NEARDATA

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

NEARDATA

(grant agreement No 101092644)

Extreme Near-Data Processing Platform

D3.2 XtremeHub Reference Implementation

Due date of deliverable: 31-10-2025
Actual submission date: 31-10-2025

Start date of project: 01-01-2023 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	108
WP/Task related to this document	WP3 / T3.1, T3.2, T3.3, T3.4, T3.5
WP/Task responsible	DELL
Leader	Raúl Gracia (DELL)
Technical Manager	Daniel Barcelona (URV)
Quality Manager	Aaron Call (BSC)
Author(s)	Raúl Gracia (DELL), Alan Cueva (DELL), Hossam Elghamry (DELL), Sean Ahearne (DELL), Ger Hallissey (DELL), Albert Cañadilla (URV), Daniel Barcelona (URV), Gerard Finol (URV), André Miguel (SCO), Aaron Call (BSC)
Partner(s) Contributing	DELL, URV, BSC, SCO
Document ID	NEARDATA_D3.2_Public.pdf
Abstract	This deliverable includes the final design and reference implementation for all tasks.
Keywords	Serverless analytics, data partitioning, data streams, data connectors, confidential computing, TEEs.

History of changes

Version	Date	Author	Summary of changes
0.1	30-09-2025	Raúl Gracia-Tinedo, Alan Cueva, Hossam Elghamry	First complete draft.
0.2	17-10-2025	Raúl Gracia-Tinedo, Alan Cueva, Hossam Elghamry	Review feedback.
1.0	31-10-2025	Raúl Gracia-Tinedo, Alan Cueva, Hossam Elghamry	Final version.

Table of Contents

1 Executive summary **2**

2 Introduction **3**

3 XtremeHub Overview **4**

4 XtremeHub Compute: Burst Computing **6**
4.1 Introduction . . . 6
4.2 Motivation: in search of burstability . . . 7
4.2.1 FaaS is holding us back . . . 8
4.3 Burst Computing . . . 9
4.3.1 Worker packing and communication . . . 10
4.4 Design and implementation . . . 11
4.4.1 Life cycle overview . . . 11
4.4.2 Developing and running bursts . . . 12
4.4.3 Application example . . . 13
4.4.4 Burst platform implementation . . . 13
4.4.5 BCM implementation . . . 14
4.5 Evaluation . . . 15
4.5.1 Burst group invocation . . . 15
4.5.2 Burst inter-pack communication . . . 17
4.5.3 Burst group collectives . . . 18
4.5.4 Burst applications . . . 18
4.6 Related work . . . 21
4.7 Discussion and Conclusions . . . 22

5 XtremeHub Compute for RAG: Serverless Vector DBs **25**
5.1 Introduction . . . 25
5.1.1 Motivation . . . 25
5.1.2 Challenge: Stateless FaaS & Dynamic Data . . . 25
5.1.3 Contributions . . . 26
5.2 Background . . . 26
5.2.1 Function-as-a-Service (FaaS) . . . 26
5.2.2 Vector DBs . . . 27
5.3 Serverless Vector DBs: An Overview . . . 27
5.3.1 Architecture . . . 27
5.3.2 Vector DB Design: Serverful vs Serverless . . . 28
5.4 Trade-offs in Data Partitioning . . . 29
5.4.1 Clustering-based Data Partitioning . . . 30
5.4.2 Block-based Data Partitioning . . . 31
5.5 Experimental Methodology . . . 31
5.5.1 Prototype Implementation . . . 31
5.5.2 Setup . . . 33
5.6 Clustering vs Block-based Data Partitioning . . . 34
5.6.1 The Cost of Balanced Data Partitions . . . 34
5.6.2 The Effect of Vector Redundancy . . . 36
5.6.3 Clustering vs Blocks: Data Partitioning . . . 37
5.6.4 Clustering vs Blocks: Query Performance . . . 38
5.6.5 Clustering vs Blocks: Cost Analysis . . . 40
5.7 Milvus vs Block-based Serverless Vector DB . . . 40

5.7.1 Partitioning and Indexing Performance . . . 40
5.7.2 Query Performance . . . 41
5.7.3 Cost analysis . . . 42
5.7.4 Scalability . . . 42
5.8 Related Work . . . 43
5.9 Discussion and Conclusions . . . 44

6 XtremeHub Streams and FaaS: FaaStream 45
6.1 Introduction . . . 45
6.1.1 Challenges . . . 45
6.1.2 Contributions . . . 46
6.2 Motivation: Stream-based FaaS Pipelines . . . 46
6.2.1 Key Insights: Not All Streams are Created Equal . . . 47
6.3 FaaStream Design . . . 47
6.3.1 FaaStream Architecture and Life-cycle . . . 47
6.3.2 FaaStream Abstractions and API . . . 48
6.3.3 Serverless Pipeline Auto-Scaling . . . 49
6.3.4 Data Shuffling via Custom Event Routing . . . 50
6.3.5 Consistent Function State under Failures . . . 50
6.4 Evaluation . . . 51
6.4.1 Implementation . . . 52
6.4.2 Experimental Setup . . . 52
6.4.3 Unifying Streaming and Batch Data Access . . . 53
6.4.4 Coordinated Auto-Scaling . . . 56
6.4.5 Stream-based Data Shuffling . . . 58
6.4.6 Stateful Pipelines upon Failures . . . 60
6.5 Related Work . . . 60
6.6 Conclusions . . . 61

7 XtremeHub Security and Streams 62
7.1 Secure Streaming in Action . . . 62
7.2 Summary . . . 62

8 XtremeHub Stream Connectors: Nexus 63
8.1 Introduction . . . 63
8.1.1 Motivation: Beyond Tiered Data Streams . . . 63
8.1.2 Data Management Challenges . . . 64
8.1.3 Contributions . . . 64
8.2 Background . . . 65
8.2.1 Event Streaming Systems . . . 65
8.2.2 The Shift towards Streaming Storage . . . 65
8.3 Nexus Design . . . 65
8.3.1 Design Principles and Insights . . . 66
8.3.2 Abstractions . . . 67
8.4 Nexus Architecture . . . 67
8.4.1 System Metadata . . . 67
8.4.2 Streamlet Execution . . . 68
8.4.3 Streamlet State . . . 68
8.4.4 Mesh-like Data Routing . . . 69
8.4.5 Fault Tolerance and Correctness . . . 70
8.5 Nexus in Action . . . 70
8.5.1 Streamlet API . . . 71

8.6 Implementation . . . 71
8.7 Validation . . . 71
8.7.1 Experimental Setup . . . 72
8.7.2 Interception Performance . . . 72
8.7.3 Enhancing Event Streaming Systems . . . 74
8.8 Related Work . . . 76
8.9 Conclusions . . . 77

9 XtremeHub HPC Connectors 78
9.1 Introduction . . . 78
9.2 The Lithops-HPC framework . . . 79
9.2.1 Architecture Implementation . . . 81
9.2.2 Programming model . . . 82
9.3 Evaluation . . . 82
9.3.1 Setup . . . 82
9.3.2 Complexity . . . 82
9.3.3 Lithops-HPC Overhead . . . 84
9.3.4 Time to service . . . 84
9.3.5 High-performance . . . 86
9.3.6 Scaling . . . 87
9.3.7 Resource management . . . 88
9.3.8 Multi-cluster deployment . . . 88
9.4 Related Work . . . 88
9.5 Discussion . . . 90
9.6 Future Work . . . 91

10 Conclusions and Next Steps 92

List of Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CAS	Configuration and Attestation Service
CC	Creative Commons
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CSV	Comma-separated values
DAG	Directed Acyclic Graph
DOI	Digital Object Identifier
FASTQ	Text-based format for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores
FLOPS	Floating Point Operations Per Second
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
LTS	Long-Term Storage
MDR	Multifactor Dimensionality Reduction
MPI	Message Passing Interface
MPP	Massively Parallel Processing
RAG	Retrieval-Augmented Generation
S3	Simple Storage Service
SDP	Streaming Data Platform
TEE	Trusted Execution Environment
VCF	Variant Call Format
VM	Virtual Machine
WAL	Write-Ahead Log

1 Executive summary

This deliverable presents the reference implementation of XtremeHub, consolidating and extending the foundational components introduced in D3.1 (*i.e.*, Lithops, Pravega, Scone, and connectors). It marks a significant step forward in the convergence of compute, stream, and security capabilities within NEARDATA's data plane. D3.2 demonstrates several advanced integrations across NEARDATA components, demonstrating how XtremeHub supports dynamic, data-intensive workloads with elasticity, performance, and simplicity across heterogeneous cloud environments.

Next, we list the software contributions described in D3.2 with some KPIs[1] that give a sense on the impact of our research:

- **Burst Computing** for efficient execution of massively parallel jobs with group-aware invocation and locality-aware communication. *KPI-2*: It improves throughput by up to 3× and reduces invocation latency by 11.5×. Published in USENIX ATC'25.
- **Serverless Vector DBs**, enabling scalable AI workloads such as Retrieval-Augmented Generation (RAG) using stateless cloud functions. *KPI-5*: Simplifies deployment, reduces operational overhead, and achieves up to 98% cost reduction compared to Milvus. Published in ACM SIGMOD'26.
- **FaaStream**, a unified framework for streaming and batch serverless pipelines built on elastic and tiered data streams. *KPI-3*: FaaStream supports coordinated auto-scaling of serverless pipelines, adapting to fluctuating workloads and outperforming Kafka-based setups in throughput and latency. Submitted for publication.
- **Secure streaming**, combining SCONE and Pravega to enable confidential computing in real-time data pipelines. *KPI-4*: SCONE-protected Pravega clients show affordable overhead (2×) for low latency workloads, making it practical for latency-sensitive applications. Published in IEEE CEC@ICNP'25.
- **Nexus**, a programmable data management mesh for tiered data streams, supporting user-defined transformations, such as routing, compression, and AI inference. *KPI-1*: Achieves 3.9× better compression ratios than built-in mechanisms in Kafka and Pulsar. Submitted for publication.
- **HPC Connectors**, integrating Lithops with supercomputing environments to support high-performance workloads. *KPI-1*: Lithops-HPC achieves up to 3× better compute performance and efficient data transfer using GKFS-backed parallel storage. Submitted for publication.

Together, these contributions demonstrate how XtremeHub can support dynamic, data-intensive workloads with elasticity, performance, and simplicity across heterogeneous cloud environments. The deliverable highlights the convergence of previously proposed components into a cohesive architecture, paving the way for their exploitation in NEARDATA's use cases.

[1] For a full review of KPIs, please see Section 10. For use-case oriented KPIs, please see D5.2.

2 Introduction

The NEARDATA project is built upon the near-data processing paradigm, which aims to bring computation closer to data sources across the cloud–edge continuum. This approach enables scalable, secure, and efficient data analytics for heterogeneous infrastructures and dynamic workloads. Within this vision, XtremeHub serves as the data plane of NEARDATA, orchestrating compute, stream, and security components to support real-time and batch analytics across diverse domains.

In D3.1, we introduced the initial design of XtremeHub, composed of modular, production-ready subsystems for serverless compute (Lithops), tiered streaming (Pravega), and confidential execution (Scone), along with connectors for integrating external systems. These components laid the foundation for a programmable and extensible data plane architecture.

This deliverable, D3.2, presents the reference implementation of XtremeHub and focuses on the convergence of its core components into a unified runtime. It demonstrates how XtremeHub evolves to support complex, data-intensive workloads with enhanced elasticity, performance, and programmability. The work reported here is tightly aligned with other Work Packages, particularly:

- WP2, which defines the overall NEARDATA architecture and early prototypes;
- WP4, which develops the Data Broker and connector ecosystem;
- WP5, which validates performance and KPIs across use cases.

The key contributions in D3.2 extend and integrate XtremeHub components in the following ways:

- *XtremeHub Compute - Burst Computing*: We propose a novel execution model for massively parallel serverless jobs with group-aware invocation and locality-aware communication.
- *XtremeHub Compute for RAG - Serverless Vector DBs*: We present a novel serverless vector DB architecture enabling scalable AI workloads such as Retrieval-Augmented Generation (RAG) using stateless cloud functions.
- *XtremeHub Streams and FaaS - FaaStream*: We design a unified framework for streaming and batch serverless pipelines built on elastic and tiered data streams.
- *XtremeHub Security and Streams*: We summarize the progress in understanding the IO performance impact of confidential execution and stream-based workloads.
- *XtremeHub Stream Connectors - Nexus*: A programmable streamlet-based data management layer for tiered data streams.
- *XtremeHub HPC Connectors*: Integration with HPC platforms to support high-performance workloads in genomics and other domains.

These components are validated via solid benchmarking methodologies and through NEARDATA's use cases in healthcare demonstrating how XtremeHub supports real-world scenarios with dynamic data flows, stringent latency requirements, and secure processing needs.

The deliverable concludes with a summary of KPIs and a roadmap for further integration and exploitation of XtremeHub, paving the way for its adoption in next-generation data-centric infrastructures.

3 XtremeHub Overview

XtremeHub is the programmable data plane of NEARDATA, designed to orchestrate compute, stream, and security components across the cloud-edge continuum. It enables scalable, secure, and elastic data processing for heterogeneous workloads, ranging from real-time analytics to batch processing and AI inference. In D3.1, XtremeHub was introduced as an architecture composed of (see Fig. 1):

- Lithops-based serverless compute, enabling stateless and parallel execution of Python functions across cloud and edge backends.
- Pravega-based tiered streaming, supporting elastic ingestion and processing of time-ordered data with strong consistency guarantees.
- SCONE-based confidential computing, providing secure enclaves for sensitive workloads using Intel SGX.
- Connector ecosystem, including Nexus for programmable data management and interfaces to external systems like HPC platforms and federated learning frameworks.

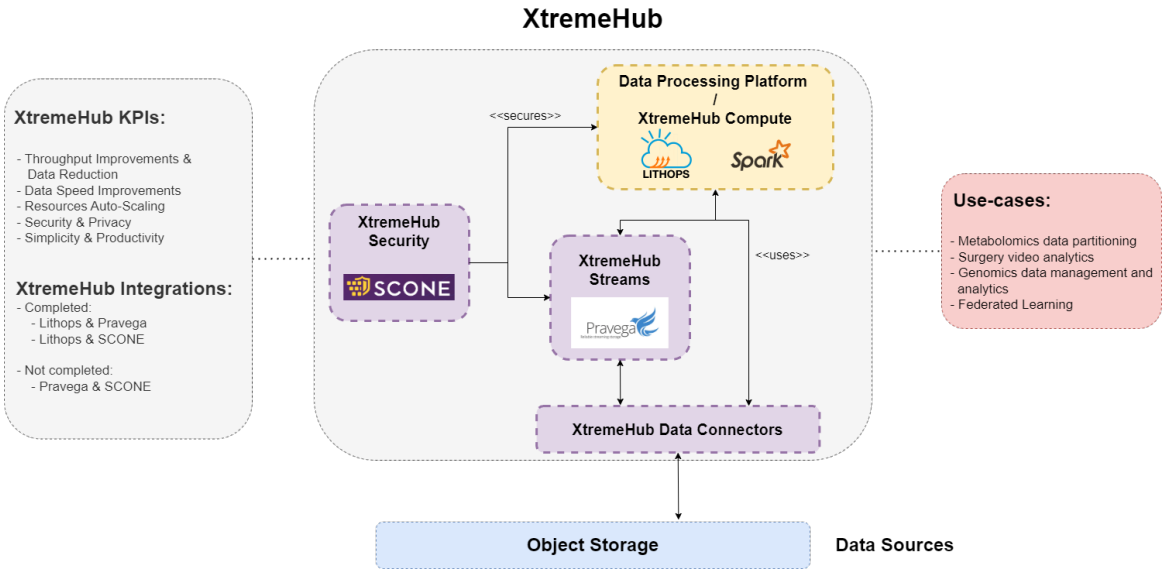


Figure 1: High-level overview of XtremeHub.

In D3.2, we show how these building blocks converge into a unified runtime that supports advanced data-centric workloads. The reference implementation demonstrates how XtremeHub evolves from a modular prototype into a cohesive system capable of powering real-world applications.

XtremeHub Compute extends the serverless paradigm with Burst Computing, a novel execution model that enables group-aware invocation and locality-optimized communication. This allows massively parallel jobs to be launched with minimal coordination overhead, improving throughput and responsiveness. Additionally, XtremeHub Compute introduces a serverless vector database architecture tailored for Retrieval-Augmented Generation (RAG) workloads, enabling scalable AI inference and search using stateless cloud functions. These innovations position XtremeHub as a powerful engine for elastic, AI-driven compute across cloud and edge environments.

XtremeHub Streams unify batch and streaming pipelines through FaaStream, a framework that leverages Pravega's tiered storage and FaaS platforms elastic compute to support dynamic workloads with varying latency and durability requirements. FaaStream demonstrates how XtremeHub can handle content-aware, low-latency processing with strong consistency and scalability.

XtremeHub Security explores the integration of confidential computing with streaming workloads, validating the use of SCONE-protected Pravega clients in real-time scenarios. The implementation shows that secure enclaves can be used effectively with acceptable performance impact. This work reinforces NEARDATA's commitment to privacy-preserving analytics, especially in sensitive domains like healthcare and federated learning.

XtremeHub Connectors evolve into a programmable data management layer through Nexus, which introduces streamlets and swarmlets for fine-grained control over data routing, transformation, and compression. This abstraction enables transparent and location-aware data operations across the cloud–edge continuum. Moreover, XtremeHub integrates with HPC platforms to support genomics and other high-performance domains, bridging cloud-native and supercomputing workflows. These connectors ensure that XtremeHub remains extensible and interoperable, ready to adapt to diverse infrastructure and application needs.

Together, these advancements position XtremeHub as a mature and versatile data plane capable of supporting NEARDATA's vision of near-data processing. The reference implementation presented in this deliverable validates the system's scalability, elasticity, and security across multiple use cases. The following sections detail each subsystem, its integration, and its validation, paving the way for future exploitation and adoption in next-generation data-centric infrastructures.

4 XtremeHub Compute: Burst Computing

4.1 Introduction

The cloud offers compute infrastructure on demand, but provisioning, adjusting, and managing these resources for large-scale data processing applications is an arduous task, especially for non-experts. Furthermore, when the load is unpredictable, dynamic, with varying volumes of data, user-driven, and sometimes interactive, finding the right scale to avoid misprovisioning [1, 2, 3] becomes very complex.

Function-as-a-Service (FaaS) has gained traction as a solution to the resource provisioning problem as it offers rapid, on-demand, no-ops scaling and a pay-as-you-go billing model at very fine granularity (MB per ms). Users do not need to set up a cluster, but the service simply accepts function invocations and fully manages the rest. Moreover, its resource burstability has set FaaS aside from traditional engines like Spark or Dask, allowing to start thousands of short-lived functions in seconds instead of minutes (see Table 1). Several research works [4, 5, 6, 7] have used FaaS for a myriad of data- and compute-intensive tasks.

This has brought a new concept in cloud computing that refers to the ability to quickly respond to sudden, parallel workloads without provisioning a cluster in advance. Authors in [5] talk about a "burstable supercomputer-on-demand" and a "burst-parallel swarm of thousands of cloud functions, all working on the same job." However, literature admits that the current FaaS model is too narrow and precluding for Massively Parallel Processing (MPP) programs [8].

Table 1: Time to provision cloud compute resources on different services and technologies.

Technology	Total vCPUs	Nodes[a]	Start-up time
EMR Spark	96	6	296 seconds
		24	431 seconds
Dataproc	96	6	95 seconds
		24	113 seconds
Dask	128	8	184 seconds
		64	253 seconds
Ray	128	8	187 seconds
		64	229 seconds
Knative (Kubernetes)	960	960	54 seconds
OpenWhisk	960	960	21 seconds
AWS Lambda (2 GB)	960	960	6 seconds
Burst Computing	960	960	1.7 seconds

[a] AWS EMR Spark and GCP Dataproc use m5 and E2-standard VM families, respectively. Dask and Ray are deployed on user-managed m6i family EC2 VMs. Knative, OpenWhisk, and Burst deployed on 20 c7i.12xlalrge VMs but start 960 functions/workers.

In NEARDATA, we highlight that the key issue of FaaS hindering burst-parallel jobs is its lack of group awareness. Indeed, FaaS users need multiple independent service calls to spawn a fleet of workers, which become strongly isolated from each other. We note that such fine-grained isolation is damaging and unnecessary for collaborative jobs, and thus propose to raise the multi-tenant boundaries to the job level.

We present burst computing, a new cloud computing model to deal with quick, sudden, massively parallel workloads, which we call bursts. To this end, we offer a group invocation primitive to handle the whole job as a unit. To the best of our knowledge, we are the first to implement this feature in a FaaS platform, clearly differing from all other research efforts that suffer the burden of

[1]The content of this section maps to tasks T3.1 and T3.2 and is related to the paper "*Burst Computing: Quick, Sudden, Massively Parrallel Processing on Serverless Resources*", published in USENIX ATC'25.

handling and orchestrating individual function invocations. A group invocation allows to optimize resource allocation, ensure worker parallelism, and perform packing: running multiple workers co-located in the same environment. In addition to speed up worker start-up latency, this enables worker locality and simultaneity, which can be exploited to improve code and data loading, and to aid powerful worker-to-worker communication patterns (e.g., broadcast, all-to-all) that seamlessly leverage shared memory channels with zero-copy mechanisms.

From the user side, a burst spawns a fleet of workers that communicate with message-passing, a simple but very powerful abstraction that creates a novel serverless substrate versatile to many applications beyond what is feasible in FaaS. This gives extensive control of the job to advanced users and allows to design compute engines or frameworks on top (e.g., DAG-based) to simplify development, manage the execution, and handle failures. Massively parallel computations are prime burst applications, especially when sudden, unpredictable, and user-driven in nature. Batch-like jobs are also good candidates when run interactively. Some examples are data processing, analytics, and machine learning workloads like exploratory model tuning, SQL, *k*-means, and large-scale sorting. Bursts may be stateless (e.g., grid search or Monte Carlo simulations) or stateful (e.g., table joins and aggregations).

We make the following contributions:

- We present burst computing, a novel cloud service model for short, sudden, massively parallel jobs (bursts). We believe that no cloud vendor or research effort has created the necessary substrate to support them.
- Burst computing evolves FaaS with a key novel group invocation primitive (a flare) that raises multi-tenant isolation from a single function invocation to the whole job. In consequence, the system launches massive process groups faster, with guaranteed parallelism, and packs workers together to exploit locality.
- We implement a burst computing platform by extending OpenWhisk, a state-of-the-art FaaS system. Our implementation includes a specialized Rust worker runtime and a burst communication middleware that seamlessly leverage worker locality with collective code/data loading and zero-copy messaging.
- Under evaluation on several burst-parallel workloads against FaaS, burst computing improves job invocation latency (up to 11.5× faster), worker simultaneity (up to 26.5× lower median absolute deviation), and group communication (up to 98% in a broadcast), for a speed-up of 13× in PageRank and 2× in TeraSort.

4.2 Motivation: in search of burstability

Many works are leveraging serverless services for massive data processing [5, 7, 4, 9, 10, 11, 12, 13] despite current (FaaS) hindrances [8, 14, 15] due to the resource burstability of this model [16, 17]. Applications benefit from quick, on-demand, no-ops resources at very fine granularity, and pay precisely for what they need, when they need it.

This has brought what we call bursts, massively parallel processing (MPP) workloads that appear suddenly and process large, variable volumes of data in a very short time (under 1 or 2 minutes). Such applications have dynamic resource needs that cannot be predicted easily, thus serverless burstability becomes essential [18, 19, 20, 21]. Consider, e.g., an interactive, scientist-driven workflow in a Jupyter Notebook, where the user dynamically explores large datasets and modifies parameters that significantly impact the workload size. Although they may resemble batch jobs, they are characterized by their sudden, sporadic occurrence, highly dynamic and unpredictable data volumes, and the expectation of low-latency execution. Representative use cases include interactive model tuning via grid search, exploratory data analysis with SQL queries and algorithms such as logistic regression, and data preparation operations like filtering and sorting.

The MilliSort and MilliQuery benchmarks [22] exemplify such short-running workloads. Additional scenarios include real-time data stream or video feed processing, where both data volume and analytical complexity may fluctuate dramatically over time.

Current data processing solutions such as Spark, Dask, Flink, or Ray fail to support bursts. A

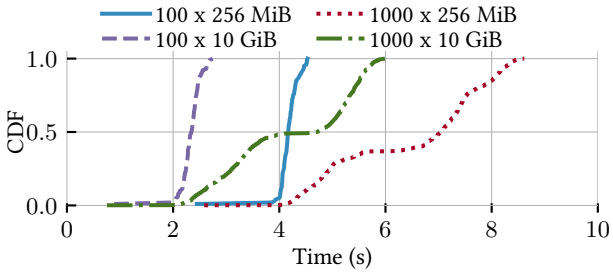


Figure 2: Start-up time (cold start) of 100 and 1000 FaaS functions in AWS Lambda for two memory sizes.

long-lived deployment of these engines is impractical, as it would easily become misprovisioned. They are not offered *as a service* by any cloud either, which could palliate the issue by multiplexing jobs from multiple tenants, and thus forces per-user deployments that are too slow to set up [17], even on cloud-managed offerings (e.g., Amazon EMR).

Table 1 shows that starting one of these technologies is intolerable for critical sporadic or dynamically sized applications.

In contrast, FaaS services provide a large-scale compute substrate much faster. Fig. 2 shows that AWS Lambda may spawn a fleet of 1000 functions in 6 seconds; a much more appropriate time range for bursts.[2] FaaS is also more attractive than Container-as-a-Service (CaaS) or managed Kubernetes services due to simpler abstractions [23] and quicker resource allocation. For example, Knative, a Kubernetes-based FaaS-like implementation, is noticeably slower in spawning workers than dedicated FaaS platforms (Table 1).

4.2.1 FaaS is holding us back

A review of the literature will show us that running bursts atop FaaS brings many challenges [14, 8, 17]. We highlight three friction points: (**F1**) worker isolation, (**F2**) job fragmentation with complex orchestration, and (**F3**) huge data movement.

To illustrate them, Fig. 3 follows the execution of a parallel job on a FaaS platform. It shows a parallel job with 6 workers. The job could be embarrassingly parallel (stateless) such as a data filtering, or require the workers to coordinate at some point (stateful) such as a table join or, more intensively due to its iterative nature, PageRank.

F1 appears because multi-tenant isolation is at the level of a function invocation. FaaS spawns function instances independently, one at a time, requiring multiple HTTP requests (A1) to obtain the 6 workers. Besides the added latency of several requests, this is an issue for parallel jobs because the platform is not aware of these workers being collaborators, and thus cannot guarantee their parallelism. This creates delays or skews between workers that potentially harm job execution. Take, for instance, Fig. 2, where the last function starts up to 6 seconds after the first one.[3] Even more, the platform populates identical environments (instances) for each invocation (A2), which stresses the system with code, dependency, and data[4] loading that creates memory duplication [24, 25].

F2 occurs when workers need to coordinate. For instance, TeraSort *à la* MapReduce includes a data shuffle amidst the job, and PageRank iteratively globally aggregates a vector. Workers cannot communicate effectively because they may not exist at the same time (**F1**). Instead, workers read and write intermediate data asynchronously through an external storage solution. This pattern (depicted in Fig. 4) creates job fragmentation (function stages) and complicates its orchestration, especially in

[2]Note that small functions (256 MB) incur higher invocation latency than large ones (10 GB). Also found on other providers (e.g., GCP), it is likely due to the overhead of scheduling finer-grained resources.

[3]Further evaluation (not shown in the plot) reveals that this dispersity may increase to 44 seconds in GCP, or 20 seconds in an OpenWhisk deployment.

[4]For instance, hyperparameter tuning uses the same data in all workers.

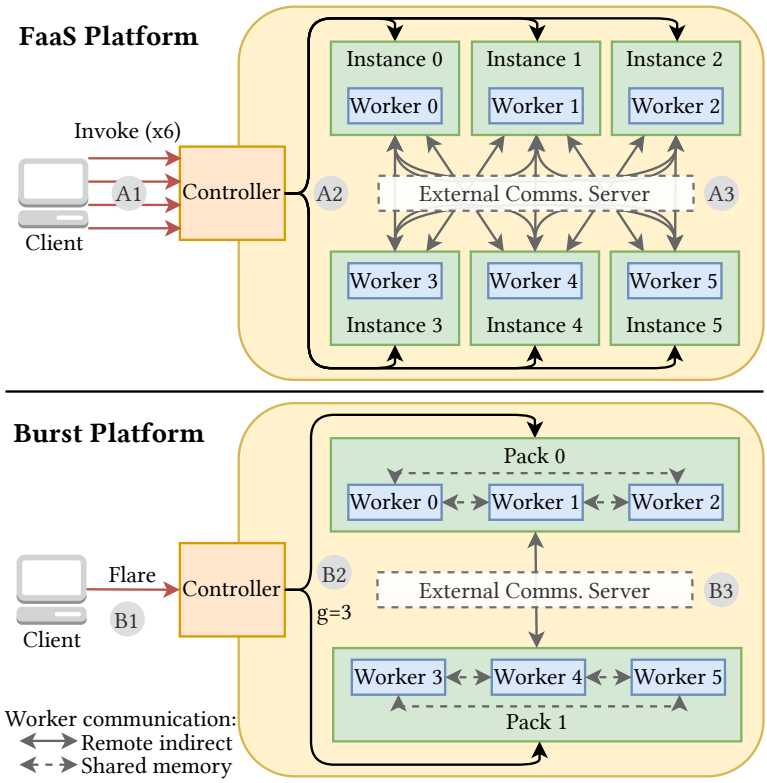


Figure 3: Running a data processing job of 6 workers in FaaS and burst computing with granularity (g) 3.

iterative algorithms like PageRank (unfeasible with this approach [26]). First, it increases data movement and requires worker recreation at each stage (adding code and data loading overhead). Second, it needs an active workflow orchestration process to monitor the state of workers and oversee the overall job progress.[5] Centralized solutions add a mostly-idle driver component while decentralized task scheduling adds a layer of complexity to applications [11, 27, 28]. Neither can solve the underlying problem of worker isolation.

These issues are emphasized by **F3**. With so many tiny isolated workers, most communication patterns (e.g., a shuffle) require numerous remote connections (A3). In data processing workloads, this may result in very large data transfers, precluded by the (FaaS) lack of direct communication [29, 30].

4.3 Burst Computing

Burst computing is a novel paradigm for running bursts in the cloud. It overcomes the above frictions with two key principles that evolve FaaS: group awareness and locality exploitation. Fig. 3 shows how this changes job invocation.

FaaS hinders worker collaboration because multi-tenant isolation is at the level of a single function (**F1**). Because a job belongs to a single tenant, it makes sense to raise isolation to the job level and handle all its workers as a group. To this end, burst computing provides a group invocation primitive, which we call flare (B1), to instantly launch massive process groups with guaranteed parallelism. To the best of our knowledge, we are the first to implement this kind of primitive in a serverless system. Flares bring group awareness to the service, which is key to perform worker packing (B2), i.e., running multiple workers of a job in the same isolated environment. Packing establishes worker locality and enables several optimizations discussed below.

[5] This can be painful since FaaS does not provide monitoring mechanisms.

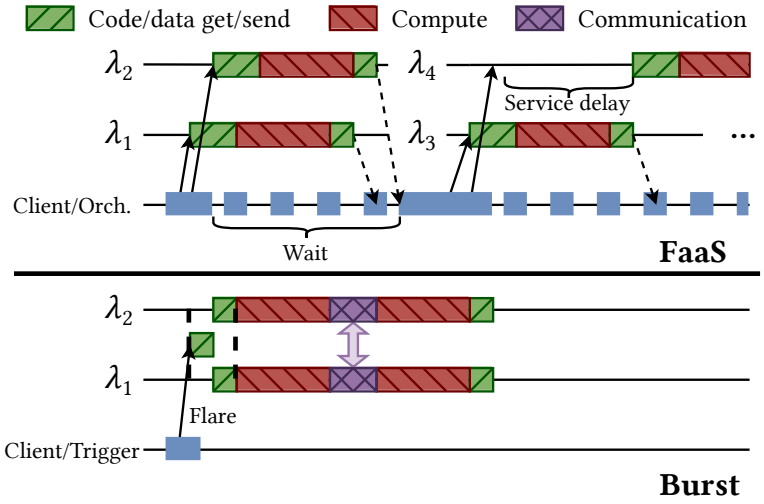


Figure 4: Timeline comparison of a parallel job with FaaS and burst computing.

In a flare, all workers have guaranteed parallelism and access to the job context (e.g., the burst size, IDs, or locality), which allows them to communicate synchronously in patterns unfeasible in FaaS, such as worker-to-worker message passing and collectives, that simplify job orchestration and avoid **F2**. This difference is depicted in Fig. 4. **F3** is addressed because communication (B3) can seamlessly exploit locality and use shared memory mechanisms between workers in the same pack, which reduces remote transfers.

4.3.1 Worker packing and communication

Worker packing To run a flare, the burst platform allocates n workers into m packs; we say that n is the burst size. The number of workers per pack is the burst's granularity ($g = n/m$). Thus, Fig. 3 shows a burst of size $n = 6$ where, by setting $g = 3$, the platform only spawns 2 packs, each with 3 workers. The higher g, the lower m, reducing the number of environment creations, which is a critical part of function invocation time in FaaS. Then, worker code and dependencies are loaded only once per pack and shared by all co-located workers. This further helps with initialization time (especially when dependencies are large) and optimizes resource usage (e.g., avoiding memory duplication [24]). A similar reasoning applies for data loading: workers processing the same data (like in hyperparameter tuning) download it just once per pack and utilize their aggregated resources to speed up the transfer (i.e., with parallel downloads).

Choosing g is a trade-off between ease of system management and locality maximization. To illustrate that, we identify three strategies[6] for worker packing: (i) *heterogeneous*, where workers are placed in containers as big as possible in the underlying system machines; (ii) *homogeneous*, where workers are placed in fixed-size containers; and (iii) *mixed*, where workers are put in fixed-size packs, but if multiple packs fall onto the same machine, they are merged into a single container. The first approach maximizes locality, but it can become a resource scheduling problem, as it is prone to fragmentation. The homogeneous packing mitigates that issue, but it restricts worker locality. The third strategy is the compromise that allows a fast and flexible management while still maximizing locality (see Section 9.3). Given this complexity, we argue that the responsibility for setting the granularity should lie with the platform rather than the user, enabling better control over resource scheduling and providing a more streamlined and user-friendly service.

Worker communication Burst applications are elastically distributed and collaborative. They are coded as a single function run by all workers that accepts any worker multiplicity transparently.

[6]Strategies must consider how many resources we assign to each worker. For simplicity, in NEARDATA we consider only vCPUs and applies 1 vCPU per worker, but the strategies work for any such assignment.

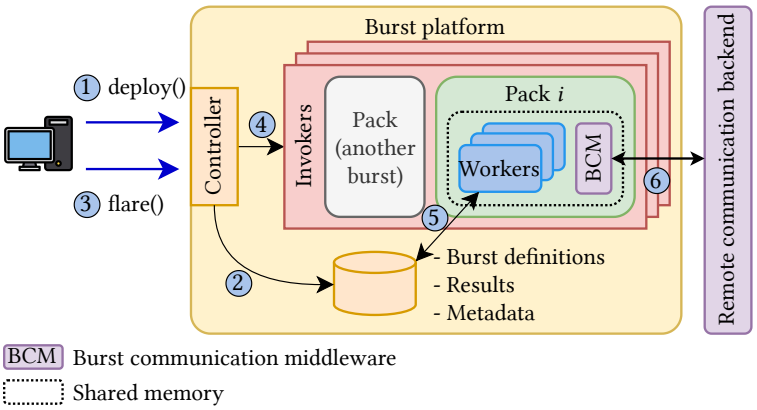


Figure 5: Burst computing platform overview.

Then, because workers are guaranteed to be parallel, they may coordinate synchronously by sending messages and with common communication patterns.

To simplify this, burst computing includes a worker-to-worker, message-passing communication middleware readily available to workers. The middleware seamlessly identifies messages between workers placed in the same pack for local communication (zero-copy). Only messages between packs are transferred remotely, and the middleware optimizes these connections (e.g., a broadcast only sends one message per pack). Remote delivery may be implemented with several technologies. Our contributions are independent of this choice because burst computing reduces any remote communication through packing. In NEARDATA, we follow the usual approach in FaaS and only consider indirect solutions using an external communication server (B3).

4.4 Design and implementation

We put the above ideas into a prototype burst computing platform and communication middleware. Here we provide the design details and implementation. Fig. 5 shows an overview of the main components and their interactions.

The burst platform extends the design of a FaaS platform to implement group invocation and worker packing. Built atop Apache OpenWhisk, our platform shares its components with important modifications (see Section 4.4.4). The controller manages user interaction with the platform, it handles inbound HTTP requests to deploy and invoke bursts, oversees system resources, and performs worker packing. A database stores the burst definitions and configuration, as well as the results and execution metadata. Computational resources in the platform are provided by the invokers, a set of machines with capacity for burst packs. Packs are run in containers that isolate a custom runtime environment to run workers.

Our burst communication middleware (BCM) has two main components: the core communication library and the remote backends. The library exposes message-based communication to workers, and it is extensible with backends to use different remote message delivery solutions.

4.4.1 Life cycle overview

Fig. 5 depicts the life cycle of the system. To deploy a new burst definition, the user first sends (1) a `deploy` HTTP request. The controller receives it and registers (2) the new definition in the database. Later, when the user desires to trigger the execution of the burst, they send (3) a `flare` HTTP request with specific parameters. The controller handles the invocation and decides worker allocation (4) based on the current state of the invoker machines. The affected invokers receive the task to spawn the required runtime environments (packs) with space for as many workers as needed. When the environments boot, their host invoker tells them which burst definition and parameters to load (5) from the database. Then, each pack spawns its workers internally, which will execute the

Table 2: Burst computing abstractions and API.

Interface	Functions
Burst Service	**deploy**(*defName*, *package*, *conf*) upload and deploy a burst definition **flare**(*defName*, [*inputParams*]) invokes a burst
Burst Function	abstract **work**(*inputParams*, *burstContext*) function to run on each worker
Burst Context	**workerID** → unique ID of this worker within the flare **burstSize** → number of total workers in the flare **packID** → unique ID of the current pack **packSize** → number of workers in the current pack **numPacks** → number of packs within the flare **belongToPack**(*workerID*) → packID returns the pack ID to which a worker belongs to **isPackLeader**() → bool returns true if this worker is its pack's leader
Comm. Primitives	**send**(*data*, *dest*) → none **recv**(*source*) → data **broadcast**(*data*, *root*) → data **allToAll**(*[data]*) → [data] **reduce**(*data*, *f*(*data*, *data*) → *data*) → data

user-defined function (`work` in Table 2) in parallel. Workers may use the BCM to coordinate and share data. This seamlessly uses shared memory or remote connections to communicate workers ⑥ in the same or a different pack, respectively. Additionally, workers may read or write data to external storage systems (e.g., object storage) or produce a result that is stored back to the database, where it may be retrieved later by users through another HTTP request.

4.4.2 Developing and running bursts

User experience is key for burst computing. As a serverless service, all resource management remains hidden. Users interact with the service through a simple interface that allows to define bursts with resource-agnostic code and to schedule their execution. This is similar to FaaS services that allow users to upload their function definitions and then set up triggers or invoke them as needed. The interface and abstractions are summarized in Table 2.

Deployment Similar to functions in FaaS, developers package and upload their burst definitions (code) to the cloud, giving them a name and configuration. The configuration includes runtime parameters and worker characteristics (such as language and memory size).

Invocation Burst definitions are triggered for execution like functions in FaaS: an event or HTTP request notifies the intent to execute a burst with specific input parameters. We call each burst invocation a burst flare (Table 2). The main difference with FaaS is that a flare will spawn a group of parallel workers (instead of a single function instance). The service ensures that all workers run simultaneously and applies packing. In our prototype, the burst size is explicit on the size of the *inputParams* array. Hence, users have direct control over it. We believe this to be important because parallelism is strictly application-specific and depends on data volume (e.g., ETL tasks), data content (e.g., dimensionality or sparsity), or algorithm configuration (e.g., the number of clusters in *k*-means). Smart burst sizing is left for future work, i.e., the platform may automatically calculate the number of workers based on application and data information.

Coding Burst definitions are coded as a single function that is run by each worker in the burst (`work` in Table 2). This function must be programmed elastically so that it accepts and runs correctly for any burst size. The code is also agnostic to the packing performed by the service.

To that end, the `work` function receives a burst context object through which each worker may obtain information about the worker distribution within the particular flare. For example, a worker can query its own unique ID, the burst size, granularity, or which workers belong to each pack (*Burst Context* in Table 2). With this information (provided by the platform invoker), the code can implement logic to apply locality optimizations at the pack and burst levels (see an example in Section 4.5.4). This context object also gives access to the BCM.

Communication interface The BCM offers simple yet powerful worker-to-worker communication through message passing similar to MPI. The abstractions are elastic (adapt to the burst size) and available through the burst context. Burst computing programs make use of two basic primitives to connect workers: send and receive. These primitives enable point-to-point communication between workers and are designed to send arbitrary volumes of data efficiently within the burst. To facilitate common communication patterns in parallel jobs, bursts may also use group collectives. As listed in Table 2, our prototype implements broadcast, all-to-all, and reduce. Primitives and collectives are locality-aware, although the programs remain agnostic to it, i.e., co-located workers (same pack) communicate on shared memory and only remote workers hit the network.

4.4.3 Application example

Fig. 6 shows an example in Rust code (simplified) of the `work` function that implements the PageRank application. The algorithm consists of an iterative process in which each worker holds a portion of the adjacency graph (relating links between web pages). In each iteration, the new global ranks are computed in parallel, aggregated, and reduced in a tree structure, then broadcasted from the root worker to the rest of them. The algorithm runs until it converges past a threshold or reaches a limit of iterations.

Similarly to the MPI computing model, all workers execute the same code but perform different logic based on the worker ID (the *rank* in MPI). The example highlights the worker accesses to the `BurstContext` object to perform collectives and obtain information about the current flare. For example, it is used to perform a collective *broadcast* to share the updated ranks vector, and later a *reduce* to aggregate the partial ranks computed among the workers. It also shows how a worker checks its ID when it needs to calculate the convergence, since this is only done by the root worker after collecting the aggregated vector in the reduce.

4.4.4 Burst platform implementation

The prototype implementation is built on top of the popular Apache OpenWhisk platform (v1.0.0). We used OpenWhisk as the basis because it is a well-known, open-source, production-tested FaaS implementation and provides higher burstability than other platforms like Knative (Table 1). Our changes amount to approximately 2 kSLOC. They affect the main components of the platform, including the controller, the invoker, and the runtime environment.

The controller now supports two new HTTP endpoints for bursts: `deploy` and `flare`. It also implements the logic to handle them (Section 4.4.1). This includes the packing strategy in the three flavors (Section 4.3): heterogeneous, homogeneous, and mixed. Granularity can be configured. In any case, the controller calculates the number and size of the packs based on the specific burst size and the resources available in the invokers.

Invokers run a new monitoring logic that can be adjusted to report their load to the controller based on CPU instead of RAM. Our prototype is set to assign 1 vCPU per worker because bursts tend to be compute-intensive jobs and we do not consider parallelism within a worker,[7] but other configurations are possible. Invokers also implement new logic to support the creation and execution of packs, spawning Docker containers of the appropriate size for each burst (by specifying resource limits) and telling each container/runtime the number of workers to run, plus their IDs and context. Containers are currently not reused across bursts.

For the runtime, we adapted the official OpenWhisk Rust environment, but it is possible to support others. The new logic allows to spawn multiple workers within it as requested by its host in-

[7] The burst size (number of workers) determines total job parallelism.

```rust
fn work(params: Input, burst: &BurstContext) -> Output {
  let num_nodes = params.num_nodes;
  let mut page_ranks = vec![1.0 / num_nodes; num_nodes];
  let mut sum = vec![0.0; num_nodes];
  let adjacency_matrix = get_adjacency_matrix(&params);
  while err < ERROR_THRESHOLD {
    page_ranks = burst.broadcast(page_ranks, ROOT_WORKER);
    for (node, links) in graph {
      for link in links {
        sum[*link] += page_ranks[*node] / out_links(*node);
      }
    }
    let reduced_ranks = burst.reduce(sum, |vec1, vec2| {
      vec1.zip(vec2).map(|(a, b)| a + b).collect()
    });
    if burst.worker_id == ROOT_WORKER {
      err = calculate_error(&page_ranks, &reduced_ranks);
      page_ranks = reduced_ranks;
    }
    err = burst.broadcast(err, ROOT_WORKER);
    reset_sums(&mut sum);
  }
  Output { page_ranks }
}
```

Figure 6: Simplified source code of the PageRank *work* function for burst computing. The accesses to the burst context to obtain the worker ID or communicate are highlighted.

voker. In particular, the Rust runtime spawns one thread per worker to provide parallelism. Finally, the runtime also includes our BCM built-in.

4.4.5 BCM implementation

The burst communication middleware (BCM) is coded in Rust in about 5 kSLOC.

It is readily available for our custom Rust runtime and we are working on a binding for Python.[8] It enables the transmission of intra-pack (zero-copy) and inter-pack (via remote backend) messages.

The BCM is instantiated by the runtime (once per pack) and made available to workers as a parameter (in the `work` function as shown in Table 2).

For local communication, BCM uses in-memory queues to send and receive data between workers in the same pack. In the Rust runtime, workers are threads and reside in the same memory space, so shared memory mechanisms are not necessary (e.g., `shm_open` or `mmap`). Instead, workers just pass memory pointers between them. Thanks to Rust's memory safety guarantees, access to shared data is thread-safe. Rust also provides a reference-counting mechanism for immutable data, so shared data is released when it is no longer used at runtime. For example, the root worker in a *broadcast* sends a read-only memory pointer to its local workers, and they safely access the message concurrently. To modify the data, one may use mechanisms such as copy-on-write.

For remote communication, each pack has a shared connection pool to the remote backend, which allows each worker within the pack to send and receive messages concurrently, with the goal of maximizing the container's bandwidth. This is especially useful in primitives like *all-to-all*, where all workers must open channels with all the others. For large messages, the data is split into smaller chunks that are sent and received concurrently. This maximizes network utilization and allows readers to start receiving data from the first chunk, instead of waiting for the full message to be available at the backend.

The BCM is extensible, allowing the implementation of more remote backends. Currently, we support Redis, DragonflyDB, RabbitMQ, and S3. The backend interface differentiates between sending direct messages (one-to-one) and broadcast messages (one-to-many). The reason is that direct messages are read only once, while broadcast messages multiple times, so we want to optimize this

[8] Other languages may be supported through bindings (Java, C++, Go...).

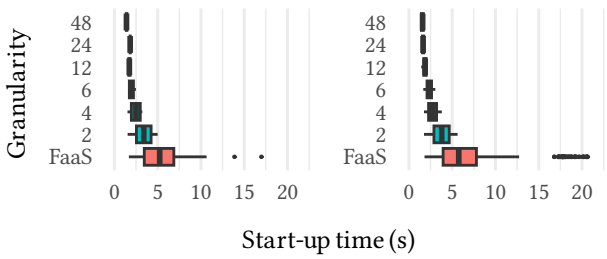


Figure 7: Worker start-up latency distribution within one job for burst computing with different packing granularity and FaaS (equivalent to $g = 1$). Left and right show, respectively, burst sizes of 48 and 960.

particular case. For instance, in RabbitMQ, one-to-one messages use direct brokers, while one-to-many use fan-out brokers.

To ensure that no messages are lost (*at-least-once* delivery semantics), the BCM relies first on the backend delivery guarantees (e.g., RabbitMQ uses durable queues to avoid dropping messages). Additionally, the BCM keeps a count of direct messages sent between each pair of workers, and for each collective operation. The middleware handles duplicate and/or out-of-order messages. For that, messages include a header with the source and destination worker, collective type, counter, and, if chunked, the number of chunks and chunk number. Messages with a counter lower than the expected value are ignored and assumed as already processed. Those with a counter greater than expected are cached locally until needed.

For chunked messages received out-of-order, a memory region is reserved for the total payload and chunks are written to their respective offset as they come in.

4.5 Evaluation

Our evaluation aims to assess burst computing against current FaaS on the three friction points described in Section 4.2.1. Importantly, we show and analyze the effects of worker packing and locality.

All experiments run on Amazon Web Services (AWS) in the `us-east-1` region.

4.5.1 Burst group invocation

Group invocation is the key element against friction **F1**. Here we evaluate how job-level isolation improves worker readiness time (invocation latency), ensures their simultaneity, and provides locality for collaborative code and data loading.

Setup: The burst platform runs on an Amazon EKS cluster, with the control plane on a `t4i.xlarge` VM (4 vCPUs and 16 GB RAM), and the invokers on up to 20 `c7i.12xlarge` VMs (48 vCPUs and 96 GB RAM). This gives us space to accommodate up to 960 workers with 1 vCPU each.

Impact on burst invocation latency First, we use the homogeneous packing policy to evaluate how assigning different granularity (g) affects burst invocation latency. The exploration is depicted in Fig. 7 for two bursts of sizes 48 (left) and 960 (right).[9] It is quickly apparent that as g increases (up to 48 in both cases), the start-up time decreases, and generally becomes more consistent across workers for all burst sizes. For instance, the latency of having all workers ready in a burst of size 960 reduces by 11.5× from $g = 1$ (FaaS) to $g = 48$. We found that container creation dominates invocation latency, hence higher g performs best. This proves that creating the biggest possible containers, and thus the less amount of them (heterogeneous packing), achieves the best start-up latency, since it creates a single container per invoker per flare. By extension, the mixed packing strategy exhibits the same results, but allows the system to manage resources more effectively in small portions to facilitate allocation and avoid resource fragmentation. To assess the impact of granularity, the rest of the evaluation uses homogeneous packing.

[9] We conducted experiments with similar results for burst sizes in-between.

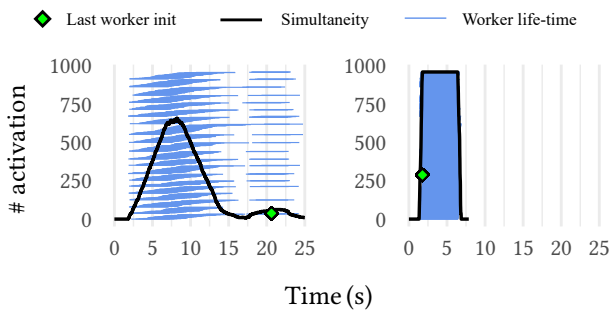


Figure 8: Simultaneity (number of workers running at an instant) in FaaS (left) and Burst with $g = 48$ (right). Each bar represents the life-time of a worker.

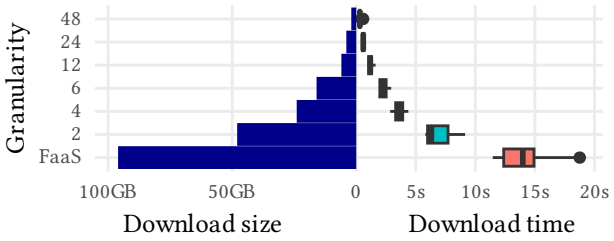


Figure 9: A burst of 96 workers loading the same 1 GB object from S3 with different granularity.

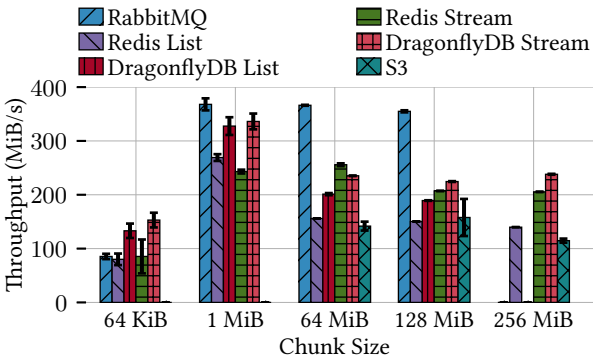
Impact on worker simultaneity We run a burst with size 960 on FaaS against burst computing with $g = 48$. For demonstration purposes, each worker performs a 5-second sleep and we plot their execution timeline in Fig. 8. The plot shows that burst computing achieves faster resource allocation and quicker readiness of workers. This ensures worker parallelism. Analyzing dispersity of worker start-up time (also in Fig. 7), the FaaS execution evinces a range of 18.8 seconds between the start of the first worker and that of the last one, with a median absolute deviation (MAD) of 2.65 seconds. In contrast, the range with $g = 48$ is just 0.44 seconds (MAD is 0.1 seconds). Compared, the range is 43× lower in burst computing, with MAD showing 26.5× lower dispersity than FaaS. Dispersity in worker start-up latency precludes FaaS to achieve full parallelism (all workers running simultaneously from start to finish), while burst guarantees it.

Impact on data loading Burst computing mitigates the FaaS problem of loading the same data on all functions (Section 4.2.1), e.g., in a grid search.

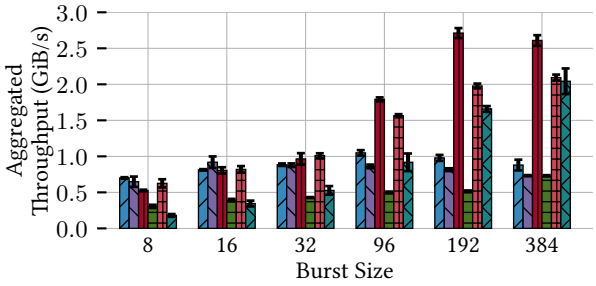
We can leverage worker access to locality information to optimize this problem and download the data only once per pack, trivially reducing data ingestion. Specifically, each worker in a pack retrieves a part of the data based on calculations from pack information in the burst context (Table 2). Then they recreate the full data in a local shared memory region. This allows to parallelize the download and complete the process faster than choosing a leader to perform it (also possible through the `isPackLeader` function in the context, which returns true for the worker with lowest ID within its pack).

We evaluate this approach on multiple g and present it in Fig. 9. Burst optimizations achieve a download time speed-up of 32.6× with $g = 48$ compared to FaaS.

Takeaway Flares eliminate friction **F1** through faster worker group initialization (11.5×) and ensured simultaneity (43× less dispersed workers) that enables locality with packing. In turn, locality may accelerate data download in applications (32.6×), tackling friction **F3**.



(a) Throughput between two remote workers sending a 1 GB payload chunked in different sizes.



(b) Aggregate throughput of two remote packs, A and B, of varying size (g =burst size/2), where each worker from pack A sends a 256 MB payload to another worker from remote pack B.

Figure 10: Throughput experiments for the different BCM backends. Median values with standard deviation (10 runs).

4.5.2 Burst inter-pack communication

Before we evaluate the effects of the BCM on frictions **F2** and **F3**, we want to ensure that an indirect communication model is feasible and to find a backend that sustains the load of bursts at scale. For this, we measure the throughput of several indirect communication backends. Specifically, we test Redis, DragonflyDB (a Redis-compatible multi-threaded alternative), RabbitMQ, and S3. Redis and DragonflyDB evaluate two flavors: using lists or streams.

Message chunk size The BCM chunks messages into several blocks to optimize network utilization and allow parallel read/write. The optimal chunk size is a trade-off between latency to first byte and operation overhead, and it varies for each communication backend. To find the optimal configuration, we measure the throughput of sending a 1 GB message between two remote workers. The workers run on two `c7i.large` machines (4 vCPUs, 8 GB) and we deploy a `c7i.16xlarge` (64 vCPUs, 128 GB) for the intermediate server. Fig. 10a plots the results. RabbitMQ offers a constant throughput for larger chunk sizes, but does not allow payloads larger than 128 MB due to AMQP protocol limitations. Redis and DragonflyDB work best at 1 MB, the latter being slightly superior. S3 offers the lowest throughput because object stores are not designed for small files (1 MB or less exceeds the allowed service request rate limits).

Maximum throughput To understand how the different backends scale under parallel load, we measure the aggregated throughput between several pairs of workers communicating simultaneously. In this experiment, we launch a group of workers (burst size from 8 to 384) split into two remote groups. Each worker in a group A sends a fixed message (256 MB) to a worker in the other, remote group B. As the burst size increases, so does the total data volume sent. Each backend uses the optimal chunk size assessed in the micro-benchmark above. Workers run on two VMs scaled to the burst size (from `c7i.xlarge` for 8 workers to `c7i.48xlarge` for 384), and the communication

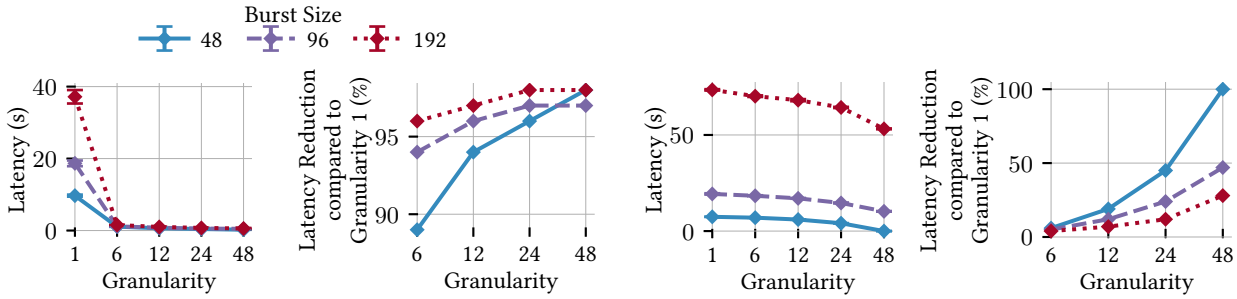


Figure 11: Latency and its reduction percentage with respect to $g = 1$ of two collectives, for varying g and burst size.

server runs on one `c7i.48xlarge` instance. The results are shown in Fig. 10b. We observe that RabbitMQ does not scale beyond 1. For the in-memory stores, the approach with lists performs better than streams. Like RabbitMQ, Redis does not scale with parallelism because it is single-threaded. In contrast, DragonflyDB does scale and achieves the highest throughput, surpassing 2.5 for large burst sizes. S3 also scales with parallelism but remains slower.

Takeaway The BCM achieves high throughput even with indirect communication, and some backends sustain up to the evaluated 384 workers with individual connections, suggesting a feasible approach. In view of the results, the rest of the evaluation uses DragonflyDB List with 1 MB chunks.

4.5.3 Burst group collectives

We assess the impact of locality on group collectives as a means to face friction **F3**. We measure end-to-end latency, i.e., the total time it takes for all workers to complete the collective, as we vary g. We present results for *broadcast* and *all-to-all*. *Reduce* behaves similar to *broadcast* because they follow the same data movement pattern.[10]

Setup: We employ one, two and four `c7i.12xlarge` VMs (48 vCPUs, 96 GB) for bursts of sizes 48, 96 and 192, respectively. g varies from 1 to 48. Each worker uses 256 MB of data for each collective call. The backend server runs on one `c7i.48xlarge` (192 vCPUs, 384 GB).

Overall, Fig. 11 shows that latency decreases as the granularity increases. This is because remote communication is the main bottleneck of collective operations and its volume decreases as g increases and more data movement becomes local.

The cost of local communication is insignificant compared to the remote one. Broadcast sends the message once, but reads it once per pack, i.e., remote data movement is directly proportional to the number of packs: if we halve the packs, we move half the data. Thus, latency quickly decreases as we increase g; near 98% reduction with $g = 48$ (Fig. 11a). All-to-all is more intensive in data traffic because all workers have a message of 256 MB for each of the other workers (48 GB total with 192 workers). This means that even if we only have two packs, half the data traffic is remote. This is clearly evident in Fig. 11b. Considering $g = 48$, burst sizes 48, 96, and 192 create one, two, and four packs; thus latency reduction is ca. 100%, 50%, and 25%, respectively.

Takeaway Locality-aware group collectives heavily mitigate friction **F3** by seamlessly reducing remote data traffic.

4.5.4 Burst applications

We evaluate three real-world bursts: hyperparameter tuning, PageRank, and TeraSort. These (or similar) applications are commonly used in the literature to assess the performance and show the

[10]Other collectives like *gather* and *scatter* are similar to *all-to-all*.

Table 3: Time to start 96 workers and gather input data in hyperparameter tuning for different granularity.

Granularity	1 (FaaS)	6	12	24	48	96
Ready time (s)	17.51	5.65	3.64	3.18	2.96	2.57

limitations of FaaS platforms for parallel jobs and serverless data analytics [12, 31, 5, 7, 32].[11] These applications clearly show all friction points while providing an overall view of the effects of burst computing compared to FaaS-based implementations. Further, they are representatives of short jobs a scientist may want to run interactively in a dynamic analysis session: sudden and quick (under 2 min).

Our baseline for comparison is thus the current FaaS paradigm available in public clouds as used in the serverless data analytics literature. Since MPI-like communication is not supported on any serverless service, we opt to use unmodified OpenWhisk and AWS Lambda as our baselines, employing external storage for communication and data sharing to align with the state of practice.

Hyperparameter tuning Grid search is a machine learning technique in which a set of hyperparameter values are evaluated to find the combination that yields the best performance for a given model. This evaluation is done in parallel, with each worker processing a full copy of the training dataset. Since there are no dependencies between parallel tasks, FaaS seems suitable for this job. However, each function would download a copy of the data, regardless of whether there are functions co-located on the same node. This results in a waste of bandwidth and memory due to duplicate data downloads. Burst computing provides an optimization opportunity by exploiting locality (see impact on data loading in Section 4.5.1).

Setup: The grid search is applied to a stochastic gradient descent model in a `sklearn` Python application and distributed to 96 workers. We use a 500 MB Amazon reviews dataset (CSV), available at Kaggle[12] and stored in an S3 bucket. AWS Lambda is the baseline (denoted as $g = 1$), with a memory configuration of 1769 MB, which provides a full vCPU. The burst platform uses a `c7i.24xlarge` instance.

Table 3 collects the "ready time", meaning the time elapsed from client-side job invocation until the input data is available to all workers and they are ready to compute. We see how burst computing quickly reduces this time as g increases. This effect has two causes. First, the group invocation primitive speeds up invocation time compared to FaaS, from approx. 4 to 1.5 seconds with $g = 96$. Second, data download can be optimized as assessed in Fig. 9. While FaaS has to download a copy of the data on each worker, workers co-located in a pack collaborate in downloading the input in parallel. Hence, as g increases, the input download time decreases, going from 14 seconds in FaaS to 1 seconds with $g > 48$.

PageRank PageRank is a well-known data analytics workload with intensive worker coordination. It involves iterative and heavy data aggregation for large datasets, which is of interest for benchmarking worker communication in burst computing. We adapted PageRank for burst computing from the Hi-Bench suite [34] (MapReduce approach). The implementation is detailed in Section 4.4.3. In this case it is not possible to make optimizations regarding data ingestion (like in Section 4.5.4), since each worker takes a different partition of the dataset. We skip reporting the MapReduce version atop FaaS because the number of (short) stages necessary to perform the iterative aggregations make it obviously slower. Spark has a similar problem [12]; evaluation on AWS EMR with an equal-sized deployment (needing 5 to start up) shows that this application takes over an hour.

Setup: This experiment uses four `c7i.16xlarge` VMs (64 vCPU, 128 GB). The graph dataset is generated with Hi-Bench consisting of 50 million nodes (ca. 30 GB) in 256 partitions. The algorithm

[11] We do not use FaaS benchmarks (e.g., FunctionBench [33]) because they focus on applications where FaaS already does a good job (independent function invocations and stateless or embarrassingly parallel workloads).

[12] `https://www.kaggle.com/bittlingmayer/amazonreviews`

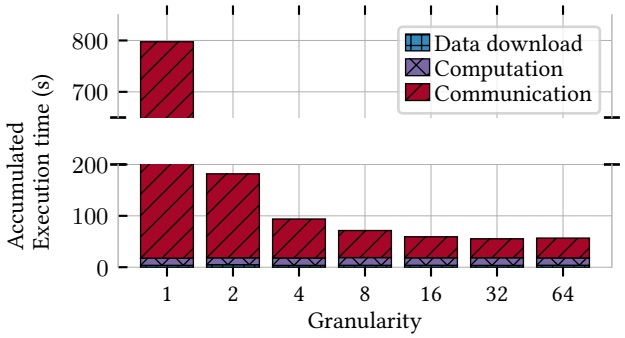


Figure 12: PageRank execution time by phase, varying g.

Table 4: Aggregated network traffic volume and percentage of traffic reduction compared to $g = 1$, varying g in PageRank.

Granularity	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Traffic (GiB)	3068	1532	764	380	188	92	44
% Reduction	n/a	50.0	75.0	87.6	93.8	97.0	98.5

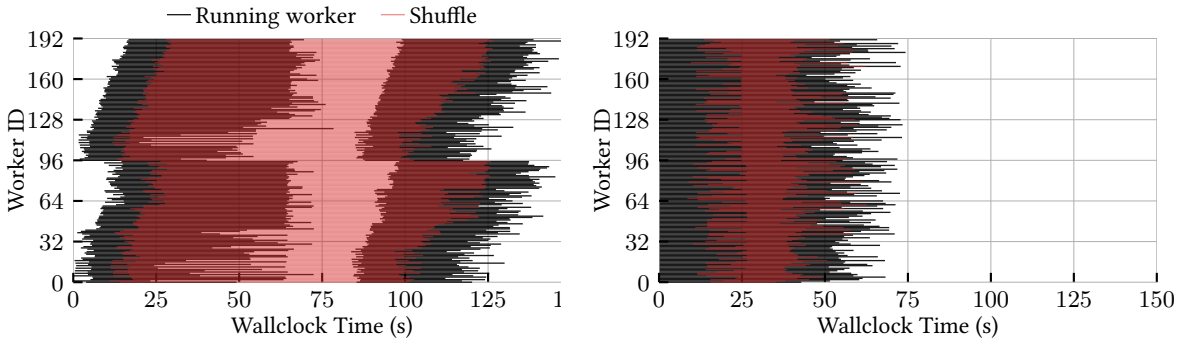
runs over 10 iterations, with a burst size of 256, varying the granularity between 1 and 64.

Fig. 12 shows the total execution time of all iterations split into phases: the time dedicated to download input data (from S3), compute the ranking, and communicate (collectives) between workers. Times for each phase are averaged across workers, and all iterations added. Table 4 shows the network traffic and % of reduction compared to $g = 1$. Communication accounts for the majority of the execution time because the rank vector must be aggregated and shared at each iteration. Our configuration uses a vector of 40 MB that is sent, received, and aggregated in a tree pattern across the workers, and then broadcast from the root worker to the rest of them. As g increases, the remote portion of this movement decreases. For example, with $g = 2$, only the first level of the (binary) reduction tree is local (the leaves), and the rest communicate remotely. With $g = 64$, there are 4 packs, so remote communication occurs only in the last 2 tree levels. With this setup, we achieve a 98.5% reduction in data traffic and a 13× speed-up compared to $g = 1$.

TeraSort We have implemented TeraSort based on the MapReduce version in Hi-Bench suite [34]. TeraSort is of particular interest because it involves a heavy data shuffle phase. We want to compare a TeraSort following the serverless MapReduce approach [13, 35, 36], with a single-stage burst computing version where we exploit locality for the shuffle phase. The main advantages of burst are: (i) the MapReduce version requires two rounds of function invocations (map and reduce), while the burst model requires a single flare, and (ii) the MapReduce version shuffles data through object storage, while burst employs the (locality-aware) *all-to-all* collective.

Setup: This experiment sorts a 100 GB dataset generated with Hi-Bench with 192 partitions. The burst platform runs on EKS with two `m7i.24xlarge` (96 vCPUs, 384 GB) invokers and a `c7i.xlarge` controller. The input data is in an Amazon S3 bucket located in the same region. For reference, an equal-sized Spark deployment solves this problem in 106 seconds average but needs 5 to start up the cluster.

Fig. 13 shows the timeline of two executions comparing serverless MapReduce and burst. The execution time of each worker is shown in horizontal black bars, stacked by *Worker ID* on the vertical axis. Superimposed in red, we see the time elapsed for the shuffle phase of the TeraSort algorithm. In the MapReduce version (Fig. 13a), we highlight (i) the dispersity in function start-up time, as we have seen in Fig. 8; (ii) a gap where no functions are running, caused by splitting the workload into two phases (map and reduce), with an externally-managed synchronization phase between them, adding further overhead; and (iii) an outlier in the map phase (worker #121), which slows down the



(a) Serverless MapReduce (b) Burst computing

Figure 13: TeraSort timeline comparison between (13a) serverless MapReduce and (13b) burst computing. MapReduce comprises two function rounds (map and reduce), with data exchange via object storage. Burst uses a single flare, exchanging data through the *all-to-all* collective.

entire workflow.

All these points are addressed in burst computing (Fig. 13b). First, the group invocation packs all workers into two containers of 96 workers, making start-up faster, and ensuring parallelism, which eliminates (latency-induced) outliers. Second, worker-to-worker collectives avoid splitting the job into two phases. Finally, remote communication is reduced thanks to locality (see Section 4.5.3). To wit, we achieve a 2× speed-up for this particular execution—1.91× mean across six executions.

For completeness, we also run TeraSort on Spark using AWS EMR on a cluster of equivalent resources. Best results are achieved with 74 executors of 4 GB of memory each. The total run time ranges between 100 and 110 seconds, placing it between the FaaS and Burst implementations. Although the execution times are relatively close, there are substantial differences that make a comparison and analysis of results complex. To start, the programming language changes (Scala to Rust), and Spark's execution model and its peer-to-peer communication capability impact application performance in different ways: Rust may be more performant, the burst execution model simpler and more effective, but Spark's direct communication is faster. Moreover, it is worth noting that cluster creation time on AWS EMR took over 5 minutes, adding additional overhead for an sporadic workload. Neither FaaS nor Burst require the user to set up a cluster.

Takeaway Frictions **F1**, **F2**, and **F3** appear in real-world applications. Hyperparameter tuning shows duplication in worker initialization due to friction **F1**, which also slows down worker start-up in PageRank and TeraSort. PageRank and TeraSort evidence the issues of friction **F2**: the iterative nature of PageRank makes it unfeasible in FaaS due to excessive stages, and TeraSort is hindered considerably due to slower coordination. Burst computing mitigates this by allowing workers to coordinate and share data in a single stage, instead of requiring multiple stages orchestrated externally that communicate asynchronously through storage. PageRank and TeraSort also emphasize friction **F3**: Page-Rank with iterative, large communication to aggregate the vector and TeraSort with a single, large data shuffle.

4.6 Related work

The concept of resource burstability has already been discussed in the serverless literature. FaaS has been referred to as a "burstable supercomputer-on-demand" [5] for "burst-parallel serverless applications" [37], albeit with limitations. "Flash bursts" in HPC [22] explore the feasibility of 1 jobs spanning a large number of servers, which is attractive but unfeasible with existing technologies in public clouds. Granular computing [38, 39, 40] explores very similar ideas to improve cluster utilization. [17] pursue serverless burstability for batch jobs, highlighting that functions are too limited for the job.

Burst computing goes a step further in defining a new way of running burst-parallel jobs in the cloud. It evolves from FaaS to exploit serverless burstability, but it unlocks the limitations of working with functions to provide a compute environment tailored for massively parallel collaborative jobs. Recent works also promote the need to evolve the FaaS model to overcome important challenges [41, 42]. To our knowledge, we are the first ones to raise the unit of management from a function to the job level in a serverless service with a group invocation abstraction to pack workers together and enable locality within the job.

Several papers [43, 44, 45] tackle handling function invocations faster. They aim to respond to thousands of function calls with very low latency. Others relax function isolation to handle multiple invocations together and cut on start-up and data sharing overhead [46, 19]. Similarly, some works perform opportunistic function packing, placing multiple function invocations from the same user on the same container to improve invocation latency and resource usage [25, 47, 48, 49, 50]. This shows that burst jobs are becoming widely accepted in FaaS settings, and locality is a key enabler. All these works only consider individual function invocations and are thus limited. Burst computing further optimizes job execution thanks to group invocations and guaranteed worker parallelism. Still, the mentioned works may be combined with burst computing to accelerate resource allocation.

A different line of work handles complex computations atop FaaS through higher-level function orchestration, task schedulers, or workflow optimizers [51, 11, 52, 27, 28, 53, 54, 55]. Since they operate with individual function invocations, they cannot be compared with burst computing, but benefit from it to guarantee task parallelism and fast start-up time. Orchestration techniques may be applied atop burst computing, e.g., to coordinate multiple bursts, manage data dependencies between them, optimize task parallelism, and handle faults.

Other works explore a hybrid approach where traditional compute engines (e.g., Spark or Flink) offload computation spikes to serverless functions [56, 57, 58] to achieve faster adaptation to dynamic loads. This requires the deployment and management of the traditional compute engine, which is not ideal to achieve serverless execution of parallel jobs. Burst computing fully hides infrastructure: users only deploy their code and run jobs, while the cloud handles the rest.

Communication and state sharing in FaaS has been widely explored. Some works combine a data store and a FaaS platform into the same system to optimize data access by placing functions where the data they use is kept [18]. This is a fundamentally different kind of locality than in burst computing as it does not support any function grouping semantics and may present resource contention problems [59]. Other solutions [32, 13, 26, 31] target applications that require function coordination and communication, close to the objectives of burst computing. However, they employ a disaggregated storage solution to relay data between stateless functions and cannot exploit any form of locality.

Boxer [60] explores direct communication, and FMI [30] builds a library of collectives between groups of FaaS functions (with NAT traversal). They only tackle current FaaS platforms and do not provide any locality optimizations (in contrast to burst computing). These solutions are orthogonal to our contributions. For instance, FMI may be used as a BCM remote backend to accelerate pack-to-pack transfers, but burst computing still contributes zero-copy communication between workers in the same pack.

Finally, we highlight a clear trend in FaaS that is aligned with burst computing: the emergence of "Big Lambdas" with multiple CPU cores (currently up to 6 vCPUs in AWS). Jobs may leverage function intra-parallelism and, similar to burst computing, worker locality becomes relevant and improves the execution of parallel applications. these high-capacity functions still lack group-aware mechanisms and parallel guarantees, requiring complex user-side handling.

4.7 Discussion and Conclusions

Burst as a cloud service One potential concern regarding the adoption of burst computing as a public cloud service is the added scheduling complexity of packing multiple workers, along with its implications for billing. Giving burst granularity control to the provider addresses this by enabling more effective resource management. The mixed packing policy we propose further enhances scheduling efficiency by targeting small resource gaps and maximizing locality through co-located

resource consolidation. Burst computing aligns with the trend toward "Big Lambda" architectures, seen in platforms such as AWS Lambda, Google Cloud Functions, and CaaS offerings like IBM Code Engine. For example, AWS already supports functions with up to 6 vCPUs, indicating that its infrastructure can already schedule resource bundles at this scale. Setting a burst granularity of 6 would thus mirror current AWS capabilities. Implementing burst computing requires the concurrent allocation of multiple such high-capacity functions, similar to existing multi-invocation FaaS handling, while optimizing locality. Though this may introduce slightly higher latency than a single function invocation, it remains more efficient than coordinating numerous remote HTTP calls. Our evaluation on OpenWhisk confirms the viability of this model, demonstrating that it preserves internal resource management and auto-scaling typical of FaaS. We see burst computing as a natural evolution of FaaS: despite the constraint of parallel provisioning, bursts are ephemeral and may be bounded in time or concurrency, clearly distinct from long-running serverful processes. Thus, we argue that burst support in public clouds is not prohibitively complex and can follow FaaS-like billing models. Moreover, there is clear evidence that users are already willing to pay for "Big Lambdas" or containers when the workloads justify it [14].

Burst applications The suitability of burst computing versus traditional FaaS depends on an application's communication patterns, performance demands, and tolerance to execution imbalances. Burst computing is designed for massively parallel workloads with collaborating workers, enabling co-location and synchronized execution. Applications with strong locality requirements and intensive data sharing—such as PageRank, SQL queries, or large-scale sorting—are prime candidates. However, even loosely coupled workloads like grid search or Monte Carlo simulations can benefit from shared data downloads (see Section 4.5.4), faster initialization, and coordinated result aggregation. Burst computing also facilitates optimizations such as function fusion, where stateless functions within a dataflow are merged into a single, stateful burst to reduce communication overhead and improve locality [61]. Interactive workloads or those requiring synchronized responses across workers similarly gain from burst simultaneity. In contrast, those with minimal data sharing or high variance in execution time across workers are typically better served by conventional FaaS. While our current prototype reclaims resources per pack—leading to potential idle resources when worker durations differ—it targets short-lived, coordinated workloads (i.e., workers typically progress in lockstep) where imbalance is rare, as seen in PageRank and MilliSort/MilliQuery [22]. Addressing this limitation via dynamically sized packs is a path for future work. Ultimately, choosing between burst and FaaS depends on locality needs, parallelism requirements, and the importance of coordinated execution.

Coding limitations While burst computing introduces new capabilities for expressing parallel and stateful workloads, it also imposes certain programming constraints that must be acknowledged. The model is designed as a low-level primitive, offering developers greater flexibility rather than prescribing orchestration logic. Similar to MPI, it executes the same code across all workers while assigning each a unique identity, enabling divergence in behavior. However, unlike MPI, bursts are triggered with FaaS-like simplicity, abstracting away explicit resource management. The primary coding complexity introduced lies in inter-worker communication, yet this is mitigated by using programming primitives that are agnostic to worker count and locality, thereby simplifying development. Although multi-burst orchestration—e.g., for full DAG-based workflows—remains future work, it can leverage well-established techniques such as dependency persistence in object storage or worker reuse across flares. A higher-level framework could encapsulate this coordination, offering abstractions for DAG scheduling and state propagation across bursts [62, 11, 51]. Many real-world applications (e.g., clustering, gradient descent, aggregation queries, N-body simulations) can already be implemented within a single burst, avoiding orchestration overhead altogether. Furthermore, libraries could expose common distributed patterns (e.g., a distributed sort) via simple APIs, hiding burst-level details from end users. Thus, while the model lacks full orchestration capabilities out of the box, it provides foundational support for powerful abstractions and ensures parallel execution—a property often missing in existing FaaS-based solutions, where developers must manually coordinate

parallelism without system-level guarantees [11, 15, 5].

Serverless clusters Burst computing poses a step towards redefining the boundary between serverless architectures and traditional cluster-based systems [16, 17]. Recent work has explored how to emulate cluster-like capabilities atop existing FaaS platforms [11, 63]. However, our approach takes a fundamentally different perspective: rather than adapting conventional cluster technologies to work around the limitations of current FaaS substrates, burst computing seeks to evolve serverless services themselves, introducing native abstractions and execution models that offer the parallelism, coordination, and locality benefits traditionally associated with clusters—yet within a fully serverless paradigm. This opens the door to realizing truly serverless versions of platforms like Spark, Dask, or Flink, moving beyond the managed, but still serverful, offerings available today.

Conclusions In NEARDATA, we have presented burst computing, a novel cloud computing model designed to address the growing demand to run sudden, variable, burst-parallel workloads without provisioning resources in advance. We have reviewed the challenges and shortcomings of current technologies (i.e., FaaS), and we have demonstrated the effectiveness and versatility of our proposed solution.

Burst computing offers several key advantages over existing FaaS technologies, primarily the addition of a group invocation primitive that allows the platform to manage jobs as a unit, instead of independent function invocations. This raises tenant isolation to the job level and allows to allocate resources en masse and apply worker packing, which in turn enables powerful locality between workers. Our experiments and performance evaluations have shown that our platform achieves significant improvements by exploiting this locality in job invocation latency, worker simultaneity, code and data loading, and worker-to-worker communication with group collectives. Further, we demonstrated speed-ups of 13×, and 2× in PageRank, and TeraSort, respectively, thereby validating efficacy in real-world scenarios.

In conclusion, burst computing represents a significant advancement for serverless data processing in the cloud, with potential to become a new paradigm of cloud services. It unlocks key limitations of FaaS, becoming the next step forward to support applications previously stymied by its restrictive model [14]. We believe that our contributions are just the substrate for further innovation such as new workflow definition tools and orchestration engines that leverage burst computing jobs and strive towards a simplified dynamic utilization of cloud resources.

5 XtremeHub Compute for RAG: Serverless Vector DBs

5.1 Introduction

The exponential growth of unstructured data has created a pressing need to leverage AI and machine learning (ML) techniques to unlock its potential. Modern AI workloads, such as those using Retrieval-Augmented Generation (RAG) models, rely heavily on generating *vector embeddings* [64] from unstructured data (*e.g.*, text, images, and audio) to enable efficient and meaningful analysis. These embeddings convert complex, high-dimensional data into lower-dimensional vectors that preserve semantic relationships, facilitating advanced tasks like natural language processing [65], image recognition [66], and recommendation systems [67]. Consequently, the ability to manage vector embeddings is key for extracting value from unstructured data in AI/ML applications.

The need for managing vector embeddings has led to the rising popularity of *vector databases* (vector DBs) [68, 69, 70, 71]. Vector DBs are specialized systems designed to store and retrieve high-dimensional vectors efficiently [72]. They enable fast and accurate similarity search, making them essential for AI and ML applications. In this sense, Pinecone [68], Weaviate [69], and Milvus [70, 71] are, among others, examples of popular vector DBs used at scale in production analytics pipelines.

5.1.1 Motivation

Despite the specialization of their search and indexing algorithms, most distributed vector DBs rely on a traditional cluster or *serverful*[13] *architecture* [72, 73]. This requires that administrators carefully manage the resources of the vector DB deployment. Under fluctuating or sparse query workloads, this architecture can result in periods of over-provisioning —leading to higher cost— and under-provisioning —causing saturation. Likewise, adapting such an architecture to handle high workload burstiness is challenging.

In response to such architectural limitations, there is an emerging trend for porting vector DBs to the *serverless paradigm*. In the industry, the main objective seems to be simplifying the provisioning of vector DBs and offering them "as-a-service" —and, in some cases, applying a pay-per-query model. Instantiations of this trend include Weaviate Serverless Cloud [69], Upstash [74], and Amazon OpenSearch Service as a Vector DB [75], to name a few.

However, the true realization of a serverless vector DB architecture goes far beyond simplified provisioning: it lies in distributing the vector DB engine across *cloud functions*. Function-as-a-Service (FaaS) providers, such as AWS Lambda [76], Azure Functions [77], or Google Cloud Functions [78], deliver a powerful substrate for executing embarrassingly parallel workloads. Crucially, such a design has the potential to overcome the elasticity and burstiness challenges that serverful vector DB architectures typically face.

Serverless vector DBs must distribute tasks across parallel cloud functions that are normally executed on compute nodes in serverful vector DBs. These tasks include:

i) *data ingestion*, which consists in adding new vector embeddings to the system; ii) *data partitioning*, which involves distributing a collection of vector embeddings into smaller pieces (*e.g.*, data objects in AWS S3); iii) *data indexing*, which generates indexes for efficient vector lookups within dataset partitions; and iv) *querying*, which performs similarity searches on vectors across dataset partitions. While promising, building a serverless vector DB presents unique challenges from a design perspective compared to serverful vector DBs (see Section 5.3).

5.1.2 Challenge: Stateless FaaS & Dynamic Data

The concept of a serverless vector DB is still in its early stages. Vexless [79] stands out as the pioneering work, demonstrating the potential of using cloud functions to parallelize vector search workloads. However, the design space for serverless vector DBs remains largely unexplored, particularly when considering the challenges introduced by *stateless FaaS platforms* and *dynamic datasets*:

[12] The content of this section maps to tasks T3.1 and T3.2 and is related to the paper "*Building Stateless Serverless Vector DBs via Block-based Data Partitioning*", published in ACM SIGMOD'26.

[13] We use the term *serverful* for architectures built for a traditional cluster of servers.

Stateless FaaS: A key dimension in this space is the nature of the FaaS platform itself. The most popular FaaS offerings —such as AWS Lambda [76], Azure Functions [77], and Google Cloud Functions [78]— are inherently *stateless*. This means that functions cannot retain state across invocations and cannot directly communicate with one another. In contrast, Vexless is built on Azure Durable Functions [80], a stateful FaaS platform that supports direct message passing and persistent workflows. This model is hard to compare or port to stateless FaaS environments. While statelessness imposes certain limitations, it also offers a highly elastic and cost-efficient execution model for embarrassingly parallel workloads that we aim to explore.

Dynamic Datasets: Another critical dimension is data management, particularly in the context of dynamic datasets that evolve over time. A practical serverless vector DB must support efficient ingestion, partitioning, indexing, and querying of data that grows continuously. In NEARDATA, we evaluate the trade-offs of clustering-based (*e.g.*, K-means) data partitioning strategies used in the state-of-the-art [79] when applied to a stateless FaaS setting.

5.1.3 Contributions

This project aims to empirically explore the feasibility of building serverless vector DBs[14] atop stateless FaaS platforms from two angles: *data partitioning* and *comparison with serverful vector DBs*. First, a core contribution is to identify the key limitations of clustering-based data partitioning for dynamic datasets and evaluate a block-based data partitioning alternative. Second, we empirically show that block-based data partitioning in serverless vector DBs enables competitive system designs compared to a serverful one. In summary, our contributions are:

- We present an overview of serverless vector DBs, highlighting their key components and design trade-offs. Given their recent emergence, this project is the first to offer a timely overview of this new family of systems (see Section 5.3).
- We identify the main limitations of using a clustering-based data partitioning scheme for serverless vector DBs, as it is the approach used in the state-of-the-art [79] (see Section 5.4).
- We provide an experimental comparison of clustering-based data partitioning with a simple, yet practical, block-based data partitioning scheme for serverless vector DBs (see Section 5.6).
- We show through experimentation that a serverless vector DB using block-based data partitioning is competitive with a serverful vector DB (Milvus) in terms of indexing time, query latency, recall, and economic cost (see Section 5.7).

We have built a serverless vector DB prototype on AWS Lambda. Our experiments show that a block-based data partitioning outperforms a clustering-based scheme in terms of partitioning performance (3.5× to 5.8×) and cost (56% to 63%), while querying times vary from 29.9% faster to 31.2% slower based on the configuration. Also, we find that techniques proposed to enhance clustering-based partitioning [79], such as balancing data partitions and vector redundancy, may not offer clear benefits overall. Finally, when comparing our prototype with Milvus, we achieve better partitioning time (9.2× to 65.6×), similar query recall, and an acceptable overhead in querying in exchange of cost reduction (66% to 99%).

5.2 Background

In this section, we provide the necessary background to understand the remainder of the section: FaaS and vector DBs.

5.2.1 Function-as-a-Service (FaaS)

The *serverless computing paradigm* is a cloud execution model where developers build and deploy applications without managing the underlying infrastructure [81]. Instead, cloud providers dynamically handle provisioning, scaling, and maintenance. This enhances agility, reduces operational complexity, and optimizes resource utilization by charging only for actual code execution.

[14]The term "serverless vector DB" in this project refers to systems built on stateless FaaS.

The most prominent form of serverless computing is *Function-as-a-Service* (FaaS) [76, 77], which allows developers to write modular functions triggered by specific events, such as HTTP requests, database updates, or message queues. FaaS follows an *event-driven programming model*, where each invocation is ephemeral, stateless, and performs a single task. This allows fine-grained, *pay-per-use* billing, based on the number and duration of invocations, typically measured in milliseconds. Unlike traditional server-based pricing models that incur costs for idle resources, FaaS promotes automatic scaling and efficient utilization. Developers can focus entirely on business logic, which is key for improving developer productivity and democratizes access to cloud platforms.

Importantly, most FaaS services are stateless. This statelessness has limitations, such as cold start latency (*i.e.*, start-up delay after a period of inactivity) [82, 83], lack of persistent state, and barriers to inter-function communication [84]. Note that Microsoft offers stateful FaaS extensions like Azure Durable Functions [80] and Durable Entities, enabling persistent state and workflow orchestration. However, these solutions are still niche and tightly coupled to a specific vendor, limiting the generalizability of the applications that use them. Building a serverless vector DB on top of a stateless FaaS introduces unique challenges that we explore in NEARDATA.

5.2.2 Vector DBs

Typical database operations consists of two main phases: (i) *Data ingestion and storage*, which uses different schemas and generates indexes for fast retrieval; (ii) *Querying*, where the database performs efficient lookups on the stored data. The architecture of vector DBs is similar to general-purpose ones. However, vector DBs store and index high-dimensional vectors, using nearest neighbor search with similarity metrics (*e.g.*, cosine similarity, dot product, or Euclidean distance). Due to the high computational cost of exact nearest neighbor search, vector DBs employ approximate nearest neighbor (ANN) algorithms with specialized index structures (*e.g.*, HNSW, IVF, PQ) to balance recall and latency [85].

When the collection of stored vector embeddings grows, vector DBs face two main scalability-related challenges: (i) the index(es) must often be recomputed when the stored data changes, and (ii) as the number of vectors grows, searching becomes slow. For this reason, in recent years, a new generation of vector DBs have emerged to address these challenges via a distributed architecture and parallel data management. In a distributed vector DB, the different tasks are performed by individual services that can be scaled independently. Moreover, a vector dataset is split into chunks, so vector DB tasks can be executed in parallel. Interestingly, this architecture shift introduces a new dimension to the vector DB operation that is the primary focus in NEARDATA: *data partitioning*.

One instance of distributed vector DB is Milvus [70, 71]. It scales each process independently as services, with nodes (servers or VMs) assigned to specific roles: Data, Index, or Query. Data nodes are responsible for ingestion, partitioning, and storage management. Index nodes create indexes for data chunks, while Query nodes load these indexes and data into memory to respond to queries. Additionally, specialized nodes such as Query Coordinators and Data Coordinators aggregate results and manage data flows, including load balancing. Although services are scaled independently, the stateful nature of this cluster architecture can be rigid and difficult to adapt to rapid workload changes. It also requires an administrator to carefully right-size the deployment. Solving such elasticity and provisioning issues is a key goal of serverless vector DBs.

5.3 Serverless Vector DBs: An Overview

Building a vector DB on top a FaaS service is an emerging trend. In this section, we contribute a general description of the architecture and design trade-offs of this new family of systems (see Fig. 14).

5.3.1 Architecture

A serverless vector DB retains the core components of a serverful vector DB but leverages serverless compute services for elasticity and cost efficiency. A serverless vector DB consists of two service types: *data services* for ingestion and storage, and *compute services* for partitioning, indexing, and

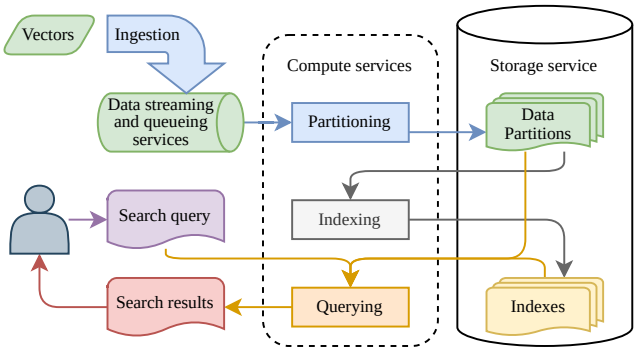


Figure 14: Key processes of a serverless vector DB.

querying. The *data ingestion component* handles vector inserts, supporting both batch and streaming data [86] through event brokers (*e.g.*, Kafka [87], AWS Kinesis) or event-driven pipelines (*e.g.*, AWS Lambda with S3 triggers). After ingestion, vectors undergo *partitioning and preprocessing* (*e.g.*, K-means clustering) before being stored in object storage (*e.g.*, AWS S3) for long-term retention.

Data partition *indexing* is a crucial process performed by index generator functions, which transform vector partitions into efficient approximate nearest neighbor (ANN) search structures. Common indexing techniques include tree-based (KD-Tree), graph-based (HNSW), and quantization-based (IVF-PQ) approaches, enabling fast vector retrieval. These indexes, along with the raw vector data, are stored in scalable object storage systems, ensuring durability and high availability. Since indexing workloads are intermittent, FaaS services are well-suited for dynamically executing index generation without maintaining persistent compute resources.

Query execution in a serverless vector DB is distributed and parallelized across multiple partitions. Typically, a *query coordinator* orchestrates the execution of serverless functions responsible for retrieving candidate vectors from different partitions. These functions compute similarity scores (*e.g.*, cosine similarity, Euclidean distance) and return partial results that are then aggregated using a *reduce function*. Given the stateless nature of cloud functions, query coordination mechanisms (*e.g.*, Step Functions) are essential for synchronizing results. The final ranked vectors are returned to the user and functions terminate their execution.

By combining cloud functions with scalable storage, a serverless vector DB eliminates the need for persistent infrastructure while maintaining high-performance search capabilities. Compared to serverful architectures, this approach automatically scales with workload demands, optimizes cost through pay-as-you-go billing, and reduces operational overhead. This makes serverless vector databases a compelling choice for applications requiring efficient and scalable similarity search, such as recommendation systems, image retrieval, and AI-powered analytics.

5.3.2 Vector DB Design: Serverful vs Serverless

Next, we discuss the architectural differences between a popular serverful vector DB (Milvus) and existing serverless counterparts (Vexless, this project). In Table 5, the systems compared represent different architectural approaches to vector DB management.

Milvus is a popular example of a serverful vector DB architecture designed to manage vector embeddings as a service. One of the primary advantages of its design is its ability to scale both horizontally and vertically, allowing the addition of more resources per node to handle increasing workloads. This makes it particularly well-suited for handling continuous query workloads with moderate fluctuations, as it provides fast interactive queries. In a typical deployment, Milvus uses event streaming systems (*e.g.*, Kafka, Pulsar) to achieve high-performance data ingestion. The low latency and high performance of service nodes are key advantages of the serverful architecture over serverless counterparts, making it an attractive option for applications that require low-latency guarantees on vector search.

Table 5: Comparison of Vector DB architectures.

Dimension	Milvus [70, 71]	Vexless [79]	This project
Architecture	Serverful	Serverless (*Stateful*, Azure Durable Functions)	Serverless (*Stateless*, AWS Lambda)
Communication	Distributed coordination (gRPC)	Stateful functions with message passing	Object storage-based communication
Elasticity	Horizontal node scaling	Automatic function scaling	Automatic function scaling
Operation	Node management/maintenance	Automatic function provisioning	Automatic function provisioning
Billing	Node/hour based	Pay per function invocation	Pay per function invocation
Data Ingestion	Data nodes (streaming-based)	Not supported (static datasets)	Supported (data object ingestion)
Data Partitioning	Shard-based (dynamic sharding with load balancing)	Clustering-based (balanced K-means with redundancy)	Block-based (fixed-size parallel blocks)
Data Indexing	Per shard segment on Index nodes (multi-algorithm)	Per cluster on cloud functions* (HNSW)	Per block on cloud functions (IVF)
Querying	Query nodes (similarity/multi-vector search) - interactive/batch	Cloud functions on partition subsets (similarity search) - interactive	Cloud functions on entire dataset (similarity search) - batch

*Vexless actually performs indexing in a centralized VM, but as we show in Section 5.6 this phase can be parallelized.

However, a serverful vector DB deployment may be cumbersome to manage under high workload fluctuations and burstiness. This is because this architecture introduces additional complexity and operational burden to adapt the service to the workload needs. Furthermore, workload sparsity can lead to low cost-effectiveness for a serverful vector DB, as resources can be underutilized during periods of low activity. In such scenarios, serverless vector DBs have emerged as an interesting alternative, building on top of FaaS offerings (*e.g.*, Azure Functions). By automatically provisioning cloud functions and billing them on a per-execution basis, serverless vector DBs can minimize operational overhead and reduce cost. As a result, serverful and serverless vector DB architectures provide trade-offs that make them ideal for different scenarios, thus requiring careful consideration of the specific use case at hand.

When focusing on serverless vector DBs, there are fundamental differences between Vexless and this project. Vexless leverages a *stateful FaaS platform* (Azure Durable Functions) for keeping data in memory and enabling direct function re-invocation, thus avoiding cold starts. These properties allow Vexless to maintain *low query times in interactive mode*, as previously loaded data can be reused across queries without repeated access to external storage. However, Azure Durable Functions represents a niche FaaS model that is not widely adopted across cloud providers. In contrast, our work explores *stateless FaaS platforms* —such as AWS Lambda, Azure Functions, and Google Cloud Functions— which represent the mainstream model today. Stateless FaaS functions are ephemeral and cannot retain state between invocations, which means that each query must fetch data from external storage (*e.g.*, object stores), introducing additional latency that can impact interactive performance. While this model imposes certain trade-offs, our experiments show that we can amortize statelessness costs via *query batching* (see §5.5). Importantly, the fundamental differences between stateful and stateless FaaS platforms make it difficult to directly compare serverless vector DBs built on top of different platforms. As such, this project evaluates exclusively stateless FaaS as a widely available foundation for serverless vector DBs.

Crucially, the design of Vexless cannot handle continuous data ingestion as it assumes static datasets. Vexless performs a clustering-based data partitioning and indexing approach that must be executed in advance, limiting its ability to handle dynamic datasets. In contrast, we propose a block-based data partitioning approach that enables continuous data ingestion to the system, providing greater flexibility and scalability. In the next section, we analyze in depth the trade-offs of data partitioning between Vexless and this project.

5.4 Trade-offs in Data Partitioning

We have overviewed the architecture of serverless vector DBs. Next, we focus on a critical aspect when distributing a vector DB engine across stateless cloud functions: *data partitioning*. Concretely, we compare the state-of-the-art clustering-based data partitioning [79] with our block-based data

partitioning scheme (see Fig. 15) regarding three dimensions relevant in a serverless scenario: i) *partitioning complexity*, ii) *load balancing*, and iii) *query performance*.

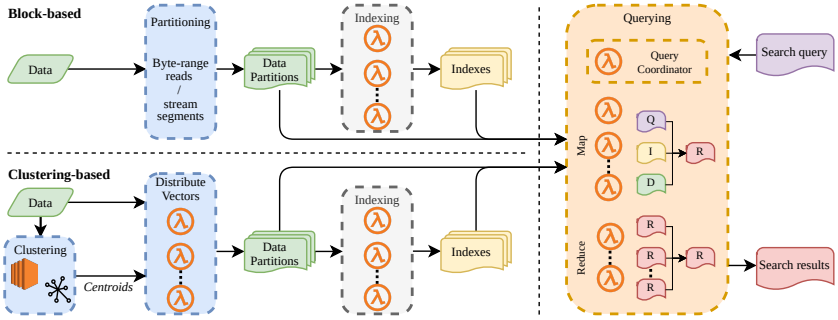


Figure 15: Life cycle of a serverless vector DB comparing block-based and clustering-based data partitioning. Querying is similar architecture-wise in both cases. Still, while the block-based scheme must query on all partitions, clustering-based partitioning allows the query coordinator to filter data based on the distance between the query vector and a partition's centroid.

5.4.1 Clustering-based Data Partitioning

Partitioning complexity: This partitioning method creates clusters of nearby vectors aiming to accelerate queries by filtering out unrelated data partitions. To this end, it uses a three-step indexing pipeline. First, an unsupervised clustering algorithm, such as K-means, is implemented to cluster nearby vectors. It is important to note that clustering algorithms like K-means, hierarchical, or spectral clustering are *computationally complex* and require the *entire dataset* to operate. In practice, while distributed versions of K-means have been investigated [88], it seems inefficient to implement it on top of cloud functions. For this reason, we found that resorting to a virtual machine for executing the clustering algorithm is a feasible alternative. Once the clustering algorithm completes, the dataset is partitioned and distributed among multiple cloud functions, along with the centroids, to classify the vectors to their clusters/partitions. Finally, multiple parallel cloud functions index the dataset partitions containing nearby vectors.

Load balancing: Clustering algorithms like K-means can lead to unbalanced dataset partitions, where some clusters may contain significantly more vectors than others. To address this issue, balanced versions of K-means have been developed, which ensure equal partitioning of datasets across clusters. As pointed out in Vexless [79], producing balanced data partitions is crucial in a serverless setting to avoid straggler cloud functions. Stragglers can delay query responses due to uneven data distribution. However, the balanced version of K-means incurs a higher computational cost compared to the standard algorithm, making it impractical for large datasets [89, 90] under a continuous ingestion workload.

Query performance: This method groups vectors based on their closeness, enabling users to discard partitions with vectors far from the query input. Upon receiving a query, a clustering-based data partitioning system identifies the centroids closest to the query vector. Based on a *data discard* parameter, only the closest centroids are returned. The similarity search is then performed exclusively on data partition indexes associated with these nearby centroids. However, querying fewer centroids reduces *query recall*, as it increases the likelihood of missing the actual closest vectors. Users must understand the trade-off between data filtering and query recall, which can be challenging since many applications require high recall (*e.g.*, $> 90\%$). To address this, Vexless [79] incorporates a data redundancy mechanism that replicates vectors near partition boundaries into adjacent partitions. In Section 5.6, we evaluate the behavior of vector redundancy for clustering-based partitioning.

5.4.2 Block-based Data Partitioning

Partitioning complexity: In a block-based scheme, the complexity of data partitioning is significantly reduced by not requiring a view of the entire dataset. This approach creates equal-sized data partitions, which simplifies the process and ensures uniform distribution of data. Additionally, the ability to partition these chunks in parallel improves scalability, as multiple chunks can be processed simultaneously without dependencies on other parts of the dataset.

Load balancing: By creating equal-sized chunks, the scheme ensures that each cloud function handles a uniform amount of data, thus preventing straggler functions. Moreover, the ability to partition chunks in parallel allows multiple cloud functions to operate simultaneously, maximizing resource utilization.

Query performance: A block-based data partitioning scheme does not consider vector relationships, requiring all partitions to be queried for similarity searches. This results in fast and simple partitioning and indexing phases, avoiding the complexity of analyzing vector relationships. However, the querying phase becomes more computationally expensive, as it must query all partitions without discarding initially dissimilar vectors. Consequently, all vectors are downloaded and queried across indexes, increasing computational requirements and potentially lengthening query times. This trade-off highlights the balance between efficient indexing and comprehensive querying for accurate results.

5.5 Experimental Methodology

In this section, we provide details on our serverless vector DB implementation, as well as the experimentation setup and methodology.

5.5.1 Prototype Implementation

We have developed a serverless vector DB prototype that leverages cloud functions (AWS Lambda) to parallelize the execution of vector DB compute services [91]. Our prototype orchestrates the invocation of stateless cloud functions and transfers data through object storage. The implementation uses Lithops [92], a serverless framework for running cloud functions in parallel in map-reduce fashion. Lithops simplifies Python cloud function development and allows seamless execution on multiple FaaS services, Kubernetes clusters, or VMs. Our prototype is highly configurable with many parameters such as the number of partitions, cloud functions for each phase, index parameters, etc. A complete list is available in the prototype repository and documentation.

Partitioning workflow The main goal of our prototype is to evaluate different data partitioning schemes in serverless vector DBs. Irrespective of the scheme, the data partitioning process reads a dataset from and stores N partitions to object storage. To generate indexes, the data partitions are distributed across a set of functions using a Lithops `map` operation. Each function creates the corresponding index (or indexes) using Faiss [93] and uploads it to object storage. We use an Inverted File (IVF) configuration of $k = 512$ and multi-probe at 32. After extensive testing, this proved to give the best results overall for the datasets explored in this evaluation.

Query workflow When querying the data, our prototype acts as a query coordinator and uses two Lithops `maps` to perform a map-reduce operation. Due to the statelessness of cloud functions, queries are processed in *batches*, with each Lithops operation (a set of cloud functions) handling a full batch. The `map` phase distributes the data partitions among parallel functions, each of which runs all queries on the corresponding index (or indexes) and generates partial similarity responses. If a function processes multiple partitions, it combines the partial results before passing them to the `reduce` phase. The `reduce` phase aggregates the map responses and produces the overall top-k similarity results. Finally, our prototype collects the results and computes their accuracy. This process enables efficient and scalable similarity search, leveraging the power of distributed computing and object storage.

Block-based approach Data partitioning is applied directly on the indexing functions. Since this approach simply splits the data into chunks, the functions read the dataset from object storage with

Algorithm 1 Vector distribution for clustering-based partitioning.

Input: Database D of m vectors in $\mathbb{R}^n$. Clusters $c_1, c_2, \ldots, c_N$ with centroids $C_1, C_2, \ldots, C_N$, and labels for each vector in D. Percentage of redundancy threshold r.
Initialize: For each cluster c_i, initialize indexing partition I_i.
for all vector v in D **do**
    $c_h \leftarrow$ label from clustering assignment of v.
    Add v to I_h.
    for $i = 1, k$ where $i \neq h$ **do**
        if $d(v, C_i) \leq (1 + r)d(v, C_h)$ **then**
            Add v to I_i.
        end if
    end for
end for

byte-range requests to get a specific partition. Indexing is then applied to each partition. For querying, the full batch of queries is sent to all parallel functions and applied to all N partitions.

Clustering-based approach Clustering-based partitioning requires three additional steps: 1. dataset clustering, 2. vector distribution with redundancy, and 3. partition filtering in querying.

First, clustering is executed on a VM and produces a set of N clusters $c_1, c_2, \ldots, c_N$ with centroids $C_1, C_2, \ldots, C_N$, and a label for each vector in the database that assigns it to a cluster. The baseline clustering algorithm is the Faiss K-means implementation. We also explore the balanced version [94] proposed in Vexless [79]. Using a VM for computing the clustering is the strategy also followed in Vexless and it is a favorable configuration, as it is faster and more practical than implementing this phase in a distributed approach on top of cloud functions.

After clustering, we must distribute the vectors to their corresponding partition. While Vexless does this on the same VM where clustering happens, we parallelize this step with a Lithops `map` operation on cloud functions to speed up the process. Each function reads a part of the input dataset from storage using byte ranges (like the blocks implementation) and applies the distribution logic layout in Algorithm 1. This logic includes vector redundancy, which adds boundary vectors to multiple partitions to improve search accuracy. We develop our own redundancy logic for vector redundancy because Vexless does not offer specific guidelines or code for this purpose. Our solution is based on a redundancy percentage[15] r that acts as threshold on how close a second centroid must be compared to the closest for that vector to also be included in the second partition. Initially, vectors are added to the partition that K-means assigns them to.[16] Then the vector is added to other partitions if the distance to their centroid is lower than to the first one extended by r. For instance, if the clustering assigns a vector v to a cluster c_h with its centroid C_h at distance $d(v, C_h) = 1$, with $r = 5\%$ we will add v to any partition whose centroid C_i is at $d(v, C_i) \leq 1.05$.

To optimize queries, we employ a data partition filtering technique that selectively discards partitions. We can configure the system to search only on N_{search} partitions, given as a number or as a percentage of the total number of partitions N. During query coordination, we rank partitions based on the proximity of their centroid to the query vector and select the top N_{search} for searching. This process generates a mapping that determines which partitions to search for each vector in the query batch, which is then sent to the query functions for execution. Querying is aware of vector redundancy and checks for potential duplicates. Although all functions are typically spawned for each batch, the search computation is reduced by $100 - N_{\text{search}}\%$, resulting in improved efficiency.

[15] As opposed to Vexless, which uses an arbitrary distance value.

[16] This is the closest centroid for the baseline K-means, but it could be another one in the balanced version of the algorithm.

5.5.2 Setup

Deployment All experiments are run on Amazon Web Services (AWS) using a combination of AWS Lambda, EC2, and S3. In the case of our serverless prototype, the coordination runs on a `c7i.xlarge` EC2 instance. All data is stored in S3, including the datasets and the intermediate results of the processes. AWS Lambda functions are configured with 10 GB of memory for indexing and 8 GB for querying, which gives them 6 and 4 vCPUs, respectively, according to service documentation.[17] The indexing process is parallelized on 16 functions for all configurations, splitting partitions evenly among them. Querying uses 4 functions (or 8 in Fig. 24) in the map phase (also splitting partitions evenly) and a single one for the reduce phase. The block-based approach does not need any additional resources. For the clustering-based approach, K-means clustering runs on a `c7i.12xlarge` instance, which also acts as coordinator in these executions. Vector distribution uses 16 parallel functions (10 GB). Datasets must be stored in an S3 bucket, while query files must be uploaded to the client EC2 instance.

The architecture of Milvus makes it impossible to run on cloud functions without considerable modifications. FaaS cannot be used to host a serverful technology due to its statelessness and a single function cannot fit a Milvus deployment capable of handling our selected datasets. Therefore, we deploy Milvus on `c7i.4xlarge` and `c7i.8xlarge` VMs.[18] This setup matches the CPU and memory resources of our serverless vector DB prototype, ensuring a fair and comparable evaluation. Milvus is set to use the same IVF configuration for indexing as our prototype. These experiments also use a `c7i.xlarge` instance as a client running the `vectordbbench` tool [96], developed by the same team as Milvus (Zilliztech). Specifically, we use a custom docker container image that includes the `vectordbbench` tool, all its requirements, and both the datasets and query vectors in the specific format.

Methodology Datasets containing collections of vectors are initially stored in object storage as a single object each. All processes of a vector DB are evaluated on different number of partitions $N = \{16, 32, 64, 128\}$, and use equivalent resources in all systems, either with cloud functions or VMs. Data ingestion, partitioning, and indexing are reported aggregated as "data partitioning time" because it is the most significant part and the focus of our evaluation. All times reported include the corresponding cloud function invocation latency. Ingestion is always from object storage and equivalent for all systems. Indexing is an independent, stateless process that only depends on the size of the partitions, not the partitioning scheme. We did not find significant variation between the evaluated approaches. Reported results are averaged across ≥ 5 executions, and whiskers in plots denote standard deviation (σ).

We evaluate search accuracy using the recall metric, which measures the proximity of the response to the true neighbors of the query vector, with 100% recall indicating a perfect score. To calculate recall, we first identify the true neighbors of all query vectors using a Flat Index on the entire dataset and store them in object storage. During evaluation, each query execution is compared to these pre-computed true results to assess accuracy. For querying evaluation, we utilize a batch of 1000 queries extracted from the original query file accompanying each dataset, with each query requesting the top-10 nearest vectors.

We calculate cost based on the processing time of the experiments and the resources being utilized at each moment. We use current AWS pricing as of March 2025. For the serverless implementation, cost is the sum of the running time of all cloud functions plus invocation fees. We account the cost of VMs as the fraction corresponding to the seconds they are actively used in the experiments. Later, we also show the cost of having a serverful system running for longer periods of time, even if idle, as in a real scenario.

Datasets We use three publicly available datasets that are commonly employed to evaluate the quality of approximate nearest neighbors search algorithms in vector DBs: (i) DEEP [97], which we

[17] Functions have one vCPU per 1769 MB and scale both resources proportionally [95].

[18] Experiments with larger VMs for Milvus (*e.g.*, `c7i.24xlarge` to match serverless indexing resources) suggest it underutilizes resources during indexing.

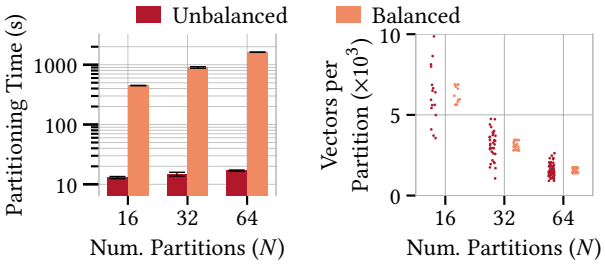


Figure 16: Partitioning time and vector dispersion across partitions for the two K-means versions (DEEP100k dataset).

use in subsets of 100k, 1 million, 10 million, and 100 million embeddings of the Deep1B with 96 dimensions normalized with L2 distance extracted from the last fully-connected layer of a GoogLeNet [98] model trained with the ImageNet [99] dataset; (ii) SIFT [100], which consists of 10 million embeddings with 128 dimensions, the Scale Invariant Feature Transform (SIFT) transforms the image data into a large number of features (scale invariant coordinates) that densely cover the image features; and (iii) GIST [101], which consists of 1 million embeddings with 960 dimensions, a gist represents an image scene as a low-dimensional vector.

5.6 Clustering vs Block-based Data Partitioning

In the first experimental section, we aim to answer the following questions regarding data partitioning in serverless vector DBs:

(1) What is the cost of achieving balanced data partitions in the clustering-based approach? (Section 5.6.1)

(2) How do query recall and storage overhead trade off with vector redundancy in clustering-based partitioning? (Section 5.6.2)

(3) How do clustering-based and block-based partitioning compare in terms of data partitioning and indexing performance? (Section 5.6.3)

(4) What are the query performance differences between clustering-based and block-based data partitioning? (Section 5.6.4)

(5) Does clustering-based partitioning amortize partitioning costs via query filtering, compared to a block-based scheme? (Section 5.6.5)

5.6.1 The Cost of Balanced Data Partitions

As a representative of clustering-based data partitioning for serverless vector DBs, Vexless emphasizes the need for balanced data partitions to prevent straggler cloud functions. To this end, it proposes a balanced K-means algorithm that maintains near-equally-sized vector partitions. We analyze the cost of keeping balanced data partitions by deploying two clustering algorithms in our prototype: unbalanced K-means (Faiss default implementation) and balanced K-means [94]. The latter formulates the cluster assignment step as a Minimum Cost Flow linear network optimization problem [102].

First, we focus on the data partitioning and indexing cost for both K-means versions. Fig. 16 (right) shows the number of vectors per partition in both cases. Naturally, the balanced K-means exhibits a significantly lower dispersion of vectors per partition (*e.g.*, $\sigma = 0.541 \times 10^3$ for 16 partitions) compared to unbalanced K-means (*e.g.*, $\sigma = 1.778 \times 10^3$ for 16 partitions). However, Fig. 16 (left) also shows the differences in computational complexity related to partitioning and indexing the DEEP100k dataset for unbalanced ($\mathcal{O}(nc)$) and balanced K-means ($\mathcal{O}((n^3c + n^2c^2 + nc^3)\log(n + c))$)

Table 6: Query performance comparison or the two K-means versions on DEEP100k dataset (single batch of 1000 queries).

Params		Balanced			Unbalanced		
N	N_{search}	Time (s)	$\pm\sigma$	Recall	Time (s)	$\pm\sigma$	Recall
16	1	7.103	0.924	71.30	7.491	1.791	70.68
16	4	6.716	0.031	92.55	6.742	0.026	92.97
16	8	6.712	0.026	95.01	6.789	0.048	95.37
16	12	6.771	0.026	95.49	6.779	0.066	95.64
16	16	6.761	0.048	95.52	6.825	0.077	95.65
32	1	7.095	0.874	63.54	7.118	0.923	65.59
32	8	6.766	0.049	94.35	6.750	0.031	94.75
32	16	6.772	0.036	95.64	6.823	0.036	95.90
32	24	6.856	0.027	95.81	6.860	0.062	96.05
32	32	6.934	0.048	95.82	7.128	0.343	96.05
64	1	7.127	0.904	60.54	6.708	0.029	60.43
64	16	6.815	0.065	96.44	7.247	0.888	96.34
64	32	7.699	0.993	97.03	8.315	0.976	96.97
64	48	8.811	0.480	97.12	9.371	1.706	97.03
64	64	9.061	0.425	97.13	10.287	2.169	97.06

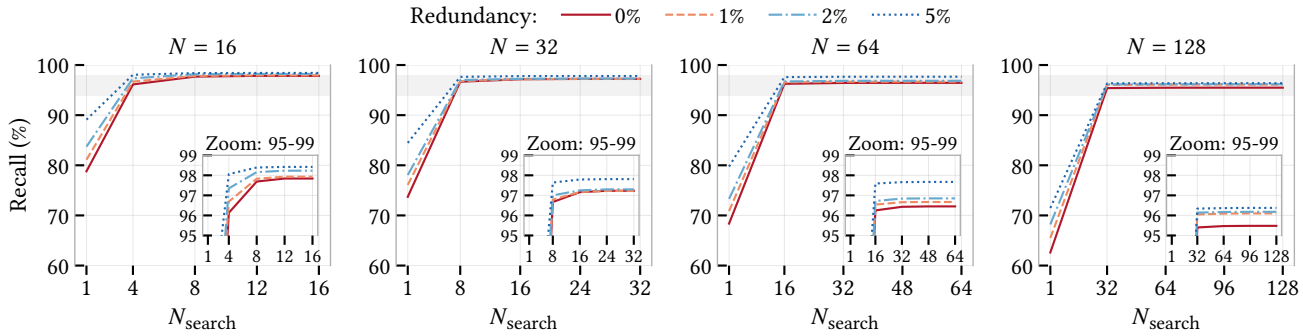


Figure 17: Effect of redundancy (r) and N_{search} on recall depending on the total number of partitions (N).

algorithms. Visibly, using the balanced version of K-means incurs from 35× up to 96× higher partitioning time, mostly due to the computation of centroids. We could not experiment with larger datasets using the balanced K-means version due to the growing clustering times.

One could think that higher partitioning and indexing cost may be linked to significant advantages in query performance. In Table 6, we observe that the balanced K-means algorithm outperforms the unbalanced version in terms of query time, with an average reduction of 2.6% across all experiments, and a maximum reduction of 13.5% for $N = 64$ and $N_{search} = 64$. In terms of recall, the balanced K-means algorithm achieves an average increase of 0.3% across all experiments, with a maximum increase of 0.8% for $N = 16$ and $N_{search} = 1$. The standard deviation of query times is also halved on average for the balanced K-means algorithm, indicating more consistent performance. However, such advantages seem modest compared to the cost related to data partitioning and indexing. This is especially true when considering large and/or dynamic datasets that require continuous indexing activity.

Conclusion. Balanced clustering algorithms are costly to compute and offer modest query gains, making them unsuitable for efficient dynamic dataset partitioning in serverless vector DBs.

Table 7: Impact of vector redundancy (r) on the vectors per data partition standard deviation (σ) for DEEP100k dataset.

Vector redundancy (r)	**Standard deviation (σ)**		
	$N = 16$	$N = 32$	$N = 64$
$r = 0\%$ (Unbalanced K-means)	1778	903	378
$r = 0\%$ (Balanced K-means)	541	264	138
$r = 5\%$ (Balanced K-means)	1491	1792	660

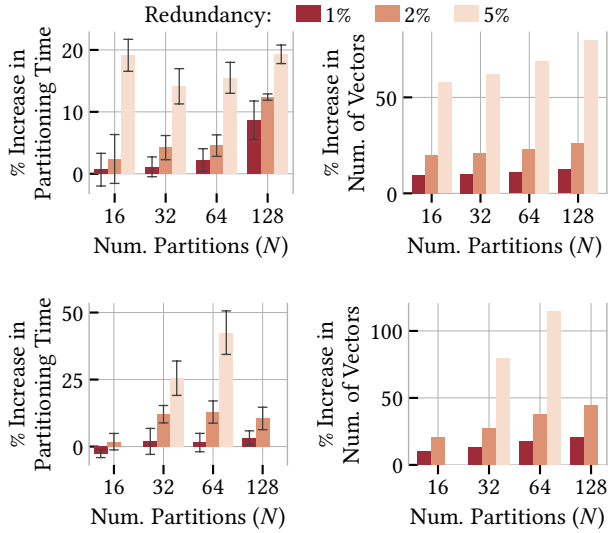


Figure 18: Percentage of increase in partitioning time and stored vectors for different r values with respect to no redundancy. DEEP10M dataset on top, GIST1M dataset on bottom.

5.6.2 The Effect of Vector Redundancy

Vexless contributes a vector redundancy mechanism to mitigate loss in query recall when filtering data partitions according to the input vector distance to centroids. Next, we provide an evaluation of the effect of incorporating vector redundancy. The vector redundancy implementation details can be found in Section 8.6.

Retaking the discussion on partition load balancing, Table 7 shows the impact of vector redundancy (r) on the dispersion of vectors per data partition (σ). Visibly, adding vector redundancy to the balanced K-means version re-introduces load imbalance across partitions. For example, in the DEEP dataset, setting $r = 5\%$ makes the dispersion of vectors per data partition (σ) to be up to 1.98× worse than the unbalanced K-means version ($N = 32$). This insight is important, as Vexless proposes using two techniques that seem to have conflicting outcomes. Note that given the previous results, the remainder of our analysis uses the unbalanced version of K-means.

Fig. 17 shows the recall of search queries based on the number of partitions available (N) and searched (N_{search}). Vector redundancy exhibits a 3% to 16% improvement in query recall compared to the baseline (*i.e.*, no redundancy) when searching in the closest data partition ($N_{search} = 1$). However, for $N_{search} = 1$, we also observe that query recall is relatively low ($< 90\%$), which may not be precise enough for many applications. At the same time, the recall improvements of vector redundancy become less evident ($< 2\%$) as N_{search} increases.

Interestingly, vector redundancy has additional cost in terms of data partitioning and indexing time, as well as storage overhead. Fig. 18 (left) shows the relative increment in data partitioning and indexing time depending on the vector redundancy level for DEEP10M and GIST1M datasets.

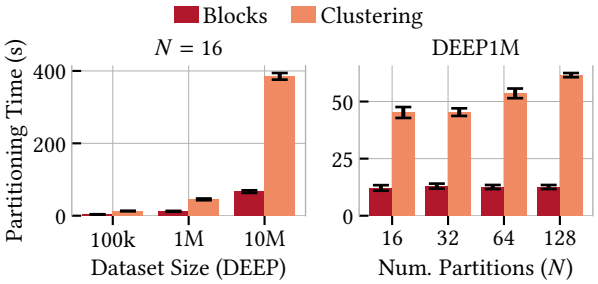


Figure 19: Partitioning time comparison for different data volume and number of partitions (N).

Table 8: Partition size (in MiB) balance for a clustering-based approach when using the entire DEEP10M dataset (Full) or only the first 1000 vectors (1k).

	16 Clusters		32 Clusters		64 Clusters	
Statistic	Full	1k	Full	1k	Full	1k
Min	90.00	61.00	52.00	1.30	27.00	1.00
Max	300.00	603.00	171.00	263.00	142.00	177.00
Mean	233.62	233.64	116.97	116.88	58.52	59.29
σ	51.94	87.18	28.49	61.87	21.19	39.22
CV	0.22	0.37	0.24	0.53	0.36	0.66

In case of DEEP, redundancy seems affordable up to $r = 2\%$ if $N \leq 64$ ($< 5\%$). However, higher r or N configurations increase processing time between 9% to 19% on average. Storage overhead (right of Fig. 18) also grows from 9% up to 80% depending on r and N. GIST, which has a higher dimensionality, suffers even more from this cost growth, reaching a 40% overhead in processing time and more than doubling stored vectors for $r = 5\%$ and $N = 64$.

Conclusion. The benefits of vector redundancy are limited to searching very few partitions (*e.g.*, $N_{search} = 1$) and still result in low query recalls ($< 90\%$) that may not meet the needs of many applications. Considering the load balancing, indexing, and storage costs, the general applicability of vector redundancy is unclear.

5.6.3 Clustering vs Blocks: Data Partitioning

Next, we compare the proposed block-based data partitioning with the clustering-based counterpart (unbalanced, no redundancy) in terms of data partitioning and indexing time (see Fig. 19). As described in Section 5.5.2, we use 16 cloud functions for indexing data partitions in both cases, whereas the clustering-based data partitioning uses an additional `c7i.12xlarge` VM for computing the centroids.

As expected, for a fixed dataset size (DEEP1M), the block-based partitioning is not sensitive to the number of partitions (see Fig. 19, right). The main reason is that block-based partitioning executes in an embarrassingly parallel fashion and can be performed with parallel byte-range reads from storage. On the other hand, the stateful nature of clustering-based data partitioning requires a complete view of the dataset and its complexity increases with the number of data partitions (*i.e.*, clusters).

However, the real limitation of clustering-based data partitioning is rendered when scaling the dataset size for the same amount of resources (Fig. 19, left). Visibly, increasing the dataset size from 100k to 1M and from 1M to 10M leads to data partitioning times 3.6× and 8.6× higher, respectively. The main reason is that the K-means clustering stage needs to load the full dataset in a single VM — not in parallel— and perform an increasingly expensive computation with dataset size. Conversely, for block-based data partitioning, partitioning time scales linearly with data volume for the same amount of resources. This is because each cloud function has more data to index in each block.

Lastly, we want to reinforce the observation that clustering-based partitioning is inherently un-

Table 9: Query recall comparison of block-based (B) versus clustering-based (C) data partitioning (DEEP10M dataset).

	Num. Partitions (N)							
	16		32		64		128	
N_{search}	B	C	B	C	B	C	B	C
25%	99.24	96.13	99.26	96.67	99.44	96.25	99.30	95.40
50%		97.69		97.16		96.43		95.47
75%		97.84		97.23		96.45		95.48
100%		97.84		97.23		96.45		95.48

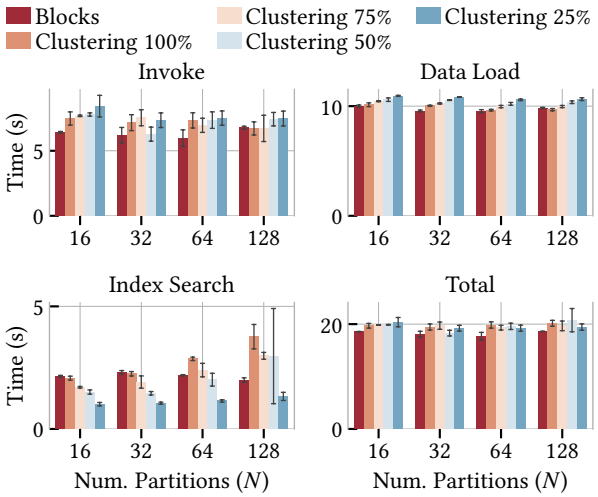


Figure 20: Querying time comparison of block-based versus clustering-based data partitioning for different number of partitions (N) and N_{search} at 100%, 75%, 50%, and 25%. Besides the three phases shown, the total time includes function invocation overhead and the reduce operation, which vary minimally and, aggregated, add ≈6 seconds on average.

suitable for dynamic data ingestion. Clustering algorithms operate on a static dataset to generate partitions. When new data arrives after the initial clustering, there are only two options: i) recompute the clustering to include the new data, or ii) continue using the existing partitioning. The former is computationally expensive, as it requires reprocessing the entire dataset and temporarily halting database operations. The latter leads to increasingly unbalanced partitions over time, degrading both performance and accuracy. Table 8 illustrates the latter issue using the DEEP10M dataset. We compare partition balance when clustering is computed on the full dataset versus only the first 1000 vectors. The results show that using a small subset for clustering significantly increases imbalance, raising the coefficient of variation by 1.7× to 2.2×. This negatively impacts query latency and recall. In contrast, block-based partitioning is well-suited for dynamic ingestion. New data can be incrementally added by splitting it into fixed-size blocks, without requiring global reorganization.

Conclusion. Block-based data partitioning is more efficient and scalable than clustering-based partitioning, especially as dataset size and number of partitions increase.

5.6.4 Clustering vs Blocks: Query Performance

In this section, we compare the implications of data partitioning on query performance: query recall (Table 9) and query times (Fig. 20).

Table 9 shows the query recall comparison between block-based and clustering-based data partitioning. We observe that the recall of the block-based scheme consistently outperforms clustering-

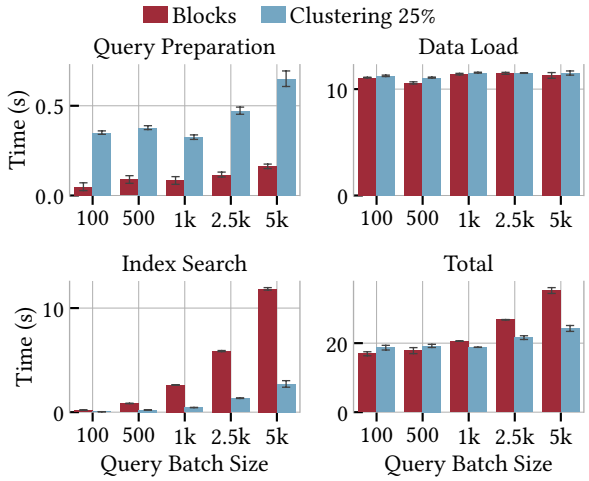


Figure 21: Querying time comparison of block-based versus clustering-based ($N_{\text{search}} = 25\%$) data partitioning for different query batch sizes, using DEEP10M with 32 partitions. Besides the three phases shown, the total time includes function invocation overhead and the reduction operation, which vary minimally and, aggregated, add ≈6 seconds on average.

based partitioning by 1.4% to 3.9% for different values of N and N_{search}. However, these query recall differences are not very significant, even for low values of N_{search}. This indicates that data filtering in queries can be effectively leveraged while maintaining acceptable query recall. Interestingly, when $N = N_{\text{search}}$, the query recall of clustering-based data partitioning is slightly lower than the block-based scheme. This may be due to the interplay of vector distribution with IVF indexes in our prototype. To inspect this, we reproduced the experiment with HNSW indexes (as in Vexless) obtaining similar results (*i.e.*, block-based partitioning shows better recall than the clustering-based scheme by 0.09% to 3.25%).

Fig. 20 provides a breakdown of query latency in our prototype for both clustering-based and block-based data partitioning schemes. Interestingly, data partitioning has an impact on the query phases of a serverless vector DB.

The `preparation` phase sets up the job, which for clustering-based partitioning involves selecting the right partitions to query based on the input vectors, which adds overhead. The `search` phase executes vector search in the cloud functions for each data partition. The data filtering in the clustering-based approach has the largest impact here. The `data load` phase retrieves data partitions and the query batch file. This is constant in all cases because all partitions are required for any query batch. The times to invoke functions and execute the reduce operation are not differentiated in the plot, but contribute to the total time. These times are constant in all configurations. In this experiment, loading the data dominates the total time, resulting in similar performance for both approaches. Specifically, our prototype exhibits query times up to 16% faster using block-based compared to clustering-based partitioning. The results are best for clustering when using 32 partitions, where it improves block-based query times by 9.47%.

The advantages of clustering-based partitioning primarily impact index search time. Query performance improvements are visible only when this phase dominates querying times. To illustrate this, we evaluated query batches ranging from 100 to 5000 vectors. Fig. 21 shows that clustering increases preparation time with larger batches, while data loading remains stable. Although search time grows in both approaches due to higher workload, clustering benefits more from partition filtering, yielding an overall performance gain of 31.18% over block-based partitioning at 5k vectors. However, such large batches may be impractical in real-world scenarios.

Conclusion. The execution complexity in clustering-based partitioning prevents it to reduce query times in a serverless vector DB, even with query filtering, unless using large query batches.

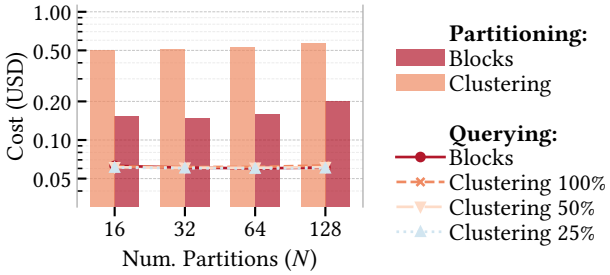


Figure 22: Total indexing and querying costs of clustering-based and block-based data partitioning with 10 batches of 5000 queries (DEEP10M dataset).

5.6.5 Clustering vs Blocks: Cost Analysis

Next, we focus on understanding the partitioning and querying costs related to applying clustering-based and block-based data partitioning in a serverless vector DB. It is important to note that, due to using stateless FaaS, queries are executed in batches. Thus, query costs are calculated per batch (not individual queries).

In terms of data partitioning and indexing, Fig. 22 shows that the longer processing times of K-means clustering, plus the additional VM needed, incur a 2.82× to 3.44× increase in partitioning cost.[19] Note that we evaluate the partitioning and indexing of a static dataset. If we consider dynamic data, clustering-based partitioning cost would increase due to the re-processing of existing data.

Notably, our prototype reveals that query cost depends on the batch size. For batches of ≤ 1000 vectors, clustering-based data partitioning does not offset its own overhead, even with per-query data filtering. This aligns with previous results (Section 5.6.4). Fig. 22 shows the cost for batches of 5000 vectors, where the effect of partition filtering is most pronounced (1.6× improvement with 32 partitions and $N_{\text{search}} = 25\%$). Interestingly, the cost reduction from clustering becomes less significant compared to the block-based approach as the number of partitions increases. This observation highlights the trade-off between query cost, data filtering, and query latency in a serverless vector DB.

Conclusion. Clustering-based partitioning incurs significantly higher indexing cost than the block-based scheme. It offers no query cost benefits for small batches ($\leq$ 1000 vectors), even with data filtering. For large batches, it can reduce search cost through partition filtering, but gains depend on batch size and configuration.

5.7 Milvus vs Block-based Serverless Vector DB

In this section, we compare our proposed serverless vector DB prototype ("SVDB" for short) with block-based data partitioning against a popular serverful vector DB system: Milvus. In particular, the following experiments aim to answer the following questions:

(1) How does vector indexing with SVDB compare to Milvus? (Section 5.7.1)

(2) How does querying with SVDB compare to Milvus? (Section 5.7.2)

(3) How does SVDB improve cost compared to Milvus? (Section 5.7.3)

(4) How does SVDB scale with data volume? (Section 5.7.4)

5.7.1 Partitioning and Indexing Performance

Next, we compare the data partitioning/indexing performance in Milvus and SVDB. To this end, Fig. 23 shows the processing time of both systems for multiple datasets and partition numbers (N).

Visibly, SVDB indexes all three datasets 9.2× to 65.6× faster than Milvus. This is because SVDB fully exploits the parallelism of cloud functions. In contrast, Milvus does not parallelize the ingestion

[19] Based on on-demand pricing; spot instance rates still lead to similar conclusions.

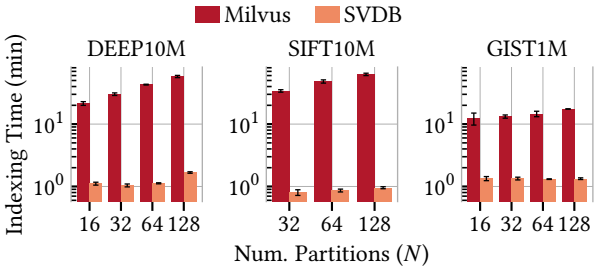


Figure 23: Partitioning and indexing time for different datasets on Milvus and SVDB.

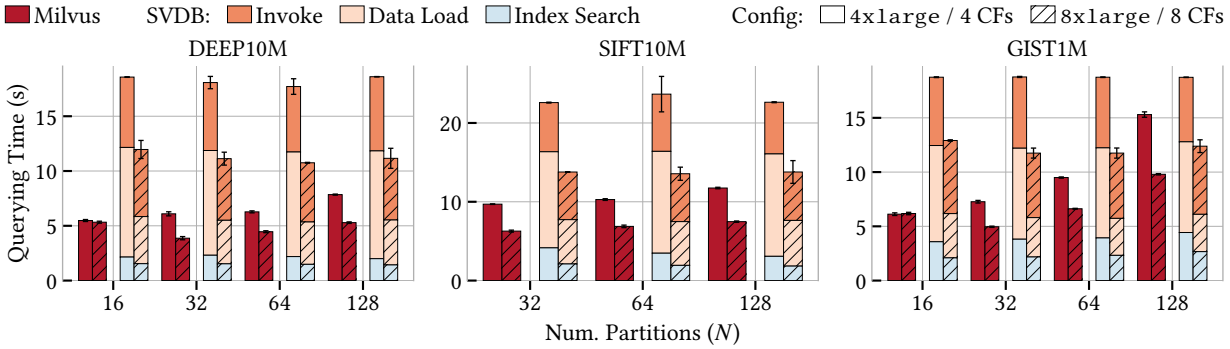


Figure 24: Querying times for 3 datasets on Milvus and SVDB. The plot splits the time of SVDB into cloud function invocation overhead (Invoke), index downloading (Data Load), and query execution (Index Search). Milvus has the query node running with the data loaded.

of a static dataset, resulting in under-utilization of available resources. Note that we experimented with deploying Milvus on different VM sizes (*e.g.*, `8xlarge`, `4xlarge`), but observed similar partitioning performance results. This may indicate some internal limitation in the indexing implementation of the system. Moreover, SVDB partitioning time is not sensitive to the number of partitions. Instead, Milvus processing time increases with the number of partitions.

Conclusion. SVDB achieves faster vector partitioning and indexing compared to Milvus for an equivalent amount of resources.

5.7.2 Query Performance

In this section, we evaluate the query performance of Milvus and SVDB in terms of querying time (Fig. 24) and recall (Table 10). As in the previous section, we evaluate query performance executing query batches (1000 vector queries/batch). For context, querying time reported for Milvus implies that the deployment is already set up and the data loaded into memory. However, reaching this state takes Milvus over 45 seconds which are not accounted in our results.

Fig. 24 compares the querying time in Milvus and SVDB for two equivalent resource configurations (*i.e.*, `4xlarge`/`8xlarge` VMs and 4/8 cloud functions). As expected, Milvus offers times 1.22× to 3.38× faster than SVDB. That is, Milvus is continuously running as a serverful service with the data loaded in memory, whereas SVDB requires to start the functions and load data on each query batch. We also observe that, in both systems, adding more resources reduces querying time. This implies that they are able to parallelize the execution of queries and fully utilize the underlying resources.

Inspecting the latency breakdown, the `index search` phase of querying in SVDB is faster than Milvus. However, querying also requires coordinating cloud functions, which is part of the `invoke` phase overhead. While part of this overhead is also present in Milvus for coordinating the queries, SVDB introduces additional function invocation latency. The `data load` phase is exclusive to SVDB. This overhead can be improved with stateful FaaS services, as proposed in Vexless [79]. Finally,

Table 10: Recall of top-10 similarity searches for a batch of 1000 queries in Milvus and SVDB.

Num. Partitions		16	32	64	128
DEEP	Milvus	99.27	99.43	99.38	99.28
	SVDB	99.24	99.26	99.44	99.30
SIFT	Milvus	-	99.80	99.70	99.60
	SVDB	-	99.37	99.41	99.50
GIST	Milvus	98.40	98.70	99.20	99.70
	SVDB	98.56	98.64	98.92	99.33

Table 11: Total cost of indexing and querying on Milvus and SVDB for the DEEP10M dataset across varying partition counts ($N = \{16, 32, 64, 128\}$, aggregated). Results for SIFT10M and GIST1M are similar. Q-Dense denotes 10 consecutive batches of 1000 queries; Q-Sparse-1 and Q-Sparse-24 spread the same batches over 1 hour and 24 hours, respectively.

	Indexing	**Q-Dense**	**Q-Sparse-1**	**Q-Sparse-24**
Milvus	\$10.71	\$0.07	\$1.43	\$34.39
SVDB	\$0.66	\$0.48	\$0.48	\$0.48

Table 10 shows that query recall is equivalent in both systems,[20] ensuring accurate vector searches.

Conclusion. Querying time in SVDB is slower than Milvus due to the expected overhead in function invocation and data loading. Query recall is virtually the same in both systems.

5.7.3 Cost analysis

Next, we focus on the economic cost of Milvus and SVDB. Table 11 shows that the indexing of the DEEP10M dataset is 16.2× cheaper in SVDB (\$0.66) compared to Milvus (\$10.71). A reason is that Milvus does not parallelize data indexing irrespective of the VM size. This induces longer processing time that translates into monetary cost.

Querying is more cost-effective on Milvus only if queries are executed as a dense workload (\$0.07 versus \$0.48 for running 10 batches of 1000 queries back to back). However, a serverful vector DB must always be running, leading to higher cost than a serverless solution for sparse workloads. For instance, if the 10 batches of queries are spread over an hour (`Q-Sparse-1` in Table 11), Milvus costs \$1.43, while SVDB remains at \$0.48. This cost difference increases with even sparser workloads. Also, the startup time for a Milvus deployment is much longer than for cloud functions ($\approx$ 45 seconds versus $\approx$ 2 seconds), making it impractical to start dynamically like SVDB, as it would severely impact query latency.

Conclusion. Compared to Milvus, SVDB enables faster indexing and on-demand query function allocation. This results in better cost-effectiveness, especially for sparse workloads.

5.7.4 Scalability

Finally, we aim to evaluate the scalability of SVDB, both in terms of data partitioning and as a system. To this end, we run both indexing and querying for two subsets of the DEEP1B dataset, containing 10 million and 100 million vectors. Accordingly, we also increase the amount of resources by 10× to process the latter. Specifically, we create 32 and 320 data partitions, indexed on 16 and 160 cloud functions, and use 8 and 80 functions for querying. Therefore, each partition is the same size in both configurations, and each function processes the same amount of data.

As expected, partitioning time is equal in both cases ($\approx$ 55 seconds) because it is an embarrassingly parallel process. Specifically, partitioning and indexing the DEEP100M dataset is about a sec-

[20] Note that SIFT has higher dimensionality compared to DEEP and cannot be processed with 16 partitions, so the experiments start at 32 for this dataset.

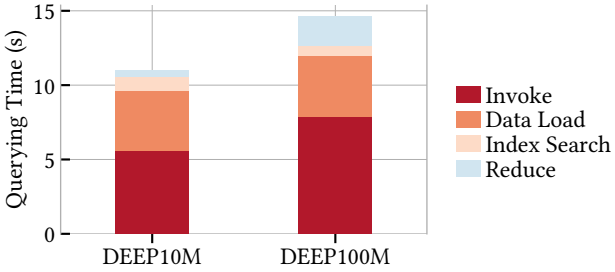


Figure 25: Querying time breakdown for DEEP10M (32 partitions in 8 CFs) and DEEP100M (320 partitions in 80 CFs).

Table 12: Cost comparison for DEEP10M and DEEP100M for total partitioning and querying with 10 batches of 1000 queries.

Dataset Size	Partitioning Cost	Querying Cost
DEEP 10M	$0.1477	$0.0527
DEEP 100M	$1.50	$0.52

ond slower due to the overhead of invoking more functions. Fig. 25 presents a breakdown for the querying phase. The plot shows that loading the indexes and searching takes the same time in both cases, demonstrating the scalability of the task. As expected, the overhead of invoking and managing more functions is higher (40%) and reduce time for collecting the results also increases (4×). The latter aspect can be explained because Lithops by default uses a single reduce function [92]. This implementation limitation can be solved by using multiple reducers.

Table 12 compares execution costs. The data partitioning process incurs a 10× higher cost due to identical execution time but a tenfold increase in resource usage. Querying scales similarly, as only the invocation and reduction phases grow —both of which have minimal impact on overall cost.

Conclusion. Our serverless vector DB with block-based data partitioning can scale the resources to arbitrarily large dataset sizes.

5.8 Related Work

Vector DBs, such as Pinecone [68], Weaviate [69], and Milvus [70, 71], have gained significant attention in recent years due to the increasing demand for efficient and scalable similarity search in various applications [103], including natural language processing [65], image recognition [66], and recommendation systems [67]. Nevertheless, most distributed vector DBs rely on traditional serverful architectures, which can be limiting when handling sparse and bursty workloads. The emergence of serverless vector DBs is a relatively recent development, and as such, none of the recent surveys in the field have provided specific coverage of this new family of systems [73, 81, 104]. This project aims to provide a timely and specific overview of the architecture of serverless vector DBs, including a systematic comparison with serverful counterparts.

The convergence of the serverless paradigm with vector DBs is the primary focus of NEARDATA. Recently, the industry has seen the emergence of vector DB services marketed as "serverless," such as Weaviate Serverless Cloud [69], Amazon OpenSearch Service as a Vector DB [75], and Upstash [74]. However, these services primarily aim to automate the provisioning of vector DB deployments. While simplifying operational complexity, this model only partially realizes the potential of a serverless architecture. A truly serverless vector DB system involves distributing the vector DB engine across cloud functions, a concept that has only been explored in Vexless [79]: the first serverless vector DB of this kind to date.

To our knowledge, this is the first experimental analysis that evaluates in depth the design space of data management in serverless vector DBs, with especial emphasis on data partitioning. We be-

lieve that the observations from our analysis can help drive new generations of serverless vector DBs to achieve better performance, efficiency, and cost-effectiveness.

5.9 Discussion and Conclusions

Our experiments validate the hypothesis that clustering-based data partitioning is generally impractical in stateless serverless vector DBs. First, data partitioning and indexing is slower compared to our block-based scheme because it requires a clustering stage, which is stateful and cannot be efficiently parallelized. Furthermore, when revisiting the state-of-the-art method for performing clustering-based data partitioning in a serverless vector DB (Vexless [79]), we found interesting insights. For example, using a balanced K-means clustering for achieving balanced data partitions incurs partitioning times that are 35× to 96× higher than unbalanced K-means. We also found that vector redundancy —a technique to improve query recall when filtering data partitions— has limited benefits (and multiple costs) beyond querying very few data partitions, which generally leads to low recall. When executing query batches in a stateless FaaS system, clustering-based data partitioning introduces additional complexity that does not benefit from data filtering —neither in query time nor cost— unless large batches are used. Based on these observations, we conclude that block-based data partitioning is a more practical and effective way of managing large and dynamic datasets in serverless vector DBs.

As a natural next step of our analysis, we compared our serverless vector DB prototype with a popular serverful vector DB (Milvus). Our results show that our prototype achieves 9.2× to 65.6× faster data partitioning and indexing time than Milvus due to its ability to fully exploit the parallelism of cloud functions. While querying time in our serverless vector DB is slower than Milvus due to the expected overhead in function invocation and data loading, this has a direct translation into economic cost. To wit, our serverless vector DB also offers better cost-effectiveness, especially for sparse workloads, as it does not require a service continuously running like Milvus. Overall, we believe that a serverless vector DB with block-based data partitioning offers an interesting alternative to serverful vector DBs, especially when considering sparse/bursty workloads and reduced infrastructure/operational cost.

Conclusion Serverless vector DBs are a promising architecture for managing sparse and bursty vector workloads while reducing operational cost. However, they are still in their infancy. We provide a timely overview of this new family of systems. Additionally, we analyze a key aspect of their operation: data partitioning. Through extensive experiments, we demonstrate that the current state-of-the-art approach for data partitioning (clustering-based) has significant limitations. To address them, we propose a simple yet practical block-based data partitioning scheme. Our findings show that a serverless vector DB with block-based data partitioning is competitive compared to a serverful vector DB (Milvus) in various aspects (*e.g.*, indexing time, query recall). We hope that the insights from this project will help driving better performance and efficiency for the next generation of serverless vector DBs.

6 XtremeHub Streams and FaaS: FaaStream

6.1 Introduction

Function-as–a-Service (FaaS) serverless platforms provide users with means to run functions at large scale while being oblivious to the underlying infrastructure [76, 105, 77]. The cost-effectiveness, scalability, and flexibility of the FaaS paradigm makes it a key component of modern application development in the cloud. Their popularity growth is undeniable, with services such as AWS Lambda and Azure Functions being fast-growing offerings in public cloud portfolios [106].

The advantages of FaaS platforms are driving their adoption for building *data-centric batch and streaming serverless pipelines* [107, 108]. It is increasingly common to find FaaS pipelines composed of multiple stages of cloud functions that deliver valuable insights. Ranging from real-time monitoring in manufacturing to genomics batch processing, serverless pipelines simplify access to large-scale cloud analytics.

However, despite abundant research in the last years, there is still significant progress to be made in fully harnessing complex serverless analytics pipelines in public FaaS platforms. Crucially, a key limitation of public FaaS platforms is the stateless nature of functions and the lack of direct inter-function communication. Although prior art has proposed custom serverless runtimes that leverage data locality and direct function communication [109, 110, 111], these solutions are constrained in their applicability to public FaaS platforms.

In this project, we focus on serverless analytics frameworks that rely on disaggregated storage for state management and indirect communication [112, 113, 114]. However, these storage services come with specific semantics (*e.g.*, object storage, in-memory key/value stores, logs) that complicate the efficient execution of heterogeneous serverless jobs (batch, streaming) within a unified engine. Consequently, while public FaaS frameworks simplify infrastructure provisioning, users are still required to determine the appropriate storage substrate for their workloads. This complexity undermines the programming simplicity that FaaS platforms aim to provide.

In a sense, serverless analytics are witnessing their own version of the "Lambda architecture" [115] problem[21], where each framework relies on different storage substrates for specific workloads. Interestingly, over the past decade, dataflow engines like Apache Flink [116] and Apache Spark [117, 118] have overcome this problem by providing a unified API for managing both streaming and batch workloads. These engines offer data primitives for building complex analytics jobs that have been crucial to their success. We believe that a similar approach can be explored in public FaaS platforms.

6.1.1 Challenges

In NEARDATA, we aim to provide a unified data processing framework for public FaaS that can efficiently support both *streaming and batch serverless jobs*. However, due to the limitations of public FaaS platforms, this entails challenges:

Identify the right storage abstraction. FaaS analytics frameworks that rely on external storage choose a specific system with a *subset of workloads* in mind. However, using some storage systems certainly restricts the kind of workloads that can be efficiently executed. For example, object storage (*e.g.*, AWS S3) provides high parallelism and throughput, but it offers sub-optimal performance when considering small data objects or when the workload at hand is latency sensitive [119]. Similarly, popular options like in-memory stores (*e.g.*, Redis) enable fast inter-function data sharing at the cost of data durability and scalability. We believe that there is still research to be done in storage substrates that support the efficient execution of heterogeneous FaaS pipelines.

Providing a unified set of data primitives. A unified data processing framework for public FaaS platforms should provide key data primitives to handle heterogeneous jobs. Inspired by cluster dataflow analytics, we believe that support for *inter-function data transfers*, *data shuffling*, and *state consistency* are crucial. The framework must build these primitives on top of shared storage and

[20]The content of this section maps to tasks T3.1 and T3.2 and is related to the paper "*FaaStream: Unifying Streaming and Batch Data Processing in Public FaaS Platforms*", submitted for publication.

[21]The Lambda architecture problem in analytics pipelines refers to the complexity of managing separate systems for batch and real-time processing.

expose them to users in a simple manner, abstracting away complexity and storage awareness from developers. This is non-trivial as the requirements to support such primitives can be disparate.

FaaS and storage coordination. Unlike cluster dataflow systems, public FaaS platforms are constrained by *unique limitations*, such as strict memory limits and the short-lived nature of tasks. Still, FaaS platforms also offer a significant advantage: their *extreme elasticity* enables them to handle workload burstiness and scale more efficiently than traditional cluster-based systems. To effectively support heterogeneous FaaS pipelines, a data analytics framework must be designed to work with the storage layer in a way that overcomes the limitations of FaaS, while also scaling storage resources in accordance with the elastic nature of cloud functions. Addressing this challenge can realize the full potential of data processing for public FaaS platforms.

6.1.2 Contributions

In NEARDATA, we introduce FaaStream: a stream-based serverless data processing framework. To our knowledge, FaaStream is the first serverless framework that builds upon *elastic and tiered data streams —i.e.*, data streams that change the number of stream partitions and offload cold data to external storage— as the storage substrate for FaaS pipelines. Tiered data streams provide an attractive latency-throughput trade-off that can meet the requirements of both streaming and batch serverless jobs. In summary, our contributions are:

- We identify elastic and tiered data streams as a powerful substrate for unifying data processing when executing heterogeneous pipelines in public FaaS platforms.
- We design FaaStream: a serverless data processing framework that unifies data processing for streaming and batch jobs. FaaStream builds key primitives on top of streaming storage such as inter-function data transfers, data shuffling, and consistent function state.
- FaaStream coordinates the parallelism between the data stream and the function pipeline, enabling transparent adaptation to workload bursts for the user.

The evaluation of FaaStream on AWS shows its effectiveness in achieving low-latency and high-throughput data processing for streaming and batch serverless pipelines, respectively. To wit, a FaaStream job can be executed in streaming and in batch with no code changes, while outperforming the analogous job using AWS Kinesis [120] in latency (80.99% lower p95 latency) and Lithops [113] in throughput (22.5×). FaaStream's coordinated auto-scaling enables efficient resource utilization under fluctuating workloads, which is not possible using alternatives like Apache Kafka for AWS Lambda. The FaaStream data shuffling primitive shows promising results, achieving execution times up to 25.80% faster than Serverless Spark [117], while outperforming Seer [121] by 20% in speed and delivering 6.25% better cost-effectiveness. Furthermore, FaaStream's failure recovery mechanism ensures function state consistency under failures. Our results highlight FaaStream's potential towards providing unified data primitives for heterogeneous pipelines in public FaaS.

6.2 Motivation: Stream-based FaaS Pipelines

Cloud functions on public FaaS platforms typically rely on external storage to manage inter-function communication and maintain function state (Fig. 37). Object storage has been the traditional choice for data-intensive serverless pipelines [113, 121] due to its high throughput, simple (and limited) API, and parallelism, but it struggles with workloads demanding low latency and fine-grained data access. In contrast, ephemeral or in-memory stores like Redis offer lower latency and have proven effective for tasks such as data shuffling [122, 111]. However, these systems face scalability challenges and do not meet the data durability needs of many critical use cases.

Interestingly, event streaming systems, such as Apache Kafka [87, 123], Apache Pulsar [124], and Redpanda [125], are popular technologies for ingesting and managing data in streaming fashion. These systems expose the *data stream* (*a.k.a.*, topic) abstraction for managing data events, internally composed of *partitions* or *segments* that enable parallelism for higher throughput. They offer support

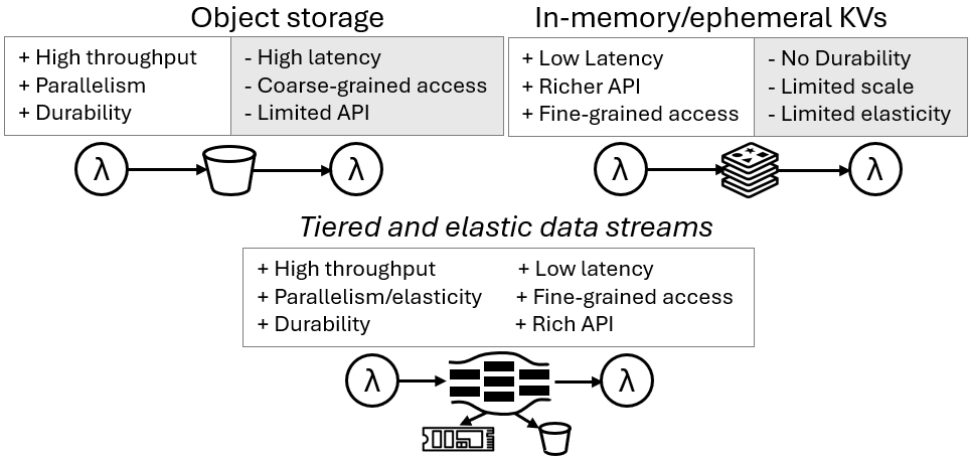


Figure 26: Storage abstractions trade-offs for FaaS pipelines.

for data durability, exactly-once semantics, and low-latency ingestion, while providing a scalable and fault-tolerant platform for high-volume data streams. Akin to the log abstraction [109, 126, 127], data streams naturally support inter-function communication. Serverless functions can also process data in a streaming manner, starting computation without waiting for previous results. Overall, we believe event streaming systems can be an attractive storage alternative for FaaS pipelines.

6.2.1 Key Insights: Not All Streams are Created Equal

While promising, not all event streaming systems offer the same features. Next, we identify the key insights that make *elastic and tiered data streams* ideal for FaaS pipelines (Fig. 37).

The data stream as unifying storage abstraction. To address the growing storage demands of streaming workloads, modern systems have adopted *tiered storage* as a key feature [128, 129]. Tiered data streams offload cold data from high-performance storage (*e.g.*, SSDs, NVMes) to scale-out storage (*e.g.*, object storage), thus bridging the gap between streaming and batch workloads. By leveraging a multi-tiered architecture, they offer *low-latency* access for real-time data processing while integrating with *high-throughput* storage for batch jobs. As for dataflow analytics [116, 118], this dual capability makes tiered data streams a powerful abstraction for efficient execution of heterogeneous serverless jobs (§6.3.1).

Rich data stream semantics for data primitives. Event streaming systems provide rich semantics that can be exploited for building data primitives in FaaS pipelines (§6.3.2). First, data streams are a natural abstraction to build *inter-function data transfers*. Moreover, most systems guarantee event order in parallel data streams on a per routing key basis. We identify that event routing can be exploited for custom *data shuffling* (§6.3.4). Similarly, event streaming systems typically provide transactions and checkpoints, which are crucial to deliver *function state consistency* (§6.3.5).

Data elasticity for FaaS pipelines. A unique characteristic of FaaS pipelines is that they are extremely elastic and exposed to bursty workloads. In this sense, some event streaming systems provide *data stream elasticity —i.e.*, dynamically change the number of parallel partitions— for handling workload fluctuations [129]. In our view, elastic data streams match the elasticity needs of FaaS pipelines (§6.3.3).

6.3 FaaStream Design

Next, we introduce FaaStream: the first serverless data processing framework built on top of elastic and tiered streams.

6.3.1 FaaStream Architecture and Life-cycle

FaaStream's architecture comprises three main layers: Control, Compute, and Storage. At the Control layer, the FaaStream Manager orchestrates the pipeline life-cycle. The Compute layer is formed by

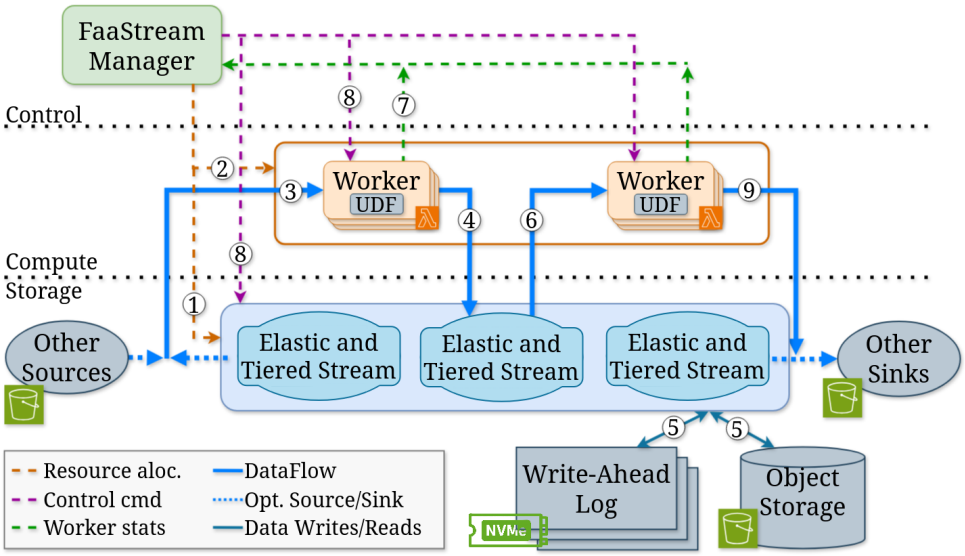


Figure 27: FaaStream architecture.

FaaStream workers executing pipeline stages. The Storage relies on FaaStream streams supported by an elastic and tiered streaming storage service. Figure 39 illustrates FaaStream's architecture and pipeline life-cycle.

The life-cycle of a FaaStream pipeline begins when a user submits a job. The FaaStream Manager first initializes the required resources in the public FaaS platform and the streaming storage system (1). For each pipeline stage, the Manager creates the input/output streams, sets the appropriate number of stream partitions, and sets up the worker coordination channels (*e.g.*, telemetry, worker management, etc.).

Once the resources are set up, the FaaStream Manager launches the workers (2). Each worker is assigned a subset of the input stream partitions (3), processes data executing user defined functions (UDF) (*e.g.*, `map`, `reduce`, `filter`), and writes the results to the output stream (4). These outputs are then consumed by the subsequent stage of the pipeline.

To adapt resource utilization, the Manager continuously monitors the pipeline workers. Workers periodically send telemetry data to the Manager (7), which uses it to dynamically adjust the number of workers per stage and scale the associated stream partitions accordingly (8).

All data in FaaStream streams is tiered for efficiency. Initially, data is written to a Write-Ahead Log stored on high-performance storage (SSD, NVMe) to enable low-latency inter-function communication. Then, data is asynchronously flushed to object storage for high-throughput reads (5).

Finally, the workers in the last stage of the pipeline write the results to the designated pipeline sink (9). Similar to the pipeline source, the sink can be a stream or any other backend supported by FaaStream (*e.g.*, S3 bucket).

6.3.2 FaaStream Abstractions and API

Inspired by the dataflow analytics model [118, 116], the FaaStream API supports both streaming and batch FaaS pipelines through a unified set of abstractions (see Table 13).

In FaaStream, the **Sources** define data ingestion inputs for a pipeline, supporting data streams (`fromStream`) or object storage (`fromS3File` and `fromS3Folder`), among others.

The **Pipeline** abstraction defines the sequence of data transformations in a serverless workflow. It provides core functions such as `map`, `reduce`, and `filter` to define individual stages of the pipeline. Users can specify the final destination for processed data, such as a stream (`writeToStream`) or an object storage (`writeToBucket`).

Each pipeline worker executes a UDF that processes data. UDFs are implemented by extending the **Transformer** interface. The Transformer initializes resources such as connections or internal state

Abstraction	Interface
Source	*fromStream(stream) -> Source* *fromBucketFile(bucket, file) -> Source* *fromBucketFolder(bucket, folder) -> Source*
Pipeline	*map(Transformer)* | *reduce(Transformer)* | *filter(input -> Boolean)* *writeToStream(stream)* | *writeToBucket(bucket, path)*
Transformer	*initialize()* | *apply(input) -> output* | *close()* | *@State*
ScalingPolicy	*addWorkerStats(stats)* | *getWorkers()* | *getPartitions()*
Manager	execute(*pipeline, scalingPolicy, exactlyOnce)*

Table 13: FaaStream API summary.

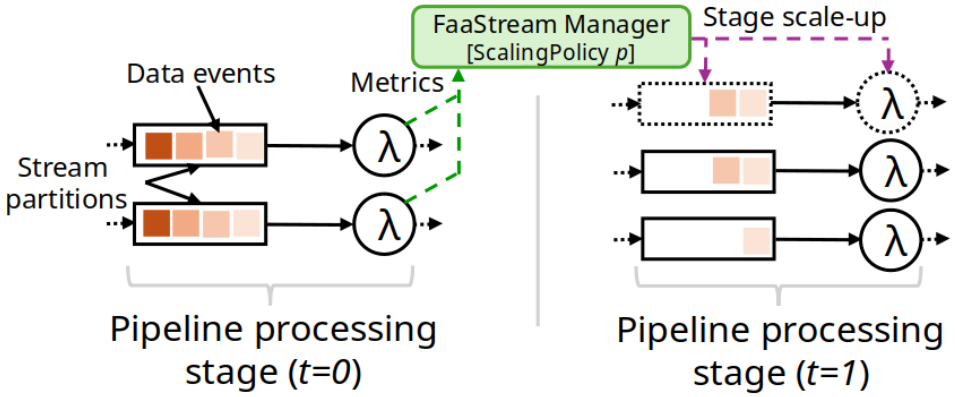


Figure 28: FaaStream's auto-scaling coordinates stream partitions and workers on a per pipeline processing stage basis.

using the `initialize` method and release them with `close`. The processing logic is defined in the `apply` method. Also, the API provides the `@State` annotation, which allows users to define the fields that form the state of the Transformer. This state is automatically managed by FaaStream, ensuring that it is persisted and recovered in case of failures when exactly-once is enabled (§6.3.5).

Scaling policies extend the **ScalingPolicy** class, using `addWorkerStats` to gather runtime data from each of the workers and *getWorkers* and *getPartitions* to determine scaling actions. FaaStream supports built-in scaling policies for coordinated auto-scaling (§6.3.3) and fixed workers.

Finally, The `execute` method in **Manager** submits a pipeline for execution, allowing optional configurations like scaling policies and function state consistency guarantees.

6.3.3 Serverless Pipeline Auto-Scaling

FaaStream builds a coordinated auto-scaling mechanism to dynamically adjust the number of workers and stream partitions in a pipeline stage based on workload demands (Fig. 28). This mechanism ensures efficient resource utilization while maintaining high performance under fluctuating workloads.

Each pipeline stage has a dedicated telemetry channel for workers to report their load metrics to the FaaStream Manager. Workers maintain an internal queue to buffer events awaiting processing, which also serves as an indicator of their load. When a worker reads an event from its assigned partition, it adds the event to this queue for processing.

Workers periodically report metrics, including queue size, events processed, processing time, and read time, to the Manager. The Manager aggregates these metrics to assess the overall load across all

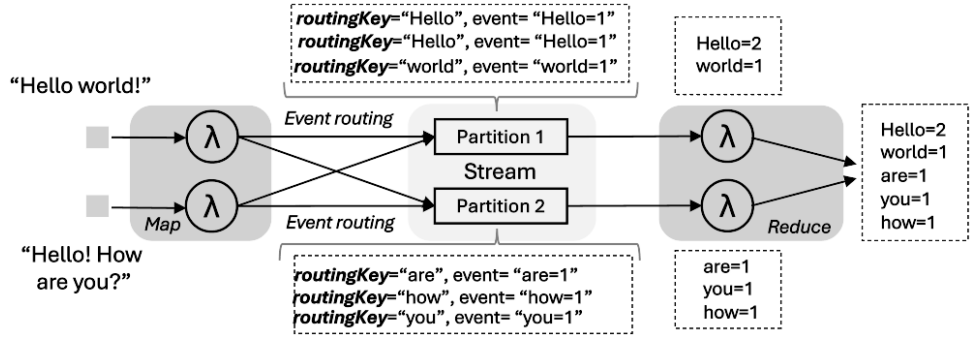


Figure 29: Data shuffling example via custom event routing. Using the words themselves as routing keys deterministically assigns them to stream partitions for reducers to read.

workers in the stage. Using this data, the Manager dynamically adjusts the number of workers and stream partitions, scaling up or down as needed to optimize resource utilization and maintain high performance.

The ScalingPolicy abstraction is responsible for defining the scaling strategy applied by the Manager. It provides methods to pass the gathered worker metrics and to retrieve the number of workers and stream partitions. This enables users to define custom auto-scaling strategies, from simple threshold-based policies (*e.g.*, scale workers by Y and partitions by Z if a metric exceeds X) to advanced approaches like time-series analysis or AI-driven workload prediction. FaaStream includes two built-in policies: a default threshold-based coordinated auto-scaling policy (evaluated in §6.4.4) and a static scaling policy for fixed resource allocation.

6.3.4 Data Shuffling via Custom Event Routing

FaaStream introduces a flexible data shuffling operator leveraging the key-based event routing mechanism in streaming systems. FaaStream allows users to specify a custom routing hash algorithm as an optional parameter when applying a Transformer (*e.g.*, `map`, `filter`). A routing hash algorithm in FaaStream is a function that maps an event to a decimal value $\in [0, 1]$. This value is then used to determine the target partition for the event. Choosing a routing algorithm directly influences how the shuffle operation is performed, enabling flexibility for addressing specific workloads.

For example, Fig. 29 shows how we can customize event routing to achieve data shuffling for a WordCount job. To wit, FaaStream can use a consistent hashing algorithm to route all events with the same "word" to the same partition. This guarantees that each worker in the reduce stage processes all events for a given word, enabling parallel execution of the reduce operation without extra function data exchanges.

As another example, in sorting jobs where event order must be preserved, the FaaStream data shuffling uses a routing hash algorithm that keeps the natural ordering of events. To it, it ensures that the i-th partition contains elements that are lower than those in the $(i + 1)$-th partition. This allows each worker in the sort stage to locally sort the elements in its assigned partitions, producing a globally sorted result at the end of the stage. We evaluate data shuffling in §6.4.5.

6.3.5 Consistent Function State under Failures

FaaStream provides consistent function state via exactly-once processing semantics that combine transactional writes and checkpointing. It leverages stream transactions to atomically commit or abort operations while maintaining consistent state across serverless functions. Each pipeline stage is treated as a transactional unit, coordinated by the Manager. The Manager ensures all workers checkpoint their state and read positions before committing a transaction. In case of errors, the Manager employs a health detector mechanism to monitor the status of each serverless function. If a failure is detected, it aborts the transaction and resets workers to the last checkpoint. The health detector ensures that the system can react promptly to failures. Users can enable exactly-once semantics

Algorithm 2 Exactly-Once Manager's logic in FaaStream

```
Require: Input/Output Stream, Workers, ReaderGroup
txn ← NewTxn(Output Stream)
while not end of Input Stream do
    Workers.UseTransaction(txn)
    WaitTxnWindow()
    nextTxn ← NewTxn()
    chkpt ← Workers.Checkpoint(txn, nextTxn)
    SaveCheckpoint(txn, chkpt)
    Commit(txn)
    txn ← nextTxn
    if error occurs then
        Abort(txn)
        r_txn ← LastTxn()
        r_chkpt ← GetChkpt(r_txn)
        Workers.Reset(r_txn)
        Workers.Restart(r_chkpt)
    end if
end while
```

Algorithm 3 Exactly-Once Worker Event Processing Loop

```
Require: Input Stream, Reader, Writer, Queue
while not end of Input Stream do
    data ← Reader.Read()
    if data is Event then
        Queue.Add(data)                    ▷ Data in queue is processed by the UDF.
    else if data is UseTransaction then
        txn ← UseTransaction.txn()
        writer ← CreateWriter(txn)
    else if data is Checkpoint then
        WaitQueueEmpty()
        Flush(txn)
        SaveState(txn, state)
    else if data is Reset then
        r_txn ← Reset.txn()
    else if data is Restart(r_chkpt) then
        state ← RecoverState(r_txn)
        Reader.Restart(r_chkpt)
    end if
end while
```

by configuring it in the PipelineManager, with FaaStream handling all underlying mechanisms.

Algorithms 2 and 3 outline FaaStream's exactly-once processing. The Manager (Algorithm 2) coordinates transactions by initiating a new transaction, forwarding it to workers, and waiting for a configurable duration for event processing. Workers checkpoint their state and read positions before the Manager commits the transaction. In case of failure, the Manager aborts the transaction and resets workers to the last checkpoint. The Worker (Algorithm 3) handles events and control messages. Events are queued for processing, while control messages like `UseTransaction`, `Checkpoint`, `Reset`, and `Restart` manage transaction initialization, state saving, and recovery to ensure failure recovery.

6.4 Evaluation

In this section, we evaluate FaaStream via experiments in AWS. Our evaluation addresses four key questions:

- Can FaaStream unify data access to streaming and batch jobs while achieving good performance? (§6.4.3)
- How does FaaStream achieve auto-scaling in heterogeneous serverless pipelines? (§6.4.4)
- How FaaStream data shuffling capabilities can optimize map-reduce serverless jobs? (§6.4.5)
- What are the data consistency trade-offs in FaaStream for serverless pipelines? (§6.4.6)

6.4.1 Implementation

FaaStream is implemented in Java (10K LoC), leveraging AWS Lambda and Pravega [129] for serverless functions and tiered streaming storage. The FaaStream implementation includes the worker (*i.e.*, IO, hashing) and the orchestrator (*i.e.,* auto-scaling, exactly-once protocol) functionality. To meet its requirements, FaaStream enhances Pravega's client with a collaborative segment distribution mechanism, ensuring faster convergence to an even distribution among readers. FaaStream also introduces a specialized reader for exactly-once processing, a custom stream writer for adaptive event routing, and optimizes Pravega's Segment Store for S3-based long-term storage (LTS). Finally, FaaStream integrates Crucial's serverless executor [130] for managing serverless function execution, providing a user-friendly API for invoking AWS Lambdas.

6.4.2 Experimental Setup

Deployment. All experiments are conducted in the AWS `us-east-1` region. Unless otherwise specified, AWS Lambda functions are configured with 1,769MB of memory, corresponding to 1 vCPU as per AWS documentation [131]. FaaStream deploys the Pravega cluster and AWS Lambda functions within the same VPC and availability zone to minimize latency and costs. A `m5.4xlarge` EC2 instance, equipped with 16 vCPUs and 64GB of RAM, serves as the client VM for running the FaaStream orchestrator. The Pravega cluster is deployed on `i3en.2xlarge` instances, each featuring 8 vCPUs, 64GB of RAM, and 2 dedicated NVMe drives, running Ubuntu 22.04. To simplify provisioning, each instance hosts a Pravega controller, a Pravega segment store, a Bookkeeper instance, and a Zookeeper instance. One NVMe drive is allocated for the Bookkeeper journal and the other for the Bookkeeper ledger and Zookeeper. For long-term storage, Pravega uses an S3 bucket via a VPC S3 endpoint.

Object storage/in-memory baselines. First, we evaluate FaaS pipelines executed in standard object storage and in-memory baselines compared to FaaStream.

S3-triggered Lambda: This baseline uses S3 as the intermediate storage, with AWS Lambda functions automatically triggered upon the creation of new objects in the S3 bucket. This setup represents the native AWS approach for serverless workflows. Note that AWS Step Functions were not used due to their 256KB data-passing limitation [132].

Lithops: We use the Lithops [113] serverless framework, a fork of PyWren [119], as one of our baselines. Lithops is evaluated with two storage backends: S3 and Redis (in-memory). The S3 bucket is located in the same AWS region as the Lambda functions to minimize latency. Redis is deployed on a `m5.4xlarge` EC2 instance (16 vCPUs, 64GB RAM) within the same VPC as the Lambda functions.

Seer: Seer [121], a state-of-the-art serverless shuffle manager, is used as a baseline for data shuffling benchmarks. For the TeraSort benchmark, we use the publicly available Seer implementation [133], configured to use its direct shuffle method. AWS Lambda functions are allocated 1,769 MB of memory, and an S3 bucket in the same region as the functions is used for intermediate storage.

Serverless Spark: EMR Serverless [134] is a serverless version of Apache Spark [118], allowing users to run Spark applications without managing infrastructure. It automatically scales resources based on workload. For consistency, we configure Serverless Spark with a fixed number of executors, each with 4 cores and 16GB of memory (4GB per core), to avoid performance degradation due to autoscaling.

Event streaming baselines. Moreover, we evaluate FaaStream using event streaming systems other than Pravega.

Kinesis: AWS Kinesis Data Streams is a fully managed real-time data streaming service. For this baseline, AWS Lambda functions read from and write to Kinesis streams. Kinesis streams are divided into shards, which determine throughput capacity. To ensure fairness, we use the `Provisioned` mode to control stream properties. Each shard costs $0.015 per hour. To match the hourly cost of an `i3en.2xlarge` instance ($0.904), we allocate 60 shards ($0.90 per hour) per VM instance used by FaaStream.

Kafka: We deploy a Kafka [123] cluster as another baseline for stream processing. AWS Lambda functions read from and write to Kafka topics. The Kafka cluster is deployed on `i3en.2xlarge` VMs

running Ubuntu 22.04. One NVMe drive is dedicated to the Zookeeper journal. For a fair comparison, we configure Kafka with durability guarantees equivalent to FaaStream, flushing data to persistent media before acknowledging writes (`log.flush.interval.messages=1`, `log.flush.interval.ms=0`), and data replication set to 1.

Benchmarks. *Embeddings generation*: We use an embeddings extraction job as a benchmark to evaluate CPU-bound jobs. The job consists of a single-stage pipeline that processes images from an input stream. Each frame is passed through a pre-trained deep learning model, ResNet-50 (trained on ImageNet), to extract feature embeddings. The extracted embeddings are then written to the output stream. For this benchmark, we use a subset of the ImageNet dataset [135], with each image having an approximate size of 35KB.

Hashing: To assess the performance of an I/O-bound workload, we employ a hash generation job as a benchmark. This task involves a single-stage pipeline that computes SHA-256 hashes for byte array events. The input data comprises raw bytes extracted from images in the ImageNet dataset [135].

WordCount: As one of the most widely used map-reduce jobs [136], WordCount serves as a cornerstone for understanding distributed data processing. Despite its simplicity, it is highly representative of a wide range of real-world map-reduce workloads [137, 138]. The task involves counting the occurrences of each word in a collection of text files. For this benchmark, we utilize a dataset comprising 120 text files from a Wikipedia dataset [139], totaling 4.3GB of text data.

TeraSort: The TeraSort benchmark [140] is a widely recognized standard for assessing the efficiency of data shuffling in serverless analytics frameworks [141, 121, 142, 143]. It involves sorting a large key-value dataset, where each record consists of 100 bytes: a 10-byte key and a 90-byte value. The dataset is generated randomly and subsequently sorted by the key. Following prior works, we evaluate using a 100GB dataset.

6.4.3 Unifying Streaming and Batch Data Access

In this section, we demonstrate that FaaStream unifies data access for streaming and batch serverless jobs while achieving good performance in a wide variety of scenarios.

Streaming IO. Streaming pipeline jobs demand low latencies to ensure timely data processing [144, 114, 145]. In this section, we assess the latency performance of FaaStream in comparison to the baselines in §6.4.2.

Impact of event size on latency: We first evaluate the latency overhead of transferring data across serverless functions. As outlined in [110], we measure the latency of a two-function chained pipeline across various data sizes. The comparison includes FaaStream (1 EC2 VM), a single-shard Kinesis stream, Lithops using either S3 or Redis (1 EC2 VM for Redis) as the storage backend, and Lambda functions triggered by S3 events. The results appear in Figure 30a.

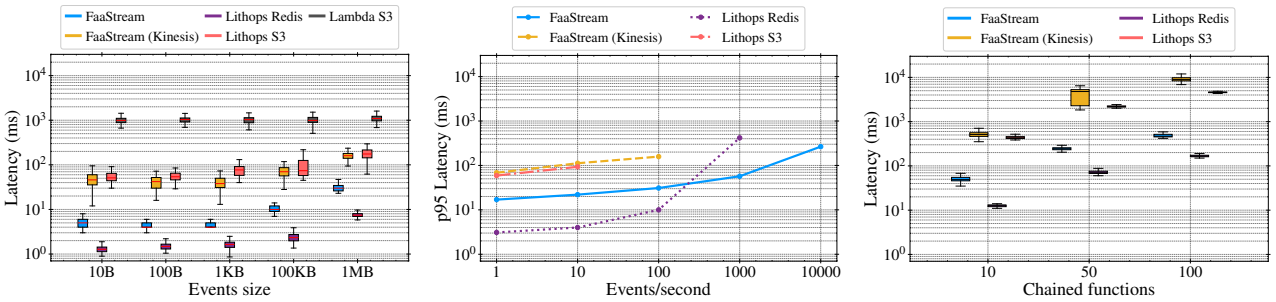
Visibly, for small event sizes (*i.e.*, $\leq$ 1KB), FaaStream exhibits consistently low latencies (*e.g.*, p95 from 11.0ms to 12.2ms). On the other hand, Lambda S3 (up to 127.8×), Kinesis (up to 6.3×), and Lithops S3 (up to 12×), which store data persistently, show higher p95 latency than FaaStream. Lithops Redis achieves p95 latencies from 2.05ms to 2.78ms, which is $\approx$ 4.75× lower than FaaStream. This is primarily because it stores data in memory rather than in storage.

For larger event sizes (100KB/1MB), FaaStream p95 latency also outperforms Lithops S3 (9.3 × /7.2×), Kinesis (4.4 × /4.9×), and Lambda S3 (65.6 × /35.6×). Again, compared to Lithops Redis, FaaStream is about 2.9 × /3.5× slower as it stores data on persistent storage rather than in memory. These results indicate that FaaStream achieves data durability and low IO latency for a range of event sizes, outperforming storage inter-function communication solutions like Kinesis and S3.

Throughput/latency trade-off: Next, we explore the latency behavior of FaaStream as we increase the IO throughput of the pipeline. We use the same two-function chained pipeline as the previous experiments with 1KB events (see Fig. 30b).

For latency-oriented workloads (*i.e.*, $\leq$ 100 events/second), Fig. 30b shows that FaaStream offers the lowest (p95) latency numbers compared to the other storage-based alternatives (Kinesis, Lithops S3). Only Lithops Redis using in-memory storage exhibits lower latency than FaaStream.

Interestingly, we observe that all the systems saturate between 10× to 1000× earlier than FaaS-



(a) Latencies for data passing between functions. Event rate: 1 event/s.

(b) Impact of event throughput in latencies. Event size: 1KB.

(c) Latencies of function chains of different lengths. Event rate: 1 event/s.

Figure 30: Impact on streaming latency of (a) inter-function data passing, (b) event throughput, and (c) function chain length.

tream as the workload becomes more throughput-oriented. That is, as Lithops lacks a batching mechanism for writes, using the S3 backend struggles to handle $\geq$ 10e/s. When switching to a Redis backend, we observe that Lithops can sustain up to 1K e/s. In this line, although Kinesis shards theoretically support up to 1K e/s, we empirically observed saturation at lower rates. This supports that using the stream abstraction in FaaStream allows us to develop batching algorithms that dynamically adapt to latency- and throughput-oriented workloads.

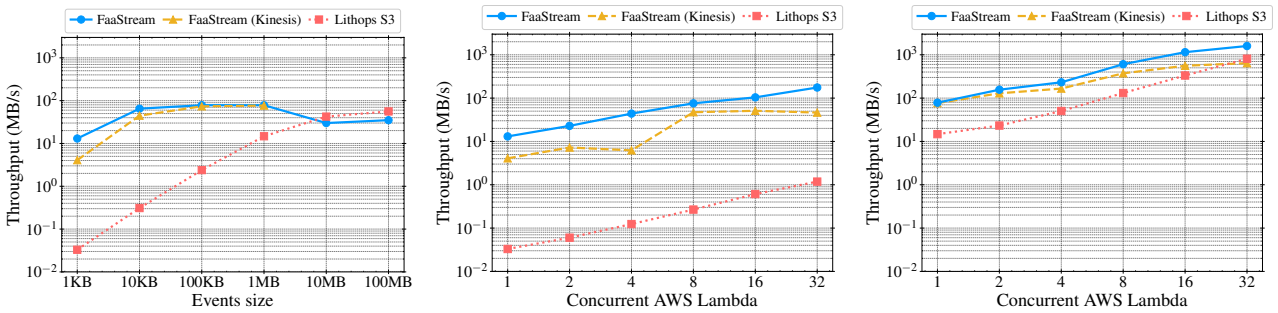
Function chains: Next, evaluate the latencies for pipelines with long function chains. Similar to [146, 110], in this experiment each function in the chain receives an integer, increments it by one, and passes it to the next function. We measure the end-to-end latency of the pipeline for different chain lengths. We used the same setup as in prior experiments.

As expected, Fig. 30c shows that FaaStream's end-to-end latencies grow linearly with the number of functions in the chain, as is also the case with Lithops S3 and Lithops Redis. However, this is not the case with Kinesis, where the p95 latency increases significantly, reaching 650 ms for 10 chained functions and 11,885 ms for 100 functions. Similarly to the previous benchmarks, FaaStream achieves latencies (*e.g.*, p95 of 552 ms for 100 chained functions) almost an order of magnitude lower than Kinesis (up to 21.5×) and Lithops S3 (up to 8.6×), while being slightly higher than Lithops Redis (up to 3.0×). These results support that FaaStream is an ideal substrate for building serverless pipelines. It ensures reliable data transfer of various sizes and rates with high performance across long function chains.

Batch IO. In contrast to streaming analytics, batch jobs typically demand high throughput. This section evaluates FaaStream's throughput performance against AWS Kinesis and Lithops with S3 when processing data in batch. The experiments assume data at rest (*i.e.*, not cached in memory). The dataset consists of raw byte array events. We use a single-stage pipeline whose functions read the byte arrays and produce timestamped events written to the output stream. We measure the pipeline's processing throughput in MB/s.

Impact of event size on throughput: We evaluate the effect of event sizes on the throughput of FaaStream compared to baselines in §6.4.2. In this experiment, we use a single concurrent AWS Lambda function and measure the batch processing throughput for various event sizes. FaaStream is configured with a single VM, while Kinesis is configured with 60 shards to match the cost of the FaaStream setup.

Figure 31a shows that FaaStream consistently achieves the highest throughput across all event sizes compared to Kinesis. For small events (1KB), FaaStream outperforms Kinesis by up to 3×. This performance gap is primarily due to Kinesis's limitation on the number of events that can be read per request. As the event size increases, FaaStream keeps its advantage, with both systems eventually converging near the AWS Lambda throughput limit ($\approx$ 80MB/s). Importantly, Kinesis



(a) Impact of event size on throughput using a single function.

(b) Batch throughput for different number of concurrent functions. Event size: 1KB.

(c) Batch throughput for different number of concurrent functions. Event size: 1MB.

Figure 31: Impact on batch throughput of (a) event granularity and throughput scalability for 1KB (b) and 1MB (c) events.

cannot manage events larger than 1MB.

Interestingly, when compared to Lithops S3, FaaStream exhibits significantly higher throughput for small events. This is because object storage systems are not optimized for small data access. For larger events, FaaStream's performance is comparable to S3. However, for very large events (10MB and 100MB), FaaStream's throughput slightly degrades compared to smaller event sizes (*e.g.*, 100KB or 1MB). This degradation is attributed to a known issue[22] in Pravega's S3 backend. We experimentally evaluated that this does not occur when using alternative long-term storage systems like AWS EFS.

Overall, our experiments highlight that using tiered data streams as a storage substrate for serverless pipelines can achieve high batch throughput for a variety of data sizes.

Throughput scalability: In what follows, we evaluate the batch throughput scalability of FaaStream as the number of concurrent AWS Lambda functions increases. Two event sizes are considered: 1KB and 1MB. The event processing throughput is measured for varying numbers of concurrent AWS Lambda functions (ranging from 1 to 32). FaaStream is deployed with a Pravega cluster consisting of 5VMs, while Kinesis is configured with 300 shards to match the hourly cost of the Pravega deployment. Figures 31b and 31c present the results for 1KB and 1MB events, respectively.

For both event sizes, FaaStream demonstrates the highest throughput, scaling linearly with the number of concurrent AWS Lambda functions. In contrast, Kinesis throughput saturates at 8 Lambda functions due to its dependency on the number of shards, which imposes a hard limit on its scalability. Interestingly, FaaStream also outperforms Lithops with S3 in both cases. While the superior performance for smaller event sizes was anticipated, FaaStream's ability to achieve up to 2× higher throughput for larger event sizes highlights its efficiency in handling high-throughput workloads.

Unified data access. Next, we evaluate FaaStream's capability to execute a serverless job seamlessly in both streaming and batch modes without requiring code changes. To this end, we use the *hash generation* job (see §6.4.2). We test two configurations: the first operates in streaming mode, generating hashes as data arrives at a rate of 100 images per second. The second runs in batch mode, processing all pre-written data. Listing 1 shows the job code using the FaaStream API.

We compare the FaaStream job in Listing 1 with the analogous implementations using Kinesis and Lithops S3. For both streaming and batch executions, we use 4 concurrent AWS Lambda functions. FaaStream is deployed using a single VM and the Kinesis stream is configured with 60 shards. In Fig. 32, the X axes measures latencies to compute the hashes in streaming and the Y axes the throughput of the batch job. Visibly, FaaStream achieves the highest throughput in batch mode —2.25× and 22.51× higher than Kinesis and Lithops S3— and the lowest latencies —80.99% and 86.02% lower p95

[22]This issue is being addressed by the Pravega community: `https://github.com/pravega/pravega/issues/7441`.

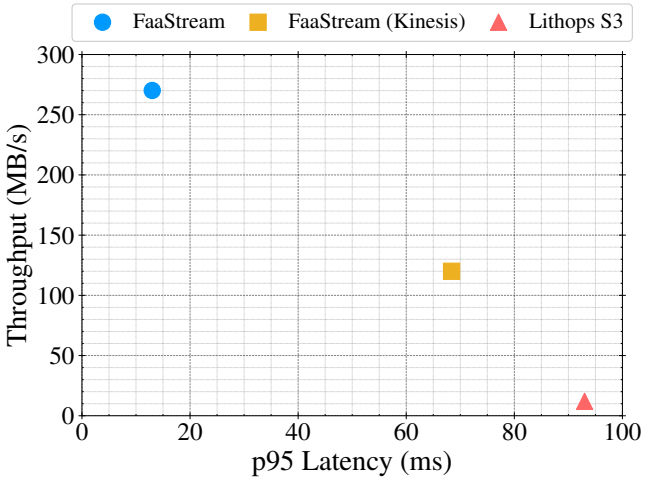


Figure 32: Unified data access performance. Event size: 35 KB, 4 concurrent AWS Lambdas.

```
Source<byte[]> source = Source.fromStream(input);
Pipeline<byte [], byte []> pipeline = source
        .map(new HashTransformer(), workers)
        .writeToStream(output);

Manager manager = new Manager();
manager.execute(pipeline, new StaticScalingPolicy());
```

Listing 1: FaaStream unified data access example.

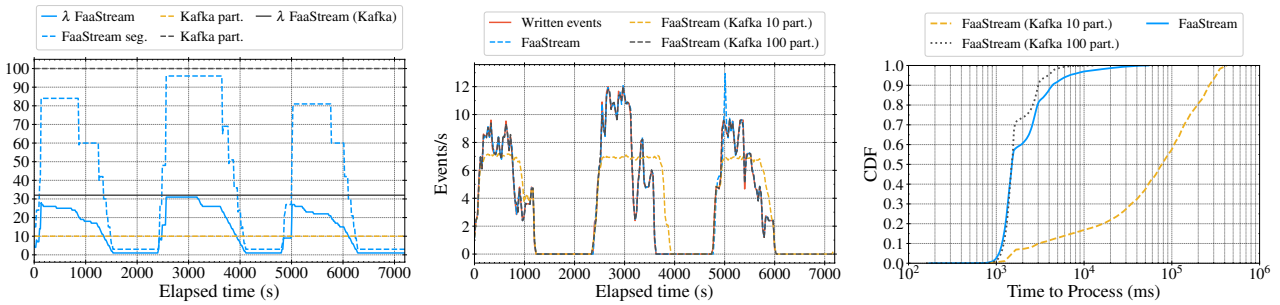
latency than Kinesis and Lithops S3— in streaming mode. This experiment supports the overarching design intent in FaaStream: a unified analytics API for both streaming and batch serverless pipelines on top of tiered data streams.

6.4.4 Coordinated Auto-Scaling

In this section, we evaluate the coordinated auto-scaling capabilities of FaaStream, which combines the inherent elasticity of FaaS with elastic data streams. To this end, we use real-world serverless job: a surgical video processing pipeline from the National Center for Tumor Diseases (NCT) in Germany [147]. Also, NCT provided traces describing the number of active operating rooms in a hospital with 1-minute granularity, which exhibits strong daily and weekly patterns. Using this trace, we generate an input stream of images with a configurable image-per-second rate that fluctuates over time. We replay these traces with two types of workloads: a CPU-bound and an IO-bound processing pipeline.

As a baseline for comparison, Lambda functions perform IO via Apache Kafka, a well-known event streaming platform in the industry. Since Kafka does not provide elastic data streams [148, 149], we use two different Kafka setups with a fixed number of topic partitions: 10 and 100 partitions.

FaaStream's coordinated auto-scaling algorithm (see §6.3.3) uses a 20-second wait period for both scaling up and scaling down. Scale-down operations occur when the average number of queued events per worker is below 0.40, terminating one worker at a time, ensuring there is always at least one worker. Scale-up operations happen when the average number of queued events per worker exceeds 1.95, increasing the number of workers by 2 to 3 times. The input stream segments are maintained at 3 to 4 times the number of workers. The queue length serves as an indicator of system load. For instance, a mean queue length of 0.5 suggests that the incoming rate is approximately 50% of the system's processing capacity, allowing for a reduction in the number of workers. Conversely, a mean queue length of 2 indicates that the incoming rate is roughly 200% of the processing capacity,

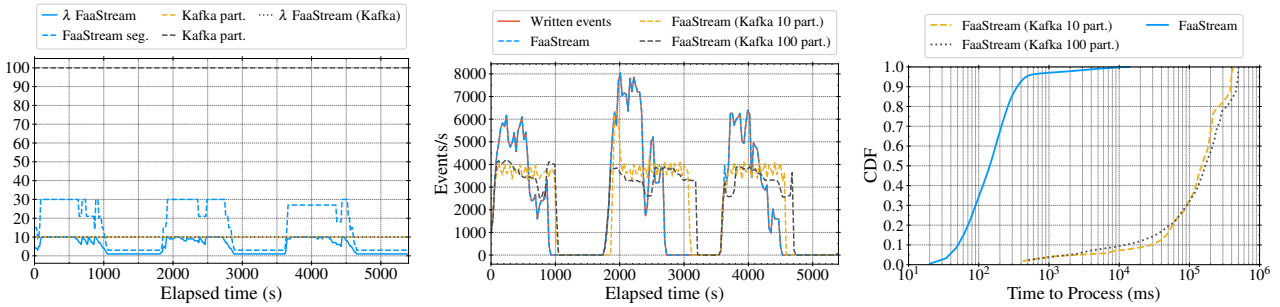


(a) Active AWS Lambda functions and segments/partitions for CPU-bound pipelines.

(b) Throughput for CPU-bound pipelines.

(c) Latency CDF for CPU-bound pipelines.

Figure 33: Coordinated auto-scaling for CPU-bound pipelines: (a) throughput, (b) active functions and partitions, (c) latency.



(a) Active AWS Lambda functions and segments/partitions for IO-bound pipelines.

(b) Throughput for IO-bound pipelines.

(c) Latency CDF for IO-bound pipelines.

Figure 34: Coordinated auto-scaling for IO-bound pipelines: (a) throughput, (b) active functions and partitions, (c) latency.

necessitating a doubling of the workers. The scale-up and scale-down thresholds are configurable, and in our setup they are set to 0.40 and 1.95, respectively. During the experiments, we monitored that the distribution of segments/partitions was uniform among workers for both FaaStream and Kafka.

CPU bound pipelines. In this experiment, we use the embeddings generation job detailed in §6.4.2, which is a CPU-bound job. We emulate a three-day period of the NCT trace in two hours. AWS Lambdas are set with 4GB of memory.

Fig. 33a illustrates the resource utilization for the serverless pipeline using FaaStream and Kafka. For the Kafka setup, we maintained a constant number of 32 Lambda functions running continuously and consuming data from the topic. Instead, FaaStream uses the aforementioned auto-scaling algorithm, with a maximum of 32 and a minimum of 1 AWS Lambda functions. Note that dynamic scaling in FaaStream involves starting and stopping Lambda functions, which may incur in cold start penalties. Fig. 33b shows the throughput of the pipelines. Visibly, the Kafka setup with 10 topic partitions achieves $\approx$ 40% lower throughput compared to FaaStream in peak periods due to the constrained partition parallelism. In Kafka, each topic partition is consumed by at most one reader at a time (*i.e.*, they cannot be processed in parallel). This limitation is problematic for CPU-bound serverless jobs, which require higher partition parallelism to distribute the load across more Lambda functions. This causes event queuing that explains the higher latencies observed in Fig. 33c.

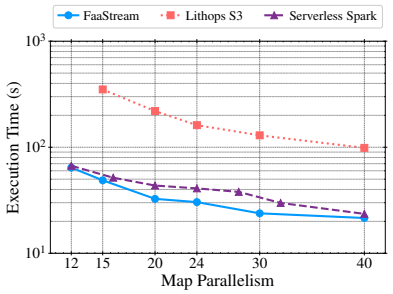


Figure 35: WordCount execution time (4.3GB of data, 120 files) for various levels of parallelism.

System	Map/Reduce Workers	Exec. time (s)
Seer	**200**	**123.28**
	225	129.66
	250	136.28
Serverless Spark	**152**	**77.72**
	176	85.66
	200	83.607
FaaStream	150	104.16
	175	**98.16**
	200	123.24

Table 14: Execution time for 100GB TeraSort benchmark.

System	Workers	Shuffling cost ($)	Perf./cost (GB/s/$)
Seer	**200**	**0.216**	**3.75**
	225	0.273	2.82
	250	0.338	2.17
FaaStream	150	0.261	3.670
	175	**0.254**	**4.00**
	200	0.360	2.25

Table 15: Shuffling costs for 100 GB TeraSort benchmark.

Moreover, Fig. 33b indicates that FaaStream achieves the expected ingestion throughput, as for Kafka with 100 topic partitions. Fig. 33b shows that FaaStream auto-scaling only incurs a small latency penalty (*i.e.*, $\approx$ 3 seconds at p95) due to cold function startup. Crucially, FaaStream achieves such a performance while using fewer resources than Kafka with 100 partitions and 32 active Lambda functions.

IO-bound pipelines. We use the hash generation job from §6.4.2 as an IO-bound serverless job. To increase the IO load, we multiply by 1000$\times$ the rate of images generated with respect to the previous experiment and reproduce the 3-day NCT trace period in 1.5 hours.

Fig. 34a shows a similar resource utilization pattern as the previous experiment: with Kafka, partitions and Lambda functions are statically provisioned, whereas FaaStream adapts stream segments and functions to workload fluctuations.

More importantly, we observe significant performance differences between using Kafka and FaaStream for running IO bound jobs, both in terms of throughput (Fig. 34b) and latency (Fig. 34c). Specifically, we observe that FaaStream achieves a throughput 2$\times$ higher than Kafka and a time-to-process latency orders of magnitude lower. Even worse, Kafka with 10 topic partitions achieves a slightly higher throughput and lower latency than the setup with 100 topic partitions. This may be due to the fact that we set Kafka with strict data durability. As Kafka uses a log file per topic partition, an increasing number of partitions results in a loss of performance due to concurrent synchronous writes [129]. This indicates that not all event streaming are equally equipped to serve serverless pipelines in terms of parallelism.

6.4.5 Stream-based Data Shuffling

Data shuffling, which redistributes data among workers, is an essential primitive for many data processing pipelines. In serverless computing, this process is particularly important due to the absence of direct communication between functions, thus requiring an intermediate communication system. In this section, we evaluate the performance of FaaStream with jobs that require data shuffling. To this end, we use two well-known benchmarks: WordCount and TeraSort. These benchmarks are representative of common data processing patterns [137] and are widely used in the literature to evaluate the performance of data shuffling systems [141, 121, 143]. We compare the performance of FaaStream with two state-of-the-art systems: Lithops [113] and Serverless Spark [134, 117].

Word Count. For WordCount, we implemented in FaaStream a custom shuffling algorithm that uses the `MurmurHash3` algorithm as the routing hash for each word (see §6.4.5). This allows us to distribute the words consistently across the segments. We configured the Lithops' reduce function with the maximum memory available in AWS Lambda (10,240 MB) to avoid memory pressure, as Lithops only supports a single reduce function. FaaStream uses a single VM instance.

Figure 35 shows the execution time of the WordCount benchmark as a function of the level of parallelism in the Map stage. We observe that FaaStream outperforms Lithops by nearly an order

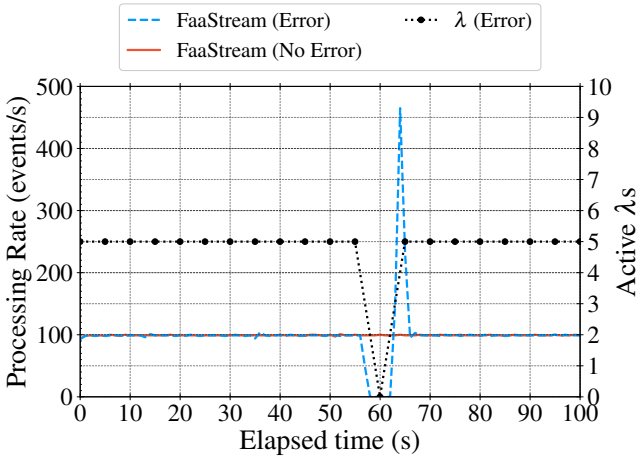


Figure 36: Exactly-once failure recovery.

of magnitude. This performance gap is expected due to the single reduce design in Lithops. Interestingly, FaaStream is also slightly faster than Serverless Spark (up to 25.80%), despite the latter performing the shuffle in memory while FaaStream's data shuffling involves disk operations. This advantage is a direct result of data pipelining enabled by tiered data streams, which allows FaaStream to overlap the map and reduce stages effectively.

TeraSort. We used Seer [121] and Serverless Spark as baselines for TeraSort. Our FaaStream shuffling implementation uses a locality-based hashing algorithm which respects the lexicographical order of the keys. This allows us to distribute the keys evenly across the segments while maintaining their order, ensuring that each segment contains a contiguous range of keys. Both FaaStream and Seer Lambdas are configured with 4GB of memory and FaaStream used 10 VMs.

Table 14 shows the execution time of the TeraSort benchmark for different configurations of Seer, Serverless Spark, and FaaStream. Note that we were not able to execute Seer with fewer than 200 workers due to Lambda functions reaching their memory limit. Results in Table 14 indicate that FaaStream is 20% faster than Seer. It is also noteworthy that FaaStream is only 20 seconds slower than Serverless Spark, which performs shuffling in-memory (*i.e.*, no data durability). Additionally, for the Serverless Spark execution time, we only considered the time elapsed from the job start to its completion, excluding the scheduling time for the job. If the scheduling time was to be considered, the execution time for Serverless Spark would be 174 seconds, which is a 77% higher execution time than FaaStream. These results support the design decision of using tiered data streams as a storage substrate for data shuffling. Also, the FaaStream data shuffling primitive can adapt to various shuffling algorithms.

Shuffling costs. Next, we also compare the intermediate storage costs used by the serverless functions to perform data shuffling in TeraSort. For Seer, we measure the cost of the S3 requests performed and for FaaStream we measure both the S3 cost and the cost for the EC2 cluster used. We excluded Serverless Spark from this comparison because its shuffling operation is performed in-memory and does not require intermediate storage. Along with the shuffling cost, we also measure the performance/cost ratio of the shuffling operation, which takes into account the performance of the shuffling operation in terms of GB/s and the cost of the shuffling operation, expressed as a GB/s/$ ratio.

Table 15 shows the USD cost of the shuffling operator for the different configurations of Seer and FaaStream. We observe that FaaStream's shuffling cost ($0.254) is only slightly higher than Seer's shuffling cost for the best configuration ($0.216). However, when it comes to the performance/cost ratio, FaaStream achieves a ratio of 4.00GB/s/$, which is 0.25 GB/s/$ higher than Seer's best configuration (3.75 GB/s/$). These results demonstrate that FaaStream's shuffling cost is competitive with state-of-the-art serverless analytics frameworks while providing a better performance/cost ratio.

6.4.6 Stateful Pipelines upon Failures

Finally, we evaluate FaaStream's failure recovery mechanism for keeping exactly-once state consistency guarantees under failures. To this end, we execute two versions of a single-stage serverless job configured with exactly-once semantics. The first version introduces a failure in all workers at a specific time, while the second proceeds normally without interruptions. The job processes a stream of integers, computes their cumulative sum, and writes the results to an output stream. We configured the job with 5 concurrent workers, processing a steady input stream of 100e/s.

Figure 36 presents the results. Around second 58, a failure is triggered, causing all workers to halt. FaaStream detects the failure and completes the recovery process in ≈ 5 seconds. This includes failure detection, relaunching the workers, restoring their checkpointed state, and resuming processing from the last checkpoint (see §6.3.5). After recovery, the system seamlessly continues processing events, with no data loss or duplication, maintaining exactly-once guarantees.

Framework	Public FaaS	Storage abstraction	Workload type	Data elasticity	Shuffling	State consistency
FaaSFlow [111]	No	Shared memory and remote store	Batch	No	No	Atomic consistency for individual steps
Boki [109]	No	Shared Log	Batch	Yes	No	Exactly-once
MinFlow [150]	No	Shared memory and Object Storage	Batch	No	Yes	N/A
Pheromone [110]	No	Shared memory and durable KV store	Batch, Streaming*	No	Yes	Not specified
SPCS [112]	Yes	Dynamo DB and SQS	Streaming	No	No	At Least Once
Lithops [113]	Yes	Object Storage	Batch	Yes	No	N/A
Seer [121]	Yes	Object Storage	Batch	Yes	Yes	N/A
Sponge [114]	Yes	No intermediate storage (VMs for its router operators).	Streaming	Yes	Yes	Exactly-once
FaaStream	Yes	Tiered data streams	Batch, Streaming	Yes	Yes	Exactly-once

* Batched streaming jobs.

Table 16: Related work comparison with FaaStream.

6.5 Related Work

Next, we review the related work. For clarity, Table 16 outlines the most relevant systems categorized by key dimensions.

Serverless frameworks. First, we find serverless frameworks designed for public clouds that use shared storage. ExCamera [151] uses thousands of AWS Lambda functions to process video streams, using a rendezvous server for inter-function communication. gg[152] is able to launch thousands of parallel functions, using external storage like S3 or Redis. It also proposes direct function communication via NAT-traversal, a technique furthered studied by FMI [153]. PyWren [119], Lithops [113], NumpyWren [154], and Sprocket [155] rely on object storage for intermediate data. Kappa [156] manages cloud function timeouts with checkpointing and uses FIFO queues on a coordinator VM for inter-function communication. Crucial [130] uses distributed shared memory for synchronizing function shared state. On the other hand, we find ad-hoc serverless runtimes with integrated storage or communication mechanisms. Pheromone [110] uses in-place shared memory object store, enabling zero-copy data sharing and direct inter-node communication. FaaSFlow [111] implements a shared memory storage for co-located functions. Cloudburst [157] uses Anna [158] with a mutable cache for stateful functions. SONIC [159] optimizes data exchanges with local storage, direct-passing between VMs, and remote storage. While effective, these works rely on controlling the runtime, limiting their applicability in public clouds.

Storage abstractions. Previous works propose various storage abstractions for intermediate data and state management. Pocket [122] and Jiffy [160] propose elastic storage systems, with Jiffy optimizing resource allocation for analytics workloads. Shredder [161] and Glider [162] enable lightweight computations within the storage layer to reduce data movement. Shared log abstractions, such as Beldi [126] and Boki [109], ensure fault tolerance and transactional consistency, with Boki co-locating its LogBook engine for low-latency state management. IndiLog [127] further enhances shared log scalability. Unlike these, FaaStream uniquely employs elastic and tiered data streams as a unified storage abstraction.

Workload types. Several frameworks have been proposed to address specific workload types in serverless environments. Flint [163] introduces a Spark execution engine for AWS Lambda, enabling batch data analytics jobs on serverless. SplitServe [164] extends Apache Spark to use both VMs and cloud functions, leveraging HDFS-based storage for data shuffling. For streaming workloads, Styx [165] focuses on enabling stateful transactions in serverless streaming jobs. Flock [166] presents a streaming query engine tailored for serverless environments. SPCS [112] proposes a stream computing framework built atop serverless architecture. Pheromone [110] supports both batch and streaming workloads, but relays on batched streaming for the latter. Durable Functions [167] introduces an actor-based model to enable stateful serverless workflows. Similarly, Dirigo [168] proposes a stream processing service built atop actors. Apache Flink Stateful Functions [169] extends Flink with an actor model for stateful serverless applications. However, it is not designed for data analytics workloads and lacks key features like programmatic abstractions or a shuffling operator. FaaStream uniquely supports both batch and streaming workloads programmatically and in terms of infrastructure, with elastic streams handling fluctuating ingestion workloads.

Data shuffling. The serverless data shuffling problem has been addressed by several previous works. Lambada [170] introduces a shuffle operator using object storage with a multi-level exchange pattern. Seer [121] optimizes the object storage exchange pattern used for faster execution. Locus [141] adds shuffle support to PyWren, combining object storage and Redis. MinFlow [150], optimizes the communication topology also combining shared memory and object storage. These works emphasize the need for an optimized serverless shuffling using external storage. FaaStream provides a novel data shuffling primitive by exploiting adaptive event routing on top of data streams.

State consistency. Multiple works focused on managing state in serverless pipelines. SPCS [112] provides at-least-once guarantees for streaming workloads, while Boki [109] ensures exactly-once guarantees for batch jobs. Sponge [114] and Styx [165] extend exactly-once semantics to streaming workloads. Similarly, FaaStream allows deploying stateful functions, delivering exactly-once guarantees for streaming workloads on top of tiered data streams.

6.6 Conclusions

In this deliverable, we introduced FaaStream: a serverless framework that unifies streaming and batch data processing in public FaaS platforms. FaaStream leverages elastic and tiered data streams to support both batch and streaming workloads at the infrastructure level, while exposing a set of unified data primitives at the programming level. Our evaluation shows that FaaStream achieves low-latency and high-throughput inter-function data transfers, adaptability to fluctuating workloads, flexible data shuffling, and exactly-once function state consistency under failures. FaaStream paves the way for freeing developers from infrastructure-related decisions when building heterogeneous FaaS pipelines in public clouds. Moreover, our results may incentivize cloud providers to expose elastic and tiered data streams as a cloud-native storage abstraction FaaS platforms.

7 XtremeHub Security and Streams

Another path of convergence in XtremeHub encompasses streams and security. In particular, we worked on a compelling demonstration of NEARDATA's cutting-edge confidential computing capabilities, showcasing how Trusted Execution Environments (TEEs) can be effectively applied to real-time data streaming with an affordable performance impact for some use cases. This *section serves as a summary of the work done to combine security and low-latency stream processing in deliverable D4.2*.

7.1 Secure Streaming in Action

Using Pravega as the streaming backbone and SCONE as the confidential computing runtime, the NEARDATA team evaluated "sconified" clients running inside Intel SGX enclaves. These clients were benchmarked against standard ones across 24 test cases, varying throughput (100–10K event/sec) and payload sizes (100B–100KB). The results reveal a nuanced performance landscape:

- At low throughput, secure clients show higher latency ($\approx 2\times$), as expected due to enclave overhead.
- At medium to high throughput, the gap narrows significantly, with secure clients approaching parity.
- At peak throughput (10K events/sec, 100KB payloads), secure clients even outperform standard ones—thanks to SCONE's optimized threading and memory management.

7.2 Summary

These results show that, even though the expected latency impact from using Scone in streaming clients, it may be justified in some health applications that require high security and confidentiality standards. In domains like surgical AI and metabolomics, where data sensitivity is non-negotiable, NEARDATA proves that confidential computing is no longer a bottleneck. The platform enables real-time analytics with strong security guarantees, making it ideal for federated learning scenarios where privacy, compliance, and performance must coexist.

In D4.2, we report the full report for this analysis. The document also dives deeper into the Data Broker's architecture, orchestration mechanisms, and integration with advanced security components like SinClave, CRISP, LLD, and Revelio.

[22]The content of this section maps to tasks T3.2 and T4.4 and is related to the paper "*Understanding the Latency-Security Tradeoff: TEE-based Confidential Computing for Streaming Workloads*" published in CEC@IEEE ICNP'25.

8 XtremeHub Stream Connectors: Nexus

8.1 Introduction

Continuous data sources—*e.g.*, IoT, sensors, server logs—generate an influx of data that needs to be ingested, processed, and stored reliably and with high performance. Typically, continuous data sources generate *data events*, which may contain unstructured contents ranging from 1 byte to multiple MBs. Managing data events generated by a source requires the system to keep order and consistency. This motivated the adoption of the *data stream* abstraction (*a.k.a.*, "topic") as the foundation for managing data from continuous data sources.

In the last decade, multiple event streaming systems have emerged to expose the data stream abstraction. For example, systems like Apache Kafka [123], Apache Pulsar [124], Redpanda [125], or Pravega [171] implement streams with key guarantees, such as durability, event order, and high performance, among others. Event streaming systems also provide connectors for analytics engines —*e.g.*, Apache Flink [116], Apache Spark [118]— to become a powerful data sink/source in production-grade stream processing pipelines [172, 173].

More recently, however, the industry raised the need of storing stream data for the long term and in a cost-effective manner (*e.g.*, batch analytics, AI training, auditing) [174, 175, 176]. Augmenting event streaming systems with long-term storage capabilities is a major challenge, with profound design implications in some cases [177]. And yet, in recent years, we have witnessed most event streaming systems providing *storage tiering* [178, 179]: the ability to move cold stream data to an external storage service (*e.g.*, AWS S3). Storage tiering for data streams allows the system to achieve a sweet spot in the infrastructure costs and performance trade-off [86].

8.1.1 Motivation: Beyond Tiered Data Streams

In general, the storage tiering process for data streams is quite simple: it consists of moving chunks[23] of stream data to/from external storage via APIs. When a chunk —*e.g.*, a few to 100s of MBs— of stream data is considered cold, a server of the event streaming system triggers a storage operation to offload it to external storage. Once stored remotely, chunks of stream data can be safely removed from the high-performance, expensive drives used to ingest events with low latency.

Crucially, we identify that tiered data streams open up an opportunity for advanced data management that remains largely unexplored beyond simple storage offloading. For instance, we could run *functions* on chunks of tiered stream data that route them to specific storage based on their contents, perform custom data transformations, or proactively prefetch/cache data for performance purposes, to name a few. Ideally, this would be *complementary and independent* to the stream processing services that event streaming systems already offer (*e.g.*, Kafka Streams API, Flink jobs). In particular, applications could continue performing stream processing at the *data event granularity* with low latency, while data management would be executed offline on *stream data chunks*.

A simple example of the opportunity we explore in NEARDATA is illustrated in Fig. 37 (details in §8.7). In Fig. 37, we show different approaches to apply data compression on a Kafka stream: i) *client*, ii) *broker*, and iii) *storage tiering* (broker). For each compression approach, we evaluate the compression ratio and write latency for various write event rates and batch sizes via OpenMessaging Benchmark [180]. Visibly, when applying compression at the data event level, there is a trade-off between compression ratio and write latency. This is natural, as larger event batches provide better compression opportunities at the cost of needing more time to complete. Note that this happens irrespective of whether data event batches are compressed at the client or broker sides. Instead, by applying compression on chunks of tiered stream data, we keep event write latency unaffected while achieving 2.6× better compression ratios than compressing data at the event level, as stream data chunks are even larger than event batches.

A key observation is that managing chunks of tiered stream data can provide benefits beyond

[22]The content of this section maps to tasks T3.2 and T3.5 and is related to the paper "*Nexus: A Data Management Mesh for Tiered Data Streams*", submitted for publication.

[23]In this project, the term "chunk" or "data chunk" refers to the units of stream data offloaded to external storage.

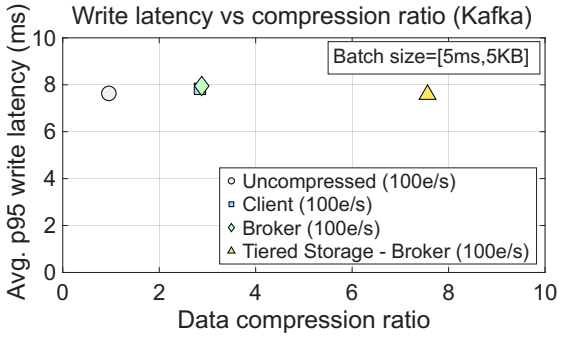


Figure 37: Write latency vs data compression trade-offs in enforcing data compression in Kafka at different points of the write path (*i.e.*, client, broker, and storage tiering).

data reduction. However, in the experiments above, we used the data compression functionality built in the storage tiering binding itself [181]. As event streaming brokers are already complex systems, attempting to build a general-purpose data management framework on top of a storage tiering binding may be inflexible and hard to maintain. This calls to rethink data management for tiered data streams as an independent service.

8.1.2 Data Management Challenges

Our goal is to enable *transparent and flexible in-transit data management* for tiered data streams. This can be achieved by intercepting storage requests for stream data chunks and applying user-defined functions, all without impacting existing event streaming systems. This approach empowers administrators to orchestrate rich data management pipelines, thereby adding value to the stream tiering process.

Still, realizing this vision entails challenges: i) *Decoupling*: Event streaming and data management should be clearly separated concerns. A strong reason supporting this decision is that event streaming systems are performance-sensitive and complex already. Thus, adding advanced data management capabilities to them could negatively impact their maintainability and performance. ii) *Transparency*: Ideally, we should perform data management on tiered data streams without the event streaming system noticing. Addressing this challenge would also benefit not just one, but multiple event streaming systems. iii) *Extensibility*: Data management is a quite broad term [182]. As such, we should provide users with means to develop a variety of data management functions for optimizing storage flows related to tiered data streams. iv) *Heterogeneity*: As event streaming systems may be deployed at the Edge and/or the Cloud, a data management solution should also be aware of such heterogeneous infrastructures.

8.1.3 Contributions

We present `Nexus`, the first system to address the data management gap between event streaming systems and external storage services. `Nexus` transparently intercepts storage operations from tiered data streams via standard APIs and executes in-line data management functions (*streamlets*) on chunks of stream data. These functions are orchestrated through *policies* and executed across heterogeneous infrastructures using clusters of workers (*swarmlets*), enabling extensible and location-aware data management. `Nexus` supports both stateless and stateful streamlets. Moreover, `Nexus` introduces *mesh-like data routing* to transparently manage streamlet execution across infrastructures. Our key contributions are:

- This is the first work addressing the data management gap between streaming systems and external stores.
- Design of `Nexus`, a system with streamlets, swarmlets, and policies for composable and location-aware in-transit data management of tiered data streams.

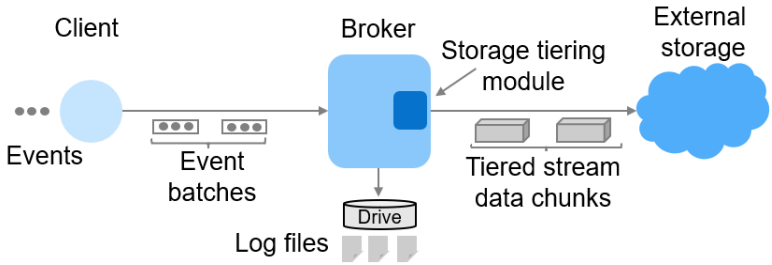


Figure 38: Architecture of an event streaming system.

- Implementation of `Nexus` and multiple streamlets to evaluate the benefits on event streaming systems like Kafka, Pulsar, and Pravega across Edge/Cloud deployments.

We validated `Nexus` through extensive experiments on AWS and on-premises clusters. `Nexus` intercepts and routes storage requests with modest overhead ($< 5\%$) and supports complex streamlet pipelines. `Nexus` achieves compression ratios up to $\approx 3.9\times$ higher than Kafka's and Pulsar's best client- or broker-side compression, all while maintaining low write latency. Additionally, `Nexus` enables privacy-aware semantic routing using AI inference with efficient stateful execution, and reduces streamlet state metadata read requests by $\approx 98\%$ via partition-aware routing.

8.2 Background

8.2.1 Event Streaming Systems

Event streaming systems, such as Apache Kafka [87, 123], Apache Pulsar [124], Redpanda [125], and Pravega [171], have emerged as a popular technology for ingesting and processing data in real-time. These systems expose the *data stream* (*a.k.a.*, topic) abstraction for managing data events. Internally, data streams are composed of *partitions* or *segments*, which can be parallelized for higher throughput. Most systems provide support for data durability and exactly-once semantics in data streams, as well as low-latency stream data ingestion.

The general architecture of event streaming systems consists of producers that write data events to streams, and consumers that read and process these events (see Fig. 38). Brokers act as intermediaries, managing the ingestion, storage, and delivery of events. Each broker persists stream data to local log files on high-performance storage devices to ensure durability and low-latency access [183]. Data is typically organized into append-only logs, segmented by partitions, and replicated across brokers for fault tolerance.

8.2.2 The Shift towards Streaming Storage

Due to the growing storage demands of streaming use cases, event streaming systems have adopted *tiered storage* as a key feature [178, 179, 184]. Tiered storage consists of offloading cold stream data from high-performance storage (*e.g.*, SSDs, NVMes) to scale-out storage (*e.g.*, object storage). This results in a sweet spot in the latency-throughput trade-off while reducing storage costs, making it a crucial feature for organizations to efficiently manage and store stream data [86].

In tiered storage, cold stream data is offloaded in *chunks*: units of larger granularity than individual events, typically ranging from a few to 100s of megabytes. These chunks are transferred asynchronously from the broker's local log files to external storage via standard APIs (*e.g.*, AWS S3). Once offloaded, the data is removed from the high-performance tier, freeing up resources for low-latency ingestion. This mechanism is transparent to producers and consumers, and enables long-term retention to historical data without impacting the performance of the hot path. However, advanced data management has not yet been considered in the tiering process.

8.3 Nexus Design

This section presents the design of `Nexus`, a novel data management mesh for tiered data streams.

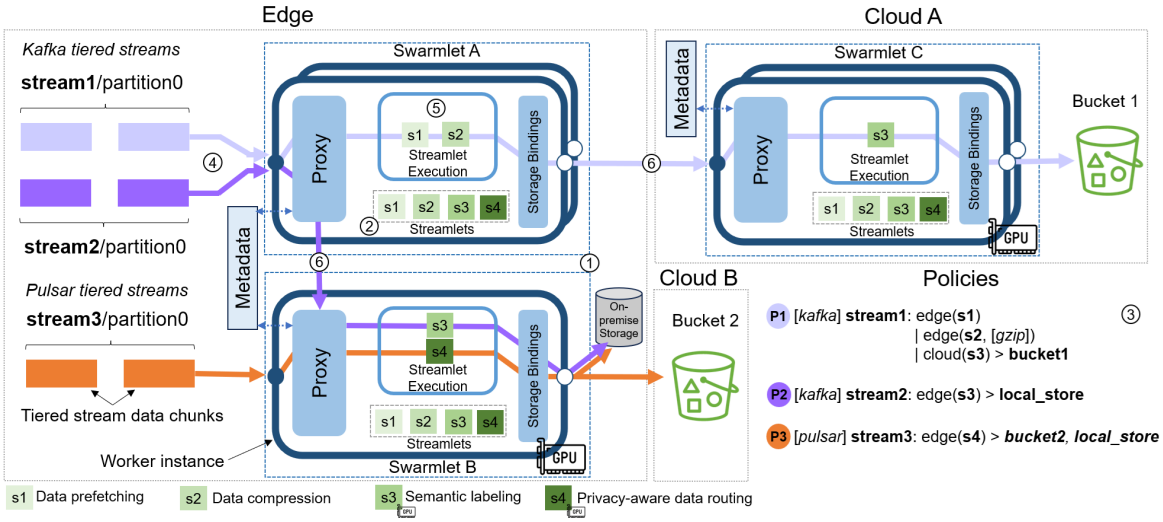


Figure 39: An architecture overview and operation of Nexus.

8.3.1 Design Principles and Insights

Event streaming systems are progressively adopting tiered storage as a key feature for providing scalable and cost-effective data storage. However, these systems are limited in terms of data management capabilities for tiered data streams. As of today, most systems just provide storage bindings for offloading chunks of stream data to external storage. To over come these limitations, we present Nexus, whose design is determined by the following set of principles.

First, Nexus is designed considering that event streaming and data management should be clearly *decoupled* concerns. A key observation supporting this design decision is that event streaming systems work at the *data event* level, whereas tiered data streams are managed at the *chunk* level. Such data granularities are totally different both in terms of size (*i.e.*, 10-1000x size difference) and processing latency expectations (*i.e.*, milliseconds vs minutes). In this sense, we realize that most event streaming systems adopt standard APIs for managing tiered stream data from external storage services (*e.g.*, AWS S3, Azure) [178, 185]. Nexus design exploits this insight by transparently *intercepting* storage operations of tiered data streams. In addition to favor a decoupled data management layer, storage API interception makes Nexus applicable to virtually any event streaming system with tiered storage.

Second, data management is a broad term with many types of applications [182]. As such, Nexus is designed to deploy and execute *user-defined data management functions* for optimizing storage operations related to tiered data streams [186, 187] (see §8.4.2). While some data management functions are expected to be stateless, Nexus design also considers supporting stateful functions. Function state may be required for storing data required for an algorithm or passing a partial event stored in previous chunk across executions[24], for example. In contrast to prior work [188], we do not attempt to keep globally consistent metadata shared across functions. As a novel contribution, Nexus exploits the sequential structure of data streams for routing data across stateful functions and linearizing function state management via consistent hashing. This relaxes the consistency requirements on shared metadata [189] (see §8.4.3).

Finally, event streaming systems are increasingly deployed at the Edge, or in Edge/Cloud environments [190, 191], for latency, cost, and privacy reasons. However, such an heterogeneous infrastructure also needs to be considered at the data management level. Nexus allows users to deploy data management functions on *specific locations* and *compose* their execution across the Cloud/Edge infrastructures via simple *policies*. Note that some functions may require specialized hardware for their execution (*e.g.*, GPU, TEE). Nexus tracks the specialized hardware available in worker instances,

[24]Some event streaming systems do not guarantee that event boundaries match with chunks of tiered stream data [86].

as well as execution requirements for transparently executing functions and routing data to the right place (see §8.4.4).

8.3.2 Abstractions

In what follows, we describe the abstractions that are key building blocks for the design of `Nexus`:

Streamlet: A streamlet is a function executed in-line on a storage request related to a chunk of tiered stream data. Streamlets are reactive: they are invoked upon the interception of storage operations. The streamlet concept is broad enough to address diverse data management requirements on data streams. We classify the streamlets into four categories: *transformers*, which change the content of data (*e.g.*, compression, encryption); *performance*, which aim to improve the IO performance of the event streaming system (*e.g.*, caching, prefetching); *routing*, which change the default destination of stream data (*e.g.*, multi-cloud replication); and *semantic*, which analyze the contents of stream data chunks and react to them (*e.g.*, AI-based metadata labeling). Note that a streamlet may fall into more than one category at the same time (*e.g.*, privacy-aware data routing).

Swarmlet: A swarmlet is set of `Nexus` worker instances for executing streamlets. Importantly, all the worker instances within a swarmlet are identical in terms of hardware resources and are located in the same infrastructure (Edge, Cloud). A `Nexus` deployment may be composed of multiple swarmlets, each one having its own access endpoint. This abstraction allows `Nexus` to easily route data and schedule the execution of streamlets based on their hardware needs.

Policy: Administrators use policies for configuring the execution of streamlets in `Nexus`. Policies can define the execution of a single streamlet or multiple ones composed in an execution pipeline [192]. Moreover, policies allow administrators to determine the location for a streamlet execution, as streamlets may need to be executed at different locations across a Cloud/Edge environment. In `Nexus`, a policy has the following elements: i) *system type*, which helps `Nexus` to identify actual chunks of stream data from other system-dependent metadata files also stored externally; ii) *target*, which defines the relevant stream(s) for the policy at hand; iii) *streamlet pipeline*, which defines the streamlet(s) to be executed, as well as the location and order of execution; iv) *output storage*, which is the eventual location of stream data chunks once executed the streamlet pipeline. Each streamlet may contain parameters for configuring its runtime behavior.

8.4 Nexus Architecture

`Nexus` runs on one or multiple clusters. Fig. 39 shows an overview of its architecture and operation. `Nexus` worker instances are deployed as swarmlets (step ①, Fig. 39) based on their infrastructure location (Edge, Cloud) and hardware resources (*e.g.*, GPU, TEE). Administrators can then deploy streamlets (step ②, Fig. 39) by loading binaries and setting up their metadata descriptor that describes key aspects for their execution, like hardware requirements. With swarmlets and streamlets in place, an administrator defines policies to orchestrate the execution of streamlets on data streams (step ③, Fig. 39). The metadata of streamlets, swarmlets, and policies is stored in the metadata store (see §8.4.1).

Event streaming systems can be configured to offload chunks of tiered stream data against a swarmlet service endpoint. `Nexus` worker instances act as a proxy by implementing standard APIs (AWS S3) for transparently intercepting storage operations from the event streaming system viewpoint (step ④, Fig. 39). Moreover, worker instances take care of the execution of streamlets (step ⑤, Fig. 39). As visible in Fig. 39, `Nexus` executes streamlet pipelines enforcing the streamlet location specified by the policy. This is possible thanks to `Nexus`' mesh-like data routing (step ⑥, Fig. 39).

In the following, we describe the system components in `Nexus` that allow administrators building data management pipelines for tiered data streams.

8.4.1 System Metadata

A swarmlet has a metadata store for managing system metadata and these are independent across swarmlets.

Swarmlet deployment. Deploying a `Nexus` swarmlet consists of creating a service composed by

one or multiple containerized worker instances. Such instances should be reachable under the same endpoint. Moreover, a swarmlet has two metadada requirements to be fully operational. First, an administrator needs to define the *location* of the swarmlet. For instance, if a swarmlet is deployed on an Edge cluster, we could associate it with the location label `edge`. Similarly, administrators should define the required *specialized hardware* for worker instances, if any. Worker instances for a swarmlet are placed on nodes with the required hardware available and labeled accordingly (*e.g.*, `gpu`). Both the location and hardware metadata of swarmlets is stored in the metadata store and used for correctly executing streamlet processing pipelines.

Streamlet metadata. Streamlets interact with metadata at multiple levels. First, an administrator installing a streamlet provides a *descriptor* specifying its main features. This includes information about if the streamlet should be executed on writes and/or reads (`on=[PUT, GET, ALL]`), the streamlet type (*e.g.*, `type=transformer`), if it is stateful (`stateful=[true, false]`), or if it has any special hardware requirements to be executed (*e.g.*, `hardware=gpu`). This descriptor helps `Nexus` to execute streamlets correctly and in the right location via data routing (see §8.4.4).

8.4.2 Streamlet Execution

Streamlets are runnable function binaries that are executed in the execution engine of worker instances. Streamlet binaries can be dynamically distributed and loaded onto swarmlets. The container image of worker instances also allows packaging third-party libraries that could be invoked by streamlets [193]. Next, we provide details on streamlet execution:

Containerized execution. `Nexus` provides container level isolation for streamlets running on the same worker instance. Streamlets scheduled on the same worker instance run in the same container and can access the shared system metadata. This approach is reasonable as `Nexus` can be administered by a single organization, which differs from public FaaS platforms [194]. The execution engine allows to pipeline several streamlets on a single storage request, based on the policy definition. To illustrate this, policy `P1` in Fig. 39 shows the pipelined execution of streamlets `s1` (prefetching) and `s2` (compression) in a worker instance at the Edge.

Storing outputs. Once its execution completes, a terminal streamlet in a pipeline stores the processed data chunk on the external storage defined in the policy. `Nexus` provides a storage binding module for abstracting the streamlets from the different the supported API implementations. Normally, a streamlet pipeline ends by storing stream data chunks in a single storage service (`P1` and `P2` in Fig. 39). Still, streamlets that route data may receive as input several storage service endpoints to store data based on the computation's output. For instance, `P3` in Fig. 39 routes data to a local object store or to a S3 bucket depending on whether it infers some potential privacy issue in the contents of stream data chunks.

8.4.3 Streamlet State

`Nexus` provides state support to streamlets. In particular, stremlets can access the following primitives:

Object tags. A new `StreamletContext` object is created upon execution of a streamlet pipeline. The `StreamletContext` provides support for managing shared key/value pairs. This yields that, within the pipeline execution in a worker instance, streamlets can manage their own key/value pairs, as well as access key/value pairs from other streamlets. This basic primitive can support interesting functionality; *e.g.*, streamlet $s1$ may route data to bucket b only if key k managed by streamlet $s2$ has a certain value. Importantly, all the key/value pairs generated during a streamlet pipeline execution are asynchronously stored along with the object itself as object tags. Upon a `GET` request, object tags are loaded in the `StreamletContext` and are available to streamlets before the pipeline starts its execution.

Stateful streamlets. Although many streamlets are assumed to be stateless (*e.g.*, data compression), complex data management functions may need to keep persistent state. To this end, `Nexus` allows administrators to define a streamlet as stateful (`stateful=true`). At that point, `Nexus` will automatically store in the system metadata any data structure (*e.g.*, list, map) annotated as `@Persistent`.

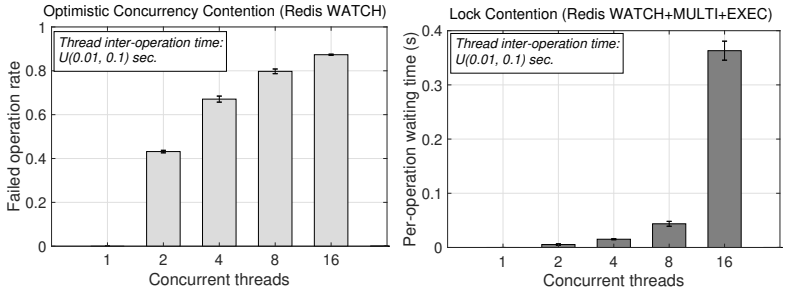


Figure 40: Redis micro-benchmarks to measure contention in concurrent updates on shared data structures (optimistic concurrency, locks). The time between checking the server state and performing the update is random $\in [0.01, 0.1]$ secs.

However, in this setting, managing the streamlet state can be expensive if done naively. For the sake of illustration, let us assume a data routing streamlet that analyzes stream data chunks and stores them in an alternative bucket based on the computation outcome. Moreover, it keeps track of chunks stored in the alternative bucket via a persistent map to serve `GET` requests. As `Nexus` assumes no control over load balancers, data chunks are randomly distributed and processed by streamlet instances across different worker nodes for the same stream. Therefore, streamlets are concurrently managing the same shared map. To keep updates to the shared map consistent, we could resort to existing synchronization primitives in metadata services. But, as visible in a set of micro benchmarks on Redis (Fig. 40), both optimistic concurrency and locking mechanisms exhibit significant contention as the number of concurrent updates grow. Similarly, consistent metadata stores proposed in the serverless literature also incur overhead in keeping shared data structures consistent [188].

Instead, to enable efficient streamlet state management, we build on the observation that a data stream partitions are *parallel* (*i.e.*, there is no dependency among them) and *sequential* (*i.e.*, the data arrival and tiering process keeps order). Therefore, `Nexus` can linearize metadata updates on a per-partition basis to avoid metadata synchronization on stateful streamlet data structures working on the same stream [195]. First, within the same worker node, storage requests for a given stream partition are processed in order of arrival. Second, for a given stream partition, `Nexus` creates individual streamlet data structures in the metadata store that are managed by the same streamlet instance. As depicted in Fig. 41, this has two benefits: i) streamlet metadata updates do not require synchronization, and ii) on `GET`s, streamlets have a fresh view of their metadata, which reduces accesses to the metadata store.

Naturally, managing the streamlet state this way also requires ensuring that `Nexus` routes storage requests based on their partition to a consistent set of worker nodes. As we describe next, this is supported via data routing.

8.4.4 Mesh-like Data Routing

Event streaming systems have limited flexibility for configuring the storage tiering endpoint [178, 148]. For this reason, `Nexus` handles the complexity of routing stream data chunks for building streamlet pipelines. Moreover, `Nexus` assumes no control over the service load balancer. Thus, the proxy component in the `Nexus` worker instance is responsible for intercepting and routing storage requests across the system. In `Nexus`, data routing considers direct data transfers across swarmlets, which differs from typical indirect communication in serverless pipelines [194, 193]. `Nexus` offers two routing types:

Partition-aware data routing: `Nexus` employs a deterministic data routing protocol to assign stream partitions to specific worker instances, thereby ensuring consistent execution of stateful streamlets. Let S be a stream composed of parallel partitions $P = \{p_1, p_2, \ldots, p_i\}$ and $W = \{w_1, w_2, \ldots, w_j\}$ be the set of available worker instances within a swarmlet. `Nexus` partitions the ID space $[0, 2^n - 1]$ into m contiguous intervals ($m \geq j$), each assigned to a worker instance. Given a stream partition

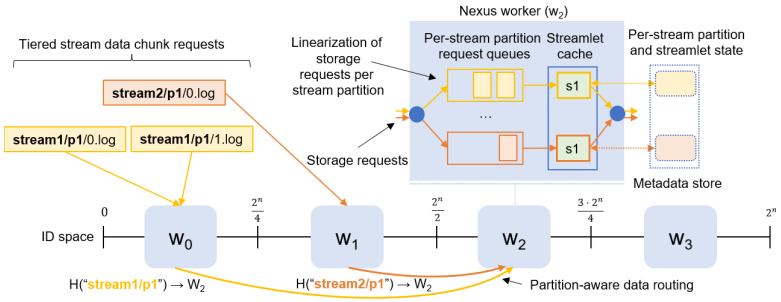


Figure 41: Example of partition-aware data routing.

identifier p_x, a consistent hash function $H : P \rightarrow [0, 2^n - 1]$ is used to compute its position in the ID space [**?**]. The storage request is forwarded to worker w_z responsible for the interval containing $H(p_x)$. Once in w_z, the storage request will be processed by the expected streamlets. An example of such a process can be seen in Fig. 41.

Location and hardware-based data routing: `Nexus` also routes stream data chunks based on policy location and hardware. First, an administrator explicitly defines in a policy where a streamlet should be executed (Edge, Cloud). The reason is that some streamlets are more effective when executed on a specific infrastructure (*e.g.*, caching data at the Edge). Based on policy metadata, `Nexus` routes storage operations to the swarmlets available by the policy location. Moreover, `Nexus` implicitly re-routes operations within the same locations to meet a streamlet's hardware requirements. To illustrate this, `P2` in Fig. 39 defines a policy in which streamlet `s3` is set for execution at the `edge` and requires a GPU. For a Kafka cluster, `swarmlet A` is defined as the storage tiering endpoint. However, `swarmlet A` does not have GPU-powered worker instances. In this situation, proxies in `swarmlet A` route storage operations to `swarmlet B`, which has worker instances with GPUs. This provides flexibility to `Nexus` deployments without impacting event streaming systems configuration.

8.4.5 Fault Tolerance and Correctness

`Nexus` is designed to target the same level of reliability as if the event streaming system was interacting with an external storage. To this end, `Nexus` processes storage requests in-line and only acknowledges them back to the event streaming system once the entire streamlet pipeline has completed successfully. This ensures that data transformations (*e.g.*, compression, routing) are fully applied before the data is considered durably stored. For transformer streamlets, which modify the content of stream data chunks, `Nexus` can re-compute the checksum headers to maintain compatibility with object storage APIs and preserve end-to-end data integrity.

`Nexus` relies on partition-aware data routing to ensure consistent streamlet execution and state management. Swarmlet membership is tracked via a consistent metadata service (*e.g.*, watches), ensuring that partition ownership is always up-to-date and validated before execution. Before processing a request, each worker checks whether it owns the partition at hand. If not, the request fails and is retried.

Finally, most metadata in `Nexus` is local to each swarmlet, simplifying management and avoiding cross-infrastructure dependencies. However, policies that span multiple infrastructures require a consistent view to ensure deterministic streamlet execution. `Nexus` enforces this via scheduled policy updates: new policy versions are propagated from the Edge to the Cloud and applied to future chunks based on stream monotonicity. Additionally, storage requests carry the policy version being executed, allowing workers to validate consistency and reject mismatched executions, triggering retries.

8.5 Nexus in Action

Next, we briefly overview the main `Nexus` APIs from a developer viewpoint.

Nexus API	Purpose	Key Methods
`ByteStreamlet`	Raw byte processing (e.g., compression, encryption)	`processPutBytes`, `processGetBytes`
`DataSourceStreamlet`	Preload or redirect data before `GET` (e.g., caching, routing)	`handlePreGet`
`EventStreamlet<T>`	Record-level processing (e.g., AI inference, annotations)	`processPutRecord`, `processGetRecord`
`Deserializer<T>`	Converts byte stream to typed records	`deserializeChunk`
`@Persistent`	Declares persistent state across executions	-

Table 17: Summary of `Nexus` Streamlet APIs

8.5.1 Streamlet API

Developers extend `Nexus` by implementing streamlets that process stream data chunks. `Nexus` supports two streamlet types: `ByteStreamlet` and `EventStreamlet` (see Table 17).

A `ByteStreamlet` allows developers to process raw byte streams during `PUT` and `GET` operations by implementing the methods `processPutBytes` and `processGetBytes`. These streamlets are suitable for functionality such as compression or encryption, for instance. `Nexus` invokes these methods in-line during storage operations, ensuring that transformations are applied before data is durably stored.

For streamlets like caching or buffering, developers can also implement the `DataSourceStreamlet` interface. This interface extends the byte-based model by allowing streamlets to intercept the data source before a `GET` operation via the `handlePreGet` method. This enables streamlets to pre-load content from alternative sources or serve cached data directly.

The `EventStreamlet<T>` API allows developers to process deserialized records from stream data chunks. These streamlets are ideal for semantic processing tasks, such as AI-based inference or content-based annotation. Developers must provide a `Deserializer<T>` implementation to convert the input stream into typed records. The streamlet then defines `processPutRecord` and `processGetRecord` to handle each record individually. Importantly, event streamlets cannot modifying the request content to preserve data integrity.

Both `ByteStreamlet` and `EventStreamlet` can maintain persistent state across executions by annotating data structures with `@Persistent`. `Nexus` automatically stores and retrieves these structures using the system metadata backend at hand.

8.6 Implementation

We have implemented `Nexus` as a middleware for the S3Proxy project [196], an extensible S3-compatible proxy. The middleware intercepts S3 requests and processes them based on system metadata (*e.g.*, swarmlets, streamlets, and policies). From the client's perspective, `Nexus` behaves like a standard S3 interface: the client receives a response only if the request is successfully stored or fails. Internally, however, `Nexus` uses asynchronous programming to concurrently process requests in a streaming fashion. The implementation spans over 7K lines of Java code, including a CLI for managing metadata and scripts for deployment and benchmarking. Redis [197] serves as the metadata backend, with local caching for read-only access. Upon updates (*e.g.*, new policies), Redis notifies the `Nexus` metadata service to apply changes. `Nexus` also supports dynamic compilation and runtime loading of streamlets and deserializers. The code is available at [198].

8.7 Validation

The evaluation of `Nexus` focuses on the following questions:

- Can `Nexus` provide high performance storage request interception and routing (§8.7.2)?
- What benefits may `Nexus` transparently bring to existing event streaming systems (§8.7.3)?

8.7.1 Experimental Setup

Cluster settings. We deploy `Nexus` in two Kubernetes-based environments. On AWS, we create an EKS cluster on top of 3 `i3en.2xlarge` instances, each with 8vCPUs, 64GB of memory, and 2 local NVMe drives to deploy all instances (`Nexus` workers, Redis, and benchmarks). We use AWS S3 as long-term storage. Our lab cluster is composed of 7 VMs running Kubernetes. Each VM has 8 CPUs, 20GB of memory, and 80GB of storage (1 master, 6 workers). One VM has a GPU available for running AI-related streamlets (Nvidia A16 with 16GB VRAM). In our experiments, Edge and Cloud are logical regions for swarmlets within the same infrastructure.

Baselines. We exercise `Nexus` via 3 streaming systems:

Apache Kafka: Apache Kafka [87, 123] is a distributed event streaming platform optimized for high-throughput, low-latency data processing via a publish-subscribe model. It structures data into topics, partitions, and offsets, ensuring scalability and fault tolerance through replication. For tiered storage, introduced in KPI-405 [184], we use Aiven's Kafka storage tiering plugin [181].

Apache Pulsar: Apache Pulsar [124] is a scalable, low-latency event streaming system. Its tiered storage offloads sealed log segments from high-performance storage (*e.g.*, Apache BookKeeper [199, 183]) to cost-efficient long-term storage, reducing costs while preserving seamless access to historical data.

CNCF Pravega: Pravega [171, 86] is a distributed stream storage system for unbounded, high-throughput data. Pravega streams support dynamic scaling, exactly-once semantics, and long-term retention. Its tiered storage integrates low-latency short-term storage via Apache BookKeeper with cost-efficient long-term storage like cloud object stores.

Benchmarks. To generate workloads in our experiments, we resort to the following benchmarks:

FIO: FIO [200] is a powerful benchmarking tool used to evaluate the performance of storage systems, including object storage services. It supports various workload types such as sequential and random read/write operations, mixed I/O patterns, and custom workloads. FIO allows users to configure parameters like block size, I/O depth, and number of jobs, and it supports multiple I/O engines (*e.g.*, libaio).

OpenMessaging Benchmark: OpenMessaging Benchmark [180] is a comprehensive toolset designed for benchmarking messaging systems in the cloud. It supports systems like Apache Kafka, Apache Pulsar, Pravega, and more.

Datasets. We used the following datasets for building stream data payloads in our experiments[25]: i) *Data compression*: For data compression experiments, we used traces from HDFS from a collection of system logs available at [201, 202]. ii) *Semantic data routing*: For image payloads, we used images from ImageNet [99] and Cholec80 [203]. iii) *Genomic data*: For the genomics data management experiment, we use FASTQ files available from SRA Toolkit [204].

8.7.2 Interception Performance

First, we evaluate `Nexus` in isolation via FIO to understand its interception and routing performance on AWS.

IO interception performance. We execute a `r/w` workload of objects against S3 via FIO as a performance baseline. Then, we reproduce the same workload when a single `Nexus` worker intercepts storage request without processing and when it executes a no-op streamlet. Fig. 42 shows the results for various levels of benchmark parallelism and object sizes.

First, the main observation in Fig. 42 is that `Nexus` does not induce a significant performance penalty when intercepting object storage requests. In most cases, the throughput reduction that FIO reports when `Nexus` intercepts requests falls below 5%. `Nexus` worst performance cases compared to the baseline seem to be for one benchmark thread and extreme object sizes (1MB and 1000MB). A possible explanation may be that `Nexus` performs metadata checks during the hot path that incur additional latency, whose impact is more pronounced for low parallelism rates. We also observe that throughput variability in `Nexus` is generally lower compared to FIO performing IO directly to

[25] We extended OpenMessaging Benchmark to load custom payloads.

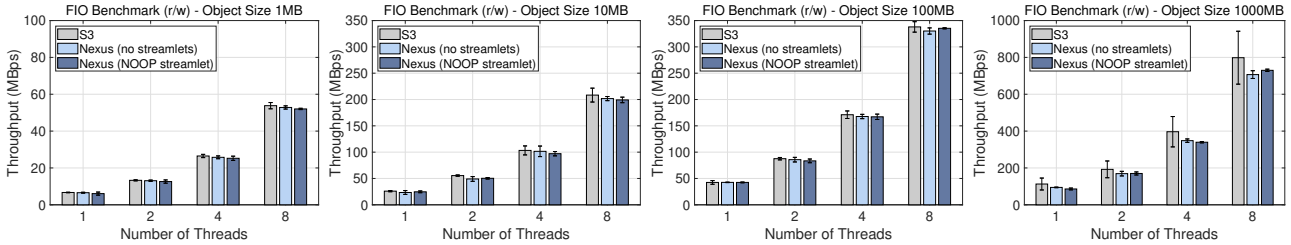


Figure 42: Interception performance of 1 Nexus instance under FIO read/write workloads on AWS.

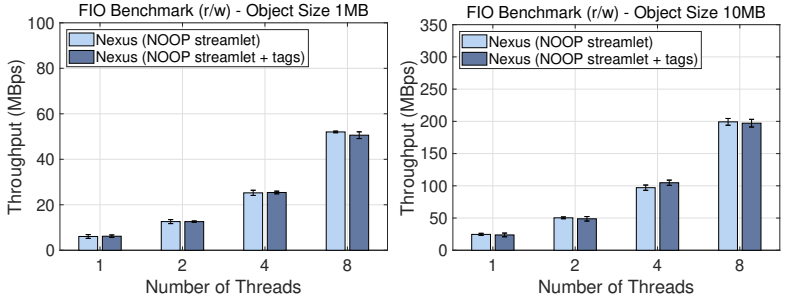


Figure 43: Performance impact of leveraging object tags using 1 Nexus instance under FIO read/write workloads on AWS.

S3. This may be due to the server buffering configurations for optimizing HTTP connections in Nexus. Overall, a single Nexus instance can execute a no-op streamlet from multiple client requests at $\approx$ 730MBps, demonstrating the feasibility of the current implementation.

Object metadata. Nexus exploits object metadata to allow streamlets annotating objects, as well as to store specific system information (*e.g.*, if the object was processed by a transformer streamlet). To inspect the impact of using object tags, Fig. 43 shows the throughput comparison of a no-op streamlet versus a no-op streamlet that stores and retrieves an object tag for PUTs and GETs, respectively.

Fig. 43 shows that the throughput reduction related to managing object tags is generally minor. Specifically, the worst-case throughput reduction observed is 2.7% and 3.2% for 1MB and 10MB objects, respectively. A key reason for minimizing the performance impact of managing object tags is that these are asynchronous operations. For instance, upon an object PUT, object tags are stored asynchronously once the actual object has been correctly stored (see §8.4.3), thus not blocking the original request. This experiment demonstrates that Nexus can effectively manage object metadata, which can be leveraged for both system and user purposes.

Streamlet pipelining overhead. Next, we evaluate the performance of Nexus intercepting object storage requests with a different number of pipelined functions in the same worker instance. To this end, we create policies pipelining up to 3 no-op streamlets for PUT requests (see Fig. 44).

Fig. 44 shows that Nexus can successfully pipeline multiple functions as a chain. Visibly, each additional no-op streamlet can induce an additional 1% to 6% of throughput reduction, depending on the case. In line with prior observations, the 3-streamlet case with the lowest levels of parallelism exhibits the most pronounced performance drop compared to the baseline (up to 6.8%). Overall, this experiment demonstrates the practicality of building complex streamlet pipelines in Nexus.

Data routing overhead. Nexus can route object requests transparently to event streaming systems based on the policies and streamlets configured (see §8.4.4). We evaluate the performance impact of executing no-op streamlets across 1 to 3 Nexus worker instances (see Fig. 45).

In the more typical 2-hop pipeline scenario (*e.g.*, Cloud-Edge deployments), Fig. 45 shows that the throughput reductions due to routing across swarmlets ranges from 6.2% to 14.8%, depending on object size and benchmark parallelism. We also assess a more demanding 3-hop configuration in

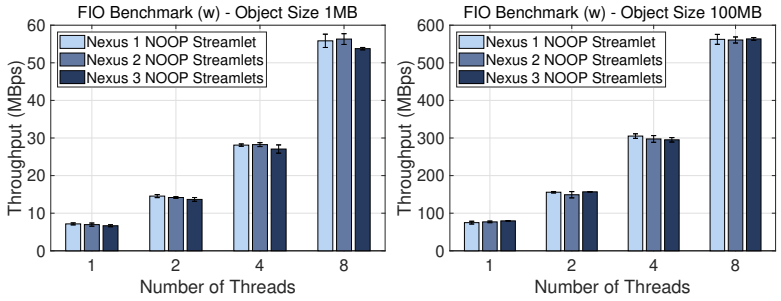


Figure 44: Processing performance depending on pipeline size of 1 `Nexus` instance under FIO write workloads on AWS.

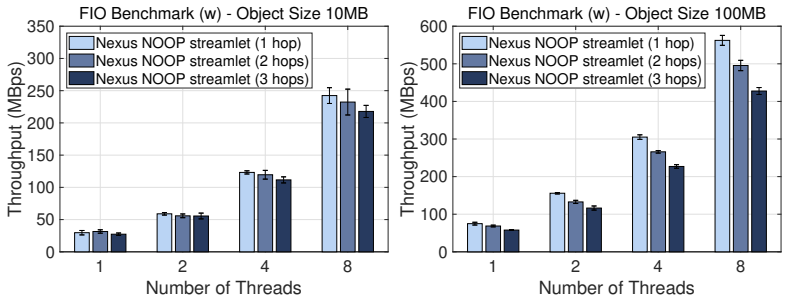


Figure 45: Performance impact of data routing hops under FIO write workloads on AWS.

`Nexus`. In this case, we observe a higher performance penalty, with throughput reductions reaching up to 25.6%, particularly for larger objects. Despite the added routing complexity, these results confirm that `Nexus` maintains acceptable performance levels and preserves transparency from the event streaming system's perspective, which is a key design goal.

8.7.3 Enhancing Event Streaming Systems

Next, we focus on illustrating the advantages that `Nexus` provides to event streaming systems in our on-premises cluster.

Infrastructure abstraction. We perform two experiments in this section: *data compression* and *semantic data routing* (see Fig. 47). For the first experiment, we deploy two `Nexus` swarmlets logically associated to Edge and Cloud regions, where data compression takes place at the Edge. Second, as may happen in a real environments, the Edge cluster is expanded with GPU-enabled nodes. Then, we create a new swarmlet in the Edge region and implemented a semantic data routing streamlet pipeline that requires GPU. The crucial insight is that such infrastructure expansion is transparent to the event streaming system, as `Nexus` re-routes storage requests across swarmlets based on hardware requirements. Therefore, `Nexus` allows administrators to manage the infrastructure and policies without requiring changing the (limited) tiered storage configuration of event streaming systems.

Transparent data compression. First, we focus on data compression. We implemented a transformer streamlet: GZip data compression (`s1`). We set event streaming systems to offload data to `Nexus swarmlet 1` in our on-premises cluster and configured `Nexus` against a MinIO bucket. We compare the impact of data compression on the event streaming systems built-in mechanisms (if any) in terms of latency and compression ratio for Kafka, Pulsar, and Pravega.

Fig. 46 shows how different compression models—client-side, broker-side, tiered storage (broker), and `Nexus`—affect compression ratio and write latency across event streaming systems. `Nexus` achieves the highest compression ratios (*e.g.*, up to 10.4× in Pravega, 9.9× in Kafka, and 8.3× in Pulsar) while maintaining low write latencies (similar to compressing data at the tiered storage module in the Kafka broker). `Nexus` achieves compression ratios up to 3.9× higher than the best client-

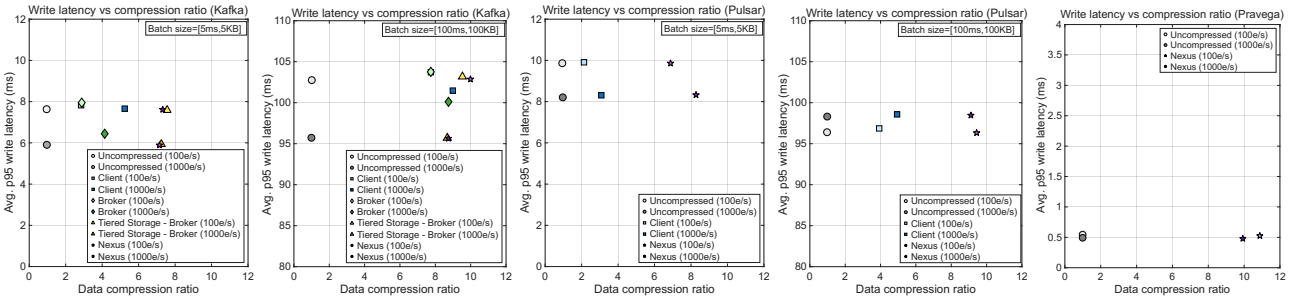


Figure 46: Data compression vs write latency of event streaming systems data compression and `Nexus`.

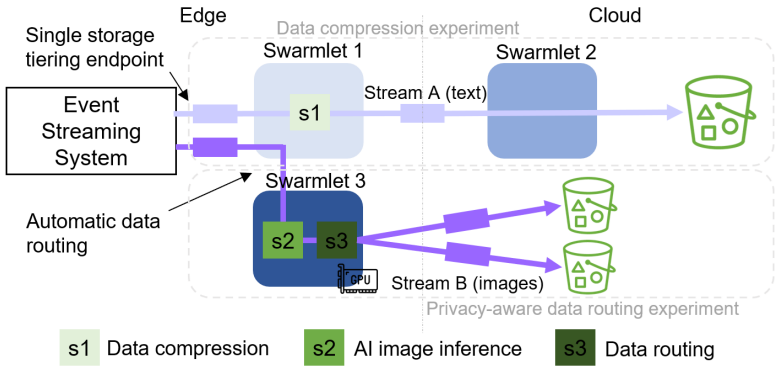


Figure 47: Experiment setup showing `Nexus` capabilities to abstract infrastructure from event streaming systems.

or broker-side compression in Kafka, and over 3.8× higher in Pulsar (Pravega has no compression implemented).

Interestingly, in Kafka and Pulsar, which use user-defined batching strategies (*e.g.*, fixed batch time and size), we observe that larger batch sizes and higher producer rates generally improve compression ratios by leading to larger event batches. For instance, in Kafka, increasing the batch size from 5ms/5kb to 100ms/100kb at 100e/s improves the compression ratio from 2.84× to 7.75× with client-side compression. However, this improvement comes at the cost of increasing 95th write latency by ≈ 15×. These results show that `Nexus` enables transparent data compression across diverse event streaming systems, without impacting hot path write latency.

Semantic data routing. We instantiated a new swarmlet (`swarmlet 3`) in our on-premises infrastructure deployed on nodes equipped with GPUs (see Fig. 47). We run a workload with OpenMessaging Benchmark in which the benchmark writes images to Kafka containing humans or not, switching randomly every 2MBs. Then, we configured a two-streamlet pipeline at the Edge: i) a semantic human detection streamlet via Yolov5 [205] model (`s2`), and ii) a data routing streamlet (`s3`) that stores a data chunk on a private bucket if any of the processed images belongs to a human. Note that `s2` is a stateful streamlet (see §8.4.3), meaning that it keeps a `@Persistent` map of the data chunks stored in the private bucket.

Fig. 48 (left) shows a time-series with the semantic decisions made by the streamlet pipeline. Visibly, `Nexus` stores stream data chunks to the right bucket based on the result of the AI inference process (when Kafka manages 1MB and 2MB chunks). Furthermore, while `s2` is an `EventStreamlet` that processes the internal content of a chunk event by event (see §8.5), its implementation allows the user to perform AI inference on a subset a chunk's events. Fig. 48 (right) illustrates `s2`'s processing time depending on the image sampling rate and the chunk size. Finally, we tested `s2` by using the proposed partitioned approach vs a shared approach in which any streamlet instance can manage

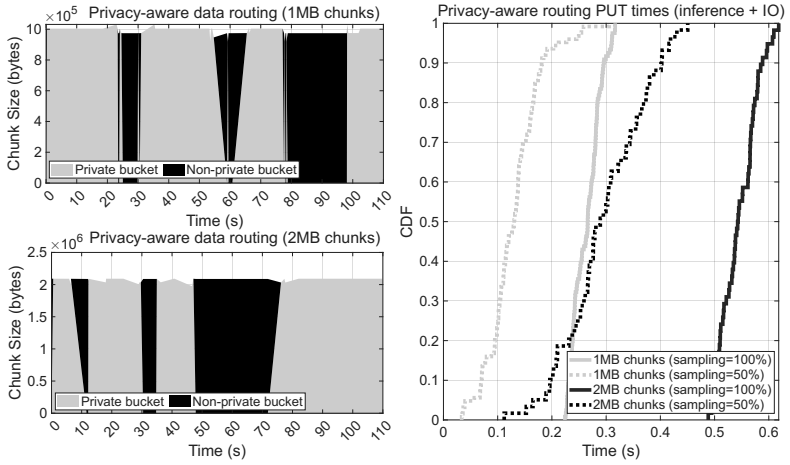


Figure 48: Storage bucket for stream data chunks decided by the semantic streamlet (left) and inference times based on chunk size and event sampling rate (right).

the streamlet state. In this sense, while the number of metadata writes is similar as any update needs to be persisted in metadata, we found that the partition-aware data routing reduced in $\approx 98\%$ read requests to the metadata in our experiment. The reason is that the same streamlet instance was in charge of managing metadata for a given stream partition thanks to `Nexus` partition-aware data routing (§8.4.4), thus avoiding having to load the most recent metadata on every request.

8.8 Related Work

Unlike existing systems, `Nexus` uniquely targets the data management gap between event streaming systems and external storage by enabling programmable, in-transit functions on tiered data streams across heterogeneous infrastructures.

Data streaming and data management. Event streaming systems such as Apache Kafka [123], Apache Pulsar [124], Redpanda [125], and Pravega [86] have evolved to support high-throughput, low-latency ingestion and durable storage of data streams [177]. Recent advancements have introduced tiered storage mechanisms that offload cold data to external object stores [184, 178, 179], enabling unified access for both streaming and batch analytics. Unfortunately, these systems provide limited support for in-transit data management during tiering. To our knowledge, only Aiven's storage tiering module for Kafka [181] offers built-in support for data compression and caching on chunks of tiered stream data. However, extending such tiering modules, which are deployed on top of event streaming brokers, introduces additional complexity and operational burden. Instead, `Nexus` proposes a programmable data management layer that operates on tiered stream data independently of the streaming system's hot path.

Software-defined storage. `Nexus` builds on software-defined storage (SDS) concepts, such as policies and processing stages that shape storage behavior. IOFlow [206] pioneered SDS with end-to-end I/O policies via queuing abstractions. Crystal [186] extended SDS to object stores using filters and a centralized controller for dynamic policy enforcement. Retro [207] generalized SDS for multi-tenant systems with resource-agnostic policies. While these systems focus on internal control and resource management, `Nexus` targets programmable data management over tiered stream data and operates by intercepting and processing storage operations across Edge/Cloud infrastructures.

Data-intensive serverless systems. Recent serverless computing research explores data-aware function orchestration, akin to `Nexus` streamlets. Sonic [192] enables application-aware data passing across chained functions, while Sion [208] introduces elasticity and locality-aware scheduling for cloud storage. Middleware systems [187] and bucket-trigger models [194] propose data-driven execution for object stores. AWS S3 Object Lambda [209] enables developers to customize the output of `GET` requests using Lambda functions, allowing for transformations such as filtering. However, it is

limited to read-time operations and lacks support for stateful orchestration. Nexus shares the goal of programmable data flows but focuses on tiered stream data. It supports efficient stateful streamlet execution via partition-aware routing, leveraging stream properties to optimize metadata access. To our knowledge, existing serverless frameworks do not address the unique challenges of managing tiered streams across Edge/Cloud infrastructures.

Data and service meshes. Nexus borrows ideas from data and service mesh systems to route storage requests to the right swarmlets for enforcing policy execution. While industry platforms such as Confluent's data mesh framework [210] adopt data mesh principles for decentralized governance and scalable data sharing, they operate primarily at the metadata and access control layer. In contrast, service meshes such as Istio [211] focus on continuous data ingestion and traffic routing across heterogeneous infrastructures. Emerging data mesh platforms such as NebulaStream [212] aim to unify data management across heterogeneous infrastructures, especially in IoT and Edge/Cloud environments. These systems focus on continuous data ingestion and distributed query execution. Nexus complements this vision by focusing on the tiering boundary (where stream data transitions from hot to cold) and enabling programmable data management at that point.

8.9 Conclusions

Nexus introduces a novel data management mesh that bridges the gap between event streaming systems and external storage. By decoupling data management from the streaming hot path, Nexus supports extensible streamlet pipelines on tiered stream data across heterogeneous infrastructures with modest overhead. Our results shows that Nexus achieves up to 3.9× higher compression ratios than built-in mechanisms in Kafka and Pulsar, supports privacy-aware semantic routing with efficient stateful execution, and reduces metadata read requests for stateful streamlets by 98% via partition-aware routing. oWe believe Nexus opens a new design space for programmable tiered stream data management that can benefit data-intensive use cases using event streaming systems.

9 XtremeHub HPC Connectors

9.1 Introduction

In this section, we present Lithops-HPC, an HPC connector that allows Lithops to access HPC resources lying on supercomputers. Such work is challenging as Lithops relies on Kubernetes or on cloud providers. Such options are unavailable on the supercomputer due to security issues concerning Docker, as well as the requirement of privileges to run certain commands.

However, Singularity, with limitations, is allowed to run as long as it is only user level. One cannot build a container within the supercomputer; however, one can run images previously built, as they can be run on the user level. Thus, we devised a connector acting as a backend to integrate Lithops into it. In the process of such work, the changes were higher than expected and actually resulted in a new concept of Lithops, while keeping its essence, Lithops-HPC.

High Performance Computing (HPC) provides users with direct access to advanced computational infrastructure. However, effectively utilizing its full potential requires expertise in resource configuration and program parallelism. As a result, HPC users often face the challenge of managing infrastructure and using low-level computing techniques, which can be particularly daunting for non-expert users unfamiliar with aspects such as job scheduling, resource provisioning, data distribution, and even parallel programming.

A growing trend in HPC user accessibility, aimed at simplifying programming, is driven by the increasing adoption of Python in supercomputing environments [213]. Understandably, most tools and libraries for scientific data processing and machine learning are available in Python, bringing them closer to non-expert users and allowing more people to explore them. Tools such as Open OnDemand [214] confirm this need by enabling a Jupyter Notebook interface to a supercomputer. Furthermore, this is motivated by recent initiatives like AI factories [215] in the EU and the National Artificial Intelligence Research Resource (NAIRR) [216] in the USA, designed to open supercomputers to the general public to drive innovation. The downside is that they do not provide an easy way for these non-experts to easily leverage supercomputing resources, resulting in ineffective utilization. Frameworks such as Pegasus [217], COMPSs [218], Parsl [219], and cluster-based dataflow engines like Dask, Spark, Flink, or Ray try to hide such complexities to the end-user. However, all of them still require the user to handle resource allocation through the SLURM manager. For the end-user, this can be challenging as the manager requests at last the amount of cores, GPUs, and wall clock time. This is not an easy decision for such a user profile, and a misprediction on such parameters can result in the workload crashing or using more resources than necessary, with the latter being the most common case, which is not desirable for the sake of the whole infrastructure nor by other users requiring such unused resources, which become unavailable for no reason.

On the other hand, cloud computing has evolved over the years to simplify distributed and parallel computing, with a particular focus on non-expert users. With the recent serverless paradigm, infrastructure management shifts from the developer to the service provider. This approach allows to creation of serverless architectures that, particularly with the Function-as-a-Service (FaaS) model, allow running massively parallel programs by coding a single function that automatically scales on demand to thousands of CPU cores without managing any resources.

Several frameworks leverage FaaS services to easily develop and scale data analytics applications, such as PyWren [220], ExCamera [221], and Lithops [92]. Among them, Lithops stands out for its intuitive Python-based programming model. With it, applications may distribute the computation seamlessly through map (and reduce) operations that automatically run a function in parallel, massively, based on the data volume.

Lithops-HPC is designed to support multiple cloud platforms, allowing users to focus on their applications rather than worrying about resources, parallelization, and the particularities and operation of specific services. This abstraction of infrastructure and the associated programming model are highly attractive to scientists in HPC, as they are applicable to many data processing applications and offer the potential to utilize advanced computing resources more effectively without necessitat-

[25] The content of this section maps to tasks T3.4 and has been submitted for publication.

ing expertise in their management. While Lithops-HPC integrates Lithops into HPC environments, it still requires user management and a certain level of knowledge of the infrastructure. An element that is also lacking in COMPSs, Dask, and other alternative frameworks targeting HPC. They still require custom, hands-on resource provisioning and management.

So in this section, we present Lithops-HPC, an innovative expansion of the Lithops data analytics framework that leverages its simple Python interface and serverless abstractions to execute massively parallel applications elastically on HPC clusters. The main goal is to not only ease the programmability of HPC workloads to the user but also to avoid the user having to have any knowledge of the infrastructure. The user simply requests to run a piece of code, and the framework takes it from there. Lithops-HPC addresses the unique challenge of bridging the gap between serverless and HPC technologies, bringing together the advantages of both. On the one hand, following a serverless model, the system reduces the complexity and expertise requirements imposed upon HPC users. On the other hand, users still have access to the full power of HPC resources that are not available in serverless services, such as high-performance processors, sophisticated storage systems, faster communication networks (*e.g.*, InfiniBand), and hardware accelerators (*e.g.*, GPUs and TPUs).

Our implementation of Lithops-HPC tackles specific challenges in the fast invocation of tens or hundreds of thousands of functions in parallel, solving bottlenecks in scalability and latency issues, and providing seamless access to specialized hardware and technologies.

We evaluated Lithops-HPC using two demanding benchmarks: a large-scale matrix multiplication test (FLOPs) and a high-throughput object storage benchmark (write/read tasks). The results show that Lithops-HPC supports serverless application development capable of scaling immediately to tens of thousands of CPU cores, outperforming popular cloud platforms, and bringing both flexibility and scalability to HPC systems. In particular, we found that Lithops-HPC allows users to deploy applications more simply than traditional HPC submissions while introducing minimal overhead compared to a naive Python-MPI implementation, even when executing more than 10,000 parallel functions. Performance results showed that Lithops-HPC achieved up to 60 GFLOPs per CPU worker and more than 1,000 GFLOPs per GPU worker—roughly 3× and 50× faster, respectively, than AWS-reported performance. In storage tests, Lithops-HPC achieved more than 600 MB/s bandwidth using a parallel file system (PFS) for both read and write tasks. These results are approximately three times higher than the throughput achieved by the AWS-S3 storage system. Additionally, the Lithops-HPC resource manager, a component enabling dynamic switching between CPU and GPU runtimes at execution time, allows applications to automatically take advantage of accelerators when they are available.

9.2 The Lithops-HPC framework

Figure 49 illustrates the Serverless Advanced Computing (Lithops-HPC) design. The Lithops-HPC model abstracts traditional HPC infrastructure configurations, such as job scheduling and resource allocation, allowing programmers to focus solely on coding. However, similar to other serverless platforms, a provider role is still necessary to handle routine infrastructure deployments and management. To facilitate these tasks, Lithops-HPC provides a command-line Application Programming Interface (API), which contains all necessary scripts to schedule jobs, allocate machines within HPC nodes, and start CPU/GPU workers (i.e, the user should just type *lithops hpc runtime_deploy <runtime name>* to start a new runtime backend). It allows customizing the runtime backend, setting the number of workers, the type of processing units (both CPU and GPU are supported), and storage properties via a configuration file. An example of this configuration is defined in the following `config.yaml`.

```
lithops:
  backend: hpc
  storage: pfs
pfs:
  storage_root: <DIR_PATH>
cluster_hpc:
```

```
  host: <hpc_login>
  username: <hpc_user>
  key_path: <SSH key pair>
hpc:
  runtime: <choose a particular runtime_name>
  runtimes: #Custom runtime parameters
    <runtime_name>:
      account: <HPC_USER>
      qos: <HPC_QOS>
      num_workers: <CPU/GPU workers>
```

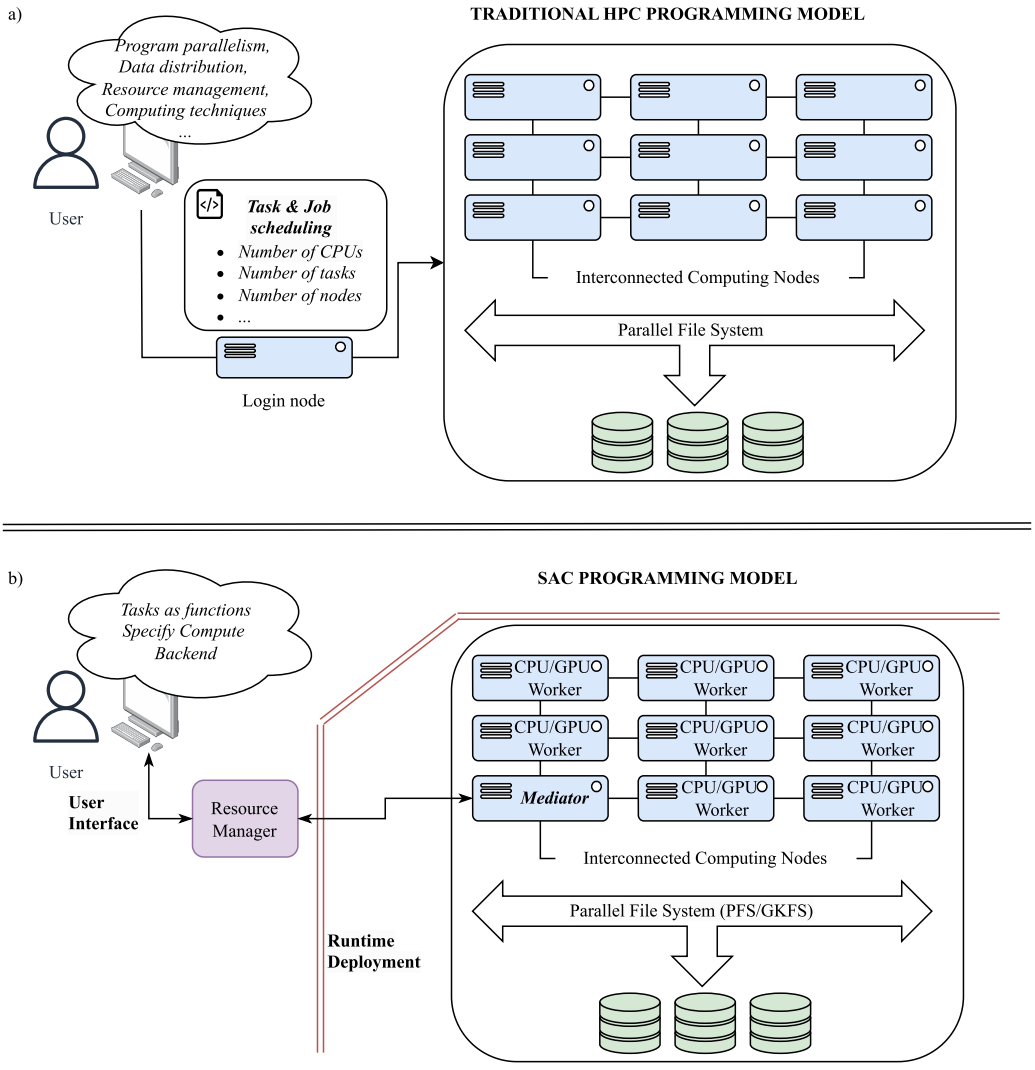


Figure 49: Using HPC resources by non-expert users. Traditional model and using Lithops-HPC.

Once a runtime is deployed, different users can use it to execute their functions. It is also possible to scale up/down the system by appending or releasing CPU/GPU workers into an existing runtime (i.e user should just type *lithops hpc runtime scaleup<up/down> <runtime name>* to scale a particular compute backend).

Later, the users, in the client role, employ the available runtimes to compute their applications. They can list all available runtimes using the command line API and choose one by editing the configuration file (i.e user should just type *lithops hpc runtime_list* to list all available compute backends). It allows clients to execute their applications, reusing a deployed backend, without interacting with

the infrastructure. However, similar to other HPC platforms, an account and credentials to connect to the supercomputer are still mandatory. To simplify the connection with the cluster, Lithops-HPC includes the necessary commands in the command line API; it abstracts the necessary steps to log in to the cluster, attach the runtimes, and mount the file system (i.e user should just type *lithops hpc connect* to connect with the HPC cluster via SSH key pair).

9.2.1 Architecture Implementation

Lithops-HPC architecture comprises the following components.

- **Compute backend:** The essential component in the compute backend is the CPU/GPU worker. A worker represents the computational unit, and it is responsible for executing the user-defined functions. Lithops-HPC is based on a modified version of Lithops' multi-cloud serverless framework.

- **Storage backend:** The storage component is responsible for storing user function instructions, input data, and the corresponding results. Lithops-HPC model supports both Parallel File System (PFS) and GekkoFS (GKFS) [222]. Lithops-HPC creates a directory inside the client machine that will be used as a mount point. It abstracts the storage tasks from the user and allows any modifications made inside that directory will be replicated to the filesystem inside the HPC machines. Besides, by using the specialized GKFS storage system, the local disk inside each compute node in the HPC cluster is aggregated to produce a high-performance storage space that can be accessed in a distributed manner.

- **Communication middleware:** A message broker software enables communication between the client and the compute backend. Lithops-HPC employees the Advanced Message Queuing Protocol (AMQP) to store the defined functions into a unique work queue (aka: Task Queue). This process encapsulates a task as a message and sends it to each CPU/GPU worker in the compute backend. Once the runtime is chosen by the user, the tasks are appended to the current task queue. Users can share a task queue across several runtimes, allowing scaling up the systems, by setting the **rmq_queue** parameter in the configuration file. It allows to scaleup the system even using several machines. Pending tasks are automatically distributed to all the runtimes with the same rmq_queue values and returned to the original queue in case some runtime is canceled (scaled down the system).

- **Client:** Lithops-HPC abstracts all the necessary steps to log in to the cluster, attach the runtimes, and mount the file system. Lithops-HPC spawns a local client on a PC and then does a double port forward to the message broker through the HPC login node to reach the compute backend. Lithops-HPC provides a complete command-line API (CLI) that allows users to manage the HPC with minimal effort.

- **Executor:** It is the main component inside the client that abstracts away the complexity of cloud backends and storage systems, allowing users to run distributed functions at scale with minimal configuration. It orchestrates the execution of tasks, launches parallel workers, packages the function and data, handles transmission to remote workers, tracks task progress, retrieves logs, stats, and finally recollects the results.

- **Resource manager:** It is a component that acts as a middle layer between the backend and the client, receiving all the jobs' petitions to run. The user interacts with the manager, indicating a lithops configuration file and the lithops job it wish to run. Then the scheduler invokes the client, which in turn deploys the jobs. Then, it reads the statistics from the Communication middleware. It uses Prometheus to retrieve data regarding the status of each of the queues using a REST API. Moreover, it reads the information regarding the backends in the lithops configuration file to estimate how much each of them can handle. Later, the manager decides which backend each of the jobs should be run on, so it is also able to enforce resource management

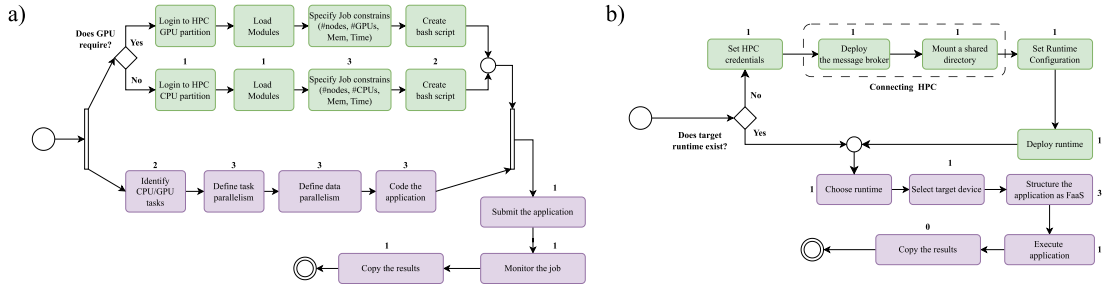


Figure 50: HPC vs Lithops Workflow deployment complexity

policies, although at the moment it merely implements a First Fit policy. The current implementation served only the purpose of demonstrating that it is possible to combine multiple backends and implement policies.

9.2.2 Programming model

The basic component in the Lithops-HPC programming model is a function. A function represents a set of instructions (aka: Task) that is applied to a set of data. Lithops-HPC supports the following execution modes. i) asynchronous call: one single function is computed in a worker. ii) map call: parallel execution of data is done across multiple workers. iii) map-reduce: first, a map function for each intermediate data is executed, then a reduce function merges all intermediate results associated.

Lithops-HPC is focused on data movement efficiency. By using Lithops, it automatically breaks the workload into smaller portions (aka: chunks), handing each portion to a separate worker. Therefore, scaling is totally dynamic and only limited by the designed program concurrency. In addition, Lithops-HPC employs decorators to handle GPU accelerators, which allows the same code can be executed in both CPU and GPU workers, only by setting the target device in the code.

9.3 Evaluation

9.3.1 Setup

The experiments described in the next lines were conducted inside three different clusters. The main cluster is the MareNostrum 5 (MN5) supercomputer. MN5 combines Lenovo ThinkSystem SD650 V3 and Eviden BullSequana XH3000 architectures, providing two partitions with different technical characteristics. The MN5-General-Purpose Partition comprises 6.408 nodes based on Intel Sapphire Rapids (4th Generation Intel Xeon Scalable Processors). The second cluster was NORD4, it is a Lenovo system composed of SD530 Compute Racks, an Intel Omni-Path high-performance network interconnected and running SuSE Linux Enterprise Server as the operating system. This general-purpose block consists of 1 rack housing 72 nodes with a grand total of 3456 processor cores. Finally, CTE-AMD is a cluster based on AMD EPYC processors, with Rocky Linux release 8.5 Operating System and an Infiniband interconnection network. It compomises 33 compute nodes, each of them 1 x AMD EPYC 7742 @ 2.250GHz (64 cores and 2 threads/core, total 128 threads per node) 1024GiB of main memory distributed in 16 dimms x 64GiB @ 3200MHz, 1 x SSD 480GB as local storage 2 x GPU AMD Radeon Instinct MI50 with 32GB Single Port Mellanox Infiniband HDR100, GPFS via two copper links 10 GBit.

9.3.2 Complexity

The first goal was to compare the perceived user-experience and its complexity compared to previous systems. Figure 50 compares the traditional HPC job submission workflow with the Lithops-HPC deployment process. Each block represents a step in the user workflow. Green lines indicate hardware interactions and purple blocks denote software-level interactions.

To quantify process complexity, each step was assigned a value between 0 and 3, inspired by complexity scoring models proposed by [223]. Cyclomatic Complexity [224], Yaqin's metric [225],

and Process Complexity [223] metrics were computed from these values. Cyclomatic Complexity measures the number of independent paths in a control flow graph, where lower values indicate simpler logic with fewer branching decisions, making systems easier to test and maintain (Eq1). Yaqin's Metric assesses the cognitive difficulty of understanding a process by analyzing structural elements based on AND, OR, and XOR branch complexities; a lower score indicates minor mental effort and fewer opportunities for user error (Eq2). Process Complexity evaluates the technical knowledge required to complete a framework, including the number of manual steps and system interactions. Lower process complexity implies a more accessible and streamlined user experience (Eq3).

$$M = E - N + 2P \tag{Eq1}$$

where:

M Cyclomatic complexity (number of independent paths)

E Number of edges in the control flow graph

N Number of nodes in the control flow graph

P Number of connected components (usually 1 for a single program)

$$Y = Ns + As + Cand + Cor + Cxor + Cyc + Cd \tag{Eq2}$$

where:

Y Yaqin's cognitive complexity score

Ns Graph size

$Cand/or/xor$ Branch complexity

$Ccyc$ Cyclical complexity

Cd Depth complexity

$$C_P = \sum(Wt) \tag{Eq3}$$

where:

C_P Process complexity score

Wt Wt is the weight for each one of the tasks

Table 18 summarizes the results. Cyclomatic-Complexity reveals that Lithops-HPC deployment requires approximately 1.5× fewer execution paths compared to the traditional HPC workflow, indicating a simpler and more streamlined process. Yaqin's metric shows that Lithops-HPC deployment introduces 1.6x fewer branches, loops, and nesting depth than traditional HPC, which means fewer cognitive activities for users. Finally, the Process Complexity score indicates that Lithops-HPC demands half as much technical knowledge to complete the workflow compared to traditional HPC submissions. These evaluations demonstrate that Lithops-HPC offers a significantly more user-friendly and less complex deployment experience than traditional HPC job submissions.

Backend Deployment	Complexity		
	Cyclomatic	Yaqin	Process
HPC	3	69	28
Lithops	2	43	13

Table 18: Complexity metrics

9.3.3 Lithops-HPC Overhead

Despite achieving the main goal of simplifying complexity, it is nonetheless important to assess whether it was achieved at a high cost or not. For this, we must assess Lithops-HPC framework compared to the traditional approach in supercomputers. To evaluate the overhead introduced by our Lithops-HPC framework, a FLOPs-intensive application, a matrix multiplication workload, was executed in two scenarios: traditional Python-MPI and Lithops-HPC using our Lithops-HPC framework. The experiment was run using 112, 1120, and 11200 tasks; each task comprises a square matrix multiplication of size 4086, with 5 loop iterations, where each task is executed on a single CPU (since each MN5 node contains 112 CPUs, 1, 10, and 100 MN5 nodes were employed, respectively). First, only the execution time was considered. The time required by the workload manager (Slurm) to allocate computational nodes was excluded from both cases. Figure 51 illustrates that the execution time per task is the same for all experiments when traditional MPI-Python is employed; however, our Lithops-HPC model introduces an overhead at the beginning and the end of each function. Figure 51 details the execution stages in the Lithops deployment. Four distinct states are shown: host submission, function start, function end, and result collection. The plot indicates that the Lithops-HPC model requires additional time to start the function and recollect the results, which increases with the number of tasks to execute. In particular, these times are required by the executor, the Lithops component in charge of invoking the worker and collecting the results (see 4.4 section). The overall overhead is up to 1.75x slower for 11200 tasks.

9.3.4 Time to service

Although our Lithops-HPC model introduces an overhead, it allows reusing the resources assigned to an application. This reusage can be enabled without requiring a new allocation through the Slurm manager, in opposition to traditional MPI-based approaches. Consequently, despite the performance to run an application may be degraded, in the overall picture, users may now submit multiple applications and or experiments without waiting to be allocated new resources. To assess how much time is gained, we make a new experiment to assess the time to service. The experiment comprises relaunching the application 10 times and calculating the total time required to run and finish them all, including resource allocation. For this experiment, we acquired resources from the Spanish Supercomputing Network (RES), which granted us our own partition in the MareNostrum 5 supercomputer. This allowed us to have priority over resource allocation and thus make a fair comparison to the priorities scientific users of the supercomputer typically get, as they typically request RES grants as well. Figure 52 illustrates the total time used in both scenarios (Python-MPI and Lithops-HPC). Subfigure a) shows the time taken when the nodes are already allocated, while b) considers as well the time required to perform the first backend allocation. Thu,s this figure shows that reusing the backend deployed by Lithops, it is possible to reduce the total execution time by 1.5x, and that, in both scenarios, it is better than using a pure-MPI approach for this reason. We must clarify, however, that if we looked at individual runs and not a collection of them, MPI would be faster. Thus, we are improving time to service when considering repetitive experiments, which is a common case in research groups where several people need to run, and Lithops-HPC allows them to reuse resources, unlike MPI.

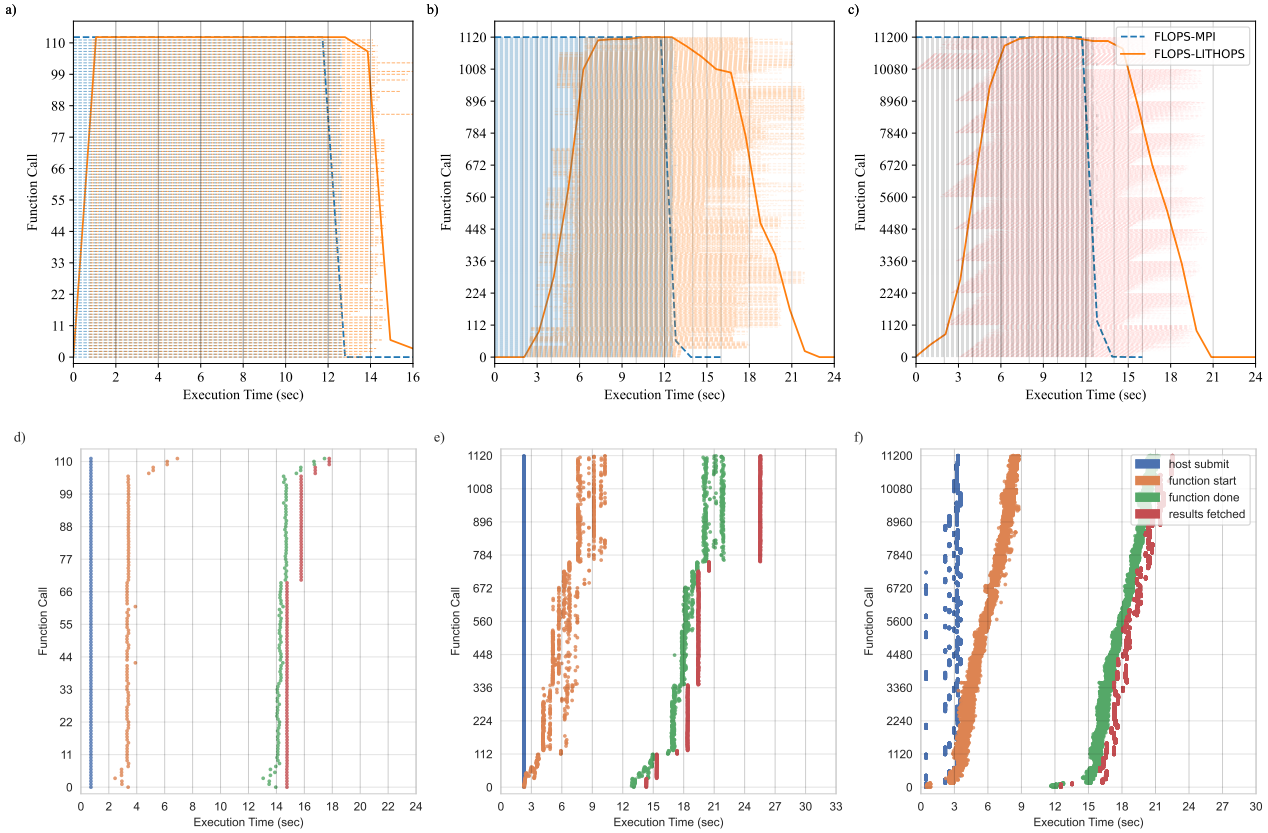


Figure 51: Python MPI execution vs Lithops execution. (a) reuses the resources allocated while (b) makes a new resource allocation - hence the extra time to complete.

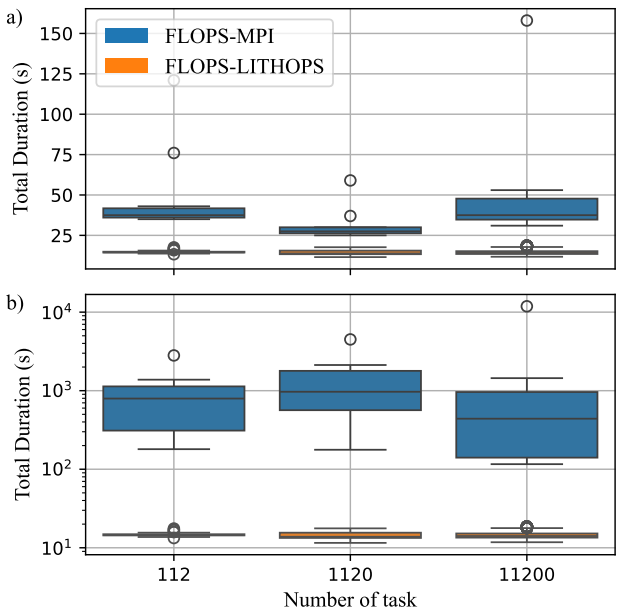


Figure 52: Python MPI execution vs Lithops execution

9.3.5 High-performance

The performance achieved by our framework was evaluated using the standard FLOPs and Object-Storage Lithops' benchmarks [226]. FLOPs application calculates the computing power by executing an n-square matrix multiplication during m iterations and reports the total of Floating Point Operations Per Second (FLOPs). The Object Storage benchmark measures the speed of data transfer by writing and reading a file of size n MB and reports the bandwidth (MB/s). We evaluated both compute and storage performance on high-performance computing (HPC) resources and compared the results against reports from popular cloud platforms at a similar scale (each task computes a matrix of size 4096 with 5 loop iterations, for the FLOPS benchmark, and writes/reads a file size of 512MB for the object-storage benchmark).To demonstrate the ability of our framework to leverage specialized resources such as accelerators, we included GPU-based experiments.

Figures 53 and 54 illustrate the performance comparison for 1,000 concurrent tasks executed in both cloud platforms and the HPC system. (Results for 100 tasks are similar and follow the same trend). The results show that the Lithops-HPC framework achieves higher performance in FLOPS than any cloud offering, up to 3× for HPC-CPU runtime and 30× for HPC-GPU runtime, faster processing than the best-performing FaaS service (AWS Lambda). Furthermore, performance is more consistent across tasks than in any other service, with AWS Lambda being closer. This allows a better predictability of application performance, which is key for HPC programs.

Similarly, storage access in HPC is faster across the board (3× AWS). In contrast, variability across functions here is higher for the HPC system due to the shared and centralized nature of the PFS. One of the attempts to further improve Lithops-HPC was to introduce GKFS, with he idea of leveraging local hard disks available on each of the supercomputers' nodes as a hot cache for GPFS (the underlying network filesystem across the supercomputer). While results were promising, the number of InfiniBand sockets GKFS needs to open diminished its results and shows no improvements with respect to GPFS. However, further work will be done regarding the required number of open sockets to achieve the expected performance.

Nonetheless, already with the GPFS filesystem, Lithops-HPC framework provides an ultra-fast storage system (30× AWS) with an excellent bandwidth (around 700 MB/s in both write and read operations), which is outstanding for data processing applications that transfer lots of data.

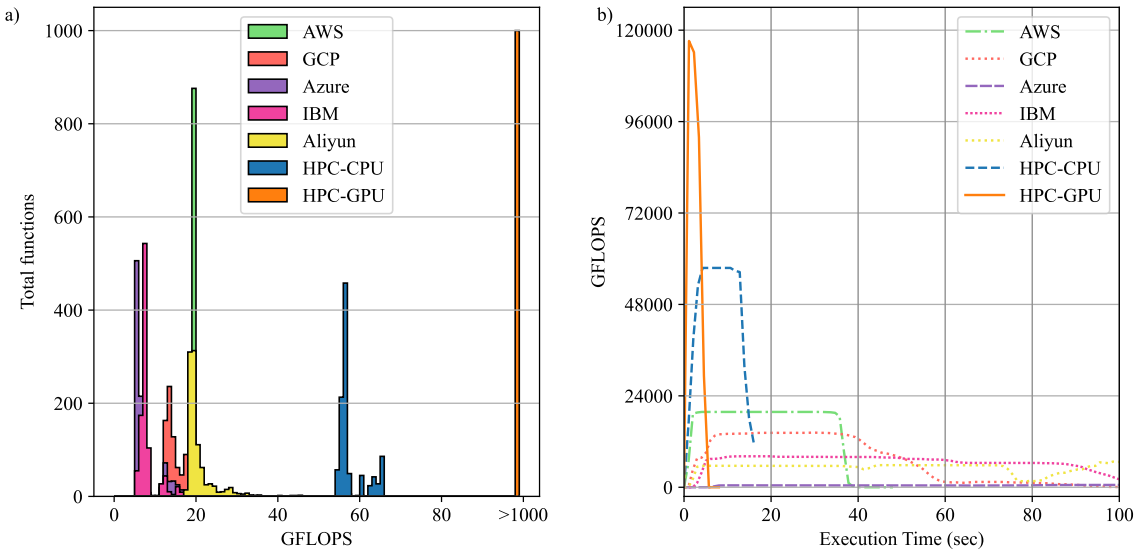


Figure 53: FLOPS benchmark on cloud FaaS and Lithops-HPC (1000 tasks). FLOPS per function histogram (left) and peak aggregate (right).

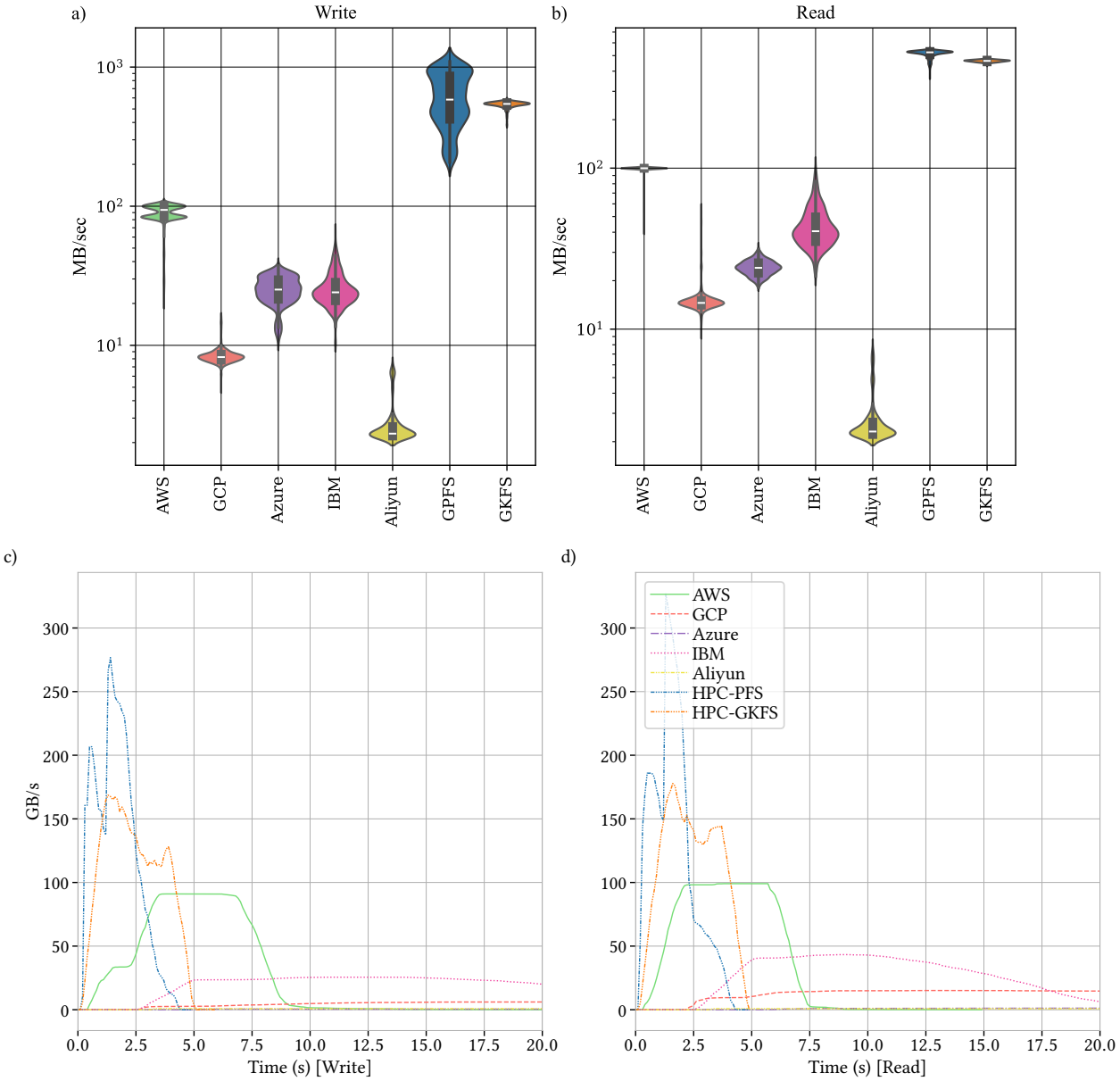


Figure 54: Storage benchmark on cloud FaaS and Lithops-HPC (1000 tasks). The top row shows a box plot of per-function storage bandwidth. Bottom row shows the aggregate of functions along wall-clock time.

9.3.6 Scaling

We repeated the previous FLOPS and storage benchmarks at large scale to demonstrate how our framework enables seamless scaling to significantly larger resource pools than typical cloud services, without additional user effort. Figure 55 compares the execution time and peak GFLOPs achieved by Lithops-HPC across varying numbers of workers (1 worker = 1 CPU running a task). The blue line represents the expected execution time, while the gray line indicates the overhead introduced. The figure also illustrates the distribution of the achieved GFLOPs across the total number of workers.

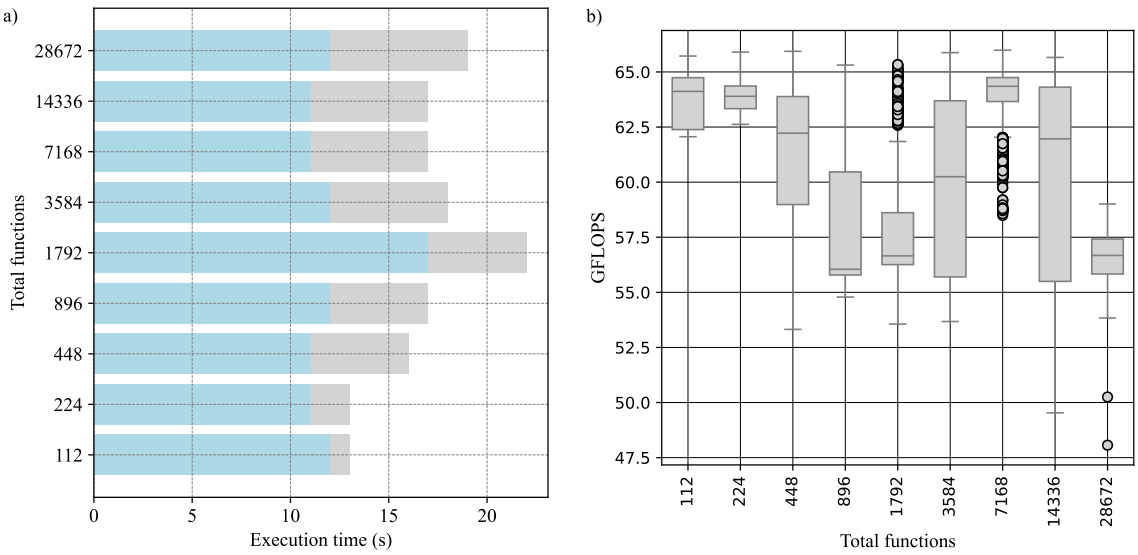


Figure 55: Massive FLOPS benchmark.

9.3.7 Resource management

To illustrate the first fit policy implemented in our resource manager (described in section 4.4) we have designed an experiment to demonstrate that Lithops-HPC can offload to other backends once the preferred one reaches the saturation point. To this end, we have run 1000 instances of FLOPS. Each of the runs are separated by 400 miliseconds. In this experiment, we set a maximum of 100 cores in the CPU backend, with so many others in the GPU backend.

Figure 56 shows the number of jobs executed in each of the backends during the launch of the aforementioned 1000 instances. It can be observed how, after around 200 seconds, the CPU backend becomes saturated, at which point jobs start to be run on the GPU backend (red line), thus offloading there. The total number of jobs run on the GPU is quite small compared to those run on the CPU, as the preferred backend is that one. This experiment demonstrates the ability to implement scheduling policies to effectively manage resources. In the future, we could implement more sophisticated policies and feed information to them using ML-based approaches.

9.3.8 Multi-cluster deployment

The concept of backend offloading can be extended to the cluster level. In this scenario, three runtimes, sharing the same task queue, were deployed across the three HPC clusters (MN5, NORD4, and CTE-AMD). Then, trhee application were submitted. Each application corresponds a FLOPs execution for 5000, 3000, and 2000 tasks, respectively. Figure 57 illustrates the execution schedule, showcasing how tasks are automatically distributed across all clusters. It shows as the task are automatically assigned when a new runtime is available and how the task are requeued when a runtime is canceled. In this scenario there is no orchestration in place: all backends share the same queue and filesystem - which is a requirement of the RabbitMQ queue system used -. Thus, when a message (i.e, a function) gets into the queue the first backend to look in it for a function to execute will. Therefore we can say the distribution is performed in a FCFS fashion where the backend with the best trade-off between resources and computational power will consume most of the workload. For this reason the GPU backends consume more than the HPC.

9.4 Related Work

Serverless data analytics have been fervently explored in the literature. These works aim at delivering a compute substrate for data processing workloads that is highly elastic and provided based on demand, at very fine granularity, so that users do not need to provision any resources and pay

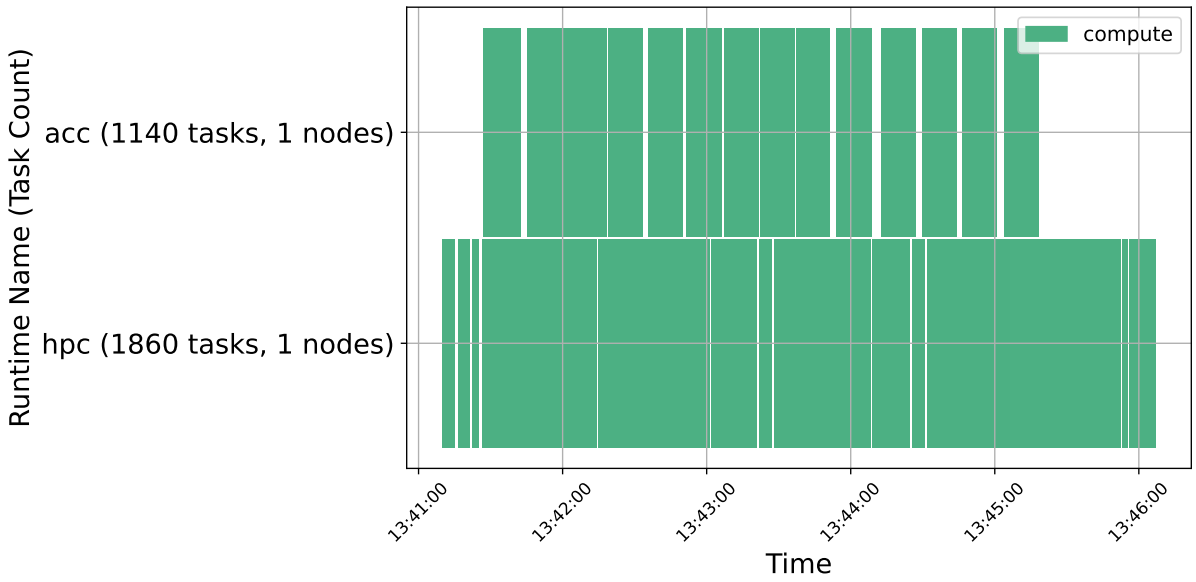


Figure 56: Offloading of tasks across HPC and GPU

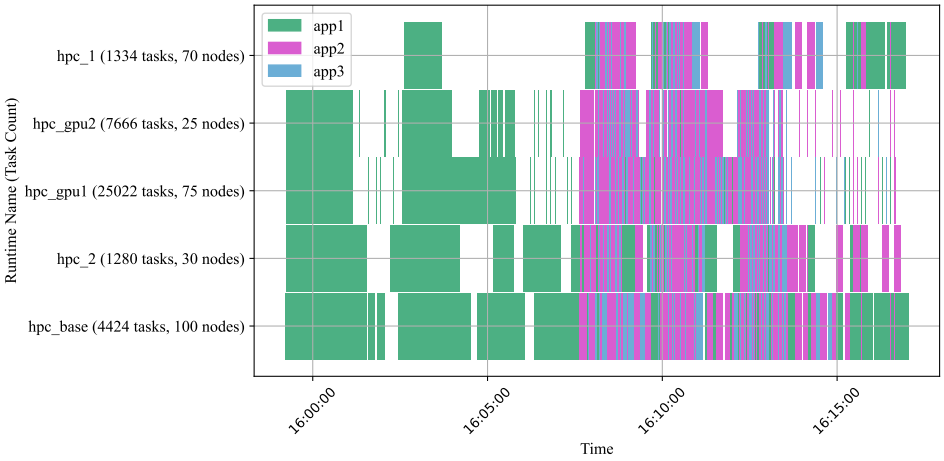


Figure 57: Application running using different runtimes.

only for the exact resources that they need, when they need them. Most serverless data processing frameworks [220, 92, 221] leverage the Function-as-a-Service (FaaS) paradigm for running stages of functions in parallel that are spawned based on data volume and are finely billed by milliseconds of active computation. Some approaches extend this idea by designing function orchestration systems to define workflows that manage function dependencies (i.e, tasks in a graph) and enable complex serverless applications [227, 228, 229, 230, 231, 232, 233, 234].

A critical challenge in serverless data analytics is in handling application state, i.e, how data moves through a workflow when functions are ephemeral and stateless. Solutions to this challenge include extending the FaaS model with a shared memory space [235, 236, 237, 238], or designing specialized external storage systems [239, 240, 84, 241, 242, 243, 189].

Several frameworks aim at simplifying the usage of HPC clusters. Most tooling and middleware in this line become wrappers around the cluster manager system (, Slurm or similar) with the goal to hide its complexities for users who are not knowledgeable in managing resources at a low level and are ready to sacrifice some performance for the sake of convenience. Among many instances of this

are Pegasus [217], COMPSs [218], Parsl [219], and cluster-based dataflow engines like Dask, Spark, Flink, or Ray.

Some interaction between serverless ideas and HPC has already been explored. HPC-Whisk [244], for example, adapts the OpenWhisk open-source FaaS platform to run serverless functions in an HPC cluster by deploying low-priority jobs in the supercomputer that operate as OpenWhisk's invoker nodes (function workers). rFaaS [245] is an RDMA-enabled FaaS implementation that uses idle supercomputer resources to execute short-running ephemeral functions dynamically, and [246] further explores resource wastage in HPC, advocating for a serverless approach to fill idle computation spaces. funcX [247] explores a federated FaaS platform to spread on multiple locations, even supercomputers. An early exploration of the Lithops model on HPC [248] uses Singularity containers to manually deploy workers on a supercomputer, showing promising results.

In contrast to literature, this work extends the full serverless experience to HPC data analytics, rather than only supporting FaaS execution. We embrace serverless usability to enable HPC users to code applications through simple Python functions that run in parallel based on data volume. This model is more familiar to non-expert users than other frameworks that require them to deploy and manage resources and code applications with complex parallel and distributed computing techniques. Whereas HPC-Whisk and rFaaS could serve as compute backends on HPC for a framework like Lithops, they primarily target short-lived functions running on idle HPC resources, which are unsuitable for data analytics workloads. Instead, our approach allocates larger compute resources for running functions as an alternative to (although not replacing) traditional resource management in supercomputers. In this scenario, function workers can be shared between users for better resource utilization, and requested or returned dynamically to the underlying platform to match demand.

9.5 Discussion

The results presented in the previous section demonstrate that the Lithops-HPC framework effectively simplifies the deployment of high-performance computing (HPC) workloads. By reducing the number of required user interactions with hardware and software, and by abstracting deployment complexity, Lithops-HPC offers a more accessible alternative to traditional sbatch-based HPC job submissions. Furthermore, Lithops-HPC does not introduce significant overhead over a naïve implementation of Python multiprocessing.

Moreover, although traditional HPC implementations (C code, MPI, etc.) can be highly optimized for performance, they demand expertise and advanced programming concepts that move away from the common environment for non-expert HPC users. Thanks to the abstraction layers provided by Lithops and its integration into the HPC infrastructure, we showed that our Lithops-HPC framework enables a much simpler and faster deployment process than manual Python parallelization via Slurm. In addition, despite the fact that the traditional MPI approach offers better performance, it does not allow the reuse of the allocated resources, which means a waiting time for each new deployment. However, using Lithops-HPC, once a runtime is deployed, it can be reused across multiple applications by the same or different users, effectively eliminating repeated interactions with the underlying infrastructure and streamlining the workflow. Therefore, our Lithops-HPC strategy clearly outperforms MPI-based approaches in scenarios where scientific teams should work together and launch many applications.

Although Lithops-HPC methodology is similar to Cloud computing, performance evaluation confirms that the Lithops-HPC framework enables users to leverage high-performance resources efficiently, achieving superior performance compared to cloud-based deployments under the same parallelism levels. In fact, the evaluation showed that our HPC implementation outperforms cloud platforms in both FLOPS and storage bandwidth. Furthermore, while cloud providers typically limit large-scale CPU access, Lithops-HPC makes it relatively easy to provision tens or even hundreds of CPUs. We demonstrated that the Lithops-HPC can easily handle workloads with more than 20,000 parallel functions without degrading performance or increasing the programming effort.

Importantly, Lithops-HPC allows users' functions to automatically detect GPU availability and run accordingly. It enables demanding applications to benefit significantly from GPU acceleration

without additional effort. Furthermore, by sharing the same task queue, Lithops-HPC exhibits strong adaptability across heterogeneous clusters, allowing functions to leverage all the computational resources and for the compute backend to adapt to workload variations. It represents a significant improvement in HPC programmability, as traditional models are often rigid and lack such dynamic resource selection.

9.6 Future Work

Despite its advantages, Lithops-HPC has some limitations that need handling. First, it introduces minimal overhead during function initialization and result collection, which affects performance compared to traditional HPC implementations. Second, while the current implementation supports a basic resource manager, switching between CPU and GPU runtimes, it lacks advanced scheduling strategies such as workload-aware balancing, energy-efficient execution, or cost-aware scheduling. Another concern is security: while authentication is required to access clusters, storage protection mechanisms are limited to directory permissions, and thus are exposed to potential vulnerabilities. Lastly, Lithops-HPC is inspired by the Function-as-a-Service (FaaS) model, which emphasizes infrastructure abstraction and event-driven executions but also introduces limitations in function concurrency, especially when tasks have complex interdependencies. Managing such dependencies or shared state between functions remains an open challenge for future development in our Lithops-HPC model and FaaS area.

10 Conclusions and Next Steps

A summary of the KPIs for XtremeHub components is illustrated in Table 19.

Component	KPI	Results
XtremeHub Compute	KPI-2	Burst Computing enables group-aware invocation and locality-optimized communication, improving throughput by up to 3× and reducing invocation latency by 11.5×.
	KPI-3	Burst Computing improves worker simultaneity by 26.5×, enabling full parallelism and simplifying deployment of interactive workloads like grid search and PageRank. It supports elastic, resource-agnostic programming via a single group invocation primitive.
	KPI-2	Our serverless Vector DBs implementation achieves sub-second batch query latency and reduces indexing time by 9.2× to 65.6× compared to Milvus.
	KPI-2	Our block-based data partitioning scheme for serverless vector DBs outperforms clustering-based (Vexless) in indexing performance by 3.5× to 5.8×, and reduces cost by 56% to 63%, while supporting continuous data ingestion and dynamic workloads.
	KPI-5	Serverless Vector DBs simplify deployment of RAG workloads, reduces operational tasks, and reduces costs up to 98% compared to Milvus.
XtremeHub Streams	KPI-3	FaaStream supports coordinated auto-scaling of serverless pipelines, adapting to fluctuating workloads and outperforming Kafka-based setups in throughput and latency.
	KPI-2	FaaStream achieves 22.5× higher throughput than Lithops with S3 for small events (*e.g.*, 1KB).
	KPI-2	FaaStream achieves up to 80.99% lower latency than AWS Kinesis in streaming IO.
	KPI-1	FaaStream achieves up to up to 25.80% faster data shuffling than Serverless Spark.
	KPI-5	FaaStream allows executing the same serverless job both in batch and streaming without almost no changes in the job's code.
XtremeHub Security	KPI-4	SCONE-protected Pravega clients show affordable latency increase (2×) at low throughput and negligible overhead at high throughput (*e.g.*, 10k events/sec), making it practical to use in latency-sensitive applications.
XtremeHub Connectors	KPI-1	Nexus streamlets enable programmable data reduction techniques like compression, achieving 2.6× to 3.9× better compression ratios than event streaming systems built-in compression (*e.g.*, Kafka, Pulsar).
	KPI-1	Nexus partition-aware data routing allows for stateful streamlets while reducing metadata reads by ≈ 98% due to linearization of operations.
	KPI-1	Lithops-HPC achieves higher FLOPs and object storage throughput than popular cloud platforms, demonstrating up to 3× better compute performance and efficient data transfer using GKFS-backed parallel storage.
	KPI-5	Lithops-HPC reduces deployment complexity compared to traditional HPC workflows: Cyclomatic Complexity drops from 3 to 2, Yaqin's metric from 69 to 43, and Process Complexity from 28 to 13.

Table 19: KPIs and results achieved by XtremeHub components in D3.2, aligned with project-level indicators.

The D3.2 deliverable marks a significant milestone in the evolution of XtremeHub, demonstrating its maturity as a programmable data plane for near-data processing across heterogeneous infrastructures. By integrating advanced components such as Burst Computing, FaaStream, Serverless Vector DBs, Nexus, and HPC connectors, the reference implementation showcases how XtremeHub can support dynamic, latency-sensitive, and data-intensive workloads with elasticity, performance, and security. The system's ability to unify compute, stream, and security capabilities—validated through rigorous benchmarking and real-world use cases—positions it as a robust foundation for next-generation serverless analytics in domains like genomics, metabolomics, and surgiomics.

The results presented in this deliverable provide strong evidence of XtremeHub's alignment with NEARDATA's KPIs. From significant throughput and latency improvements in ETL and video analytics, to demonstrable auto-scaling, confidential computing, and user-centric platform simplicity, XtremeHub delivers on its promise of enabling efficient and secure data processing at scale. The convergence of its components into a cohesive runtime paves the way for broader exploitation in NEARDATA's use cases and future adoption in International Health Data Spaces. Looking ahead, the deliverable sets the stage for further integration, orchestration, and deployment of XtremeHub across diverse infrastructures and application domains.

References

[1] G. París, P. García-López, and M. Sánchez-Artigas, "Serverless Elastic Exploration of Unbalanced Algorithms," in 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), (Los Alamitos, CA, USA), pp. 149–157, IEEE Computer Society, Oct. 2020. ISSN: 2159-6190.

[2] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: enabling quality-of-service in serverless computing," in Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, (New York, NY, USA), pp. 311–327, Association for Computing Machinery, Oct. 2020.

[3] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized Core-granular Scheduling for Serverless Functions," in Proceedings of the ACM Symposium on Cloud Computing, SoCC '19, (New York, NY, USA), pp. 158–164, Association for Computing Machinery, Nov. 2019.

[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, (New York, NY, USA), pp. 445–451, Association for Computing Machinery, Sept. 2017.

[5] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), (Boston, MA), pp. 363–376, USENIX Association, Mar. 2017.

[6] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the IBM cloud," in Proceedings of the 19th International Middleware Conference Industry, Middleware '18, (New York, NY, USA), p. 1–8, Association for Computing Machinery, 2018.

[7] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), (Renton, WA), pp. 475–488, USENIX Association, July 2019.

[8] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A Berkeley view on serverless computing," Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[9] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in Proceedings of the ACM Symposium on Cloud Computing, SoCC '18, (New York, NY, USA), pp. 263–274, Association for Computing Machinery, Oct. 2018.

[10] S. Werner and S. Tai, "A reference architecture for serverless big data processing," Future Generation Computer Systems, vol. 155, pp. 179–192, 2024.

[11] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: a scalable and locality-enhanced framework for serverless parallel computing," in Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, (New York, NY, USA), p. 1–15, Association for Computing Machinery, 2020.

[12] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful Serverless Computing with Crucial," ACM Transactions on Software Engineering and Methodology, vol. 31, pp. 39:1–39:38, Mar. 2022.

[13] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), (Boston, MA), pp. 193–206, USENIX Association, Feb. 2019.

[14] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," 2018.

[15] D. Barcelona-Pons and P. García-López, "Benchmarking Parallelism in FaaS Platforms," Future Generation Computer Systems, vol. 124, pp. 268–284, Oct. 2020.

[16] P. Garcia Lopez, A. Slominski, B. Metzler, M. Berhendt, and S. Shillaker, "Serverless end game: Disaggregation enabling transparency," in Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies, SESAME '24, (New York, NY, USA), p. 9–14, Association for Computing Machinery, 2024.

[17] I. Müller, R. Bruno, A. Klimovic, J. Wilkes, E. Sedlar, and G. Alonso, "Serverless Clusters: The Missing Piece for Interactive Batch Applications?." Presented at 10th Workshop on Systems for Post-Moore Architectures (SPMA 2020), Apr. 2020.

[18] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: stateful functions-as-a-service," Proceedings of the VLDB Endowment, vol. 13, pp. 2438–2452, July 2020.

[19] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in 2020 USENIX Annual Technical Conference (USENIX ATC 20), (Boston, MA), pp. 419–433, USENIX Association, July 2020.

[20] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, (New York, NY, USA), p. 281–295, Association for Computing Machinery, 2020.

[21] B. Wadie, L. Stuart, C. M. Rath, B. Drotleff, S. Mamedov, and T. Alexandrov, "Metaspace-ml: Context-specific metabolite annotation for imaging mass spectrometry using machine learning," Nature Communications, vol. 15, no. 9110, 2024.

[22] Y. Li, S. J. Park, and J. Ousterhout, "MilliSort and MilliQuery: Large-Scale Data-Intensive computing in milliseconds," in 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), (Boston, MA), pp. 593–611, USENIX Association, Apr. 2021.

[23] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards serverless as commodity: a case of Knative," in Proceedings of the 5th International Workshop on Serverless Computing, WOSC '19, (New York, NY, USA), p. 13–18, Association for Computing Machinery, 2019.

[24] W. Qiu, M. Copik, Y. Wang, A. Calotoiu, and T. Hoefler, "User-guided Page Merging for Memory Deduplication in Serverless Systems," in 2023 IEEE International Conference on Big Data (BigData), (Los Alamitos, CA, USA), pp. 159–169, IEEE Computer Society, Dec. 2023.

[25] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, "MXFaaS: Resource sharing in serverless environments for parallelism and efficiency," in Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[26] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS track: Building stateful distributed applications with serverless architectures," in Proceedings of the 20th International Middleware Conference, Middleware '19, (New York, NY, USA), pp. 41–54, Association for Computing Machinery, dec 2019.

[27] Y. Li, L. Zhao, Y. Yang, and W. Qu, "Rethinking deployment for serverless functions: A performance-first perspective," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[28] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service workflows," in 2021 USENIX Annual Technical Conference (USENIX ATC 21), (Boston, MA), pp. 805–820, USENIX Association, July 2021.

[29] F. Lu, X. Wei, Z. Huang, R. Chen, M. Wu, and H. Chen, "Serialization/deserialization-free state transfer in serverless workflows," in Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24, (New York, NY, USA), p. 132–147, Association for Computing Machinery, 2024.

[30] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler, "FMI: Fast and cheap message passing for serverless functions," in Proceedings of the 37th International Conference on Supercomputing, ICS '23, (New York, NY, USA), p. 373–385, Association for Computing Machinery, 2023.

[31] D. Barcelona-Pons, P. García-López, and B. Metzler, "Glider: Serverless ephemeral stateful near-data computation," in Proceedings of the 24th International Middleware Conference, Middleware '23, (New York, NY, USA), p. 247–260, Association for Computing Machinery, 2023.

[32] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.

[33] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), (Los Alamitos, CA, USA), pp. 502–504, IEEE Computer Society, July 2019.

[34] B. Huang, S. Huang, J. Dai, J. Huang, and T. Xie, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), (Los Alamitos, CA, USA), pp. 41–51, IEEE Computer Society, Mar. 2010.

[35] M. Sánchez-Artigas, G. T. Eizaguirre, G. Vernik, L. Stuart, and P. García-López, "Primula: a practical shuffle/sort operator for serverless computing," in Proceedings of the 21st International Middleware Conference Industrial Track, Middleware '20, (New York, NY, USA), p. 31–37, Association for Computing Machinery, 2020.

[36] B. Liston, "Ad hoc big data processing made simple with serverless MapReduce," 2016.

[37] S. Thomas, L. Ao, G. M. Voelker, and G. Porter, "Particle: ephemeral endpoints for serverless networking," in Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, (New York, NY, USA), pp. 16–29, Association for Computing Machinery, Oct. 2020.

[38] C. Lee and J. Ousterhout, "Granular computing," in Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '19, (New York, NY, USA), p. 149–154, Association for Computing Machinery, 2019.

[39] C. Segarra, S. Shillaker, G. Li, E. Mappoura, R. Bruno, L. Vilanova, and P. Pietzuch, "GRANNY: Granular management of Compute-Intensive applications in the cloud," in 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25), (Philadelphia, PA), pp. 205–218, USENIX Association, Apr. 2025.

[40] Z. Ruan, S. Li, K. Fan, S. J. Park, M. K. Aguilera, A. Belay, and M. Schwarzkopf, "Quicksand: Harnessing stranded datacenter resources with granular computing," in 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25), (Philadelphia, PA), pp. 147–165, USENIX Association, Apr. 2025.

[41] M. Copik, A. Calotoiu, G. Rethy, R. Böhringer, R. Bruno, and T. Hoefler, "Process-as-a-service: Unifying elastic and stateful clouds with serverless processes," in Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC '24, (New York, NY, USA), p. 223–242, Association for Computing Machinery, 2024.

[42] G. T. Eizaguirre, D. Barcelona-Pons, A. Arjona, G. Vernik, P. García-López, and T. Alexandrov, "Serverful functions: Leveraging servers in complex serverless workflows (industry track)," in Proceedings of the 25th International Middleware Conference Industrial Track, Middleware Industrial Track '24, (New York, NY, USA), p. 15–21, Association for Computing Machinery, 2024.

[43] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen, "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), (Boston, MA), pp. 497–517, USENIX Association, July 2023.

[44] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, (New York, NY, USA), pp. 152–166, Association for Computing Machinery, Apr. 2021.

[45] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rFaaS: Enabling high performance serverless with rdma and leases," in 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), (Los Alamitos, CA, USA), pp. 897–907, IEEE Computer Society, may 2023.

[46] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in 2018 USENIX Annual Technical Conference (USENIX ATC 18), (Boston, MA), pp. 923–935, USENIX Association, July 2018.

[47] R. Basu Roy, T. Patel, R. Liew, Y. N. Babuji, R. Chard, and D. Tiwari, "ProPack: Executing concurrent serverless functions faster and cheaper," in Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC '23, (New York, NY, USA), p. 211–224, Association for Computing Machinery, 2023.

[48] Z. Wu, Y. Deng, Y. Zhou, J. Li, S. Pang, and X. Qin, " FaaSBatch: Boosting Serverless Efficiency With In-Container Parallelism and Resource Multiplexing ," IEEE Transactions on Computers, vol. 73, pp. 1071–1085, Apr. 2024.

[49] C. Jin, Z. Zhang, X. Xiang, S. Zou, G. Huang, X. Liu, and X. Jin, "Ditto: Efficient serverless analytics with elastic parallelism," in Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23, (New York, NY, USA), p. 406–419, Association for Computing Machinery, 2023.

[50] M. Abdi, S. Ginzburg, X. C. Lin, J. Faleiro, G. I. Chaudhry, I. Goiri, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette Load Balancing: Locality Hints for Serverless Functions," in Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys '23, (New York, NY, USA), pp. 365–380, Association for Computing Machinery, May 2023.

[51] 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 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), (Carlsbad, CA), pp. 303–320, USENIX Association, July 2022.

[52] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), (Boston, MA), pp. 653–669, USENIX Association, Apr. 2021.

[53] D. H. Liu, A. Levy, S. Noghabi, and S. Burckhardt, "Doing more with less: Orchestrating serverless applications without an orchestrator," in 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), (Boston, MA), pp. 1505–1519, USENIX Association, Apr. 2023.

[54] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), (Boston, MA), pp. 1489–1504, USENIX Association, Apr. 2023.

[55] S. Mohanty, V. M. Bhasi, M. Son, M. T. Kandemir, and C. Das, "Faastloop: Optimizing loop-based applications for serverless computing," in Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC '24, (New York, NY, USA), p. 943–960, Association for Computing Machinery, 2024.

[56] A. Flink, "Stateful functions," 2023.

[57] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun, "Sponge: Fast reactive scaling for stream processing with serverless frameworks," in 2023 USENIX Annual Technical Conference (USENIX ATC 23), (Boston, MA), pp. 301–314, USENIX Association, July 2023.

[58] A. Jain, A. F. Baarzi, G. Kesidis, B. Urgaonkar, N. Alfares, and M. Kandemir, "SplitServe: Efficiently splitting apache spark jobs across faas and iaas," in Proceedings of the 21st International Middleware Conference, Middleware '20, (New York, NY, USA), p. 236–250, Association for Computing Machinery, 2020.

[59] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Middleware '17, (New York, NY, USA), pp. 121–133, Association for Computing Machinery, Dec. 2017.

[60] M. Wawrzoniak, I. Müller, G. Alonso, and R. Bruno, "Boxer: Data Analytics on Network-enabled Serverless Platforms," in 11th Conference on Innovative Data Systems Research, CIDR 2021, (Chaminade, USA), CIDR, Jan. 2021.

[61] T. Schirmer, J. Scheuner, T. Pfandzelter, and D. Bermbach, "Fusionize: Improving Serverless Application Performance through Feedback-Driven Function Fusion," in 2022 IEEE International Conference on Cloud Engineering (IC2E), (Los Alamitos, CA, USA), pp. 85–95, IEEE Computer Society, Sept. 2022.

[62] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez, "Outsourcing Data Processing Jobs With Lithops," IEEE Transactions on Cloud Computing, vol. 11, pp. 1026–1037, Jan. 2023.

[63] M. Wawrzoniak, G. Moro, R. Bruno, A. Klimovic, and G. Alonso, "Off-the-shelf data analytics on serverless," in Proceedings of the 14th Conference on Innovative Data Systems Research, CIDR 2024, (Chaminade, USA), CIDR, 2024.

[64] M. Grohe, "word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data," in Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS'20, (New York, NY, USA), p. 1–16, Association for Computing Machinery, 2020.

[65] M. T. Pilehvar and J. Camacho-Collados, Embeddings in natural language processing: Theory and advances in vector representations of meaning. Switzerland: Morgan & Claypool Publishers, 2020.

[66] B. Klein, G. Lev, G. Sadeh, and L. Wolf, " Associating neural word embeddings with deep image representations using Fisher Vectors ," in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (Los Alamitos, CA, USA), pp. 4437–4446, IEEE Computer Society, June 2015.

[67] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: a hybrid analytical engine towards query fusion for structured and unstructured data," Proc. VLDB Endow., vol. 13, p. 3152–3165, Aug. 2020.

[68] Pinecone, "Pinecone.io," 2025.

[69] Weaviate, "Weaviate.io," 2025.

[70] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A purpose-built vector data management system," in Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, (New York, NY, USA), p. 2614–2627, Association for Computing Machinery, 2021.

[71] R. Guo, X. Luan, L. Xiang, X. Yan, X. Yi, J. Luo, Q. Cheng, W. Xu, J. Luo, F. Liu, Z. Cao, Y. Qiao, T. Wang, B. Tang, and C. Xie, "Manu: a cloud native vector database management system," Proc. VLDB Endow., vol. 15, p. 3548–3561, Aug. 2022.

[72] R. Shan, " A Deep Dive into Vector Stores: Classifying the Backbone of Retrieval-Augmented Generation ," in 2024 IEEE International Conference on Big Data (BigData), (Los Alamitos, CA, USA), pp. 8831–8833, IEEE Computer Society, Dec. 2024.

[73] J. J. Pan, J. Wang, and G. Li, "Survey of vector database management systems," The VLDB Journal, vol. 33, no. 5, pp. 1591–1615, 2024.

[74] Upstash, "Upstash - serverless data platform," 2025.

[75] AWS, "Amazon opensearch service as a vector database," 2025.

[76] "Aws lambda." `https://aws.amazon.com/en/lambda/`, 2024.

[77] "Microsoft azure functions." `https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview`, 2024.

[78] Google, "Cloud run functions," 2025.

[79] Y. Su, Y. Sun, M. Zhang, and J. Wang, "Vexless: A serverless vector data management system using cloud functions," Proc. ACM Manag. Data, vol. 2, May 2024.

[80] Azure, "Azure durable functions," 2025.

[81] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," ACM Comput. Surv., vol. 54, Nov. 2022.

[82] J. Manner, M. EndreB, T. Heckel, and G. Wirtz, " Cold Start Influencing Factors in Function as a Service ," in IEEE/ACM UCC Companion '18, (Los Alamitos, CA, USA), pp. 181–188, IEEE Computer Society, Dec. 2018.

[83] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 5, pp. 1–29, 2023.

[84] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler, "FMI: Fast and cheap message passing for serverless functions," in Proceedings of the 37th International Conference on Supercomputing, ICS '23, (New York, NY, USA), p. 373–385, Association for Computing Machinery, 2023.

[85] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in Proceedings of the 24rd International Conference on Very Large Data Bases, VLDB '98, (San Francisco, CA, USA), p. 194–205, Morgan Kaufmann Publishers Inc., 1998.

[86] R. Gracia-Tinedo, F. Junqueira, T. Kaitchuck, and S. Joshi, "Pravega: A tiered storage system for data streams," in Proceedings of the 24th International Middleware Conference, Middleware '23, (New York, NY, USA), p. 165–177, Association for Computing Machinery, 2023.

[87] J. Kreps, N. Narkhede, J. Rao, et al., "Kafka: A distributed messaging system for log processing," in NetDB '11, vol. 11, pp. 1–7, 2011.

[88] R. M. Esteves, T. Hacker, and C. Rong, "Competitive k-means, a new accurate and distributed k-means algorithm for large datasets," in Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01, CLOUDCOM '13, (USA), p. 17–24, IEEE Computer Society, 2013.

[89] M. I. Malinen and P. Fränti, "Balanced k-means for clustering," in Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshop, S+ SSPR 2014, Joensuu, Finland, August 20-22, 2014. Proceedings, (Berlin, Heidelberg, Germany), pp. 32–41, Springer, Springer, 2014.

[90] R. de Maeyer, S. Sieranoja, and P. Fränti, "Balanced k-means revisited," Applied Computing and Intelligence, vol. 3, no. 2, pp. 145–179, 2023.

[91] "Stateless serverless vector database," 2025.

[92] J. Sampé, M. Sánchez-Artigas, G. Vernik, I. Yehekzel, and P. García-López, "Outsourcing data processing jobs with lithops," IEEE Transactions on Cloud Computing, vol. 11, no. 1, pp. 1026–1037, 2023.

[93] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," 2025.

[94] J. Levy-Kramer, "k-means-constrained," Apr. 2018.

[95] AWS, "Configure lambda function memory," 2025.

[96] Zilliz, "VectorDBBench: A Benchmark Tool for VectorDB," 2025.

[97] A. B. Yandex and V. Lempitsky, " Efficient Indexing of Billion-Scale Datasets of Deep Descriptors ," in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (Los Alamitos, CA, USA), pp. 2055–2063, IEEE Computer Society, June 2016.

[98] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, " Going deeper with convolutions ," in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (Los Alamitos, CA, USA), pp. 1–9, IEEE Computer Society, June 2015.

[99] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, " ImageNet: A large-scale hierarchical image database ," in 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops), (Los Alamitos, CA, USA), pp. 248–255, IEEE Computer Society, June 2009.

[100] D. G. Lowe, "Distinctive image features from Scale-Invariant keypoints," International Journal of Computer Vision, vol. 60, pp. 91–110, Nov. 2004.

[101] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," International Journal of Computer Vision, vol. 42, pp. 145–175, May 2001.

[102] "K-means constrained," 2025.

[103] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," IEEE Transactions on Big Data, vol. 7, no. 3, pp. 535–547, 2019.

[104] J. J. Pan, J. Wang, and G. Li, "Vector database management techniques and systems," in Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS '24, (New York, NY, USA), p. 597–604, Association for Computing Machinery, 2024.

[105] "Google cloud functions." `https://cloud.google.com/functions`, 2024.

[106] Forrester, "Introducing forrester's function-as-a-service platforms landscape." `https://www.forrester.com/blogs/introducing-forresters-functions-as-a-service-landscape/`, 2023.

[107] "Aws serverless applications scenarios." `https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/scenarios.html`, 2024.

[108] "Serverless reference architecture: Real-time stream processing." `https://github.com/aws-samples/lambda-refarch-streamprocessing/`, 2024.

[109] Z. Jia and E. Witchel, "Boki: Towards data consistency and fault tolerance with shared logs in stateful serverless computing," ACM Trans. Comput. Syst., Sept. 2024. Just Accepted.

[110] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in USENIX NSDI '23, (Boston, MA), pp. 1489–1504, USENIX Association, Apr. 2023.

[111] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: enable efficient workflow execution for function-as-a-service," ASPLOS '22, (New York, NY, USA), p. 782–796, Association for Computing Machinery, 2022.

[112] Z. Cai, Z. Chen, X. Chen, R. Ma, H. Guan, and R. Buyya, "Spsc: Stream processing framework atop serverless computing for industrial big data," IEEE Transactions on Cybernetics, pp. 1–9, 2024.

[113] J. Sampé, M. Sánchez-Artigas, G. Vernik, I. Yehekzel, and P. García-López, "Outsourcing data processing jobs with lithops," IEEE Transactions on Cloud Computing, vol. 11, no. 1, pp. 1026–1037, 2023.

[114] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun, "Sponge: Fast reactive scaling for stream processing with serverless frameworks," in USENIX ATC 23, (Boston, MA), pp. 301–314, USENIX Association, July 2023.

[115] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in IEEE international conference on Big Data, pp. 2785–2792, IEEE, 2015.

[116] "Apache flink." `https://flink.apache.org`, 2023.

[117] "Apache spark." `https://spark.apache.org`, 2023.

[118] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al., "Spark: Cluster computing with working sets.," USENIX HotCloud '10, vol. 10, no. 10-10, p. 95, 2010.

[119] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in ACM SoCC '17, (New York, NY, USA), p. 445–451, Association for Computing Machinery, 2017.

[120] "Amazon kinesis." `https://aws.amazon.com/es/kinesis`, 2023.

[121] M. Sánchez-Artigas and G. T. Eizaguirre, "A seer knows best: Optimized object storage shuffling for serverless analytics," in ACM/IFIP Middleware '22, (New York, NY, USA), p. 148–160, Association for Computing Machinery, 2022.

[122] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in USENIX OSDI '18, (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.

[123] "Apache kafka." `https://kafka.apache.org`, 2023.

[124] "Apache pulsar." `https://pulsar.apache.org`, 2023.

[125] "Redpanda." `https://redpanda.com`, 2023.

[126] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in OSDI '20, pp. 1187–1204, USENIX Association, Nov. 2020.

[127] M. Wiesholler, F. Dinu, J. Picorel, and P. Bhatotia, "Indilog: Bridging scalability and performance in stateful serverless computing with shared logs," in ACM SYSTOR '24, (New York, NY, USA), p. 1–13, Association for Computing Machinery, 2024.

[128] "Apache pulsar - overview of tiered storage." `https://pulsar.apache.org/docs/2.11.x/tiered-storage-overview`, 2023.

[129] R. Gracia-Tinedo, F. Junqueira, T. Kaitchuck, and S. Joshi, "Pravega: A tiered storage system for data streams," in ACM Middleware '23, (New York, NY, USA), p. 165–177, Association for Computing Machinery, 2023.

[130] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful serverless computing with crucial," ACM Trans. Softw. Eng. Methodol., vol. 31, Mar. 2022.

[131] "Documentation for memory configuration in aws lambda." `https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html`, 2024.

[132] "Aws step functions service quotas." `https://docs.aws.amazon.com/step-functions/latest/dg/service-quotas.html`, 2023.

[133] "Seer terasort implementation." `https://github.com/GEizaguirre/terasort-lithops`, 2023.

[134] "Amazon emr." `https://docs.aws.amazon.com/emr/`, 2023.

[135] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in 2009 IEEE Conference on Computer Vision and Pattern Recognition, pp. 248–255, 2009.

[136] "Hadoop mapreduce tutorial: Example: Wordcount v1.0." `https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`, 2023.

[137] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in IEEE ICDEW '20, pp. 41–51, 2010.

[138] G. Vernik, M. Factor, E. K. Kolodner, P. Michiardi, E. Ofer, and F. Pace, "Stocator: Providing high performance and fault tolerance for apache spark over object storage," in IEEE/ACM CCGRID '18, pp. 462–471, 2018.

[139] D. S. (ltcmdrdata), "Plain text wikipedia 2020-11," 2020. Accessed: 2025-03-25.

[140] O. O. Yahoo!, "Terabyte sort on apache hadoop," Mai 2008 2008.

[141] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in USENIX NSDI '19, (Boston, MA), pp. 193–206, USENIX Association, Feb. 2019.

[142] M. Sánchez-Artigas, G. T. Eizaguirre, G. Vernik, L. Stuart, and P. García-López, "Primula: a practical shuffle/sort operator for serverless computing," in ACM Middleware '20 Industrial Track, (New York, NY, USA), p. 31–37, Association for Computing Machinery, 2020.

[143] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in 18th USENIX NSDI '21, pp. 653–669, USENIX Association, Apr. 2021.

[144] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," The Bulletin of the Technical Committee on Data Engineering, vol. 38, no. 4, 2015.

[145] G. Chantzialexiou, A. Luckow, and S. Jha, "Pilot-streaming: A stream processing framework for high-performance computing," in IEEE e-Science '18, pp. 177–188, 2018.

[146] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in ACM SoCC '20, (New York, NY, USA), p. 30–44, Association for Computing Machinery, 2020.

[147] "National center for tumor diseases (germany)." `https://www.nct-dresden.de/en/homepage`, 2025.

[148] "Apache kafka - documentation." `https://kafka.apache.org/documentation`, 2023.

[149] R. Gracia-Tinedo, F. Junqueira, B. Zhou, Y. Xiong, and L. Liu, "Practical storage-compute elasticity for stream data processing," in ACM Middleware '23 Industrial Track, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2023.

[150] T. Li, Y. Li, W. Zhu, Y. Xu, and J. C. S. Lui, "MinFlow: High-performance and cost-efficient data passing for I/O-intensive stateful serverless analytics," in USENIX FAST '24, (Santa Clara, CA), pp. 311–327, USENIX Association, Feb. 2024.

[151] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads," in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), (Boston, MA), pp. 363–376, USENIX Association, Mar. 2017.

[152] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in USENIX ATC '19, (Renton, WA), pp. 475–488, USENIX Association, July 2019.

[153] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler, "Fmi: Fast and cheap message passing for serverless functions," in ACM ICS '23, (New York, NY, USA), p. 373–385, Association for Computing Machinery, 2023.

[154] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in ACM SoCC '20, (New York, NY, USA), p. 281–295, Association for Computing Machinery, 2020.

[155] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in ACM SoCC '18, (New York, NY, USA), p. 263–274, Association for Computing Machinery, 2018.

[156] W. Zhang, V. Fang, A. Panda, and S. Shenker, "Kappa: a programming framework for serverless computing," in ACM SoCC '20, (New York, NY, USA), p. 328–343, Association for Computing Machinery, 2020.

[157] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: stateful functions-as-a-service," Proc. VLDB Endow., vol. 13, p. 2438–2452, July 2020.

[158] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A kvs for any scale," in IEEE ICDE '18, pp. 401–412, 2018.

[159] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in USENIX ATC 21, pp. 285–301, USENIX Association, July 2021.

[160] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: elastic far-memory for stateful serverless analytics," in Proceedings of the Seventeenth European Conference on Computer Systems, (New York, NY, USA), p. 697–713, Association for Computing Machinery, 2022.

[161] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in ACM SoCC '19, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2019.

[162] D. Barcelona-Pons, P. García-López, and B. Metzler, "Glider: Serverless ephemeral stateful near-data computation," in ACM Middleware '23, (New York, NY, USA), p. 247–260, Association for Computing Machinery, 2023.

[163] Y. Kim and J. Lin, " Serverless Data Analytics with Flint ," in IEEE CLOUD '18, (Los Alamitos, CA, USA), pp. 451–455, IEEE Computer Society, July 2018.

[164] A. Jain, A. F. Baarzi, G. Kesidis, B. Urgaonkar, N. Alfares, and M. Kandemir, "Splitserve: Efficiently splitting apache spark jobs across faas and iaas," in ACM Middleware '20, (New York, NY, USA), p. 236–250, Association for Computing Machinery, 2020.

[165] K. Psarakis, G. Christodoulou, G. Siachamis, M. Fragkoulis, and A. Katsifodimos, "Styx: Transactional stateful functions on streaming dataflows," in ACM SIGMOD '25, Association for Computing Machinery, 2025.

[166] G. Liao, A. Deshpande, and D. J. Abadi, "Flock: A low-cost streaming query engine on faas platforms," 2023.

[167] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: semantics for stateful serverless," Proc. ACM Program. Lang., vol. 5, Oct. 2021.

[168] L. Xu, D. Saxena, N. J. Yadwadkar, A. Akella, and I. Gupta, "Dirigo: Self-scaling stateful actors for serverless real-time data processing," 2023.

[169] "Apache flink stateful functions." `https://nightlies.apache.org/flink/flink-statefun-docs-stable/`, 2025.

[170] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive data analytics on cold data using serverless cloud infrastructure," in Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, (New York, NY, USA), p. 115–130, Association for Computing Machinery, 2020.

[171] "Pravega." https://cncf.pravega.io, 2023.

[172] "Confluent." https://www.confluent.io/, 2024.

[173] "Streamnative." https://streamnative.io/, 2024.

[174] "Storage reimagined for a streaming world." https://cncf.pravega.io/blog/2017/04/09/storage-reimagined-for-a-streaming-world/, 2017.

[175] "Infinite storage in confluent platform." https://www.confluent.io/blog/infinite-kafka-storage-in-confluent-platform/, 2020.

[176] "Streaming lakehouse: Introducing pulsar's lakehouse tiered storage," 2023. https://streamnative.io/blog/streaming-lakehouse-introducing-pulsars%-lakehouse-tiered-storage.

[177] A. Povzner, P. Mahajan, J. Gustafson, J. Rao, I. Juma, F. Min, S. Sridharan, N. Bhatia, G. Attaluri, A. Chandra, et al., "Kora: A cloud-native event streaming platform for kafka," Proceedings of the VLDB Endowment, vol. 16, no. 12, pp. 3822–3834, 2023.

[178] "Apache pulsar - overview of tiered storage." https://pulsar.apache.org/docs/3.3.x/tiered-storage-overview/, 2024.

[179] "Redpanda - tiered storage." https://docs.redpanda.com/current/manage/tiered-storage/, 2024.

[180] "Openmessaging benchmark." https://github.com/openmessaging/benchmark, 2025.

[181] "Remotestoragemanager for apache kafka tiered storage." https://github.com/Aiven-Open/tiered-storage-for-apache-kafka, 2025.

[182] Y. Mansouri, A. N. Toosi, and R. Buyya, "Data storage management in cloud environments: Taxonomy, survey, and future directions," ACM Computing Surveys (CSUR), vol. 50, no. 6, pp. 1–51, 2017.

[183] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," ACM SIGOPS operating systems review, vol. 47, no. 1, pp. 9–15, 2013.

[184] "Kip-405: Kafka tiered storage." https://cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage, 2025.

[185] "Tiered storage in confluent platform." https://docs.confluent.io/platform/current/clusters/tiered-storage.html, 2024.

[186] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom, "Crystal: Software-defined storage for multi-tenant object stores," in USENIX FAST'17, pp. 243–256, 2017.

[187] J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," in ACM/IFIP/USENIX Middleware'17, pp. 121–133, 2017.

[188] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful serverless computing with crucial," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 31, no. 3, pp. 1–38, 2022.

[189] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in USENIX NSDI'19, pp. 193–206, 2019.

[190] "Unlocking the edge: Data streaming goes where you go with confluent." https://www.confluent.io/blog/data-streaming-at-the-edge/, 2024.

[191] A. da Silva Veith, M. D. de Assuncao, and L. Lefevre, "Latency-aware strategies for deploying data stream processing applications on large cloud-edge infrastructure," IEEE Transactions on Cloud Computing, 2021.

[192] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang, et al., "Sonic: Application-aware data passing for chained serverless applications," in USENIX ATC'21, pp. 285–301, 2021.

[193] J. Sampé, M. Sánchez-Artigas, G. Vernik, I. Yehekzel, and P. García-López, "Outsourcing data processing jobs with lithops," IEEE Transactions on Cloud Computing, vol. 11, no. 1, pp. 1026–1037, 2021.

[194] M. Yu, T. Cao, W. Wang, and R. Chen, "Following the data, not the function: Rethinking function orchestration in serverless computing," in USENIX NSDI'23, pp. 1489–1504, 2023.

[195] R. Halalai, P. Sutra, É. Rivière, and P. Felber, "Zoofence: Principled service partitioning and application to the zookeeper coordination service," in IEEE SRDS'14, pp. 67–78, 2014.

[196] "S3proxy." https://github.com/gaul/s3proxy, 2025.

[197] "Redis." https://redis.io, 2025.

[198] "Nexus repository." https://github.com/neardata-eu/nexus, 2025.

[199] "Apache bookkeeper." https://bookkeeper.apache.org, 2023.

[200] "fio - flexible i/o tester." https://fio.readthedocs.io, 2025.

[201] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," in IEEE ISSRE'23, pp. 355–366, 2023.

[202] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in ACM SOSP'09, p. 117–132, 2009.

[203] D. M. J. M. M. D. M. N. P. Andru Twinanda, Sherif Shehata, "Endonet: A deep architecture for recognition tasks on laparoscopic videos," IEEE Transactions on Medical Imaging, vol. 36, 02 2016.

[204] "Sra toolkit." https://hpc.nih.gov/apps/sratoolkit.html, 2025.

[205] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, Y. Kwon, K. Michael, J. Fang, Z. Yifu, C. Wong, D. Montes, et al., "ultralytics/yolov5: v7.0-yolov5 sota realtime instance segmentation," Zenodo, 2022.

[206] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: A software-defined storage architecture," in ACM SOSP'13, pp. 182–196, 2013.

[207] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in USENIX NSDI'15, pp. 589–603, 2015.

[208] J. Zhang, A. Wang, X. Ma, B. Carver, N. Newman, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, et al., "Sion: Elastic serverless cloud storage," in VLDB'23, 2023.

[209] "Amazon s3 object lambda." https://aws.amazon.com/s3/features/object-lambda, 2025.

[210] "Confluent blog - an introduction to data mesh." https://www.confluent.io/blog/benefits-of-data-mesh-and-how-to-get-started, 2025.

[211] "Istio.io." https://istio.io/, 2025.

[212] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl, "The nebulastream platform for data and application management in the internet of things," in CIDR'20, 2020.

[213] V. Amaral, B. Norberto, M. Goulão, M. Aldinucci, S. Benkner, A. Bracciali, P. Carreira, E. Celms, L. Correia, C. Grelck, H. Karatza, C. Kessler, P. Kilpatrick, H. Martiniano, I. Mavridis, S. Pllana, A. Respício, J. Simão, L. Veiga, and A. Visa, "Programming languages for data-intensive hpc applications: A systematic mapping study," Parallel Computing, vol. 91, p. 102584, 2020.

[214] O. OnDemand, "Open ondemand," 2025.

[215] E. Comission, "Ai factories," 2025.

[216] N. S. Foundation, "National artificial intelligence research resource," 2025.

[217] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," Future Generation Computer Systems, vol. 46, pp. 17–35, 2015.

[218] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," The International Journal of High Performance Computing Applications, vol. 31, no. 1, pp. 66–82, 2017.

[219] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in python," in Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '19, (New York, NY, USA), p. 25–36, Association for Computing Machinery, 2019.

[220] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: distributed computing for the 99%," in Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, (New York, NY, USA), pp. 445–451, Association for Computing Machinery, Sept. 2017.

[221] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in USENIX NSDI'17, pp. 363–376, 2017.

[222] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs - a temporary distributed file system for hpc applications," in 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 319–324, 2018.

[223] M. Lăcătușu, A. D. Ionita, F. D. Anton, and F. Lăcătușu, "Analysis of complexity and performance for automated deployment of a software environment into the cloud," Applied Sciences, vol. 12, no. 9, p. 4183, 2022.

[224] M. Shepperd, "A critique of cyclomatic complexity as a software metric," Software Engineering Journal, vol. 3, no. 2, pp. 30–36, 1988.

[225] M. A. Yaqin, R. Sarno, and S. Rochimah, "Measuring scalable business process model complexity based on basic control structure," International Journal of Intelligent Engineering & System, vol. 13, no. 6, pp. 52–65, 2020.

[226] L. Team, "Lithops: Compute and storage benchmarks," 2020.

[227] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: a scalable and locality-enhanced framework for serverless parallel computing," in Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, (New York, NY, USA), p. 1–15, Association for Computing Machinery, 2020.

[228] A. Arjona, P. G. López, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," Future Generation Computer Systems, vol. 124, pp. 215–229, 2021.

[229] A. W. Services, "Step functions," 2016.

[230] Azure, "Durable functions," 2016.

[231] 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 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), (Carlsbad, CA), pp. 303–320, USENIX Association, July 2022.

[232] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), (Boston, MA), pp. 653–669, USENIX Association, Apr. 2021.

[233] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating Function-as-a-Service workflows," in 2021 USENIX Annual Technical Conference (USENIX ATC 21), (Boston, MA), pp. 805–820, USENIX Association, July 2021.

[234] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Mashup: making serverless computing useful for hpc workflows via hybrid execution," in Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '22, (New York, NY, USA), p. 46–60, Association for Computing Machinery, 2022.

[235] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in 2018 USENIX Annual Technical Conference (USENIX ATC 18), (Boston, MA), pp. 923–935, USENIX Association, July 2018.

[236] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in 2020 USENIX Annual Technical Conference (USENIX ATC 20), (Boston, MA), pp. 419–433, USENIX Association, July 2020.

[237] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: stateful functions-as-a-service," Proceedings of the VLDB Endowment, vol. 13, pp. 2438–2452, July 2020.

[238] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the gap between serverless and its state with storage functions," in Proceedings of the ACM Symposium on Cloud Computing, SoCC '19, (New York, NY, USA), pp. 1–12, Association for Computing Machinery, 2019.

[239] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful Serverless Computing with Crucial," ACM Transactions on Software Engineering and Methodology, vol. 31, pp. 39:1–39:38, Mar. 2022.

[240] A. Khandelwal, Y. Tang, R. Agarwal, A. Akella, and I. Stoica, "Jiffy: Elastic far-memory for stateful serverless analytics," in Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, (New York, NY, USA), pp. 697–713, Association for Computing Machinery, 2022.

[241] D. Barcelona-Pons, P. García-López, and B. Metzler, "Glider: Serverless ephemeral stateful near-data computation," in Proceedings of the 24th International Middleware Conference, Middleware '23, (New York, NY, USA), p. 247–260, Association for Computing Machinery, 2023.

[242] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.

[243] G. T. Eizaguirre and M. Sánchez-Artigas, "A seer knows best: Auto-tuned object storage shuffling for serverless analytics," Journal of Parallel and Distributed Computing, vol. 183, p. 104763, 2024.

[244] B. Przybylski, M. Pawlik, P. Zuk, B. Lagosz, M. Malawski, and K. Rzadca, " Using Unused: Non-Invasive Dynamic FaaS Infrastructure with HPC-Whisk ," in SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, (Los Alamitos, CA, USA), pp. 1–15, IEEE Computer Society, Nov. 2022.

[245] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rFaaS: Enabling high performance serverless with rdma and leases," in 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), (Los Alamitos, CA, USA), pp. 897–907, IEEE Computer Society, may 2023.

[246] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler, "Software resource disaggregation for hpc with serverless computing," 2024.

[247] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "funcx: A federated function serving fabric for science," in Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '20, (New York, NY, USA), p. 65–76, Association for Computing Machinery, 2020.

[248] A. A. Benavides, T. D. Coll, A. Call, P. García López, and R. Nou Castell, "Enhancing hpc with serverless computing: Lithops on marenostrum5," in 2024 IEEE 32nd International Conference on Network Protocols (ICNP), (Los Alamitos, CA, USA), pp. 1–6, IEEE Computer Society, 2024.