function is invoked (1), the control plane notifies the session controller on an available subprocess to allocate a session (2), and the serverless system performs all configuration and authorization tasks at the point. The user receives connection information to be able to easily and quickly reconnect (3). Each consecutive function in the same session is invoked via a direct connection to the process (4), as long as the session stays alive. Function invocation received data from the user to the session’s state memory, and the zero-copy approach helps to achieve high throughput on larger payloads. Thus, invocation latency is bounded only by the network fabric and the performance of function thread allocation.

3.4 Scaling Sessions

The PraaS process is a virtual, distributed entity. To ensure a practical, efficient implementation we introduce subprocesses, representing the actual containers used to host sessions. Subprocesses are invisible to users, and are automatically managed by the cloud provider. The cloud provider can spawn additional subprocesses to handle an increased load if the number of concurrent sessions within a process is high, can remove subprocesses with no active sessions, and even compact sessions from multiple subprocesses unto one subprocess to improve resource utilization. For this purpose, the process also contains a global controller keeping track of subprocesses along with the sessions they hosted.

Migration & Compaction.  Sessions distributed across subprocesses to allow for better load balancing and resource retrieval by the cloud provider. Clients with active connections to a session being migrated receive a packet with migration details, and they can establish a connection to the new communication channel.

4 Serverless Communication in PraaS

Serverless computing has been adopted to support distributed applications and workflows, but the limitations of the FaaS programming model require complex and domain-specific optimizations in scheduling and communication [47,49]. Most advances in serverless com-
puting focus on optimizing north-south communication rather than east-west communication.

FaaS applications include a few communication patterns - message-passing, usually implemented as propagation of a function's output to the input of a consecutive invocation [49]; an aggregation of the outputs from many functions [46, 50], and using data cached in nearby instances for bypassing storage and faster access [42, 51].

In contrast, PraaS provides global memory, an efficient and direct communication model for serverless applications (Sec. 4.1) bypassing cloud storage but backed up by it. Our communication model does not concentrate on process invocations since they have a limited lifetime, but it is focused on data movement operations, allowing dynamically reshappable applications to benefit from peer–to–peer communication.

### 4.1 Global Memory

In PraaS, we provide a global communication abstraction layer to enable optimized process session-to-session data exchange and hide the complexity of serverless communication. This communication layer is inspired by MPI Remote Memory Access [52]. We present the overview of the global memory system in Figure 6. Apart from the memory sessions can privately use, which is guaranteed to be located on the subprocess where the session is hosted, sessions can also write to global memory. The system provides no guarantees on which subprocess a region of global memory is allocated. The global controller stores the location of global memory objects. The global memory is backed up by persistent cloud storage. Memory objects are accessed through a memory interface which hides data operations conducted with storage and peer–to–peer communication.

**Global controller** The global memory layout implements the message-passing paradigm in serverless by replacing storage operations with direct communication. We include a global controller service as an external, persistent component of the system (Fig. 6), and make it responsible for monitoring global memory object location across subprocesses and relaying requests from sessions to local memory controllers within subprocesses.

**Interface** Process functions interact with the global process memory using a simple interface that resembles memory addressing operations in OS-level processes. In particular, read and write operations accept an identifier from a global namespace (think of the address space in OS-level processes). It is also possible to allocate and de-allocate global memory using an interface similar to malloc and free.

**Communication** Storage-based communication patterns are slow and expensive [10, 53, 54], but the function–to–function communication is difficult to maintain in FaaS due to the ephemeral nature of workers. Sessions can communicate with other sessions by reading and writing memory objects in the global process memory (Fig. 6). To establish such a communication channel, a session requests the location of an object from the global communication controller (Fig. 6). The global communication controller instructs the local memory controller on the subprocess where the object is hosted and the originating session to create the communication channel (Fig. 6). Therefore, any such operation does not require sender and receiver parties to push updates to external services and poll for changes, respectively. Since the global memory objects identify communication targets, we establish message-passing without identifying and enumerating ephemeral and unreliable functions.

Furthermore, the communication replaces storage read operations with sharing objects between process sessions since p2p communication achieves higher bandwidths and lower costs.

**Example** We consider a group of functions that compute partial results, apply a global reduction to synchronize the progress, and continue with the next batch of tasks, as is the case in the distributed machine learning training [4]. In FaaS, functions must resort to storage to communicate results. Furthermore, the lack of state means that functions are invoked again for the iteration and users accept the penalty of reinitialization, or functions are artificially kept alive to keep the data locality and they incur costs from unnecessary waiting in the communication round. The PraaS model provides an alternative that is more affordable and more efficient at the same time: functions store objects corresponding to their partial results in the shared session or process memory, allowing them to move reduction data directly to the destination, and they stop incurring computing charges immediately afterward. However, since session state is not immediately evicted, functions benefit from the warm environment and better data locality in the next iteration.

**Dynamic subprocess membership** In PraaS, changes to the subprocess configuration can be triggered by the
cloud provider to scale the system up or down according to demand. Each change must be atomic to ensure the validity of exchanged messages. The global memory controller stores where global memory objects are located across all subprocesses, and this information is updated when the overall subprocess configuration changes. We define this as an epoch change, and results in a two step operation. The controller announces to all subprocesses that a configuration change is necessary, the subprocesses notify their sessions. Once all outstanding communication of the last epoch is handled the sessions notify their subprocesses. Then, the subprocesses notify the controller, the configuration is changed and the subprocesses and their sessions are then notified of the location of all subprocess sessions. Thus, each change in status begins a new epoch, and the global memory controller distributes epoch change announcements to all participating subprocesses. The change notification allows applications to readjust messaging policies gracefully and avoid data failures in the presence of ephemeral workers.

5 Praas in Practice

In this section we will discuss how our proposed design can be implemented in practice, focusing on four major aspects: the container serving our runtime, the control plane, how global memory works and the client library.

5.1 Control Plane

The main responsibility of the control plane is to accept user requests and manage processes and sessions. The outward facing part of the system is an HTTP frontend with the following functionality:

- allocate_session Creates a new session. The allocation must specify a maximum amount of memory the session can use. The session is hosted on a subprocess with enough resources to support it;
- retrieve_session Loads session state from storage;
- invoke Allocates a new session or retrieves an existing one and passes the function invocation payload;
- support for process management such as creating a new process or deleting one.

Processes, sessions and invocation are all uniquely identified, and have a hierarchical structure: invocations are hosted on sessions, sessions are hosted on subprocesses and a group of subprocesses forms a process.

The control plane relies on an in-memory Redis cache to store information about which sessions belong to which process as well as which sessions are live and which are swapped.

Execution backend Subprocesses are managed by the system, and scaled up and down as the load on the Praas system varies. To run and manage containers running these subprocesses, we deploy executor servers that are connected to the control plane. The executor server fulfills requests to create additional subprocesses if the system load is high, or requests to compact sessions from multiple subprocesses with a low load. The subprocesses can be allocated to servers in a round-robin fashion, but other optimized allocation policies can be implemented. Our system does not have any restrictions in this regard, so low-latency schedulers like the one in Lambda can be supported [29]. The executor server can take the form of either a VM or a serverless function.

Subprocess The subprocess runs a session controller, that accepts requests from the control plane and manages local sessions. When receiving an allocation request, it forks a new session. It also reports sessions being swapped out or deleted to the control plane. The subprocess also runs the local memory controller, which we detail in Sec. 5.3.

5.2 Session

The main components of sessions are locally available memory that can be swapped out to cloud storage (AWS S3 in our implementation), and the data plane TCP connection. Invocation payload is received directly into memory and can be accessed by the invoked function. To run user code, a thread pool dispatches the invoked functions. Each session has an upper cap for the number of active functions that can be set by the cloud provider, for example in relation to memory used - limiting the thread pool to \( N \) threads. Functions queued beyond this number will wait until resources become available.

Sessions make usage metrics available to the session controller via session memory. The controller uses these data plane invocation metrics to decide when sessions are no longer in use and can be swapped out. A session also shuts down when the user closes the connection to it on her side.

Code Deployment We propose that users define containers with both code and libraries necessary to deploy in the Praas system. This container is extended by our monitoring system and runs with restricted permissions, similar to Lambda C++ containers. Only the Praas controller runs with root privileges.

5.3 Global Memory

To implement a global view of memory physically distributed across subprocess, each subprocess runs a local memory controller that acts as a simple server to answer session’s requests for create, put, get, and delete
operations of objects. While the interface is simple, the mechanism behind it is more involved: when a session requests an object, it will first request it from the local memory controller. If the object is not on the same subprocess, a request to the global memory controller is made, and a NAT hole punching operation begins. The global memory controller runs a page table with information linking objects to subprocesses composed of tuples like \{Object\_name, Location\}. The global memory informs the local memory controller owning the object to finalize the connection using NAT hole punching on its side. Once the connection exists, future communication can occur without involving the global memory controller.

5.4 Billing and Accounting

Similarly to current FaaS and IaaS platforms, PraaS allows users to create sessions with a predefined amount of memory and CPU resources. In addition specifying the resources for local sessions, users also need to indicate how large the process memory should be. To make this task simpler, PraaS allows users to specify process memory increments per session so that the total process memory grows linearly with the number of sessions. Separating session memory from process memory allocation results in better resource utilization and reduces the amount paid for unused resources \([11]\). Finally, to support session swapping, PraaS also charges users for the storage required to save swapped sessions.

Invocation accounting is performed by subprocesses, which collect usage metrics across their lifetime for computation, memory usage and I/O. We use the Docker container metrics for this purpose on the subprocesses and collate them across all subprocesses to allow for both remote observability and granular billing.

5.5 Client Library

The client library implements the user-end of HTTP requests to manage processes and sessions and binary data is transmitted as payload over TCP connections. Clients can serialize the data using their choice of procedure. A further feature of the client library is the possibility for the user to implement handlers to be executed if operations time-out or objects or sessions are not available.

5.6 Implementing PraaS

We implement PraaS as a group of extensions on top of existing CaaS and FaaS platforms to facilitate widespread adoption in existing cloud systems. Cloud providers automatically manage a fleet of process containers running the dedicated PraaS worker (Sec. \([5,1]\)). In particular, we demonstrate PraaS with a prototype implementation on Knative \([55]\), a serverless platform deployed with Kubernetes \([56]\) (Sec. \([5,6]\)). However, our design with modified containers and scheduling policies applies to other platforms as well, and the implementation can be ported seamlessly to other systems.

The changes to existing platforms encompass multiple aspects. The scheduling must be adjusted to use an in-memory cache that checks for process and session IDs to make decisions. The subprocess scaling decision must take data plane metrics provided by the sessions, and not eliminate subprocesses with active sessions. k8s supports scaling down by removing selected containers through the mechanism of deletion cost. Finally, our control plane can be deployed to provide to provide the session-process API, and uses the kubernetes API to allocate containers when needed.

6 Evaluation

In this section, we evaluate PraaS. In our experiments, we always repeat each measurement at least 100 times, and the plot median along with the 99% non-parametric CI. In each part of the evaluation we focus providing answers to relevant questions about PraaS.

6.1 Data Plane

Does the data plane improve invocation latency?

Our approach allows fast direct invocations by using an optimized serverless data plane. To evaluate this claim, we compare the round-trip time for processing an invocation with various payload sizes in PraaS to processing the same invocation on AWS Lambda. The benchmark is executed from a t3.medium VM and invokes warm AWS Lambda and PraaS functions. The results in Fig. 7 show a consistent, significant benefit for using PraaS, as AWS Lambda is approximately 32 times slower. AWS Lambda overheads increase with payload size quite significantly, which suggests that mul-
time copies of the payload on the critical path, which makes large-payload invocations increasingly slow.

Furthermore, we show that *PraaS* has effectively no overhead by providing the TCP baseline gathered using netperf for communicating the payload itself between the user and the executor server. **Bypassing the control plane** therefore provides a major improvement in round trip time for function invocations.

### 6.2 Fast function communication

**Does the use of session memory for communication between functions improve latency?**

An important concept in serverless workflows is chaining functions to pass the output of one as input to the other one. We claim that using session memory is a fast way to allow for functions to communicate. We evaluate this by launching a function with varied input size, passing the result of to another function, which in turn returns data to the user. For this purpose, DynamoDB is expensive and too limited (400 kB max element size) so cannot be considered an option. The options left are to either resort to accessing S3 storage or using Lambda’s function I/O. Figure 8 shows that using *PraaS* session memory for function communications is significantly faster by two orders of magnitude than using S3 storage for any data size, and about ten times faster than Lambda function I/O. The results are not surprising, and underline the need for fast, local communication rather than reliance on cloud storage solutions for function communication. **Fast communication between functions** using session memory as a medium is much faster than existing alternatives.

### 6.3 Session Management

**Does swapping sessions incur high overheads?**

To be practically useful, our system for swapping session in and out of storage must be efficient. To evaluate it, we measure the time needed to allocate a clean session, restoring a swapped session or to swap out a running session.

We swap out sessions either to S3 storage or to a gp2 instance of Elastic Block Storage attached to EC2 (a general purpose SSD). In the latter version, the cloud provider can then swap the session to the disk attached to the virtual machine to remove the overhead of swapping from the critical path, and store the swapped session to S3 storage asynchronously. S3 has very high durability, the EBS has annual failure rate of 0.1-0.2% [57].

This has the further benefit that sessions can be restored quickly when restoring from the disk. Restoring a session has a lower latency than allocating a container [58, 59] and can be done concurrently, effectively hiding the time required to load the state from memory.

As can be seen in Fig. 9a, restoring a session from disk takes the same time as allocating a fresh session, and even restoring from S2 is done in less then 150ms, even for the largest session sizes. Swapping out a session also only takes between 25 and 200 ms, depending on session size. Therefore, session swapping **does not introduce significant overheads**.

### 6.4 Global Memory

**Does communication over global process memory improve performance over using S3 storage?**

We evaluate the communication between sessions using global process memory by comparing its performance to Lambda functions communication over S3, Redis or even establishing direct communication channels. All measurements were performed on a Lambda instance with 2 GB of memory. For *PraaS*, we measure a function performing get and put operations of the same. We distinguish between the warm local scenario - data is on the local memory controller and the warm remote - data is on another subprocess on a different machine. The median ping to the other machine was 350 microseconds.

The result for both bandwidth (Fig. 10a) and latency (Fig. 10b) show clear benefits for using *PraaS*. The only solution that shows similar results is the direct p2p connection using Lambda functions. We expect it to be faster than S3 and Redis as it does not involve additional services, just one copy of the data. However, this solution

![Figure 8: PraaS vs FaaS: comm. between functions.](image)

![Figure 9: Overhead of session allocation and swapping.](image)
Figure 10: Latency and bandwidth for communication between Lambda functions and the global memory put-get operations in PraaS.

<table>
<thead>
<tr>
<th></th>
<th>$V_m$</th>
<th>$C_m$</th>
<th>$F_s$</th>
<th>$G_V$</th>
<th>$G_C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>3.53 × $10^{-3}$</td>
<td>4.45 × $10^{-3}$</td>
<td>1.5 × $10^{-2}$</td>
<td>76.43%</td>
<td>70.33%</td>
</tr>
<tr>
<td>Azure</td>
<td>3.87 × $10^{-3}$ ± 0.005</td>
<td>4.45 × $10^{-3}$</td>
<td>1.14 × $10^{-2}$</td>
<td>66.05% ± 0.05</td>
<td>60.96%</td>
</tr>
</tbody>
</table>

Table 3: Decreased costs of PraaS sessions in comparison to FaaS provisioned instances.

PraaS does not fit the serverless paradigm – functions must be online at the time of transfer and they must be aware of each other’s existence, making practical use difficult. Furthermore we confirm the previous result that Lambda shows significant I/O variability [11, 35]. PraaS provides fast, efficient communication even across physically separated machines.

6.5 Costs

What are the system costs?

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

To estimate the cost of keeping PraaS sessions alive in memory, we use the memory of other cloud services as a proxy for the cost to the cloud operator. First, we compare the cost $V_m$ of changing from a compute-optimized to a memory-optimized virtual machine instances, and divide it by the size of gained memory, which estimates the additional cost of gaining one gigabyte of DRAM. We compare c6g and x2gd instances on AWS and F5 and Eadesv5 series on Azure, and we found that $V_m$ almost the same for each instance size. Then, we select the cost of allocating additional memory when deploying PraaS on managed container systems $C_m$, and we consider AWS Fargate and Azure Container Instances. By comparing the memory costs $V_m$, $C_m$ of PraaS deployment to the cost of provisioned storage $F_s$ on FaaS, we estimate the cost decrease $G_V$ and $G_C$ of moving session state from provisioned FaaS to VMs and containers, respectively. The results presented in Table [5] prove that PraaS sessions can be offered at a lower cost, by up to 76%, and the estimation covers the cloud provider costs and profit included in the price of a VM instance. Furthermore, the memory-optimized instances come with additional SSD storage, which could be used to implement a low-latency tier for swapped sessions at no additional cost.

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

Can FaaS functions run on PraaS?

We select the thumbnail generator as an example of general-purpose image processing and the image-recognition benchmark performing ResNet-50 prediction as an example of integrating deep-learning inference into applications, both from the SeBS benchmark [11].

We implement the thumbnail generation with OpenCV 4.5, and evaluate functions with two inputs, a 93 kB image and a 3.6 MB one. For the AWS Lambda function, we need to submit the binary image data as a base64-encoded string in the POST request, which adds significant overheads due to encoding and conversions. On the other hand, PraaS benefits from a payload format that is not constrained by cloud API requirements.

The image recognition is implemented with the help of PyTorch C++ API, using OpenCV 4.5, libtorch 1.9, and torchvision 0.1. We evaluate functions with two inputs, a 53 kB image and a 230 kB one. Results (Fig. 11a and 11b) prove that PraaS outperforms all AWS instances we compared against.

6.7 Discussion

Given the features PraaS provides, it could be used as a platform to implementing distributed logistic regression with gradient averaging stochastic gradient descent in both an asynchronous and synchronous fashion. The performance of the system could improve compared to the current implementation [60], due to using direct communication. The PraaS model should make it easier to express asynchronous computations. Finally, PraaS could dynamically change the number of functions being invoked at a given time, while LambdaML [60] relies on a fixed number of functions throughout the execution.

7 Related Work

Stateful Functions open the spectrum of applications that can benefit from FaaS by allowing functions to keep state, even if disaggregated. Researchers have built stateful functions on top of key-value stores specialized to Serverless [18, 61], and elastic ephemeral caches [51, 62, 63] which combine different placement strategies to manage cost and performance. For many applications, however, statefulness is not enough as functions can fail at any time. Hence, systems such as Beldi [19], and Boki [20] have been proposed to help developers build consistent and fault tolerant systems atop ephemeral functions. Instead of offering yet another storage option to persist function state or propose a new technique to handle faults, PraaS strikes a new balance between IaaS and FaaS by proposing the concept of sessions that retain state. Within a process, sessions can exchange messages. PraaS eliminates the need for state synchronization in stateful functions by partitioning state into ownership domains managed by sessions. Sessions can be swapped at any time but their state is not lost.

Function Orchestration and Data Locality are also being extensively studied. Systems such as Speedo [64] and Nightcore [65] optimize function orchestration by either accelerating the control plane [63] or by completely skipping it [65] 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 [66], or by avoiding moving data by allowing multiple functions to share the execution environment over time [67].

PraaS, on the other hand, proposes that users should be able to directly connect to sessions and thus completely avoid the control plane after the first initialization step. In addition, by partitioning state among sessions, process functions can exchange data directly among them instead of relying on external storage proposed by previous systems [21]. Although taking advantage of direct communication between functions whenever possible, PraaS offers a global memory space abstraction to users instead of directly exposing network primitives to developers [68].

Serverless Sandboxes utilize specialized virtualization engines [68, 69] that drastically reduce the startup time, and memory footprint when compared to traditional virtual machine managers. However, to continue improving scalability and elasticity of serverless applications, soft-isolation systems [70, 72] 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 to execute concurrently inside a single session. By doing so, resource redundancy is reduced and new opportunities for local communication arise. However, PraaS’s design does not preclude other orthogonal optimization techniques such as image pre-initialization [69, 73, 75] to optimize session startup time and memory footprint, orunikernels [76, 78] to optimize process startup and memory footprint.

8 Conclusions

This work proposes PraaS, a new cloud computing abstraction that brings persistent state to ephemeral workers. By taking advantage of processes and sessions, applications can now benefit from low-latency state, fast invocations that bypass the control plane, and fast communication between sessions. We showed that PraaS is an efficient abstraction to achieve stateful functions with direct connections and fast communication options.
References


