AccelFaaS: Accelerating FaaS via Pre-Warmed Memory and Control Channel Offloading

Seong-Joong Kim, Seungwon Shin, *Member, IEEE*, and Myoungsung You

***Abstract*—Serverless computing offers a cost-efficient and scalable architecture for cloud applications by processing client requests through a set of functions. However, the existing serverless data plane suffers from significant cold-start latency and degraded inter-function data transfer performance due to its inherent complexity. Previous studies using function pre-loading and shared-memory channels partially mitigate these issues but remain limited by their incompatibility with scale-from-zero and substantial memory inefficiencies. We propose *AccelFaaS*, a fast and memory-efficient serverless data plane. *AccelFaaS* identifies that control channel setup is the dominant bottleneck in cold-starts and decouples this procedure from the critical path, significantly reducing startup latency without relying on pre-loading. *AccelFaaS* improves inter-function communication by introducing a cached shared memory channel that reduces page faults and fragmentation, while dynamically resizing its capacity according to workload demands to minimize memory usage. Evaluations show that *AccelFaaS* reduces cold-start latency by 11–27×, increases chaining throughput by 4×, and lowers memory consumption by 2–16× compared to existing systems.**

***Index Terms*—Serverless computing, cold-start latency, inter-function communication.**

I. Introduction

SERVERLESS computing [1], particularly in the form of Function as a Service (FaaS) [2], has emerged as a promising paradigm for developing and deploying large-scale cloud services, such as microservices and distributed machine learning, due to its flexible billing model and high scalability. Consequently, major cloud providers (e.g., AWS and Azure) have already deployed FaaS platforms, underscoring its widespread adoption and practical relevance. In a typical FaaS deployment, client requests are processed by a workflow of ephemeral functions. Upon request arrival, a function is initialized together with a reverse proxy (i.e., a sidecar) responsible for inter-function communication. The function processes the request, forwards intermediate results to the next function in the workflow, and is terminated once execution completes. This *scale-from-zero* execution model, in which resources are provisioned exclusively at demand time, is a defining characteristic of FaaS, enabling both elastic scalability and a flexible billing model by charging clients solely for active execution. Despite its advantages, FaaS platforms suffer from two fundamental performance challenges rooted in their architectural design.

Cold-start Latency: The first limitation is long cold-start latency. While scale-from-zero supports flexible pricing and resource efficiency, it requires reloading functions with each new request, resulting in cold-starts. This cold-start involves various time-consuming operations, such as function/sidecar spawning and request dispatching. Among them, our analysis reveals that the control channel establishment is the main bottleneck. Specifically, a sidecar should establish various control channels with the control plane, necessitating multiple network interactions. Consequently, request dispatching is delayed until all these channels are fully established, resulting in a long cold-start latency (Section II-B). Previous studies have explored predictive techniques, such as function snapshotting [3], [4], [5], [6], or function pre-loading [7], [8], [9], [10], [11]. However, function snapshots focus on reducing spawning time rather than control channel setup, and pre-loading functions undermines the scale-from-zero model by consuming resources during idle periods.

Data Transfer Overhead: The second limitation is the overhead encountered during serverless workflow. Sidecars introduce additional network hops, causing significant overhead when passing data between functions. Previous studies [7], [12], [13], [14], [15] have attempted to mitigate this by using shared memory for direct data transfers. However, these methods still suffer from memory inefficiencies, particularly in scenarios with large data exchanges. For instance, Pheromone [15] and FAASM [12] experience up to 90% degradation in data transfer throughput when the data size exceeds 5 MB (Section II-B), due to fragmented memory and frequent page faults in their shared memory pools. SPRIGHT [7] attempts to address this by using physically contiguous memory with persistent huge pages. However, it incurs high memory usage, consuming tens of gigabytes even during idle periods (Section II-B), as huge pages are unswappable. Additionally, SPRIGHT requires functions to remain active with pre-allocated huge pages to avoid initialization delays, which contradicts the scale-from-zero model.

Our approach: To overcome these two limitations, this paper introduces *AccelFaaS*, a fast and memory-efficient serverless data plane for serverless workflow. It decouples control channel

Received 3 August 2025; revised 5 March 2026; accepted 11 March 2026. Date of publication 16 March 2026; date of current version 9 June 2026. This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by Korean Government (MSIT) under Grant RS-2024-00437004. This work was supported by the 2025 Research Fund of the University of Seoul. Recommended for acceptance by Q. Chen.*(Corresponding author: Myoungsung You.)*

Seong-Joong Kim is with National Security Research Institute, Daejeon 34102, South Korea (e-mail: sungjungk@gmail.com).

Seungwon Shin is with the School of Electrical Engineering, KAIST, Daejeon 34141, South Korea (e-mail: claude@kaist.ac.kr).

Myoungsung You is with the School of Electrical and Computer Engineering, University of Seoul, Seoul 02504, South Korea (e-mail: famous@uos.ac.kr).

Digital Object Identifier 10.1109/TCC.2026.3674472

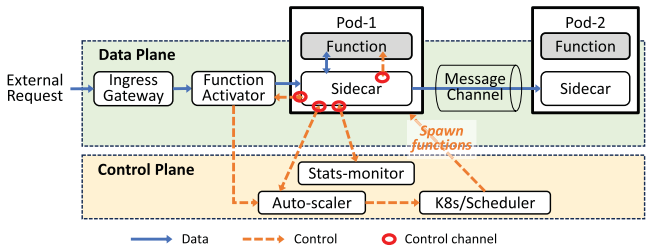


Fig. 1. Common FaaS platform architecture.

TABLE I
COMPARISON OF SOLUTIONS FOR COLD-START LATENCY AND DATA TRANSFER OVERHEAD. *FAULT/FRAG.* DENOTES PAGE FAULTS AND MEMORY FRAGMENTATION OVERHEAD.

Solution	Function cold-start		Data transfer	
	Scale-from-zero	Latency	Fault/Frag.	Mem. usage
FAASM [12]	○	Low	High	Low
SPRIGHT [7]	✖	Low	Low	High
Pheromone [15]	✖	High	High	Low
vHive [4]	○	Low	High	Low
AccelFaaS (ours)	○	Low	Low	Low

setup from function cold-start by implementing a per-node proxy (*AccelFaaS* agent) that proactively establishes necessary channels with the serverless control plane. When a function is initialized, essential metrics of the function are promptly relayed to the *AccelFaaS* agent through a per-function lightweight in-kernel channel, eliminating the need for establishing network-based control channels. This optimization significantly reduces cold-start latency while preserving the scale-from-zero. Furthermore, *AccelFaaS* employs a novel memory-efficient shared memory channel composed of transparent huge pages (THP) for serverless workflow. It uses a pre-warmed memory pool composed of physically contiguous, hot pages, which are served to functions in the chain based on demand. This channel reduces page faults and physical memory fragmentation during data transfer, enhancing performance and optimizing memory usage.

Contributions: Our contributions are outlined as follows:

- We conduct an in-depth analysis of the current serverless data plane, identifying critical performance bottlenecks in (i) cold-start latency and (ii) data-sharing overhead.
- We design *AccelFaaS*, which mitigates cold-start latency by decoupling control channel establishment and accelerates serverless workflow by utilizing a cached memory pool.
- We implement the full prototype of *AccelFaaS* and conduct extensive evaluations in real-world serverless environments. The results show that *AccelFaaS* reduces cold-start latency by 11–27× and achieves a 4× improvement in data transfer throughput, while reducing memory utilization by 2–4× compared to state-of-the-art solutions.

The remainder of this paper is organized as follows. Section II introduces the background of FaaS and analyzes the limitations of prior work in function cold-start and data transfer. Section III presents the detailed design of *AccelFaaS*. Section IV evaluates *AccelFaaS* using real-world FaaS workloads and compares its performance against existing solutions. Section V reviews related studies, while Section VI discusses the current limitations of *AccelFaaS*. Finally, Section VII concludes the paper.

II. BACKGROUND AND MOTIVATION

A. FaaS Architecture

Fig. 1 shows the common FaaS architecture [16], [17], [18], [19], [20] comprising data and control planes.

Data Plane: The green box of Fig. 1 shows FaaS data plane components. These components are responsible for handling data transmission between functions. The *ingress gateway*, which serves as the entry point of the FaaS platform, receives external requests and routes them to the *activator*. The *activator* enqueues these requests and initiates the appropriate set of functions to process the queued requests. As functions are initialized, each instance is paired with a *reverse proxy (sidecar)*, which manages inter-function data exchanges and interacts with the control plane. During initialization, the sidecar establishes multiple *control channels* with the control plane to report the function's readiness and to enable telemetry collection. Upon receiving the readiness confirmation, the activator dispatches a queued request to the function via its sidecar. The function processes the request and generates an intermediate result, which is then forwarded by the sidecar to the next function in the workflow using a dedicated data channel. This process continues iteratively until the final function returns a response to the client.

Beyond readiness signaling and data relaying, sidecars also provide essential data plane functionalities. They periodically collect metrics, such as function liveness and network telemetry, and report them to the control plane (*metric serving*). Also, sidecars temporarily buffer incoming requests when the function is busy processing previous requests, thereby ensuring ordered execution (*request queuing*).

Control Plane: The control plane (depicted in the yellow box of Fig. 1) is responsible for managing function orchestration and request routing. The *auto-scaler* dynamically adjusts the scaling of function instances based on telemetry metrics received from sidecars. For instance, it supports scale-from-zero by initializing functions only when client requests arrive, releasing resources (e.g., CPU and memory) during idle periods. In response to fluctuations in request traffic, the *auto-scaler* can spawn additional function instances to maintain optimal performance levels. The scheduler within the control plane allocates functions and resources across the cluster, selecting execution nodes based on available resources and orchestrating function placement dynamically. Together, these components ensure automatic scaling, efficient message routing, and effective resource management.

B. Challenges in FaaS Data Plane

The FaaS data plane is responsible for orchestrating data transfers among functions, yet it becomes a major performance bottleneck due to its inherent complexity. Previous studies have attempted to mitigate these issues, but they fail to address the following two critical challenges (see Table I).

TABLE II
COLD-START LATENCY BREAKDOWN OF AN ECHO FUNCTION

Stages	*queue*	*spawn*	*prepare*	*dispatch*	*execute*	*return*
Time(s)	0.1	0.28	0.88	0.1	0.08	0.09

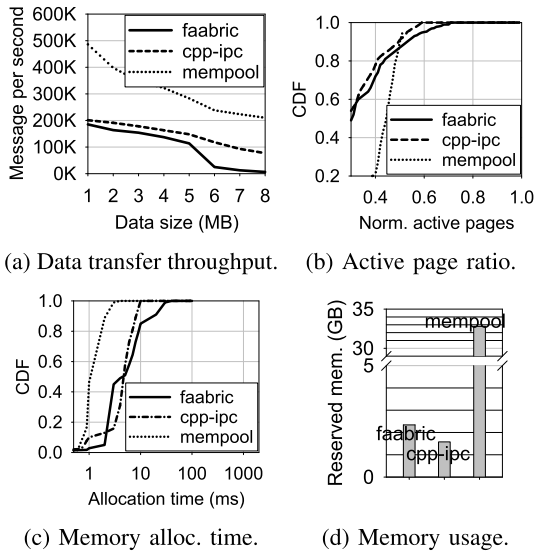
Long Cold-start Latency: Cold-start latency refers to the delay incurred when handling a request for a function that is not yet initialized. Under the scale-from-zero model, each incoming request to an inactive function triggers a re-initialization, which consists of multiple time-consuming steps. To quantify the overhead incurred during cold-start, we deploy a simple echo function and measure its end-to-end startup latency. Details of the experimental environment are provided in Section IV. The results show that the cold-start latency is approximately 1.56 seconds, which far exceeds typical serverless requirements (<1 s). We further decompose the latency into six stages, as shown in Table II:

- *Queuing:* the time from when the request reaches the ingress gateway to when it is queued by the activator.
- *Spawning:* the time required to launch the pod, including sidecar and function containers.
- *Preparation:* the time taken to establish control channels between the sidecar and the control plane.
- *Dispatching:* the time taken to dispatch the incoming request to the target function.
- *Executing:* the time spent executing the function logic.
- *Returning:* the time required for the response from the function to return through the ingress gateway.

Among these stages, the *preparation* stage dominates the overall cold-start latency. The control channel between a sidecar and the control plane is typically implemented over HTTP [21], which requires multiple TCP packet exchanges. Thus, control channel establishment incurs repeated network stack traversals, delaying readiness signaling and substantially inflating cold-start latency.

Despite its significance, most prior studies [7], [8], [22] focus on optimizing the *spawning* stage. For instance, vHive [4] accelerates function initialization using snapshot-based restoration. However, even after snapshot restoration, a restored function must establish a new control channel, since network states such as IP addresses and active sockets are mutable states and cannot be captured in snapshots. Thus, the preparation stage overhead remains unaddressed.

Other approaches rely on warm-start techniques that keep functions continuously active. Pheromone [15], for example, maintains one function per workload in a warm state to immediately serve incoming requests, while SPRIGHT [7] keeps all functions always warm using an event-driven execution model. Although effective in reducing cold-start latency, warm-start techniques compromise the scale-from-zero model, a core advantage of FaaS, as pre-warmed functions continue to consume resources (e.g., memory) even during idle time. Moreover, the latency benefit is inherently limited: a FaaS workloads typically consist of multiple functions, and only a small subset of functions can be pre-warmed due to memory overhead. Once the number of inactive functions exceeds the number of warm ones, the performance improvement becomes marginal, whereas aggressively increasing the number of warm functions directly amplifies memory overhead. Due to these limitations, scale-from-zero remains the default execution model in most commercial FaaS platforms (e.g., AWS Lambda).



(a) Data transfer throughput. (b) Active page ratio.

(c) Memory alloc. time. (d) Memory usage.

Fig. 2. Data transfer performance of existing solutions.

Data Transfer Overhead: Another major performance bottleneck in the FaaS data plane arises from data transfer between functions. Such communication is typically mediated by sidecars, which introduce extra network hops and degrade data transfer performance [7] due to redundant network stack traversals. To mitigate these overheads, most previous studies [23], [24], [25], [26] co-locate as many functions of the same workload as possible on a single machine to maximize data locality, and provide a shared memory channel (e.g., shared memory IPC) for direct data exchange between co-located functions without involving sidecar. Although shared memory channels eliminate sidecar overheads during data transfer, we observe that previous studies still suffer from severe memory inefficiencies, including memory fragmentation, frequent page faults, and excessive memory usage. To clearly show these inefficiencies, we evaluate throughput, active page ratio, memory allocation latency, and total memory usage during inter-function data transfer. We examine three shared memory channels used by state-of-the-art systems: *faabric* [27] (used in FAASM), *cpp-ipc* [28] (used in Pheromone), and *mempool* [29] (used in SPRIGHT). We exclude vHive [4] from this analysis, as it relies on gRPC communication rather than shared memory.

As shown in Fig. 2(a), both *faabric* and *cpp-ipc* exhibit a substantial throughput drop, exceeding 50%, as the data size increases. This performance degradation primarily stems from memory inefficiencies during inter-function data transfer. Each data transfer involves read and write operations on data chunks allocated in the shared memory channel. However, as shown in Fig. 2(b), about 60% of the pages in the shared memory channel remain cold. As a result, accessing these chunks frequently triggers page faults, which significantly increases chunk access latency. In addition, *faabric* and *cpp-ipc* suffer from

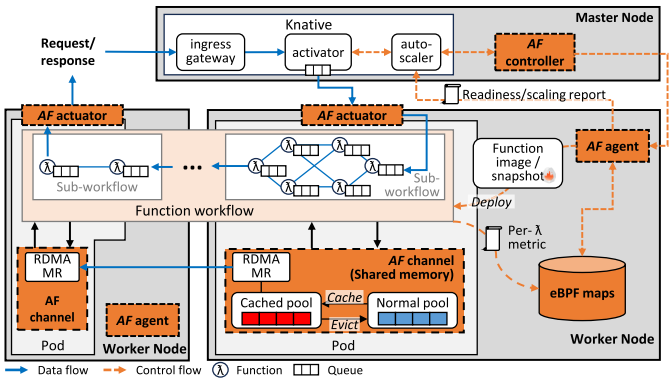


Fig. 3. The overall *AccelFaaS* system architecture.

memory fragmentation because they operate over physically non-contiguous memory regions. Thus, both systems experience prolonged chunk allocation times (Fig. 2(c)), as the memory allocator increasingly struggles to locate suitably sized contiguous regions. This often leads to costly compaction and de-fragmentation operations, further exacerbating performance degradation [23], [24], [25], [26].

Notably, *faabric* exhibits a throughput drop exceeding 90% at data sizes around 6 MB. Our analysis reveals that this sharp decline stems from limitations of *dlmalloc*, which *faabric* uses to manage shared memory. *dlmalloc* employs a single global lock for shared memory allocation and switches to a slow path, involving treebin logic and multiple system calls, when handling large chunks ($>$ 6 MB). As a result, functions are serialized behind the lock during the long chunk allocation, thereby significantly reducing data transfer throughput.

SPRIGHT achieves higher throughput than other solutions by leveraging physically contiguous memory backed by persistent huge pages. However, this performance advantage comes at the cost of excessive memory consumption. As shown in Fig. 2(d), *mempool* consumes about 32 GB of memory even when executing a workload with a single function. Furthermore, persistent huge pages must be statically provisioned at boot time and cannot be resized dynamically on runtime. This rigidity prevents SPRIGHT from adapting its memory footprint to runtime workload fluctuations: insufficient provisioning prevents new workloads from acquiring memory channels, while over-provisioning leaves large portions of memory idle, especially since SPRIGHT keeps all functions continuously active.

III. *AccelFaaS* Design

A. Design Goals

To address the two challenges identified in Section II, *AccelFaaS* is designed with the following goals.

G1. Decoupling Control Channel Setup: A significant portion of cold-start latency arises from the synchronous establishment of control channels during function initialization. To eliminate this overhead, *AccelFaaS* aims to decouple control path setup from the critical path of function startup, allowing newly spawned functions to reuse pre-established node-level channels without setup delays.

G2. Minimizing Memory Inefficiencies: The current shared-memory data planes for FaaS often suffer from memory fragmentation, frequent page faults, and high idle resource consumption. *AccelFaaS* aims to mitigate these inefficiencies by utilizing physically contiguous memory regions with pre-warmed pages while dynamically resizing the memory pool based on real-time demand.

G3. Supporting Scale-from-zero: While prior studies achieve high performance through function pre-loading, such approaches incur idle resource costs and complicate failure recovery. *AccelFaaS* is designed to maintain the scale-from-zero model while reducing cold-start latencies and improving data transfer performance without function pre-loading.

B. AccelFaaS *Overview*

Fig. 3 shows the overall architecture of *AccelFaaS*, which consists of a centralized *AccelFaaS controller* and three data-plane components: *AccelFaaS agents*, *actuators*, and *channels*. Together, these components are designed to minimize function cold-start latency and inter-function communication overhead while preserving the scale-from-zero model.

Upon receiving an invocation request through the ingress gateway, the activator triggers a *AccelFaaS* agent on the target worker node. The agent is responsible for workflow-level orchestration on the node and initializes all functions belonging to a workflow within a single pod. This co-location strategy improves data locality and reduces inter-function communication overhead during workflow execution [13], [14], [30]. During pod initialization, an eBPF-based *AccelFaaS actuator* is attached to the pod's network interface. Operating entirely in the kernel, the actuator replaces traditional heavyweight sidecar proxies by providing essential data-plane functionalities with minimal overhead. Unlike conventional designs that performs complex control-channel setups for each function, the actuator directly records function readiness information into an in-kernel eBPF map. The agent then immediately reads this readiness metric and notifies the controller through a persistent control channel without establishing a new one. By decoupling control-channel establishment from per-function initialization, *AccelFaaS* eliminates handshake-induced delays from the critical cold-start path, thereby achieving **G1**.

Inter-function data transfer within a workflow is handled by a *AccelFaaS channel* associated with each workflow pod. The channel is implemented as a shared memory pool backed by Transparent Huge Pages (THP) allocated in physically contiguous memory regions [31]. To ensure immediate availability during execution, the pages are pre-warmed during the cold-start phase, effectively avoiding runtime page faults. Furthermore, the channel dynamically adapts its size based on runtime workload characteristics: actively used pages are retained, while idle pages are reclaimed to reduce memory footprint, satisfying **G2**. For workflows spanning multiple nodes, *AccelFaaS* transparently extends the channel using RDMA, enabling low-latency, low-overhead data transfers across nodes without exposing additional complexity to the application. In contrast to prior systems that trade off the scale-from-zero model for reduced latency [7],

AccelFaaS preserves scale-from-zero execution while simultaneously mitigating cold-start and data-transfer overheads (**G3**). Specifically, *AccelFaaS* reduces cold-start latency by decoupling control-plane setup from function initialization, rather than relying on speculative pre-loading of functions. Moreover, by supporting runtime adaptation of shared memory channels, *AccelFaaS* enables flexible and efficient auto-scaling in response to dynamic workload variations.

C. AccelFaaS *Controller*

As shown at the top of Fig. 3, the *AccelFaaS* controller orchestrates other *AccelFaaS* components. To this end, it communicates with the FaaS control plane via APIs provided by the underlying platform, monitoring changes in nodes and functions, and accordingly updating relevant *AccelFaaS* components. For example, when a workload is invoked, the controller delivers the workload metadata to the *AccelFaaS* agent on the target node, enabling the agent to seamlessly attach an *AccelFaaS* actuator and instantiate a channel within the corresponding pod.

Another key role of the controller is to maximize data locality during data transfer between functions. When functions within the same workflow are scheduled across multiple nodes, data transfer requires inter-node communication, significantly degrading transfer performance. To mitigate this issue, the controller employs a topology-aware scheduling strategy in cooperation with the FaaS platform's native scheduler. By default, it attempts to co-locate all functions in a workflow on a single node [13], [14], [30], [32], [33]. If this is infeasible due to resource constraints, the controller partitions the workflow into multiple sub-flows, each of which is scheduled on a single node, as shown in Fig. 3.

To identify appropriate partition points, the controller analyzes the workflow topology (e.g., DAG). A typical workflow includes linear points, fan-in points, and fan-out points. At linear points, a function forwards data to a single successor, whereas fan-in and fan-out points respectively aggregate inputs from multiple predecessors or distribute outputs to multiple successors. The controller avoids selecting fan-in or fan-out points as partition boundaries whenever possible, instead preferring linear points. This strategy ensures that fan-in and fan-out functions remain co-located with their neighboring functions, thereby reducing inter-node communication and improving data transfer performance. For effective scheduling, the controller relies on metadata describing each function's resource requirements and the workflow topology, which is typically provided by tenants during function deployment in existing FaaS platforms.

D. AccelFaaS *Agent*

AccelFaaS (AF) agents operate on each worker node and are responsible for three core tasks: (i) attaching *AccelFaaS* components to newly deployed function pods, (ii) relaying probe metrics to the control plane, and (iii) facilitating RDMA-based inter-node data transfer.

When the *AccelFaaS* controller schedules a new function pod, it transmits the pod metadata to the designated AF agent on the target node. The agent then locates the newly created pod and installs a dedicated AF actuator and AF channel instance. The actuator is attached to the pod's network interface, while the channel is deployed as a shared library alongside the function binary. The attached AF actuator periodically collects runtime probe metrics from each function in the pod and stores them in two in-kernel eBPF maps. The *contrack map* maintains network-level statistics such as active TCP sessions, which are useful for tracking function liveness and performance. The *telemetry map* contains application-level statistics, including packet sizes and response timestamps, supporting control plane decisions, such as circuit breaking and auto-scaling.

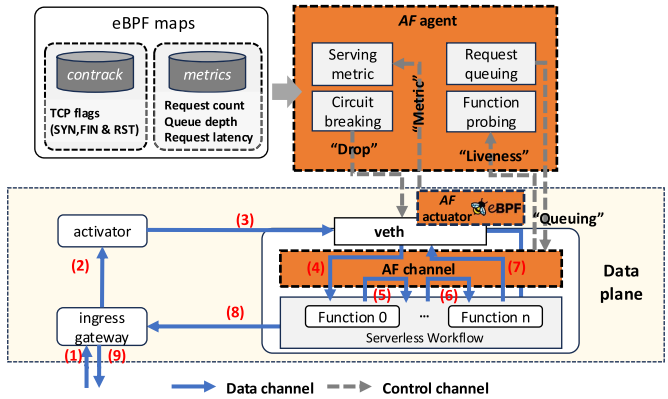


Fig. 4. The operation of *AccelFaaS* actuators.

For readiness checks during cold-start and liveness checks during runtime, the AF agent accesses both the conntrack and telemetry maps and relays the collected metrics to the control plane. Each AF agent maintains a long-lived TCP (or TLS) connection to the control plane and reuses this connection for metric forwarding. As a result, individual functions are not required to establish a new control channel during startup. This design eliminates the overhead of control-channel establishment during function cold-starts, thereby significantly reducing startup latency.

For a function workflow that spans multiple nodes, the AF agent extends the AF channel between functions using RDMA [34]. Persistent RDMA connections are maintained across nodes and reused across function invocations. When inter-node data transfer is required, the AF agent maps RDMA memory regions (MRs) into the AF channel. This allows the final function of a sub-flow to directly write output data to MR-backed pages, enabling efficient zero-copy transfers to the subsequent function on another node. This design preserves the high throughput of shared memory-based communication while minimizing cross-node latency.

E. AccelFaaS *Actuator*

An AF actuator is a per-function pod proxy running in the kernel space. As shown in Fig. 4, by cooperating with the node-level AF agent, it replaces heavy-weight per-function sidecars with lightweight per-pod in-kernel proxies, improving data plane performance while maintaining the scale-from-zero model. Specifically, the AF actuator is responsible for the following two key operations.

Requests Queuing: In FaaS environments, incoming requests are initially buffered in the activator's global queue until the target function is ready to process them. This situation arises when the function is either busy serving prior requests or has not yet been instantiated. Since the activator queue is a shared resource across all functions in the cluster, timely dispatch of requests is critical for minimizing both end-to-end response time and cold-start latency. Any delay in dispatching can lead to head-of-line (HoL) blocking, especially during request bursts. Most previous studies [7] mitigate this issue by bypassing the activator altogether. For example, SPRIGHT keeps all functions pre-warmed and routes requests directly from the ingress gateway to the activated functions. While this design effectively eliminates HoL blocking, it fundamentally compromises the scale-from-zero model, consuming memory resources even during idle time.

AccelFaaS takes a different design approach that preserves the scale-from-zero model while mitigating the queuing bottleneck. Specifically, *AccelFaaS* integrates the AF agent and the actuator to decouple request dispatching from function readiness. Upon the arrival of a new request, the AF agent immediately extracts it from the activator's global queue and forwards it to the AF actuator co-located with the destination function workflow, regardless of whether the head function in the workflow is ready or not. The AF actuator maintains a dedicated, per-function waiting queue within the AF channel that acts as a localized buffer. If the head function is not yet available, the request is held in this local waiting queue until the function becomes ready. Otherwise, the head function can process the request immediately upon activation. To accommodate varying traffic conditions, the depth of each local waiting queue is dynamically adjusted at runtime by the AF agent based on contrack and telemetry maps. This adaptive queuing mechanism ensures that the local waiting queue remains unblocked even during traffic surges, avoiding request drops without sacrificing on-demand execution.

By offloading queuing responsibility from the central activator to per-workflow AF actuators, *AccelFaaS* avoids HoL blocking at the global queue, thereby improving request dispatching and cold-start latency on request burst scenarios.

Metric Serving: Similar to previous studies [7] that utilize in-kernel proxies, the AF actuator collects various runtime metrics from each function within the workflow pod. These metrics include readiness, network-level statistics, and performance indicators, all of which are gathered directly within the kernel and are stored in the two eBPF maps: the contrack and telemetry maps. For example, the AF actuator monitors probe packets issued by the underlying container platform (e.g., Kubernetes readiness probes) to assess function readiness. It also tracks the number of pending requests in the waiting queue of each function, providing insights into local load conditions. These metrics serve as the basis for various runtime control decisions, including whether to initiate or suspend request dispatching to the head function and whether to dynamically adjust the depth of each function's waiting queue. The AF agent periodically relays these metrics to the control plane by using the pre-established control channel.

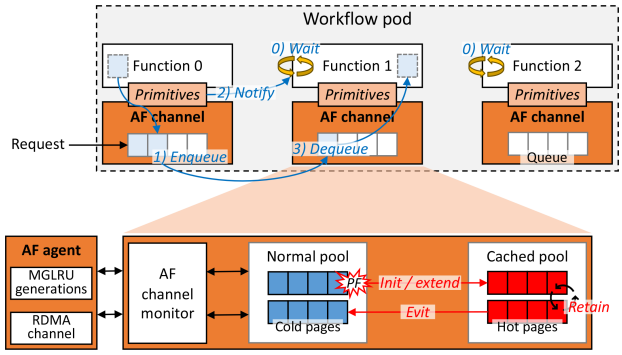


Fig. 5. Inter-function data transfer over an AF channel.

To address scalability concerns raised by metric collection, *AccelFaaS* employs a hybrid strategy that bounds eBPF usage and avoids kernel-map contention. Specifically, readiness and scheduling metrics are sampled once per invocation and updated in an eBPF map only by the head function upon receiving ingress packets, ensuring that the number of BPF map updates is constant and independent of fan-out degree. For subsequent co-located functions in the workflow, execution metrics (e.g., execution duration and data-transfer size) are recorded at the completion of each function via append-only logging in a pre-allocated shared-memory pool. This event-driven logging replaces periodic polling, effectively bounding the sampling frequency to the request rate. These logs are asynchronously aggregated by the *AccelFaaS* agent and reported to the auto-scaler, keeping the critical execution path free of eBPF map locking even under high fan-out.

F. AccelFaaS *Channel*

Fig. 5 illustrates the overall design of inter-function data transfer over the AF channel. The AF channel is designed to facilitate efficient data transfers between functions within the same workflow by leveraging a cached shared memory pool and RDMA. During function cold-start, the AF channel is loaded together with each function binary in the workflow. Since *AccelFaaS* deploys functions (containers) belonging to the same workflow within a single pod on the same node, these functions communicate through the AF channel by sharing IPC namespaces. When functions are deployed across different nodes, the AF channel between them is transparently extended over RDMA to support remote data transfer. Note that AF channels are isolated at the pod (i.e., workflow) level using namespace isolation, ensuring that unauthorized inter-pod memory accesses are prevented by the container platform. The AF channel is managed by a channel monitor, which controls the lifecycle of the channel memory through four operations: *Initialize*, *Extend*, *Retain*, and *Evict*.

Initialize: During pod creation, the channel monitor allocates a memory pool backed by transparent huge pages (THPs), as shown in the lower part of Fig. 5. This pool is initially a normal state (a *normal pool*), consisting of physically contiguous pages but remains cold because the pages have not yet been accessed. While physical contiguity ensures efficient chunk allocation,

TABLE III
AccelFaaS Channel Primitives. *User-Class APIs* are Exposed to Function Developers, While *Internal-Class APIs* are Hidden From Developers

Class	API	Description
User	void* af_init()	Initializes a shared memory pool.
User	void af_destroy(inst)	Destroys the shared memory pool and reclaims the associated memory resources.
User	void* af_alloc(size)	Allocates a memory region from the pool.
User	void af_free(ptr)	Returns the allocated memory region.
Internal	void af_wait(func_id)	Blocks the current process until a notification is received on a condition variable.
Internal	void af_notify(func_id)	Notifies the condition variable to wake up a blocked process.
User	size_t af_enq(data, func_id)	Enqueues the given memory region into the target function's queue.
User	size_t af_deq(data, func_id)	Dequeues a memory region from the target function's queue.
Internal	size_t af_send_out(data, func_id, chain_id)	Sends data to the *AccelFaaS* agent for inter-node data transfers.

cold pages can still incur runtime overhead due to page faults during data transfer. Thus, the channel monitor proactively converts a subset of cold pages into hot pages by intentionally triggering page faults. Specifically, it accesses a predefined number of pages (default 200 MB), causing them to be faulted in, populated, and cached by the kernel. These warmed pages form the *cached pool*, which is then used to construct dedicated shared memory queues for inter-function data transfer. Pages in the cached pool are organized into fixed-size memory chunks aligned to the huge page size.

Extend: Pre-warming all pages in the normal pool would unnecessarily increase cold-start latency. Therefore, *AccelFaaS* pre-warms only a subset of pages during initialization and expands the cached pool on demand. When the cached pool does not provide sufficient capacity for newly spawned functions or bursty data transfers, the channel monitor incrementally extends it by accessing additional pages from the normal pool. This extension is performed in predefined increments, inducing page faults that promote the selected pages into the cached pool as hot pages. By growing the cached pool only when necessary, the AF channel effectively balances cold-start overhead with runtime performance.

Retain: If hot pages in the cached pool are reclaimed by the kernel, inter-function data transfers incur costly runtime page faults, severely degrading transfer performance. To preserve memory locality and keep the cached pool resident, the channel monitor leverages the Multi-Generation LRU (MGLRU) reclamation mechanism, which classifies pages into multiple generations based on both access recency and frequency. To prevent cached pages from being demoted to older generations, the channel monitor periodically re-accesses pages in the cached pool, refreshing their access metadata and promoting them to younger MGLRU generations. Under MGLRU, memory reclamation targets pages in the oldest generation; thus, pages that are consistently re-touched (or accessed by functions) remain protected from eviction. As a result, the cached pool maintains a stable working set pages across function invocations, avoiding unnecessary page reclamation and reducing page fault overhead during data transfer.

Evict: The channel monitor dynamically adjusts the size of the cached pool at runtime according to the data transfer demands of functions. Pages in the cached pool that become infrequently accessed naturally age out under MGLRU and are returned to the normal pool. In addition, when memory demand decreases due to idle or terminated functions, the channel monitor evicts unused pages from the cached pool to the normal pool, shrinking the cached pool. This adaptive eviction enables memory elasticity, reducing unnecessary memory consumption while preserving sufficient cached pages for active data transfers. Specifically, the normal pool serves as a reserved region for promoting pages when contiguous memory becomes scarce, ensuring the continued availability of high-order pages. Since MGLRU management requires elevated privileges, the channel monitor cooperates with the *AccelFaaS* agent running in the root namespace to safely perform privileged operations.

G. AccelFaaS Channel Primitives.

AccelFaaS exposes developer-friendly APIs, called AF channel primitives, to allow developers to transmit data over our cached shared-memory pool using a familiar message-passing abstraction. Complex operations, including synchronization and inter-node data transfer tasks, are hidden from developers and transparently managed by the AF channel library. Table III summarizes the AF channel primitives, which are categorized into two classes: *User* and *Internal*.

User-class primitives: From a function developer's perspective, adapting an existing function workflow to *AccelFaaS* requires minimal code changes. Conventional I/O mechanisms (e.g., sockets) are replaced with user-class AF channel primitives, while the overall workflow structure and control flow remain unchanged. This design choice is consistent with prior studies [7], [12], [15] that employ shared-memory channels, where replacing socket I/O with specialized APIs is a common requirement to exploit efficient memory-level data transfer.

Specifically, a function initializes the shared-memory pool using `af_init()` and allocates a chunk (buffer) via `af_alloc()`. It then transmits data to the next function by calling `af_enq()` with data (or a pointer) and the target function's ID. The function can also receive data from the previous function via `af_deq()`. After data transfer, the function releases resources by calling `af_free()` and `af_destroy()`. These workflows are akin to those of conventional socket I/O, allowing developers to implement inter-function communication in a familiar manner. Data serialization, synchronization, and low-level memory management are hidden from developers and handled internally by the AF channel. As a result, adopting *AccelFaaS* requires only simply replacing conventional socket APIs with AF channel primitives, without restructuring main function logic.

Internal-class primitives: Internal-class AF primitives are invoked exclusively by the AF channel instance to support synchronized data transfer over the cached shared-memory pool. When a function calls `af_enq()` or `af_deq()`, the AF channel instances in the source (SRC) and destination (DST) functions internally invoke `af_notify()` and `af_wait()`

to coordinate safe and ordered data transfer. For example, when the SRC function enqueues a message, the message is placed into the DST function's queue, and the DST function is notified to dequeue and process the message, as shown in the top of Fig. 5. This ensures correct synchronization semantics without exposing concurrency control to developers.

Internal-class primitives also provide transparent support for inter-node communication over RDMA. If the DST function is not found in the local AF channel, the AF channel instance automatically switches to RDMA-based data transfer. Upon a call to `af_enq()`, the AF channel instance determines whether the DST function is local or remote. In remote scenarios, `af_send_out()` is internally invoked to transmit the message over a pre-established RDMA connection managed by the AF agent. During this process, the RDMA identifiers of the DST function are automatically resolved, enabling seamless inter-node communication without requiring any changes to the application-level code.

IV. Evaluation

A. Prototype Implementation

We have implemented a full prototype of *AccelFaaS* on top of Knative [35]. *AccelFaaS* components, including the AF controller, the AF agent, and the AF channel are implemented with a combination of C++ and Golang. These components are encapsulated into Docker containers to facilitate seamless deployment and integration within Knative. For example, the AF controller is implemented as Kubernetes custom resource [36]. The AF actuator is implemented as a set of eBPF/XDP programs that are dynamically attached to the veth interface of each pod in Knative. On each node within the Knative cluster, *AccelFaaS* mounts a shared memory volume, dedicated to the AF channel, for each workflow pod. To facilitate seamless integration, *AccelFaaS* rebuilds functions with AF channel primitives and pre-deploys them to Knative as containers.

B. Experimental Setup

Testbed: We configure a local cluster of four physical nodes, each equipped with an Intel Xeon Gold 6242R processor and NVIDIA ConnectX-7 100 Gbps NICs, running Ubuntu 22.04 LTS with Linux kernel 6.1. The nodes are interconnected in a star topology via an HPE FlexFabric 5945 switch, providing 100 Gbps non-oversubscribed connectivity over RoCEv2. In this configuration, network performance is bounded by the NIC and the end-host network stack rather than switch capacity. We deploy Knative (v1.13) as the FaaS platform on this cluster. One node serves as the master node, hosting the ingress gateway and the *AccelFaaS* controller, while the remaining three nodes act as worker nodes running functions as well as the *AccelFaaS* agent, actuator, and *AccelFaaS* channel.

Baselines: We evaluate *AccelFaaS* against five representative systems: Knative [35], vHive [4], FAASM [12], SPRIGHT [7], and Pheromone [15]. Knative serves as the canonical baseline, representing the standard scale-from-zero model. vHive represents the state of the art in cold-start optimization, enabling fast function startup through function snapshotting based on Record-and-Prefetch (REAP). Note that REAP is not supported on Linux kernel versions 6.1 and later, which are required to run our proposed memory pool. We instead use Firecracker-based snapshotting to approximate vHive's cold-start optimization. FAASM, SPRIGHT, and Pheromone serve as baselines for efficient inter-function data transfer over shared memory channels.

TABLE IV
Details of Evaluation Workloads Selected From Two Widely Used Benchmark Workloads for FaaS [37], [38]

Category	Workload	Objective
Single function	*Chameleon, PyAES, RNN-serving, CNN-serving*	Cold-start latency under scale-from-zero
Synthetic chain	N-hop *chained-function-serving*	Data transfer performance on isolated environments
Realistic workflow	*video-analytics*, *stacking-training*	Data transfer performance on complex fan-in/fan-out

Evaluation Goals: We evaluate *AccelFaaS* to answer the following four research questions:

- *Q1:* How much can *AccelFaaS* reduce the cold-start latency while preserving the scale-from-zero model? (Section IV-C)
- *Q2:* Can *AccelFaaS* maintain low cold-start latency when dispatching burst requests across multiple nodes? (Section IV-D)
- *Q3:* How much can *AccelFaaS* accelerate inter-function data transfer performance? (Section IV-E)
- *Q4:* How much can *AccelFaaS* reduce memory inefficiencies during data transfer? (Section IV-F)

For **Q1** and **Q2**, we focus on systems that natively support the scale-from-zero execution model, as cold-start latency is meaningful only in this context. Accordingly, we compare *AccelFaaS* with Knative and vHive. To ensure a fair comparison with vHive, which employs the Firecracker microVM runtime, we additionally evaluate *AccelFaaS* using the same Firecracker runtime. The remaining baselines are excluded from cold-start latency experiments, as they cannot support the scale-from-zero model. These systems are instead used to evaluate inter-function data transfer performance (**Q3** and **Q4**). Note that FAASM partially supports the scale-from-zero model, but it relies on a non-standard invocation model in which functions are triggered through system-call-like mechanisms rather than network requests (e.g., HTTP). This difference renders direct comparison with *AccelFaaS* infeasible for cold-start latency analysis. Thus, we exclude it from cold-start latency evaluations.

Workload: To evaluate the performance of *AccelFaaS* under realistic serverless environments, we select seven representative workloads drawn from two widely used serverless benchmark suites, FunctionBench [38] and vSwarm [37]. Table IV summarizes the selected workloads. For evaluating cold-start latency, we use *Chameleon*, *pyaes*, *RNN-serving*, and *CNN-serving*. The first two workloads represent simple functions with lightweight computation, serving as baseline cold-start overheads. In contrast, *RNN-serving* and *CNN-serving* are compute-intensive workloads that involve model loading and initialization, allowing us to examine whether *AccelFaaS* reduces cold-start latency for heavy functions with substantial initialization costs. For

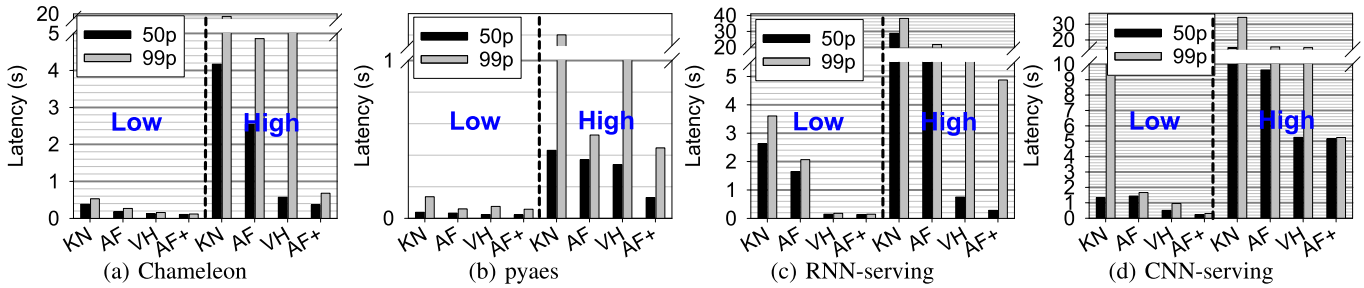


Fig. 6. Cold-start latency measurement results of Knative(*KN*), *AccelFaaS* (*AF*), vHive(*VH*), and *AccelFaaS* with Firecracker snapshots (*AF+*) under low (100) and high (500) concurrent request scenarios.

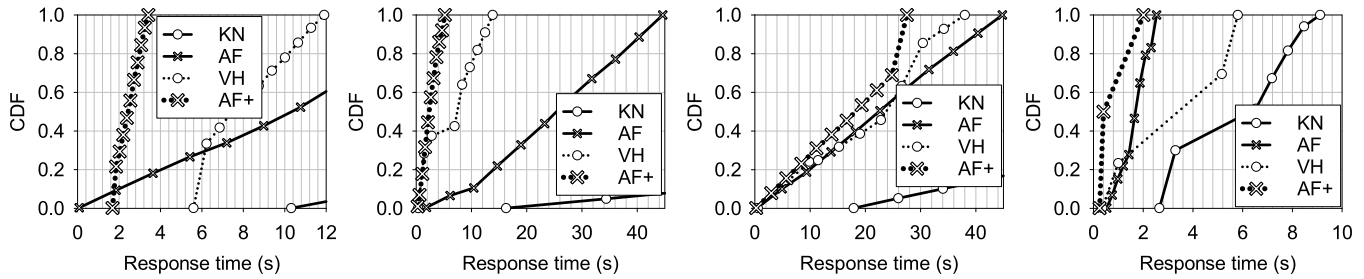


Fig. 7. Cold-start latency measurements of Knative (*KN*), *AccelFaaS* (*AF*), vHive (*VH*), and *AccelFaaS* with Firecracker snapthots (*AF+*) under 1,000 concurrent requests.

evaluating inter-function data transfer performance, we employ both a synthetic workload and two realistic workloads. The synthetic workload consists of a set of functions in which each function simply forwards data to the next function through a shared-memory channel without performing application-level computation (i.e., a hello-world). By minimizing function execution time, this workload isolates communication and memory-management overheads, thereby exposing the upper bound of achievable performance gains from platform-level optimizations. Realistic workloads involve compute-intensive function logic, library loading, model initialization, and complex fan-in/fan-out communication patterns. These workloads capture behaviors of practical serverless workloads, where individual function executions often last hundreds of milliseconds.

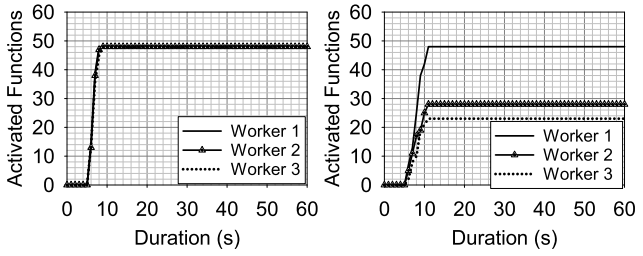
C. Cold-Start Latency Reduction

Here, we evaluate the effectiveness of *AccelFaaS* in reducing cold-start latency by measuring the end-to-end response time of *Chameleon*, *pyaes*, *RNN-serving*, and *CNN-serving*. We consider two traffic scenarios by increasing the number of concurrent requests from 100 (*Low*) to 500 (*High*). Here, a single function processes these requests without horizontal auto-scaling, simulating single-node bottleneck scenarios.

Overall Performance: Fig. 6 reports the P50 and P99 cold-start latencies of four systems: Knative (KN), *AccelFaaS* (AF), vHive (VH), and *AccelFaaS* with Firecracker snapshotting (AF+). In the Low scenario, *AccelFaaS* consistently outperforms Knative across all workloads. For example, under *Chameleon*, *AccelFaaS* achieves a P50 latency of 181 ms and a P99 latency below 270 ms, compared to 381 ms and 529 ms for Knative. This advantage persists in the High scenario, where *AccelFaaS* achieves up to 4× lower P99 latency, demonstrating robust scalability under bursty workloads. These performance gains stem from decoupling control channel establishment from the function cold-start path.

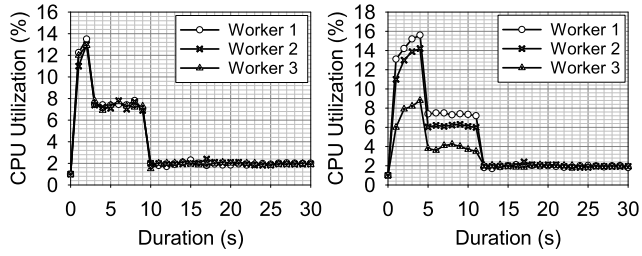
vHive outperforms both Knative and *AccelFaaS* by leveraging Firecracker snapshotting to accelerate the function spawning stage. However, snapshotting alone cannot eliminate the overhead of the preparation stage, as control channels cannot be captured in snapshots. As a result, vHive exhibits substantial tail latency due to control channel setup overheads, as shown in Fig. 6. In contrast, *AccelFaaS* combined with snapshotting (AF+) eliminates this remaining overhead. In the Low scenario, AF+ shows approximately 20% and 30% lower P50 and P99 latencies than vHive. In the High scenario, AF+ achieves more than a 5× reduction in P90 latency compared to vHive. These results indicate that even function spawning is accelerated via snapshots, control channel setup still acts as the dominant contributor to cold-start latency, and eliminating this overhead is critical for achieving optimal performance.

Request Dispatching: Rapidly dispatching incoming requests from the global activator queue to target functions is critical for mitigating cold-start latency caused by head-of-line (HoL) blocking, particularly under bursty traffic. In *AccelFaaS*, the AF actuator efficiently dispatches incoming requests from the global queue to per-function local queues. To evaluate the effectiveness of the AF actuator, we measure cold-start latency for the same



(a) *AccelFaaS* (b) vHive

Fig. 8. Activated functions under 500 concurrent requests.



(a) AccelFaaS (b) vHive

Fig. 9. Node CPU utilization under 500 concurrent requests.

four workloads under 1,000 concurrent requests, explicitly inducing HoL blocking in the activator queue.

Fig. 7 shows the CDF of cold-start latency. Across all workloads, *AccelFaaS* consistently outperforms Knative. For *CNN-serving*, more than 90% of requests complete within 25 seconds under *AccelFaaS*, whereas the same percentile exceeds 40 seconds under Knative. Similarly, for *RNN-serving*, the median latency decreases from approximately 15 seconds in Knative to 6.5 seconds in *AccelFaaS*, highlighting the impact of eliminating HoL blocking. As in previous experiments, vHive outperforms both Knative and *AccelFaaS* by leveraging snapshot-based function spawning. For example, vHive reduces the P90 latency of *pyaes* and *Chameleon* to approximately 2.0 and 3.2 seconds, respectively, compared to 6.8 and 9.5 seconds under Knative. However, vHive still suffers from HoL blocking in the centralized activator queue, which limits its responsiveness under high concurrency. In contrast, *AccelFaaS* combined with Firecracker snapshots (AF+) achieves the best performance across all systems. For *RNN-serving*, AF+ reduces the P95 latency to 8.5 seconds, compared to 12.5 seconds in vHive, while for *CNN-serving*, it lowers the P90 latency to below 20 seconds, outperforming vHive by more than 10 seconds. These results demonstrate that snapshotting alone is insufficient to address cold-start bottlenecks, and that *AccelFaaS* achieves optimal performance by combining snapshot-based spawning with per-function local request queueing and control-channel decoupling.

D. Multi-Node Scalability and Resource Efficiency

To verify that the cold-start improvements of *AccelFaaS* are not limited to single-node scenarios and to ensure they are not artifacts of hardware saturation, we analyze the system's scalability and resource utilization under a burst of 500 concurrent requests across three worker nodes. Following production-grade serverless configurations [39], [40], we employ request multiplexing with a concurrency limit of 10 per instance. In this setup, 500 concurrent requests are effectively mapped to approximately 50 active instances, ensuring the workload remains within the physical capacity of each worker node (60-cores) in our cluster.

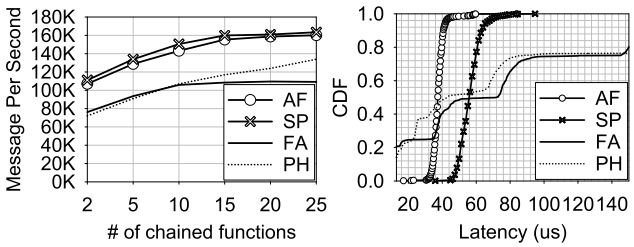
Fig. 8 reports the distribution of activated instances. *AccelFaaS* achieves near-perfect load balancing, with each worker activating approximately 48 instances (Fig. 8(a)). In contrast, vHive exhibits a pronounced imbalance (Fig. 8(b)), where Worker 1 is prematurely saturated with 48 instances while Workers 2 and 3 remain underutilized (28 and 22 instances). This imbalance in vHive stems from its reliance on coarse-grained, proxy-based metric aggregation (e.g., 2 s intervals), which creates a feedback lag during bursts. *AccelFaaS* overcomes this via the AF actuator, which detects burst signals directly in the kernel side, and the AF agent, which relays these metrics to the control plane without aggregation delay, enabling immediate load redistribution.

Fig. 9 further confirms that the observed performance gains are not caused by mitigating single-node CPU saturation or queuing bottlenecks. Under peak load, *AccelFaaS* maintains a balanced CPU utilization of approximately 13–14% per worker, with memory and I/O usage remaining below 15% of capacity. Notably, even in vHive, CPU utilization on all workers remains well below saturation despite its skewed distribution. These results confirm that the high cold-start latencies observed in existing systems are not due to hardware resource exhaustion, but rather to architectural bottlenecks. By ensuring balanced scaling and timely scheduling decisions, *AccelFaaS* demonstrates robust scalability and resource efficiency in multi-node environments.

E. Inter-Function Data Transfer Performance

We evaluate the efficiency of *AccelFaaS* in inter-function data transfer using both synthetic and realistic workloads (Table IV). The synthetic workload isolates the data plane's performance by minimizing application-level processing, while realistic workloads, characterized by complex fan-in/out topologies, represents real-world FaaS execution scenarios.

For fair comparisons, we adapt FAASM and Pheromone to support these workloads, as they require workload-level integration with their runtime-specific APIs for inter-function communication. For FAASM, which relies on WebAssembly-based isolation, we implement a lightweight data transfer wrapper within the *faabric* executor. This allows direct pointer passing between functions, bypassing global state synchronization bottlenecks while fully utilizing FAASM's shared-memory buffers. For Pheromone, we integrate the workloads with its data-centric bucket API and optimized the *cpp-ipc* memory pools to match vSwarm payload sizes, preventing runtime allocation overheads. These adaptations allow FAASM and Pheromone to efficiently execute the evaluated workloads while preserving their original communication mechanisms. SPRIGHT is used without modification, as its eBPF-based data channel transparently supports our workloads.



(a) Throughput (b) Per-hop latency

Fig. 10. Inter-function data transfer performance of *AccelFaaS (AF)*, FAASM *(FA)*, Pheromone *(PH)*, and SPRIGHT *(SP)*.

Synthetic Workload: We first evaluate the synthetic workload (*chained-function-serving*) by varying the number of functions (the workload length) and payload sizes. Each hop's payload is sampled from a uniform distribution (4 KB to 4 MB) to simulate diverse data exchange patterns. We measure end-to-end throughput and per-hop latency across all configurations.

As shown in Fig. 10(a), *AccelFaaS* achieves superior throughput compared to FAASM and Pheromone across all settings. These baselines suffer from memory fragmentation and frequent page faults inherent in their non-contiguous, cold-page-based shared memory pool. *AccelFaaS* mitigates these overheads by leveraging a cached shared memory pool backed by physically contiguous huge pages. While SPRIGHT achieves slightly higher throughput in specific regimes, it relies on a static, over-provisioned memory pool, leading to excessive memory consumption (Section II-B). In contrast, *AccelFaaS* dynamically scales its memory footprint based on load, offering a more resource-efficient architecture.

Fig. 10(b) shows the per-hop latency CDF (time taken to pass data from one function to the next). *AccelFaaS* maintains stable performance across the majority of payload sizes. Although it exhibits higher latency than other systems for the smallest 20% of samples (e.g., 4 KB) due to the fixed setup costs of THP mapping, this overhead becomes negligible as payload size increases. In realistic scenarios with mixed data sizes, *AccelFaaS* 's ability to prevent throughput collapse and latency variance results in superior overall scalability.

Realistic Workload: We further evaluate *AccelFaaS* using two realistic workloads: *video-analytics* and *stacking-training*.

The *video-analytics* workload consists of a four-stage pipeline (ingestion, pre-processing, object detection, and post-processing), with each stage running in a dedicated function instance. We increase the concurrency K (the number of concurrent pipelines) from 20 to 100 to test scalability under high concurrency. High values of K increase the number of activated functions, which in turn causes contention on the shared memory channel. The upper row in Fig. 11 shows the evaluation results. Fig. 11(a) presents the P50 latency. At $K = 20$, *AccelFaaS* with snapshotting achieves a P50 latency of 248 ms, outperforming SPRIGHT (261 ms) and other systems. As concurrency increases to $K = 60$, the P50 latency of all systems rises due to increased data channel contention; however, *AccelFaaS* maintains the lowest latency at 287 ms, while other systems range from approximately 301 to 412 ms. At $K = 100$, *AccelFaaS* reaches 361 ms, whereas SPRIGHT records 396 ms and FAASM and Pheromone exceed approximately 498–536 ms. Fig. 11(b) shows the P99 latency. At $K = 20$, *AccelFaaS* achieves a P99 latency of 309 ms, compared to 341 ms for SPRIGHT. Under high concurrency ($K = 100$), where queuing delays dominate, *AccelFaaS* maintains a P99 latency of 501 ms, while other systems degrade significantly, reaching 620–753 ms. This corresponds to a P99 latency reduction of up to 33%. Throughput results (Fig. 11(c)) further confirm this trend. *AccelFaaS* achieves up to 16.4 frames/s, which is 15–35% higher than other systems.

The *stacking-training* workload represents a fan-in/fan-out topology ($G_1 \rightarrow G_{2,1}, \ldots, G_{2,K} \rightarrow G_3$), where K independent models are trained in parallel and their results are aggregated. This workload incurs bursty and highly concurrent data transfers, placing significant stress on the data channel shared across functions. The lower row of Fig. 11 shows the evaluation results. As shown in Fig. 11(a), *AccelFaaS* consistently achieves the lowest P50 latency across all concurrency levels. At $K = 20$, *AccelFaaS* records 387 ms, outperforming SPRIGHT (401 ms). As concurrency increases to $K = 100$, *AccelFaaS* reaches approximately 728 ms, while SPRIGHT rises to around 772 ms and other systems exceed approximately 912–937 ms. Similar trends are observed in tail latency. In Fig. 11(b), *AccelFaaS* achieves a 7–20% P99 latency reduction at $K = 20$ and maintains a P99 of 981 ms at $K = 100$, whereas baseline systems exceed 1.1 s. Throughput results (Fig. 11(c)) follow the same trend, with *AccelFaaS* achieving up to 60% higher throughput than other systems.

These performance improvements in realistic workloads stem from *AccelFaaS* 's data-plane design. The cache-friendly shared-memory data channel enables efficient inter-function communication with improved data locality, leading to lower P50 latency and higher throughput. Under high concurrency, *AccelFaaS* maintains stable P99 latency through per-function local queues and an efficient eBPF-based dispatch mechanism, which mitigate contention and prevent queue amplification.

Inter-node Data Transfer: While *AccelFaaS* mainly optimizes intra-node data transfer, it transparently extends its shared memory pool across multiple nodes via RDMA to support inter-node data transfer. We evaluate the overhead introduced by this extension using a hello-world workload consisting of two functions deployed across two physical nodes. These nodes are interconnected by a 100 Gbps RoCEv2 fabric, with a baseline network round-trip latency of approximately 4 μs (measured via RDMA ping-pong). Here, *AccelFaaS* is compared with Pheromone, the only baseline supporting inter-node transfers via gRPC, as others like SPRIGHT lack native inter-node support. To isolate inter-node transfer overhead, we measure warm-start response times where functions are pre-activated on each node before receiving requests.

Fig. 12 presents the results for both intra-node and inter-node scenarios. In the intra-node case, *AccelFaaS* achieves a latency of 0.3 ms, outperforming Pheromone's 0.35 ms. When distributed across nodes, *AccelFaaS* maintains a latency of 1.2 ms, outperforming Pheromone's 1.67 ms by 28%. These results demonstrate that *AccelFaaS* scales to multi-node data transfer with manageable overhead by leveraging pre-established RDMA channels and pre-mapped memory regions.

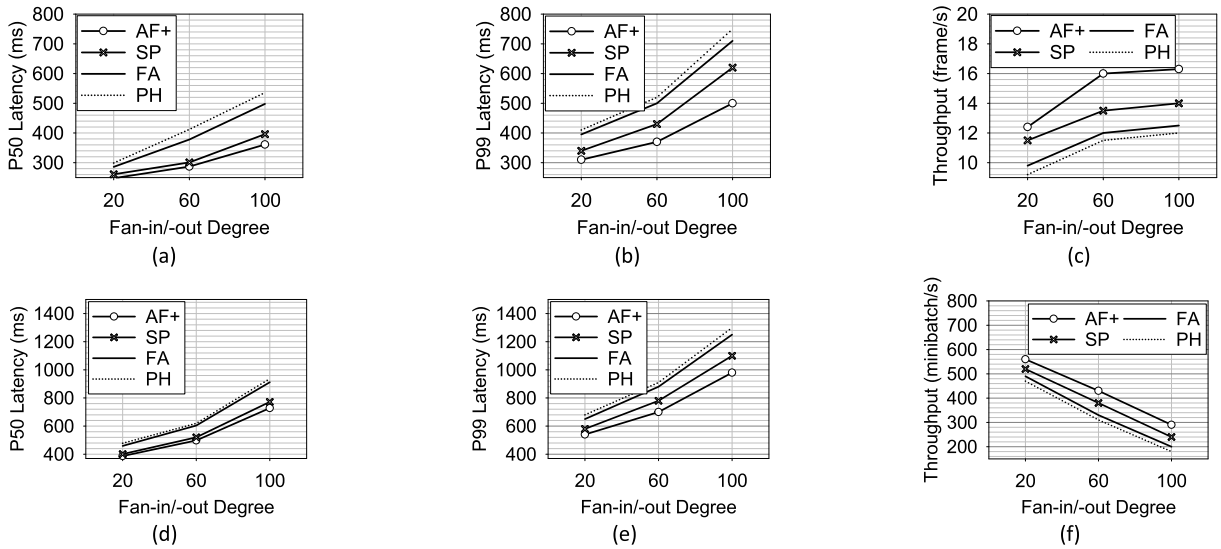


Fig. 11. Performance comparison of *AccelFaaS* (with Firecracker snapthos) and other FaaS systems with two realistic workloads. The upper row corresponds to *video-analytics*, while the lower row corresponds to *stacking-training*.

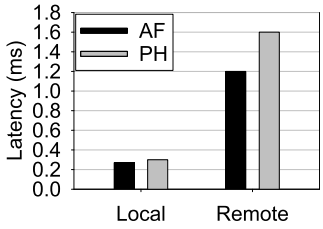


Fig. 12. Inter-node function invocation latency.

It is important to distinguish *AccelFaaS* from specialized RDMA-based FaaS systems such as rFaaS [41] and RMMap [42]. While those systems focus on maximizing network utilization for bulk data movement or streaming, *AccelFaaS* centers on transparently extending intra-node optimizations to inter-node cases rather than providing a novel RDMA-specific data transfer mechanism. We consider the integration of advanced RDMA optimizations from these specialized studies as a promising direction for future work to further enhance *AccelFaaS*.

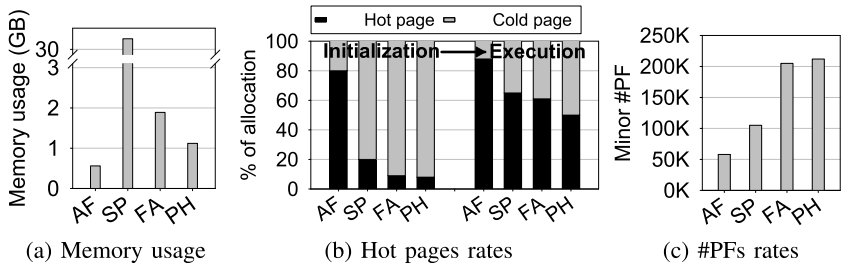
F. Memory Inefficiencies and Metric Collection Overheads

Here, we measure memory inefficiencies of each system during inter-function data transfer, identifying the root causes of the performance differences observed in Section IV-E. We then evaluate the metric collection overhead caused by AF actuators. For this, we deploy a *chained-function-serving* workload and increase the number of functions in the workload from 2 to 25 gradually during data transfer.

Memory Usage: Efficient memory usage is critical in FaaS, where workloads and resource demands fluctuate dynamically. Fig. 13(a) shows the total memory usage of each system observed during data transfer. SPRIGHT exhibits the highest memory usage due to its reliance on statically allocated `hugetlb` pages. To avoid prohibitive huge-page initialization overhead, it maintains its entire memory pool statically without dynamic (de)allocation mechanisms, leading to excessive memory usage irrespective of actual runtime load. Consequently, SPRIGHT consumes over 32 GB of memory, rendering it impractical for real-world FaaS environments serving numerous concurrent workflows. In contrast, *AccelFaaS* exhibits the lowest memory usage by dynamically resizing its cached shared memory pool based on runtime load and the number of active functions. By leveraging THP and adaptive scaling, *AccelFaaS* provides superior resource efficiency and elasticity for variable serverless workloads compared to SPRIGHT.

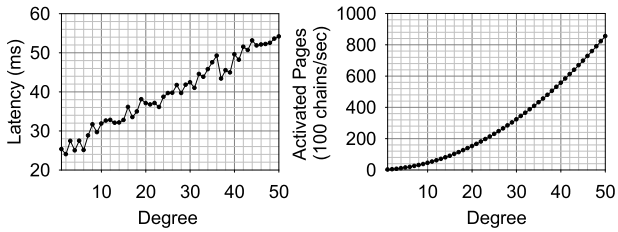
Hot Page Ratio and Page Faults: The initial state of the shared memory pool is a critical factor in data transfer performance. Fig. 13(c) shows the hot page ratio for each system's shared memory pool at initialization time and after workload execution. *AccelFaaS* maintains a high hot page ratio from the outset due to its page pre-warming mechanism, further enhanced by the use of physically contiguous THP regions that improve spatial locality. In contrast, SPRIGHT begins with predominantly cold pages because it lacks an explicit warming phase, despite its use of contiguous `hugetlb` regions. FAASM and Pheromone exhibit the lowest initial hot page ratios, as they forgo pre-warming and allocate memory in non-contiguous regions using conventional allocators.

This initial shared memory state directly translates into the page fault behaviors observed in Fig. 13(b). *AccelFaaS* incurs only $\approx$ 51 K minor page faults, as most of its pages are pre-warmed as hot pages during the initialization time. Thus, SPRIGHT suffers from twice as many faults as *AccelFaaS* because its pages must be warmed on-demand during execution, leading to a performance drop. FAASM and Pheromone incur the highest fault rates, as their cold and fragmented memory pools trigger frequent overheads for populating page tables and resolving address mappings [43], [44]. By combining contiguous memory layouts with page pre-warming, *AccelFaaS* ensures high-performance and stable data transfers.



(a) Memory usage (b) Hot pages rates (c) #PFs rates

Fig. 13. Memory overhead measurement results of *AccelFaaS (AF)*, SPRIGHT *(SP)*, FAASM *(FA)*, and Pheromone *(PH)* during data transfer.



(a) Latency (b) Activated Pages

Fig. 14. Metric collection overheads.

Metric Collection Overheads: AccelFaaS deploys AF actuators within each function pod to collect fine-grained metrics during data transfer. We evaluate the performance implications of these actuators by measuring end-to-end latency and accessed memory pages in *video-analytics* while varying the fan-in/fan-out degree from 1 to 50. As shown in Fig. 14(a), the end-to-end execution latency, inclusive of monitoring overheads, exhibits a linear correlation with the scaling degree, increasing from approximately 25 ms at degree 1 to just over 50 ms at degree 50. This trend reflects the cumulative cost of our hybrid collection strategy, utilizing both in-memory logging and kernel eBPF maps, rather than systemic contention within kernel data structures. Crucially, the per-hop baseline latency remains minimal (25–55 μs), demonstrating that our metric collection framework avoids superlinear latency penalties even under high-concurrency scenarios.

We further quantify the memory overhead introduced by the AF actuator's in-memory logging. Fig. 14(b) shows the number of activated pages when processing *video-analytics*. Unlike latency, activated pages for logging grow quadratically with the fan-in/fan-out degree, increasing from a few pages at degree 1 to over 800 pages at degree 50. This behavior arises because a chain of degree K generates K log entries, and each entry grows with accumulated metadata. Thus, *AccelFaaS* deliberately trades additional memory usage for bounded monitoring overhead and predictable latency scaling, avoiding eBPF contention while preserving μs-scale dispatch latency under highly parallel FaaS workloads.

V. Related Work

A. Cold-Start Latency

Function Snapshotting: Snapshot-based systems [3], [4], [45] accelerate cold-start by restoring the saved execution state of a previously initialized function instance. REAP [4] uses a custom page fault handler to track page faults during request execution and records the hot pages accessed by the function; these pages are then restored from the snapshot to accelerate subsequent cold-starts. Although snapshotting removes initialization of the sandbox and function binary, it cannot eliminate performance bottlenecks beyond these stages. Even when a function resumes from a snapshot, it must still establish a fresh control channel with the control plane. This is because the control channel cannot be snapshotted: a resumed instance always receives a new network identity (e.g., IP and MAC), and previously created sockets or RPC endpoints bound to old addresses cannot be restored or reused. Consequently, snapshotting cannot decouple control-channel setup from the critical path. Nevertheless, snapshot techniques complement *AccelFaaS*, as the two target orthogonal sources of overhead. For example, snapshots can accelerate sandbox initialization, while *AccelFaaS* eliminates control channel setup, jointly reducing end-to-end cold-start latency.

Function Pre-loading: Another line of work adopts function pre-loading techniques. SPRIGHT [7] maintains pre-warmed function instances to remove shared memory setup delays entirely, and SAND [46] collapse isolation boundaries to keep functions continuously active within shared sandboxes. Speculative pre-activation [47], [48] further attempts to forecast invocation patterns and launch hot functions ahead of demand. These methods effectively reduce cold-start latency but compromise key properties of serverless computing: keeping functions always on or launching them speculatively undermines the scale-from-zero model and can incur unnecessary resource consumption when predictions are inaccurate. In addition, launching a multiple functions within the same sandbox eliminates isolation boundaries, resulting in security issues in multi-tenant cloud environments.

Dirigent [49] reduces part of the cold-start latency by relocating per-function queues from sidecars to the data plane, thereby removing queue initialization overhead. However, this optimization addresses only one step in the startup sequence. Empirical workload studies [32], [50] show that under bursty traffic the dominant bottleneck shifts from queue construction to the slow delivery of requests into newly created sandboxes. Dirigent provides no specialized mechanism (e.g., zero-copy transfer from the queue to the dispatcher) to accelerate this dispatch process, leaving substantial delays in routing requests to fresh function instances. In contrast, *AccelFaaS* introduces a fast dispatching mechanism that immediately forwards incoming requests to per-function shared-memory queues through *AccelFaaS* actuators, ensuring that both startup and early request handling are accelerated.

B. Data Transfers Between Functions

Inter-node Communication: Inter-function data transfer is a critical performance factor in FaaS workflows. Prior work has primarily optimized data transfer between functions deployed on different nodes by designing efficient shared storage systems [25], [51], [52], [53], [54], [55]. These systems reduce inter-node communication overhead by employing techniques such as locality-aware object placement or lightweight, distributed key-value storage. While these methods are effective for inter-node scenarios, they are unsuitable for data transfers between functions co-located in the same node. When such co-located functions communicate indirectly through these distributed storage, they incur unnecessary storage interactions that significantly degrade chaining performance [56].

Intra-node Communication: Another line of studies center on communication between functions co-located on the same node. Most of them adopt shared memory-based channels [7], [12], [15]. While these systems accelerate communication within a serverless workflow, they suffer from memory fragmentation and frequent page faults during intensive chaining workloads, as discussed in Section II. SPRIGHT [7] achieves relatively strong performance by employing a shared memory channel built on DPDK's `rte_mempool` with hugetlb-based memory pools. However, it requires functions to remain always active to avoid costly hugetlb-based memory pool initialization, violating the scale-from-zero model. Moreover, accessing hugetlb memory requires functions running with root privileges, raising serious security concerns in multi-tenant environments. DataFlower [56] optimizes data transfer for co-located functions using an IPC-based local pipe connector. It streams data larger than 16 KB through kernel pipe buffers and temporarily stores in-flight data in a host-level data sink until the target function becomes available. Although effective for streaming workloads, DataFlower still involves the network stack for data smaller than 16 KB due to socket-based transfer, and its pipe connector is fundamentally copy-based, making it unsuitable for memory-intensive workloads where functions exchange data thousands of times per second. Faastlane [14] accelerates serverless communication by introducing shared memory channels between functions within the same process. However, its shared memory design is statically provisioned at initialization. As a result, Faastlane does not address dynamic scaling, bursty fan-in/fan-out patterns, or allocator contention under high concurrency. Moreover, Faastlane focuses on data transfer optimization and does not redesign the request dispatch or preparation path, which we show to be the dominant bottleneck in modern FaaS platforms.

Unlike these systems, *AccelFaaS* employs a cached shared memory channel built over physically contiguous memory regions and pre-warms channel pages to eliminate fragmentation, reduce page faults, and support efficient data transfer for co-located functions at scale.

VI. Discussion and Limitation

Limitations of Transparent Huge Pages: AccelFaaS leverages THP to allocate its channel within physically contiguous memory regions, thereby improving memory locality and reducing fragmentation. Promoting standard pages to huge pages requires contiguous physical memory blocks of the corresponding large page size. However, THP has three known limitations. First, workloads with sparse memory access patterns can lead to fragmentation over time, reducing THP effectiveness. Second, under memory pressure, THP may fall back to standard 4 KB page allocations, compromising the contiguity of the cached memory pool. Third, as the system runs longer, memory fragmentation increases, making large contiguous regions more difficult to allocate.

Despite these challenges, their impact is mitigated in practice because most serverless workloads exhibit contiguous memory access patterns that favor efficient data sharing [57]. To address fragmentation and allocation concerns, *AccelFaaS* includes mechanisms to retain its constituent pages, considering high-order page availability. This design allows for an efficient memory pool when the pages are used by the serverless workflow. Additionally, techniques such as *khugepaged* [58] and GCMA [59] can promote fragmented pages or improve the likelihood of contiguous allocations. Integrating these techniques into *AccelFaaS* is planned as future work.

Compatibility with Other FaaS Platforms: Although our current prototype is built on Knative, *AccelFaaS* is designed to be readily extensible to other serverless platforms. Specifically, it assumes (i) a sidecar-based data plane and (ii) function instances with access to shared memory on Linux kernel version 6.1 or later, which supports a cached memory pool. These assumptions align with common configurations in modern cloud environments, where sidecar proxies are widely adopted and shared memory is generally accessible. To ensure safe deployment, each serverless workflow is encapsulated within a pod sandbox that securely exposes the shared memory channel without requiring elevated privileges. Furthermore, to support compatibility with existing serverless applications, the channel primitives are implemented in C++ and exposed through language bindings, enabling seamless integration with functions written in other languages.

Fault Tolerance: AccelFaaS is designed to uphold standard FaaS fault-tolerance, such as at-least-once semantics and infrastructure-managed retries. The AF actuator is a stateless component sharing fate with its host pod; thus, any failure triggers a standard Kubernetes restart without complex recovery logic. Furthermore, AF channels are strictly ephemeral fast-path optimizations. Since durability is anchored in the underlying platform's persistent queuing, any loss of in-flight state in shared memory simply causes the request to be retried via proven baseline recovery paths. While the current design seamlessly integrates with existing mechanisms, further refining fine-grained state checkpointing for long-running workflows remains a promising direction for future work.

Function Co-location: Co-locating functions within the same workflow in a single pod or on the same node is a common and well-established optimization in existing FaaS systems [7], [12], [13], [32]. In practice, functions within the same workflow are typically owned by the same tenant and collaboratively execute a single application pipeline. Therefore, co-locating these functions does not introduce additional security risks

beyond those already present in conventional serverless execution models. Isolation across different workflows and tenants is enforced by underlying Kubernetes mechanisms, including namespace separation, container isolation, and network isolation (e.g., VXLAN), which prevent unauthorized access across pods. As a result, co-location in *AccelFaaS* should be viewed as a practical optimization that improves data locality and communication efficiency without weakening standard isolation guarantees. Moreover, *AccelFaaS* can be combined with stronger runtime [60], [61] and network isolation techniques [62], [63] to further reduce potential cross-tenant attacks. We leave such integration as future work.

VII. Conclusion

This paper presents *AccelFaaS*, a high-performance and memory-efficient data plane architecture that addresses two significant challenges in serverless computing: long cold-start latency and memory inefficiencies during inter-function data transfer. *AccelFaaS* mitigates cold-start latency by decoupling control channel setup from function initialization and replacing sidecars with lightweight in-kernel alternatives. A cached shared memory pool with pre-warmed THP further reduces page faults and fragmentation during chaining. Evaluations demonstrate up to 11–27× faster cold starts, 8–16× lower chaining latency, and 4× higher throughput compared to state-of-the-art solutions.

References

[1] Y. Li, Y. Lin, Y. Wang, K. Xu, and C. Xu, "Serverless computing: state-of-the-art, challenges and opportunities," *IEEE Trans. Serv. Comput.*, vol. 16, no. 2, pp. 1522–1539, 2022.

[2] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *J. Syst. Softw.*, vol. 170, 2020, Art. no. 110708.

[3] D. Du et al., "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 467–481.

[4] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 559–572.

[5] Y. Bai, Z. Yang, and F. Gao, "Faast: An efficient serverless framework made snapshot-based function response fast," in *Proc. 33 rd Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2024, pp. 174–185.

[6] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: Skip redundant paths to make serverless fast," in *Proc. 14th Eur. Conf. Comput. Syst.*, 2020, pp. 1–15.

[7] S. Qi, L. Monis, Z. Zeng, I. Chin Wang, and K. Ramakrishnan, "SPRIGHT: Extrating the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing," in *Proc. ACM SIGCOMM 2022 Conf.*, 2022, pp. 780–794.

[8] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with Greedy-dual caching," in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 386–400.

[9] Knative, "Configuring scale to zero," 2024. [Online]. Available: https://knative.dev/docs/serving/autoscaling/scale-to-zero/

[10] OpenFaaS, "Extended timeouts," 2024. [Online]. Available: https://docs.openfaas.com/tutorials/expanded-timeouts/

[11] AWS, "Configure Lambda function timeout," 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-timeout.html

[12] S. Shillaker and P. Pietzuch, "FAASM: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 419–433.

[13] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 152–166.

[14] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 805–820.

[15] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2023, pp. 1489–1504.

[16] "OpenFaaS: Chaining OpenFaaS functions," 2017. [Online]. Available: https://ericstoekl.github.io/faas/developer/chaining_functions

[17] "Eventing knative," 2025. [Online]. Available: https://knative.dev/docs/eventing/

[18] "OpenWhisk: Creating action sequences," 2024. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/actions.md#creating-action-sequences

[19] "AWS step functions," 2026. [Online]. Available: https://aws.amazon.com/stepfunctions/

[20] "Azure durable functions," 2026. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/

[21] P. Sahu, S. Wei, N. J. Yadwadkar, and M. Tiwari, "Understanding sidecars in cloud orchestration," in *Proc. 3 rd Workshop SErverless Syst., Appl. Methodologies*, 2025, pp. 1–11.

[22] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proc. 11th USENIX Workshop Hot Topics Cloud Comput.*, pp. 21–26, 2019.

[23] J. Jiang et al., "A systematic evaluation of machine learning on serverless infrastructure," *VLDB J.*, vol. 33, pp. 425–449, 2024.

[24] G. Liu et al., "Fuyao: DPU-enabled direct data transfer for serverless computing," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2024, pp. 431–447.

[25] V. Sreekanti et al., "Cloudburst: Stateful functions-as-a-service," in *Proc. Int. Conf. Very Large Data Bases*, 2020, pp. 2438–2452.

[26] D. Barcelona-Pons, P. Garcia-Lopez, and B. Metzler, "Glider: Serverless ephemeral stateful near-data computation," in *Proc. 24th Int. Middleware Conf.*, 2023, pp. 247–260.

[27] "Messaging and state layer for distributed serverless applications," 2022. [Online]. Available: https://github.com/faasm/faabric

[28] "A high-performance inter-process communication library using shared memory on Linux/Windows," 2021. [Online]. Available: https://github.com/mutouyun/cpp-ipc

[29] rte_mempool, "Memory pool library," 2024. [Online]. Available: https://doc.dpdk.org/guides/prog_guide/mempool_lib.html

[30] Y. Zhang and S. Angel, "Quilt: Resource-aware merging of serverless workflows," in *Proc. ACM SIGOPS 31st Symp. Operating Syst. Princ.*, 2025, pp. 907–927.

[31] J. Corbet, "Transparent hugepages," 2009. [Online]. Available: https://lwn.net/Articles/359158/

[32] A. Sahraei et al., "XFaaS: Hyperscale and low cost serverless functions at meta," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 231–246.

[33] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "Sonic: Application-aware data passing for chained serverless applications," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 285–301.

[34] D. Kim et al., "{FreeFlow}: Software-based virtual {RDMA} networking for containerized clouds," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2019, pp. 113–126.

[35] "Knative is an open-source enterprise-level solution to build serverless applications." 2025. [Online]. Available: https://knative.dev/docs/

[36] "Custom resources," 2025. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[37] "vSwarm-Serverless benchmarking suite," 2021. [Online]. Available: https://github.com/vhive-serverless/vSwarm

[38] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2019, pp. 502–504.

[39] Knative, "Knative Docs: Configuring concurrency," 2025. [Online]. Available: https://knative.dev/docs/serving/autoscaling/concurrency/

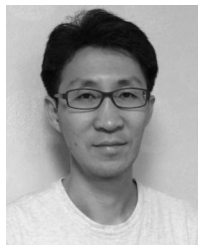
[40] G. Cloud, "Google cloud run - about concurrency," 2025. [Online]. Available: https://cloud.google.com/run/docs/about-concurrency

[41] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rFaaS: Enabling high performance serverless with RDMA and leases," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2023, pp. 897–907.

[42] F. Lu, X. Wei, Z. Huang, R. Chen, M. Wu, and H. Chen, "Serialization/deserialization-free state transfer in serverless workflows," in *Proc. 19th Eur. Conf. Comput. Syst.*, 2024, pp. 132–147.

[43] C. Tirumalasetty, C. C. Chou, N. Reddy, P. Gratz, and A. A. Wafa, "Reducing minor page fault overheads through enhanced page walker," *ACM Trans. Archit. Code Optim.*, vol. 19, pp. 1–26, 2022.
[44] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 1063–1075.
[45] L. Ao, G. Porter, and G. M. Voelker, "FaaSnap: Faas made fast using snapshot-based VMS," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 730–746.
[46] I. E. Akkus et al., "SAND: Towards high-performance serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 923–935.
[47] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs}," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 303–320.
[48] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 356–370.
[49] L. Cvetković, F. Costa, M. Djokic, M. Friedman, and A. Klimovic, "Dirigent: Lightweight serverless orchestration," 2024, *arXiv:2404.16393*.
[50] T. Yu et al., "Characterizing serverless platforms with serverlessbench," in *Proc. ACM Symp. Cloud Comput.*, 2020, pp. 30–44.
[51] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ML workflows," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 13–24.
[52] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 427–444.
[53] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in *Proc. ACM Int. Conf. Manage. Data*, 2020, pp. 115–130.
[54] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2019, pp. 193–206.
[55] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," in *Proc. Int. Conf. Very Large Data Bases*, 2019, pp. 624–638.
[56] Z. Li et al., "Dataflower: Exploiting the data-flow paradigm for serverless workflow orchestration," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 57–72.
[57] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloudd provider," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.
[58] T. L. Kernel, "HugeTLB pages," 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.5/admin-guide/mm/hugetlbpage.html
[59] S. Park, M. Kim, and H. Y. Yeom, "GCMA: Guaranteed contiguous memory allocator," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 390–401, Mar. 2019.
[60] S.-J. Kim, M. You, B. Kim, and S. Shin, "Cryonics: Trustworthy function-as-a-service using snapshot-based enclaves," in *Proc. ACM Symp. Cloud Comput.*, 2023, pp. 528–543.
[61] M. Li, Y. Xia, and H. Chen, "Confidential serverless made efficient with plug-in enclaves," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit.*, 2021, pp. 306–318.
[62] M. You, J. Nam, M. Seo, and S. Shin, "HELIOS: Hardware-assisted high-performance security extension for cloud networking," in *Proc. ACM Symp. Cloud Comput.*, 2023, pp. 486–501.
[63] M. You et al., "Hardwhale: A hardware-isolated network security enforcement system for cloud environments," in *Proc. IEEE 44th Int. Conf. Distrib. Comput. Syst.*, 2024, pp. 496–507.

Seong-Joong Kim is a senior researcher with the Attached Institute of ETRI. Before that, he interned with Samsung Electronics in the capacity of a software engineer. He is a recipient of the Samsung Electronics Scholarship. His research interests are all aspects of computer security, with a current focus on making Linux distributions more secure.



Seungwon Shin (Member, IEEE) received the BS and MS degrees in electrical and computer engineering from the KAIST, and the PhD degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University. He is an associate professor in the School of Electrical Engineering at KAIST. He is currently an executive vice president at Samsung Electronics, leading the security team in the IT & Mobile Communications Division. His research interests span the areas of Software-defined networking security, DarkWeb analysis, and Cyber threat intelligence.

Myoungsung You received the BS degree in computer science from Chungbuk National University, the MS degree in computer science from the School of Computing, KAIST, and the PhD degree in electrical engineering from KAIST. He is an assistant professor in the School of Electrical and Computer Engineering, University of Seoul. His research interests include cloud networking, programmable network data planes, network security, and privacy-preserving communication.