NEARDATA

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

NEARDATA

(grant agreement No 101092644)

Extreme Near-Data Processing Platform

D3.1 XtremeHub first release and documentation

Due date of deliverable: 30-04-2024
Actual submission date: 30-04-2024

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	46
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	Xavier Roca (URV)
Quality Manager	Aaron Call (BSC)
Author(s)	Raúl Gracia (DELL), Sean Ahearne (DELL), Ger Hallissey (DELL), Xavier Roca (URV), André Miguel (SCO), Aaron Call (BSC)
Partner(s) Contributing	DELL, URV, SCO, BSC
Document ID	NEARDATA_D3.1_Public.pdf
Abstract	The deliverable describes the design of XtremeHub: the data plane component of the NEARDATA platform.
Keywords	Serverless analytics, data partitioning, data streams, data connectors, confidential computing, TEEs.

History of changes

Version	Date	Author	Summary of changes
0.1	30-03-2024	Raúl Gracia (DELL), Sean Ahearne (DELL), Ger Hallissey (DELL), Xavier Roca (URV), André Miguel (SCO), Aaron Call (BSC)	Internal version to review.
1.0	30-04-2024	Raúl Gracia (DELL), Sean Ahearne (DELL), Ger Hallissey (DELL), Xavier Roca (URV), André Miguel (SCO), Aaron Call (BSC)	Final version.

Table of Contents

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
S3	Simple Storage Service
SDP	Streaming Data Platform
TEE	Trusted Execution Environment
VAF	Variant Call Format
VM	Virtual Machine
WAL	Write-Ahead Log

1 Executive summary

This deliverable presents the initial design and implementation of XtremeHub: the data plane component of NEARDATA. First, we provide a general overview of XtremeHub as well as the interactions of its main sub-components: Lithops, Pravega, and Scone. Second, we provide an in-depth overview of each of the XtremeHub sub-components and Data Connectors, with special emphasis on performance results and KPIs related to the research work carried out through the project so far. Finally, we provide some guidance on next steps in the development of XtremeHub for the second half of the project.

2 Introduction

The NEARDATA project is rooted in the near-data processing paradigm, which can be defined as the strategy of moving computational resources close to the data, instead of transferring the data to the processor. NEARDATA aims to establish a robust near-data infrastructure that acts as a mediator for data flows between object storage and data analytics platforms throughout the compute continuum.

A key element for realizing this vision is the innovative data plane component of NEARDATA, namely XtremeHub (a complete architecture overview of NEARDATA is available in *D2.2 NEARDATA Architecture Specs and Early Prototypes*). XtremeHub serves as an intermediate data service that captures and enhances data flows (S3 API, stream APIs) using high-performance near-data connectors (Cloud/Edge). Moreover, it allows to perform secure and private computations close to the data via containerized technologies exploiting specialized hardware (*e.g.*, Trusted Execution Environments).

In this deliverable, we provide an in-depth overview of XtremeHub as well as the main subcomponents that implement the system functionality: Lithops (multi-cloud serverless function execution engine), Pravega (tiered streaming storage engine), and Scone (confidential computing layer). We also overview key Data Connectors developed in XtremeHub for the project use cases. The combination of all these sub-components allows XtremeHub to offer efficient access, partitioning, and discovery of data objects in an object storage service, as well as streaming analytics with seamless tiering and management of data streams to object stores. Moreover, computations can be secured and isolated from external adversaries with advanced attestation capabilities. We will not only describe these sub-components (further details can be also found in deliverables *D2.2 NEARDATA Architecture Specs and Early Prototypes*, *D4.1 Data Broker release and documentation*, and *D5.1 First release of KPI benchmarks in all use cases and data connector libraries*), but also provide a detailed analysis on key aspects of their performance based on the research work of the consortium. We conclude this deliverable by outlining potential research avenues to be addressed in the second half of the project, specially related to work-package tasks to be completed in the next months of project execution.

3 XtremeHub Overview

XtremeHub is the vision for the integrated data plane components in NEARDATA. XtremeHub stands at the forefront of near-data analytics, offering a cutting-edge toolkit that seamlessly integrates powerful processing and storage components to build advanced data analytics capabilities. At its core, XtremeHub is a fusion of advanced technologies, namely Lithops for serverless compute, Pravega for streaming storage, and Scone for containerized and trusted execution environment (TEE) capabilities. Let's delve into each component and explore how they interact to deliver efficiency and security in near-data analytics. An integrated view of XtremeHub components is depicted in Fig. 1.

Lithops represents the computational core of XtremeHub, providing serverless computing capabilities that scale seamlessly to handle massive workloads across the cloud continuum (see Section 4). By leveraging Lithops, users can execute complex analytics tasks without the burden of managing infrastructure, as it dynamically provisions resources based on demand. One of Lithops' key interactions lies in its integration with object storage, where it efficiently retrieves and stores data from and to various storage back-ends, ensuring smooth data processing workflows.

Complementing Lithops' compute capabilities, Pravega serves as the backbone for storing and managing streaming data in XtremeHub (see Section 5). Its unique design enables seamless ingestion, storage, and retrieval of high-volume streaming data with strong consistency guarantees. Pravega seamlessly interfaces with object storage, efficiently storing and retrieving data streams while ensuring fault tolerance and scalability. This integration ensures that data is readily available for analysis, facilitating real-time insights generation and decision-making.

Security and privacy are paramount in near-data analytics, and Scone addresses these concerns by providing a trusted execution environment (TEE) within XtremeHub (see Section 6). With Scone, users can execute sensitive computations in a secure enclave, safeguarding data confidentiality and integrity. Leveraging Scone's private computation capabilities, users can perform critical analytics tasks on sensitive data without compromising privacy, ensuring compliance with stringent regulatory requirements.

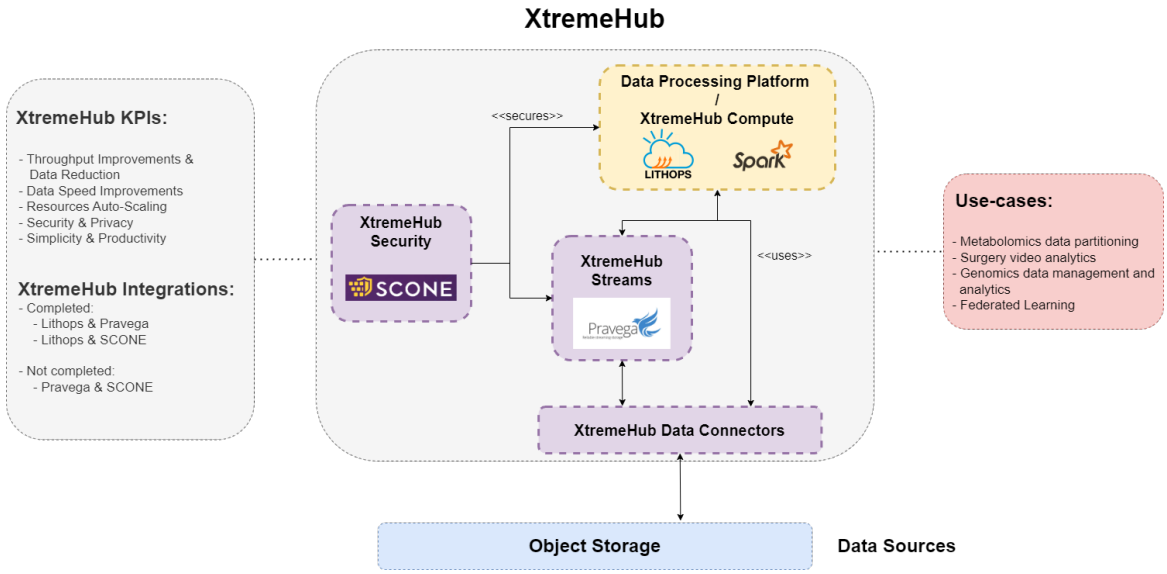

Figure 1: High-level overview of XtremeHub.

Finally, the NEARDATA project coins its own novel notion of Data Connector (see *D2.2 NEARDATA Architecture Specs and Early Prototypes*): it establishes communication between the Data Provider and the Data Consumer to deal with extreme data efficiently using new ETL techniques (see Section 7). In this deliverable, we overview some of these Data Connectors and their potential for optimizing workloads related to the project use cases.

In the following, we will describe and evaluate the components that build XtremeHub, as well

as the new progress related to them based on our research work so far. While in some cases we also align the evaluation with the project's use cases, for full details on how XtremeHub addresses the challenges of our use cases we refer to *D5.1 First release of KPI benchmarks in all use cases and data connector libraries*.

4 XtremeHub Compute: Lithops

In this section, we will show a brief overview of the Lithops serverless computing framework and evaluate its performance according to the different cloud services. Note that in deliverable *D2.1 Initial Architecture Specifications* Lithops is described more extensively specifying in detail its components.

4.1 Lithops Summary

Lithops is a Python multi-cloud serverless computing framework that allows unmodified local Python code to run at massive scale on all major serverless computing platforms. Moreover, its multicloud-agnostic architecture ensures portability across cloud providers.

Lithops brings great value to data-intensive applications such as Big Data analytics and embarrassingly parallel jobs. It is especially suited for highly parallel programs with little or no need for inter-process communication.

One of the main features of Lithops is programming simplicity. This way, programmers who are experts or not in the cloud can benefit from the use of this tool. Lithops is shipped with two types of APIs in its architecture. The Compute API, which allows the user to make calls to execute parallel tasks in the selected compute backend and the Storage API, to operate with the storage backend.

Thanks to a built-in on-the-fly data partitioning, data consumption from object storage is facilitated. The user would only have to provide the list of object keys comprising the dataset to be processed. In this way, lithops ensures automatic data partitioning and data discovery for common data formats such as CSV, thus facilitating parallel processing of this data.

Finally, three types of execution mode for Lithops computing backends are presented. Localhost, allows users to execute functions on their local machine. Serverless Mode, allows users to run functions on serverless computing services (Amazon Lambda, IBM Cloud functions). Standalone Mode, allows users to run functions on one or multiple Virtual Machines locally or in the cloud.

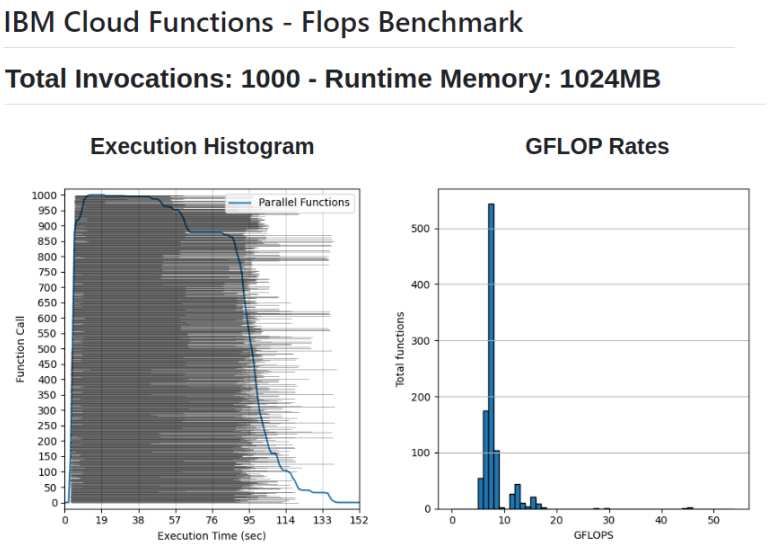

Figure 2: IBM Cloud Functions - Flops Benchmark.

4.2 Experimental evaluation

Whenever considering the use of cloud computing and/or storage, users should explore which cloud provider provides the best services and which ones best suit their needs. Because Lithops offers support for multiple clouds, it allows users to perform performance tests on different backends to observe their features and performance in detail.

In the following, we will describe two benchmarks performed on the different APIs presented in Lithops. These benchmarks were performed on two of the most used cloud service providers, such as Amazon Web Services and IBM Cloud.

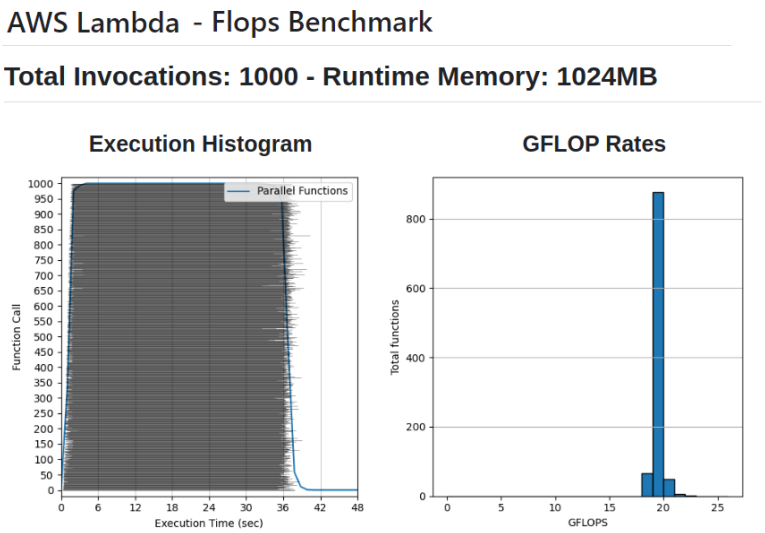

Figure 3: AWS Lambda - Flops Benchmark.

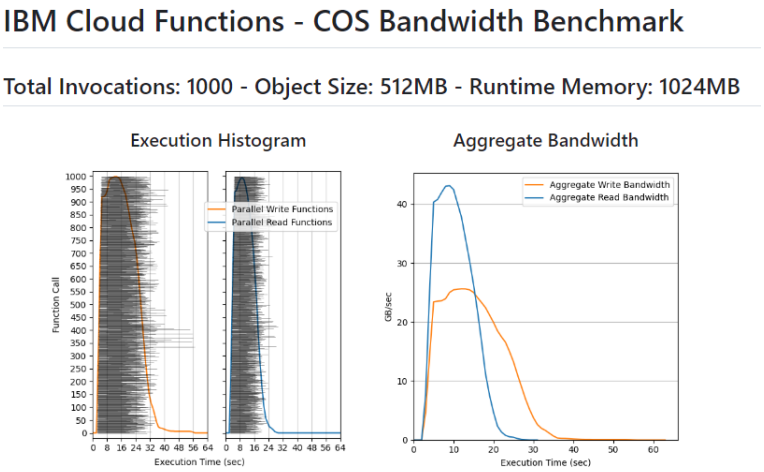

Figure 4: IBM Cloud Functions & COS - Bandwidth Benchmark.

Serverless Functions Performance The first benchmark is based on evaluating the performance of the serverless compute backends, AWS Lambda and IBM Cloud Functions, offered by the aforementioned cloud providers from the use of the Compute API offered by Lithops.

To measure the computational performance of these backends, a flop benchmark was performed. A total of 1000 functions were invoked with a memory size of 1024MB where each function multiplies two matrices a given number of times. The parameters set for this benchmark were a matrix size of $4096 * 4096$ with a total of 5 repetitions for matrix multiplications.

Figures 2 and 3 show the results of the serverless functions benchmark. First of all, the execution histogram is shown. This is key to understand the parallelism and performance obtained in each of the different cloud providers. It should be noted that the closer the executions are together, the higher the parallelism, and the shorter the execution time, the higher the performance. The second plot shows the GFLOPS rates achieved distributed by the number of functions. It allows us to identify the stable performance of the functions and their variability.

If we focus on the results obtained, figure 2 shows how IBM Cloud offers an irregular parallelism and performance since a variability of execution time in the different functions can be observed. This

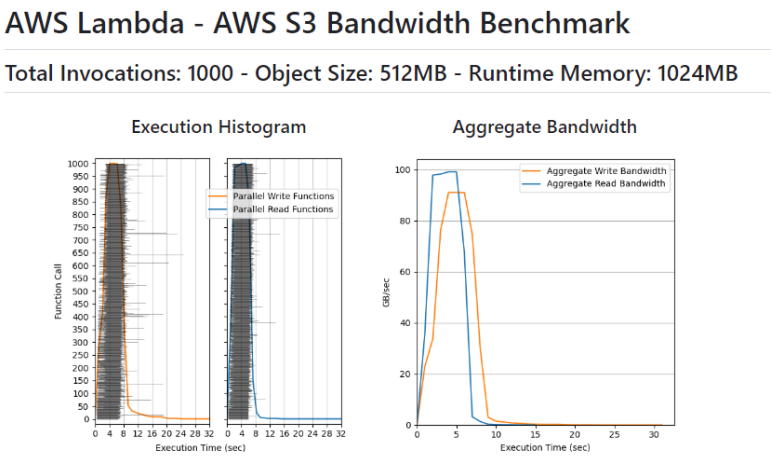

Figure 5: AWS Lambda & S3 - Bandwidth Benchmark.

case can also be observed in the GFLOP rates plot, where we find values between 5 and 45 GFLOPS. In the case of AWS, figure 3, high parallelism is observed and more stable with low execution times. It can also be visualized in the GFLOP Rates plot where we find all the results close to 20 GFLOPS.

Cloud Object Storage Performance This second benchmark is based on evaluating the bandwidth provided by the object storage backends, Amazon S3 and IBM Cloud Object Storage, of the two different cloud providers from the use of the Storage API offered by Lithops.

To perform this benchmark, I/O bound operations were evaluated in two different phases. Both phases used the same number of functions to perform their tasks. In the first phase, each function was in charge of generating a total of *N*MB that were uploaded to the cloud object storage backend. In the second phase, each function received by parameter a key referring to the object to be downloaded.

Figures 4 and 5 show the results of the cloud object storage benchmark. In the same way as in the previous test, the histogram of the execution is shown in a first instance to observe the parallelism of the cloud backend. Next, a plot is shown in relation to the bandwidth obtained in the write and read operations performed.

Looking deeper at the results, the Figure 4 shows the irregularity in the parallelization and performance of the IBM Cloud Functions. In relation to bandwidth, read operations show a bandwidth 50% higher than write operations. Ensuring a maximum of 40GB/s for reading and 25GB/s for writing. For the AWS case, shown in the Figure 5, the high parallelization is maintained with a very similar bandwidth for both read and write operations, reaching a maximum of $90 - 100$GB/s.

KPI-3 (Resource Auto-scaling). The serverless data processing platform offers the ability to develop cloud architecture solutions that are fully scalable and flexible to the resources needed to run workloads. The parallelization of serverless functions from two different cloud providers is analyzed to provide the user with a transparent evaluation of resource deployment.

5 XtremeHub Streams: Pravega

This section describes the progress done in Pravega as the streaming layer for XtremeHub. In deliverable *D2.1 Initial Architecture Specifications* we provided an extensive description of Pravega and its internal components, as well as the client interactions with the system. For this reason, in the beginning of this section we provide a brief overview of Pravega to help the reader understanding the evaluation of the system, but we defer to *D2.1 Initial Architecture Specifications* for full details. Moreover, in addition to the evaluation of Pravega against other event streaming systems, we overview advanced features related to the project use-cases: i) we explore the coordinated auto-scaling of Pravega streams with Flink task managers, ii) we provide an in-depth evaluation of the Byte API in Pravega and its advantages in some scenarios compared to event interfaces, and iii) we evaluate Pravega as a storage substrate for serverless analytics pipelines.

5.1 Pravega Design Recap

Pravega is a distributed, tiered storage system for data streams that provides the streaming layer in NEARDATA. Pravega stores *data events* in *streams* (like a "topic" in messaging systems). A stream is a durable, elastic, append-only, unbounded sequence of bytes achieving good performance and consistency. Internally, streams are divided into *segments*. A stream segment is a partition of the data within a stream. The allowed operations on segments include append, truncate, seal, merge, and delete (but not update). It is worth mentioning that a stream may have multiple segments open for appending events at a given time, which enables higher throughput. When working with streams having parallel segments, users requiring event order guarantees are expected to use *routing keys* for writing data. To wit, parallel segments in a stream are assigned to partitions of the key space as a result of a hash function (e.g., $h(k) \in [0,1)$). The writer API accepts as input a user-provided routing key to consistently select the segment for appending events to and order writes with the same key.

The architecture of Pravega is shown in Fig. 6. First, Pravega offers client libraries implementing the server APIs, such as *writers* and *readers*, which interact with Pravega server instances either within the same cluster or externally. These client libraries allow stream processing engines to manage *data events* from Pravega.

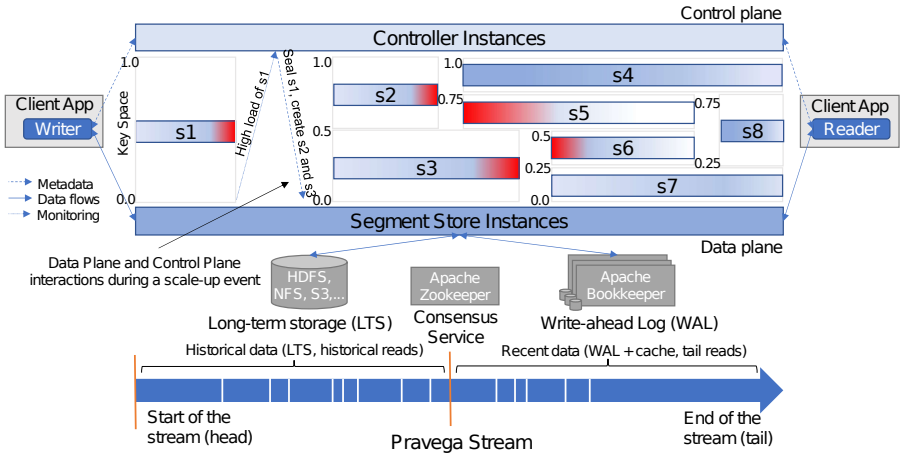

Figure 6: Architecture of Pravega consisting of clients, controllers, and segment store instances. Segment stores temporarily write data to WAL and then move the data to LTS. The figure also shows an example of stream auto-scaling.

On the server side, we find the Pravega *control plane* formed by *controller instances*. The control plane is primarily responsible for orchestrating all stream lifecycle operations, like creating, updating, scaling, and deleting streams. Pravega streams are policy-driven. Currently, the system offers two types of *stream policies*: *retention policies*, which automatically truncate a stream based on size or time bounds; and *auto-scaling policies*, which allow the system to automatically change the segment

parallelism of a stream based on the ingestion workload (events/bytes per second). The control plane takes care of enforcing such stream policies. As we describe later on (see Fig. 6), for stream auto-scaling policies, Pravega builds a feedback loop between the control and data planes, so the control plane can react to the load monitored by the data plane.

The *data plane* in Pravega handles data requests from clients and is formed by *segment store instances*. Segment stores play a critical role in making segment data durable and serving it efficiently. Note that segment stores only work with segments and are agnostic to the concept of stream, which is an abstraction of the control plane. The data plane distributes the segment-related load based on *segment containers*. Segment containers perofrm the heavy lifting on segments and the main role of segment store instances is to host segment containers. A segment is mapped during its entire life to a segment container using a stateless, uniform hash function that is known by the control plane. Thus, segment ids belong to a *key-space* that is partitioned across the available segment containers.

The segment store has two storage tiers: *Write-Ahead Log (WAL)* and *Long-Term Storage (LTS)*. The main goal of WAL (implemented via Apache Bookkeeper [1, 2]) is to guarantee durability and low latency of incoming writes and keep that data temporarily for recovery purposes. Segment stores asynchronously move data to LTS. Once some data is stored in LTS, the corresponding log file from WAL is truncated. Pravega has an LTS tier for a couple of key assumptions that determined its design: data streams are potentially *unbounded* and the system should be able to store a large number of segments in a *cost-effective manner*. Pravega achieves both goals by storing historical stream data in a scalable storage service.

Finally, Pravega uses a consensus service (Apache Zookeeper [3, 4]) for leader election and general cluster management purposes.

5.1.1 Experimental Evaluation

After having described the design and architecture of Pravega, we present an extensive evacuation of its event API. We summarize here the configuration used in our experiments on AWS (see Table 2). We compare the performance of Pravega against the most popular counterpart systems: Apache Kafka [5] and Apache Pulsar [6]. The workloads in all cases are executed with OpenMessaging Benchmark [7].

Implementation. Pravega is an open-source CNCF project in sandbox state [8]. The project was open-sourced in 2016 and it provides to the public not only the core storage engine but also deployment tools (*e.g.*, Kubernetes operators) and a connector ecosystem to integrate Pravega with a variety of analytics engines.

Deployment. We employ the same type of EC2 instances for deploying the analogous components of each system. It is especially relevant to mention that for the components in charge of writing data (*i.e.*, brokers, Apache Bookkeeper), we use an instance type with access to local NVMe drives. To evaluate the efficiency of the write path across these systems, we use one drive for the Bookkeeper journal (Pulsar/Pravega) and the Kafka broker log.

Replication. The data replication schemes differ between Kafka (leader-follower model) and Pravega/Pulsar (Bookkeeper replicates across bookies [1]). Still, we can achieve a similar data redundancy layout across them. We configured all the systems to create 3 replicas of every message, requiring at least 2 of such replicas to be confirmed by the servers before considering a write as successful.

Durability. There is an important difference between the default behavior of Kafka versus Pravega/Pulsar on data durability. Kafka by default does not flush data (*i.e.*, `fsync` system call), thus trading-off durability in favor of performance. We also evaluate Kafka with similar durability guarantees that both Pravega and Pulsar satisfy by default (*i.e.*, by setting `flush.messages=1`, `flush.ms=0`).

Storage tiering. Pravega inherently moves ingested data to LTS. In our experiments, we used an NFS volume backed up by an AWS Elastic File Service (EFS) instance. For fairness, we also have enabled the storage tiering plug-in in Pulsar to compare against Pravega unless otherwise stated. We have configured for Pulsar an AWS S3 bucket and defined the ledger rollover to happen within 1 and 5 minutes, so tiering activity occurs while the benchmark takes place. We also set Pulsar topics to start the offloading process immediately (`setOffloadThreshold=0`), as well as to remove the data

	Pravega	**Kafka**	**Pulsar**
Version	Pravega 0.9.0, Bookkeeper 4.11.1	Kafka 2.6.0	Pulsar 2.6.0, Bookkeeper 4.11.1
Replication	ensemble=3, writeQuorum=3, ackQuorum=2	replication=3, acks=all, min.insync.replicas=2	ensemble=3, writeQuorum=3, ackQuorum=2
Durability	Yes (default)	No (default)	Yes (default)
Tiering	Yes (AWS EFS)	No	Yes (AWS S3)
Instances	Controller (m5.large)=1, Segment Store + Bookie (i3.4xlarge)=3, Zookeeper (t3.small)=3, Benchmark (c5.4xlarge)=2	Broker (`i3.4xlarge`)=3, Zookeeper (`t3.small`)=3, Benchmark (`c5.4xlarge`)=2	Broker + Bookie (`i3.4xlarge`)=3, Zookeeper (`t3.small`)=3, Benchmark (`c5.4xlarge`)=2
Journal Drives	1 NVMe	1 NVMe	1 NVMe
Client Batching	Yes (dynamic)	Yes (time/size based)	Yes (time/size based)

Table 1: Experiments configuration unless otherwise stated.

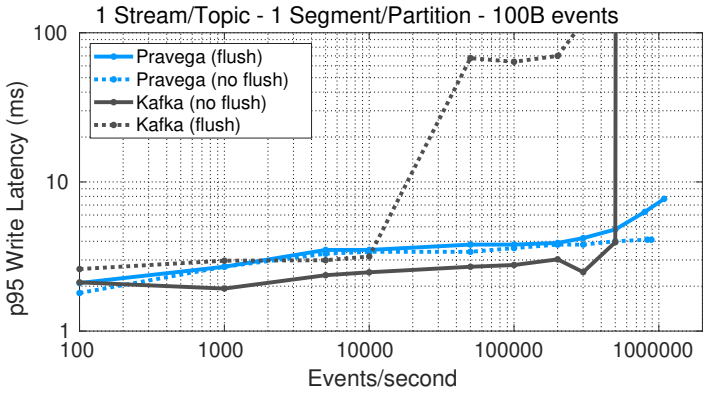

Figure 7: Impact of data durability on write performance (1 segment/partition, 1 writer/producer).

from Bookkeeper as soon as it is migrated to long-term storage (`setOffloadDeleteLag=0`). At the time of this writing, Kafka did not provide storage tiering in its open-source edition.

Routing keys. By default, our workloads use (random) routing keys on writes. We use routing keys in our workloads to ensure per-key event order, frequently a requirement of streaming applications for correctness. We also execute workloads without routing keys to understand the potential impact of routing keys on performance.

Client configuration. Pulsar and Kafka clients implement a batching mechanism that can be parameterized via "knobs" that enable the producer to buffer a certain number of messages or wait until some timeout before performing the actual write against the broker. The goal of this feature is to improve a producer's throughput for small messages, despite inducing extra latency in scenarios where the workload is not throughput-oriented. By default, we use in both systems a similar configuration: 128KB as batch size and 1ms as batch time. We also compare the behavior of the Pulsar producer with and without this feature, as well as the impact of larger batches in Kafka. We did not enable compression in the Pulsar/Kafka clients as such techniques may have an important role in performance depending on the data at hand [9].

Workloads. We are interested in understanding the behavior of these systems based on two parameters: *event size* and *number of partitions/segments*. We use event sizes from 100B to 10KB, as they can be considered typical in many streaming applications (*e.g.*, IoT, logs, social media posts). We configure our experiments to run OpenMessaging Benchmark [7] producer and consumer threads distributed across the benchmark VMs. Each of the producer and consumer threads uses a dedicated Kafka, Pulsar, or Pravega client instance. Benchmark producer threads use producers (Kafka and Pulsar) or writers (Pravega), while benchmark consumer threads use consumers (Kafka and Pulsar) or readers (Pravega).

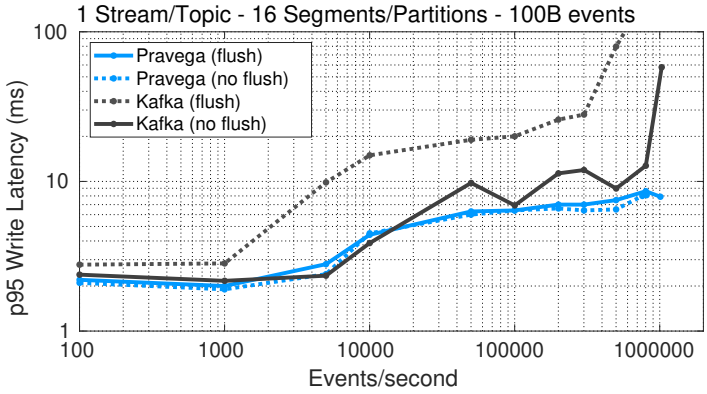

Figure 8: Impact of data durability on write performance (16 segments/partitions, 1 writer/producer).

Writer Performance and Data Durability. It is natural for an application to expect that data is available for reading once writing the data is acknowledged, despite failures. Durability is critical for applications to reason about correctness. While Pravega provides durability by default (*i.e.*, flushing data upon acknowledgement), this is not the case for Kafka. Not enabling data durability may be undesirable for enterprise applications as correlated failures do happen and servers do not always stop gracefully [10].

In Fig. 7 and Fig. 8, we show latency and throughput to compare the performance implications of non-default durability options. Specifically, we run a set of experiments comparing enabling and disabling durability in Pravega (disabling journal flushes in Bookkeeper, namely "`no flush`") and Kafka (by setting `flush.messages=1`, `flush.ms=0` to enable durability, namely "`flush`"). For simplicity, these experiments are executed with a single producer/writer.

Visibly, Fig. 7 shows that for a stream/topic with one segment/partition, the Pravega writer (`flush`) reaches a maximum throughput 73% higher than Kafka (`no flush`). Note that this performance improvement in Pravega is achieved while guaranteeing data durability. Following this comparison, for 16 segments/partitions (see Fig. 8) the maximum throughput of both Pravega and Kafka is similar (over 1 million events/second for a single writer/producer).

Note that enforcing data durability for Kafka (`flush`) has a major performance impact on write latency. This is especially visible for moderate/high throughput rates. Kafka flushes messages according to a time (`log.flush.interval.ms`) or a message (`log.flush.interval.messages`) interval. When the message interval is set to 1, all messages are flushed individually before being acknowledged, inducing a significant performance penalty. With Bookkeeper, data is persisted before being acknowledged, but they are opportunistically grouped upon flushes [1].

Regarding write latency, Kafka (`no flush`) only gets consistently lower ($\approx$ 1ms at p95) values than Pravega (`flush`) for 1 segment/partition up to 500k e/s. In the rest of the cases, the Pravega writer shows lower latency than Kafka. The performance gain for Pravega of not flushing data to the drive in Bookkeeper writes is modest, which justifies providing durability by default.

KPI-2 (Data Speed Improvements): The Pravega writer achieves good write performance compared to the Kafka producer while guaranteeing data durability.

Evaluating Batching Strategies. Batching enables a trade-off between throughput and latency. Ideally, applications should not need to reason about convoluted parameters to benefit from batching. Instead, it should be the work of the client along with the server to perform this task. The dynamic batching heuristic the Pravega writer implements has the goal of calibrating the batch sizes over time.

In Fig. 9 and Fig. 10, we plot latency and throughput to understand the impact of Pravega writer batching on performance. We compare Pravega against Pulsar and Kafka (`no flush` by default), which are systems that require an application to choose whether to use batching or not, and configure it accordingly.

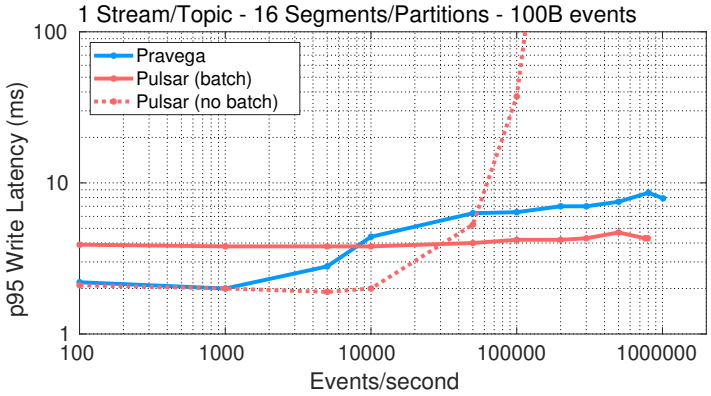

Figure 9: Evaluation of client batching strategies (1 segment/partition, 1 writer/producer).

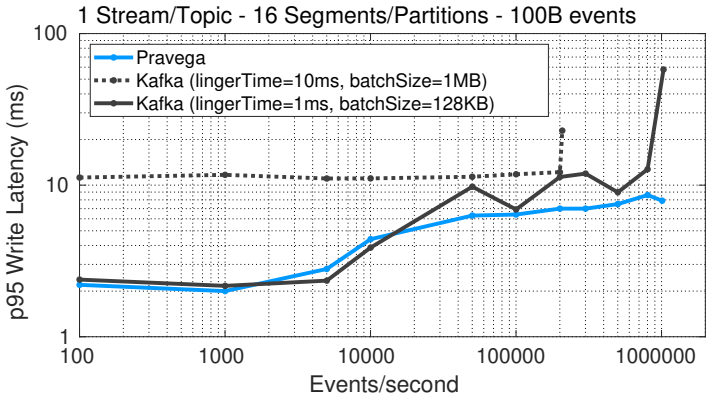

Figure 10: Evaluation of client batching strategies (16 segments/partitions, 1 writer/producer).

Fig 9 shows that the Pulsar producer is able to target either low latency or high throughput, but not both. This forces the user to choose between a latency-oriented (namely `no batch`) or a throughput-oriented configuration (namely `batch`). In contrast, the Pravega writer achieves both lower write latency than Pulsar (`batch`) for the lower-end throughput rates (*e.g.*, $<$ 10k e/s) and higher maximum throughput than Pulsar (`no batch`).

Interestingly, increasing the batch size and the wait time for Kafka (10ms linger time, 1MB batch size) to enable more batching has the opposite expected effect (see Fig. 10). The throughput drops compared to the default configuration with 1ms linger time and 128KB batch size. To understand this result, we inspected the maximum throughput of a Kafka producer writing to a 16-partition topic and using the same batch configuration, but not using routing keys when writing data. In this case, the client achieved a throughput $6x$ higher (120MBps). We consequently attribute the lower batching performance observed to the use of (random) routing keys. We focus on this aspect specifically in subsequent sections.

KPI-2 (Data Speed Improvements), KPI-3 (Simplicity and Productivity): Dynamic batching in Pravega allows writers to achieve an excellent balance between latency and throughput. Furthermore, Pravega does not require the user to decide the performance configuration of the writer ahead of time.

Writer Performance for Large Events. Events are often small in real applications (*e.g.*, $<$ 1KB), but there are also use cases using larger events. For such scenarios, batching is less effective, and the key metric is the byte throughput. In Fig. 11 and Fig. 12, we use 10KB events and we compare latency and byte throughout of the Pravega writer against both Pulsar and Kafka producers.

For a single-segment/partition stream/topic (see Fig. 11), Pravega (160MBps) and Pulsar (300MBps) writers achieve a much higher write throughput compared to Kafka (70MBps). In the case of 16 segments/partitions (see Fig. 12), Pravega shows the highest throughput (350MBps) compared to Kafka

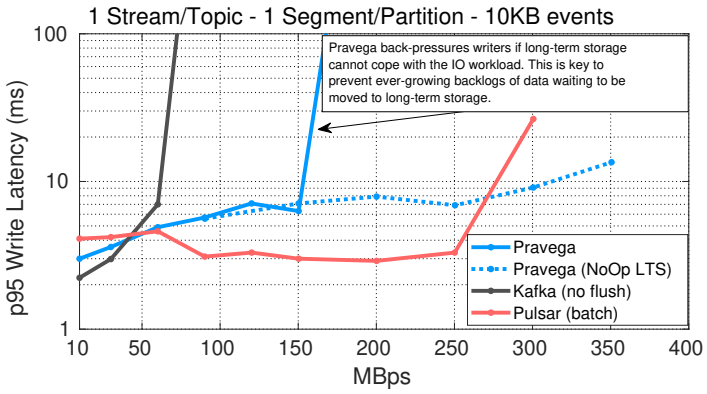

Figure 11: Write performance for larger events (1 segment/partition, 1 writer/producer).

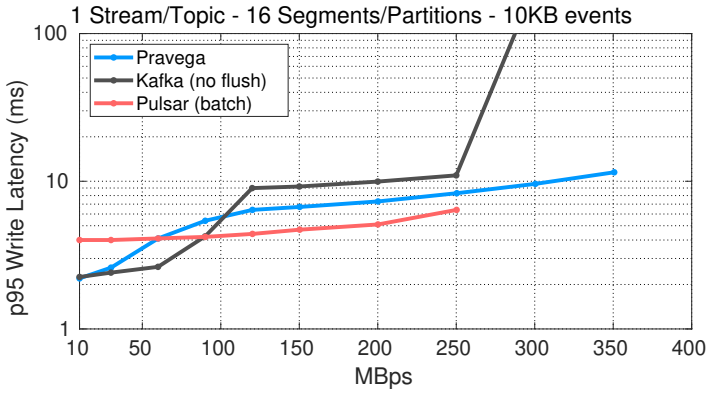

Figure 12: Write performance for larger events (16 segments/partitions, 1 writer/producer).

(330MBps) and Pulsar (250MBps).

However, the Pravega writer cannot get more than 160MBps for the single-segment case. The reason is that it is bottlenecked by the movement of data to LTS (AWS EFS). To validate this statement, we have conducted an experiment using a test feature that allows Pravega to write only metadata to LTS and no data (namely, `NoOp LTS`). From the results in Fig. 11, skipping the data writes to LTS enables much higher throughput for Pravega. Pulsar performs better with storage tiering enabled for the single-segment case because it does not throttle producers when LTS is saturated. That is, storage tiering is not an integral part of Pulsar's ingestion pipeline as it is in Pravega. While this might be seen as an advantage now, not throttling writers if LTS is saturated can lead to an ever-growing backlog of data waiting to be moved to LTS. The problem becomes evident when we present historical read results in further sections.

KPI-1 (Throughput Improvements): The Pravega writer achieves high write throughput when using multiple segments. The throughput of Pravega depends on the throughput capacity of LTS, and in the case LTS saturates, Pravega applies backpressure to avoid building a backlog of data.

Tail Reads and Routing Keys. For many applications, the time must be short between the event being generated and the time that it is available for reading and processing. We refer to such a time interval as end-to-end latency. In Fig. 13 and Fig. 14, we plot the end-to-end latency and throughput for 100B events of Pravega, Pulsar, and Kafka readers/consumers.

In Fig. 13, Pravega and Kafka exhibit lower end-to-end latency compared to Pulsar up to the saturation point. In fact, Pulsar does not achieve end-to-end latency values under 12ms (95th percentile), even with batching. On the other hand, read throughput for a stream/topic with a single segment/partition is much higher for Pravega (72%) and Pulsar (56%) than for Kafka.

Interestingly, in the case of 16 segments/partitions (see Fig. 14), Pulsar shows a 76% drop in read

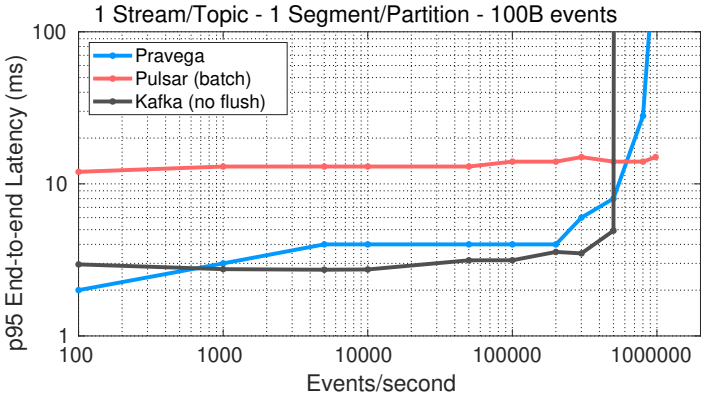

Figure 13: Performance of a tail readers/consumers (1 segment/partition, 1 writer/producer).

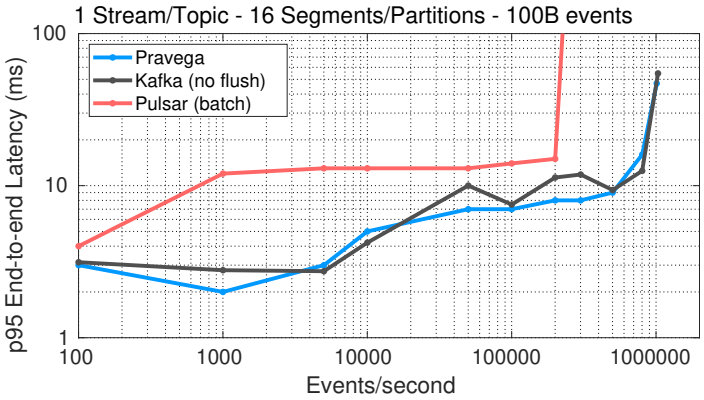

Figure 14: Performance of a tail readers/consumers (16 segments/partitions, 1 writer/producer).

throughput than the single partition case, despite configuring one consumer thread per segment/-partition in all systems (higher read parallelism). This may be a limiting factor in highly-parallel scenarios. In the case of Kafka and Pravega, managing more segments increases end-to-end latency for medium to high throughput rates.

We also want to understand the impact of using routing keys when writing and reading data from these systems. This is important as per-key event ordering is desirable for a number of applications to enable the correct processing of the events while providing parallelism. All of Pravega, Pulsar and Kafka use routing keys and guarantee total order per key. In Fig. 15 we depict the difference in performance of the readers based on whether no routing keys or random routing keys are used in the workload.

Fig. 15 shows that using random routing keys across several topic partitions induces a significant read latency overhead in Pulsar compared to not using routing keys (*e.g.*, 3.25x higher p95 end-to-end latency at 10k e/s). Still, Pulsar's read throughput remains the same even without using routing keys, which indicates that the performance limitation has another root cause. Similarly, when Kafka does not guarantee order or durability, read (and write) throughput is 59.6% higher. In contrast, Pravega performance is consistent irrespective of the use of routing keys. As many applications use routing keys for ordering purposes, it is crucial to highlight the performance differences induced by routing key access distributions.

KPI-2 (Data Speed Improvements): The Pravega reader achieves both low end-to-end latency and high throughput compared to Kafka and Pulsar for the cases tested. Pravega is virtually insensitive to the distribution of routing keys, as opposed to the other systems.

Handling High Parallelism. Next, we focus on the performance evaluation of Pravega in the presence of multiple writers appending to streams with many segments. We are interested in the append

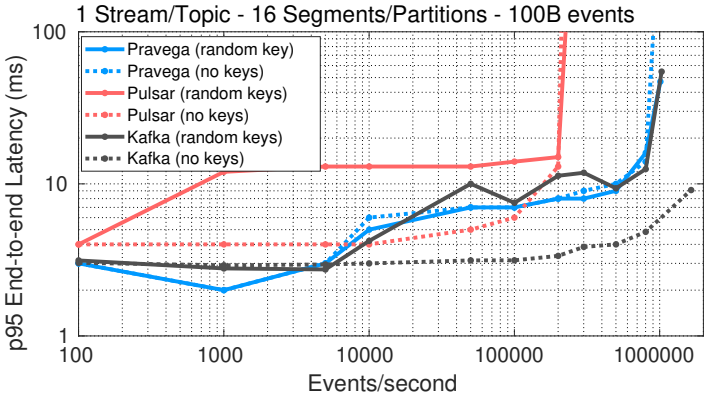

Figure 15: Impact of routing keys on read performance.

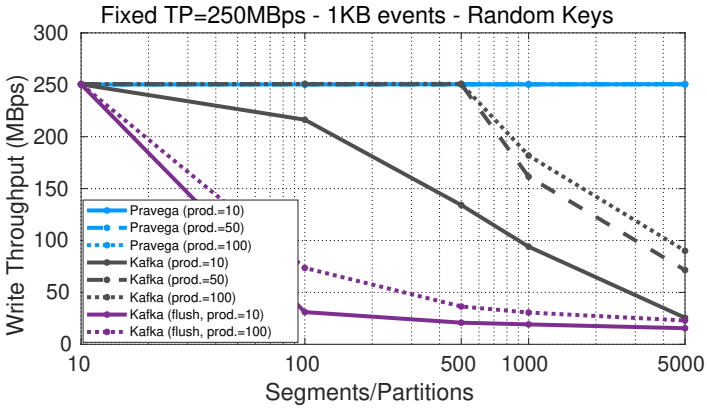

Figure 16: Impact of parallelism on write performance (Pravega vs Kafka).

path, which is critical for ingesting streaming data effectively. We chose to fix an ingest workload rate of 250MBps (1KB events) and show how the different systems behave when varying the number of writers/producers and segments/partitions. Also, this set of experiments slightly differs in deployment compared to the previous ones: i) the number of benchmark instances is 10 to support multiple clients (instead of 2); ii) to prevent CPU bottlenecks we use `i3.16xlarge` instances in segment stores, brokers, and bookies (instead of `i3.4xlarge`). In this experiment, Pulsar does not perform storage tiering.

Fig. 16 and Fig. 17 show throughput for Pravega, Kafka, and Pulsar with a varying number of stream segments and producers. Each line corresponds to a workload with a different number of producers appending to a single stream/topic. For Kafka and Pulsar, we also plot lines for alternative configurations that give more favorable results to those systems, at the cost of functionality.

Visibly, Fig. 16 and Fig. 17 show that Pravega is the only system able to sustain the target throughput rate of 250MBps for streams with up to 5k segments in a stream and 100 writers. It suggests that the design of the append path of Pravega, and specifically, the batching and multiplexing of small appends from many writers and segments at segment containers, is efficiently handling workload parallelism.

The Kafka throughput drops as we increase the number of topic partitions (see Fig. 16). Adding producers for Kafka yields higher throughput up to a limit. There is a significant difference between 10 and 50 producers, while between 50 and 100 producers, the throughput difference for Kafka is marginal. This result is likely because the lack of partition multiplexing in Kafka. To wit, high levels of write parallelism directly translate into an equivalent number of log files writing to the drive that can lead to degraded performance. Furthermore, when we enforce durability in Kafka (`flush`),

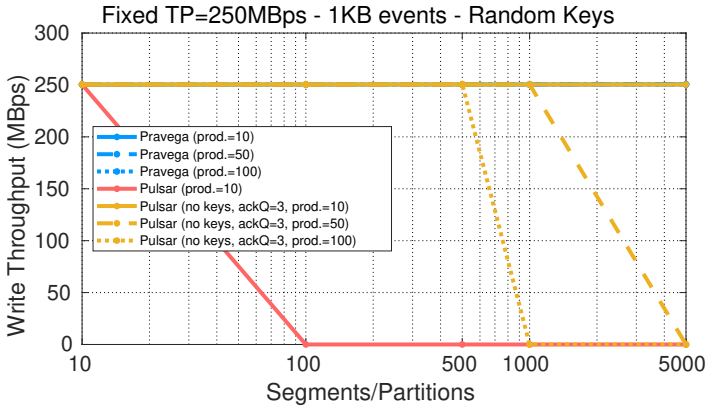

Figure 17: Impact of parallelism on write performance (Pravega vs Pulsar).

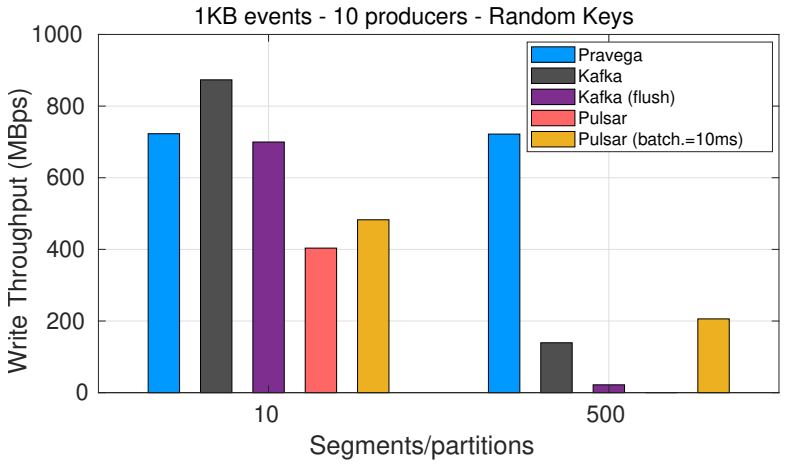

Figure 18: Max throughput achieved by systems under test.

throughput is much lower (*e.g.*, −80% for 100 producers and 500 partitions). While some penalty is expected from flushing messages to the drive, this experiment shows that enforcing durability for more than ten topic partitions penalizes throughput significantly.

Unfortunately, Pulsar crashed in most configurations we have experimented with (see Fig. 17). To understand the root cause of Pulsar's stability problems, we tried a more favorable configuration that: i) waits for all acknowledgments (`ackQ=3`) from bookies to prevent out-of-memory errors; ii) do not use routing keys to write events (*i.e.*, sacrifices event ordering and reduces the actual parallelism on writes). With this new configuration, Pulsar can get better results compared to the base scenario. However, it is still showing degraded performance and eventual instability when the experiment reaches high parallelism, especially when increasing the number of producers. Note that not using routing keys on writes seems to be the main contributor to Pulsar's improvement with the favorable configuration.

While we have used a fixed target rate in our previous experiments, we also want to understand the maximum throughput that these systems can achieve in our scenario. To narrow down the analysis, in Fig. 18 we pick 10 and 500 segments/partitions as a baseline, along with 10 producers and 1KB events.

Pravega can get a maximum throughput of 720MBps from the benchmark perspective for both 10 and 500 segments, translating into roughly 780MBps at the drive level. The difference is due to the metadata overhead added by Pravega (*e.g.*, segment attributes) and Bookkeeper. Note that this is very close to the maximum throughput we can get with synchronous writes on the drives used (we

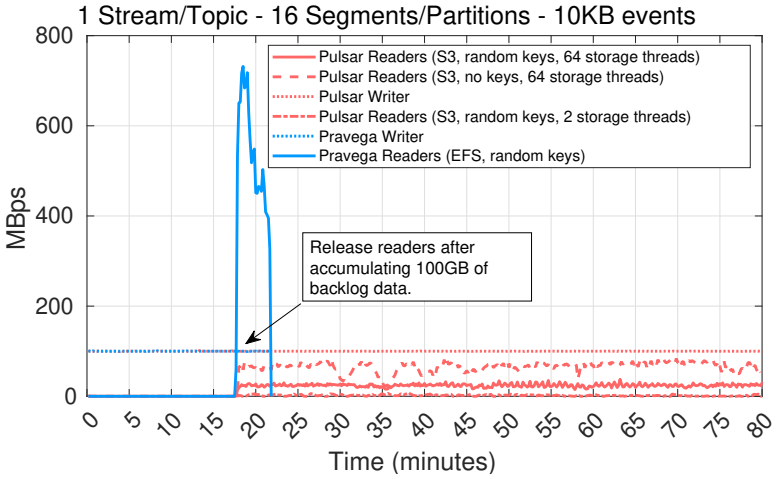

Figure 19: Historical read performance.

measured it via `dd` tool to approximately 800MBps). This confirms that Pravega (and Bookkeeper) can efficiently use the drives.

For Pulsar, with the defined configuration, we can reliably get almost 400MBps of throughput at the benchmark level. We have also explored increasing the client batching time to 10ms, which translates into a moderate improvement in throughput (20%). Still, we observe that this is far from the maximum capacity of drives, and we suspect that it is due to the use of routing keys, as it reduces the batching opportunities for Pulsar clients. Even worse, we also see that the Pulsar throughput is significantly limited as we increase the number of partitions. This result suggests that relying mainly on the client for aggregating data has important limitations.

For the 10-partition case, we observe that Kafka can achieve up to 700MBps and 900MBps, when it guarantees durability and when it does not, respectively. In the latter case, writing to "page cache" and letting the OS write data in larger blocks to the drive helps to get a higher maximum throughput. But note that this only happens for low parallelism, as for 500 segments, the throughput drops dramatically to 22MBps and 140MBps, respectively.

KPI-1 (Throughput Improvements): Pravega is the only system that achieves consistent throughput for the number of producers and segments tested, while guaranteeing event ordering and data durability. Also, it efficiently exploits drive throughput irrespective of the degree of parallelism.

Historical Read Performance. In this set of experiments, we analyze the performance of readers when requesting historical data from LTS in Pravega and Pulsar (at the time of this writing, Kafka did not provide this functionality in open source). We designed the experiment as follows. OpenMessaging Benchmark has an option that holds readers until writers have written the amount of events specified. Readers are subsequently released, and the experiment is complete when the backlog of events is consumed. We exercised this option by configuring writers to write 100MBps (10KB events) to a 16-partition/segment topic/stream at a constant rate until achieving a backlog of 100GB. Note that writers continue to write data when readers are released, so readers should read faster to eventually catch up.

In Fig. 19, we observe that Pravega achieves much higher historical read throughput than Pulsar by exploiting parallel chunk reads (peaking at 731MBps). For Pulsar, none of the configurations tested resulted in a historical read throughput higher than the write throughput. While parameters like the number of offloading threads or the routing keys used in Pulsar influence the performance of historical reads, we did not find clear guidelines for users to configure tiered storage. We also have discarded that the reason for the read performance difference is due to the LTS system used, as we tested both EFS and S3 to achieve very similar throughput rates for file/object transfers (*i.e.*, 160MBps approx.).

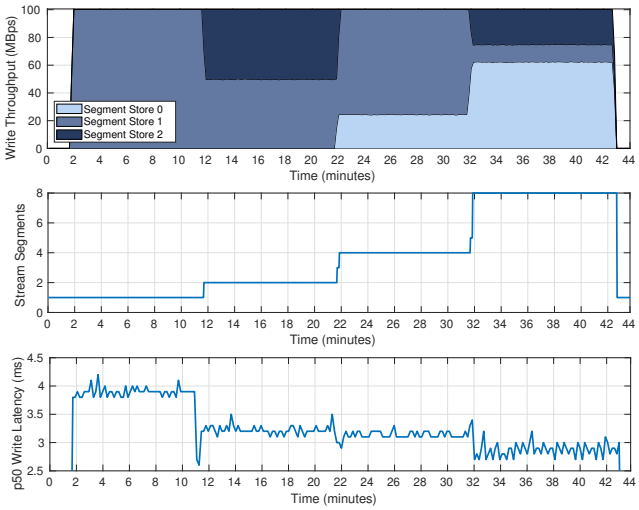

Figure 20: View of stream auto-scaling role on performance.

Another interesting observation from Fig. 19 is the following: Pulsar does not throttle writers in case LTS cannot absorb the ingestion throughput, which may lead to situations of imbalance across storage tiers. To wit, if writers write to Bookkeeper faster than the brokers offload data to LTS, the backlog of events waiting to be moved to LTS would grow without bounds. In the long run, this may have negative consequences for users who rely on timely data offloading to LTS.

KPI-1 (Throughput Improvements): Both Pravega and Pulsar provide means to move historical data to long-term storage. However, Pravega achieves much higher historical read throughput compared to Pulsar without user intervention or complicated configuration settings.

Stream Auto-scaling. Accommodating workload fluctuations over time through stream auto-scaling is a unique feature of Pravega. The absence of this feature in systems like Pulsar and Kafka, which are primarily on the front line of big data ingestion, may induce considerable operational pain to users, particularly at scale.

We focus in this section on the performance implications of stream auto-scaling. We configured auto-scaling in the test stream and we set a target rate of events per second on segments to 2k (or 20MBps, given that we used 10KB events). The benchmark tool wrote at a speed of 100MBps and the stream initially had one segment. Note that to generate some of the plots below we used the Pravega metrics exports. The three plots in Fig. 20 show different aspects of stream auto-scaling in Pravega: i) the write workload per segment store, ii) the number of segments in the stream, and iii) the write latency (p50) perceived by the benchmark instance.

Fig. 20 reveals that as the stream splits and adds more segments, the load is distributed across the available segment stores, thus causing latency to drop. As in this experiment all the segments receive the same number of writes, the actual load distribution across segment stores mainly depends on the placement of segments across segment containers. As the actual placement strategy of segments is stateless (based on consistent hashing), for a small number of segments there may be situations of load imbalance. This is the reason why not all the segment stores receive the same amount of load as stream auto-scaling progresses. Fortunately, a larger number of segments increases the probability of an even distribution of load across segment stores and segment containers [11].

KPI-3 (Resource Auto-scaling), KPI-5 (Simplicity and Productivity): Pravega is the first streaming storage system that provides elastic streams: data streams that are automatically re-partitioned according to the ingestion load and the scaling policy.

5.2 Streaming Storage-Compute Auto-Scaling

Stream processing pipelines need to handle workload fluctuations (e.g., daily patterns, popularity spikes) by scaling up/down the resources contributed to running jobs. This is likely the case in NEARDATA use cases; *e.g.*, surgery rooms occupancy (NCT for computer-assisted surgery) or even genomic data generation (UKHSA for genomics data storage and analytics) may show strong daily patterns due to standard work shifts. While there have been efforts proposing auto-scaling mechanisms for stream processing engines, prior work has overlooked the role of the storage system in ingesting and serving stream data. The absence of effective scaling for data streams is problematic given that the number of parallel partitions of a data stream limits both streaming data ingestion throughput and read parallelism for downstream streaming jobs.

In NEARDATA, we propose to augment the auto-scaling notion of stream processing engines with information about the source data stream. The key novelty of our approach lies in exploiting *elastic data streams* to ingest data, which is a unique feature of Pravega: a storage system for data streams part of the Dell's Streaming Data Platform. Pravega streams can dynamically change their parallelism based on the ingestion workload, and such information can in turn be exploited for auto-scaling the streaming job downstream. To this end, we have developed an Apache Flink connector for Pravega, as well as an auto-scaling orchestrator that feeds on data stream metrics. Our experiments show how a stream processing pipeline auto-scales by coordinating data stream and processing parallelism under workload fluctuations, with low operations cost.

5.2.1 Connecting Flink with Pravega

Apache Flink [12, 13] is a widely used real-time stream processing engine that provides unified batch and real-time analytics. Flink achieves high-throughput, low-latency streaming data processing, as well as support for complex event processing and state management. In a nutshell, the Flink runtime program is a DAG of stateful operators connected with data streams. There are two main APIs in Flink: the `DataSet` API for batch processing (i.e., finite data sets), and the `DataStream` API for stream processing (i.e., unbounded data sets). A Flink cluster is formed by the client(s), the job manager, and at least one task manager. The client transforms the program code into a DAG and submits it to the job manager. The job manager coordinates the distributed execution of the dataflow. It tracks the state and progress of each operator, schedules new operators, and coordinates checkpoints and recovery. The actual data processing takes place in the task managers. A task manager executes one or more operators that produce streams of processed data to other operators and reports its status to the job manager [13].

Both Pravega and Flink share the design principle of treating the data stream as a first-class primitive, which makes them well-suited building blocks to jointly construct stream data processing pipelines. For enabling Pravega to be a Flink data source/sink for data streams, we have developed a Flink connector for Pravega [14]. As we show next, the connector offers seamless integration with Flink instances, thereby ensuring parallel reads/writes, checkpointing, and guaranteeing exactly-once processing with Pravega.

Pravega data source for Flink. In Pravega, readers are organized into *reader groups*. A reader group is a named collection of readers, which together perform parallel reads from a given Stream. Pravega guarantees that each event written to a stream is sent to exactly one reader within the reader group. There could be one or more readers in the reader group, and there could be many different reader groups simultaneously reading from any given stream. Each reader in a reader group is assigned zero or more stream segments. The reader assigned to a stream segment is the only reader within the reader group that reads events from that stream segment. As visible in Fig. 21, readers within a reader group can dynamically rebalance the assignment of segments upon a membership change or when the number of parallel stream segments changes due to stream auto-scaling. The `FlinkPravegaReader` implementation internally uses the Pravega reader API to serve events to Flink jobs.

The Flink connector for Pravega also ensures failure recovery for streaming jobs. To wit, Flink

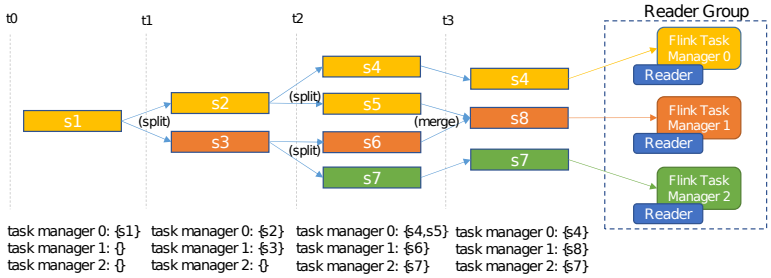

Figure 21: Example of Flink job executed across 3 task managers that reads events from a Pravega stream. Visibly, the stream segments are dynamically distributed within the reader group as stream auto-scaling takes place.

implements asynchronous periodic checkpoints [15] (Chandy-Lamport algorithm [16]) to make both Flink state and stream positions recoverable. Similarly, Pravega also has its own *checkpoint* concept that applies to a reader group reading from a stream. In Pravega, a reader group checkpoint creates a consistent reference for a position in the stream that an application can roll back to. In our connector, we combine the two notions of checkpoints for recovering a stream processing job. We worked with the Flink community to add a new interface, namely `ExternallyInducedSource` [17], to allow such external calls for checkpointing, and, finally, the connector integrated this interface to guarantee failure recovery.

Pravega data sink for Flink. The `FlinkPravegaWriter` implementation allows Flink jobs to output results from a streaming computation to Pravega in a consistent, durable, and ordered manner. A key feature of our connector when used as a sink for Flink jobs is that it provides "exactly-once" semantics (i.e., each incoming event is guaranteed to be effectively processed only once). Enabling exactly-once semantics involving an external streaming storage system as a data sink, however, requires cooperation between storage and compute layers. The way Flink implements exactly-once involves retries, which implies that the output might be partially written. Therefore, the sink of an external system that a Flink job outputs to must support commits and rollbacks.

Pravega supports *transactional writes* [18], which matches the requirement of committing and rolling back (aborting a transaction). Pravega transactions allow applications to prepare and then commit a set of events that can be written atomically to a stream. Pravega transaction semantics are such that upon a commit, it processes all events of the transaction to enable them for reading. If a transaction aborts instead, then no transaction event becomes available for reading. Pravega transactions enable a Flink job to align the checkpointing process with committing the output, thus allowing us to achieve exactly-once processing pipelines. Still, building such a solution is non-trivial: the main difficulty is to have the coordination between Flink and the Pravega sink. One common approach for coordinating commits and rollbacks in a distributed system is the two-phase commit protocol [19]. We followed this path, worked with the Flink community, and implemented the sink function via a two-phase commit coordinated with Flink checkpoints [20, 21].

5.2.2 Auto-scaling Orchestrator

Previously, we have described two key building blocks for providing storage-compute elasticity in stream data processing pipeline with Pravega and Flink: i) Pravega auto-scaling streams, and ii) a Flink connector for Pravega. Next, we describe the orchestrator component that allows us to scale compute and storage services independently, based on policies and data ingestion metrics.

Fig. 22 describes the architecture of the orchestrator component. First, the orchestrator is in charge of consuming monitoring information from both storage and processing engines. This information is the foundation for making auto-scaling decisions. Second, the orchestrator accepts user-defined scaling policies based on monitoring metrics. It continuously evaluates the metrics related to the existing policies for reacting and scaling up or down a specific service accordingly. Concretely, we

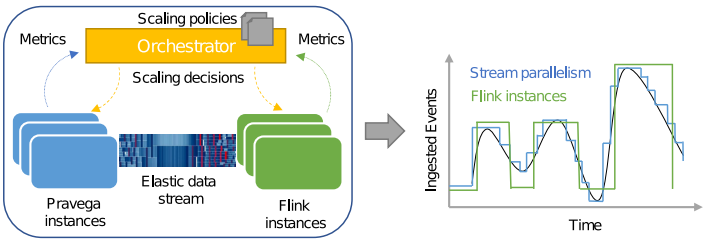

Figure 22: Orchestrator component and its interactions with user-defined scaling policies, Pravega, and Flink. Moreover, we show an example of exploiting the information about the parallelism of a Pravega stream for right-sizing a Flink job.

have implemented the orchestrator component via the Kubernetes horizontal pod autoscaler [22] consuming monitoring information from Prometheus [23].

While having an orchestrator component is a well-known practice in modern streaming processing pipelines, the novelty of our approach lies in allowing it to exploit metrics related to the storage side, such as the data stream parallelism, for taking scaling decisions about the stream processing job (and *vice-versa*). Fig. 22 illustrates precisely this. Visibly, the orchestrator consumes metrics from both Pravega and Flink. A user can also define scaling policies that involve metrics from both systems. In Fig. 22, we focus on a policy that addresses a common problem: keeping the the number of Flink task managers equal to the number of segments of the source data stream for load-balancing reasons. As depicted in Fig. 22 (right), due to the fluctuations in the ingestion workload, the Pravega stream changes the number of parallel segments dynamically. Due to the policy set in our orchestrator, the Flink job scales the number of task managers accordingly. Note that such a mechanism is generic enough to build policies that combine metrics from the data stream with other storage and/or processing-related metrics.

5.2.3 Experimental Evaluation

Setup. To run our experiments, we use a virtual VMware Tanzu Kubernetes cluster made of 6 virtual nodes (16 CPUs, 16GB of RAM each node). We deployed Pravega (0.12.0), Apache Bookkeeper (4.14.1), and Apache Zookeeper (3.7.1). For long-term storage in Pravega, we used a network file system (DellEMC Isilon H600 [24]), whereas for provisioning regular storage volumes we used VMware vSan. We also deployed Apache Flink (1.13.6) in reactive mode [25], so we can modify the number of task managers on-the-fly. The workload is generated by an application that uses the Pravega writer to simulate a continuous data source. Concretely, the data being written are samples (FASTQ format) of Pathogen DNA sequences from UK Health State Agency [26]. The data generator writes sequences as individual events. We instructed the data generator to follow a sinusoidal throughput pattern exhibiting a 1000x difference between peak and valley loads within 30-minute intervals. This is more aggressive than some real workloads [27]. For processing, we wrote a Flink application (1 slot per task manager) that performs multiple operations on a per-sequence basis (i.e., finding multiple patterns, counting DNA bases). Our goal is to evaluate a CPU-intensive scenario in which scaling out the streaming processing engine is an important requirement upon workload fluctuations.

Estimating operations cost. First, we evaluate the complexity of augmenting the stream processing pipeline with our storage-compute auto-scaling mechanism from the user's viewpoint. Essentially, the user needs to reason about two parameters: i) the *stream scaling policy rate*, and ii) the ratio of *segments to task managers*. With these two parameters, a user has full control of both the IO load received per task manager, as well as the data ingestion throughput for the streaming pipeline.

We target a CPU-intensive Flink job, and we infer the actual performance of a task processing genomic data. Fig. 23 shows that a task manager (running a single task) saturates with 100% CPU utilization when it reaches $\approx$ 1.4K events/second in our cluster. With this data point, we choose a scaling policy with a rate of 0.8K events/second, which translates into a CPU utilization within the

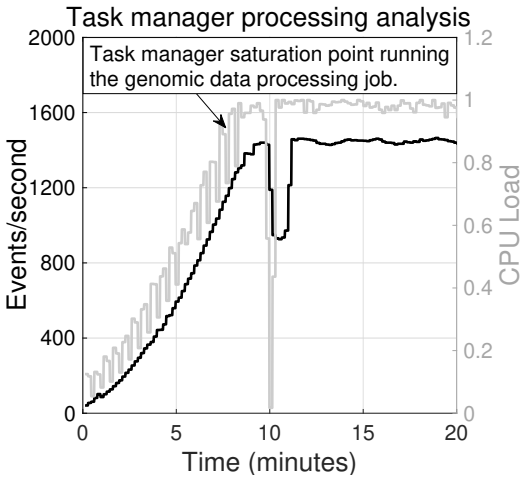

Figure 23: Task manager performance running the genomic test job.

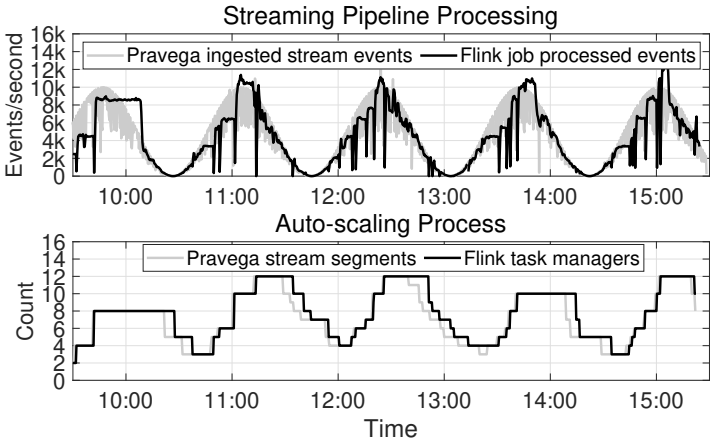

Figure 24: Data ingestion and processing overview of the streaming pipeline with our storage-compute auto-scaling mechanism.

range of 0.4 and 0.6 according to Fig. 23. Besides, given that the bottleneck is at the processing end, we define a one-to-one relationship between stream segments and task managers as a policy in the orchestrator component. Naturally, for IO-heavy workloads in which Flink jobs perform lightweight computations, it would be reasonable to have multiple segments per task manager. Overall, reasoning about these two parameters —which can be modified at runtime– is all a user needs to materialize storage-compute auto-scaling of streaming pipelines.

KPI-5 (Simplicity and Productivity): Administrators can reason about the storage-compute auto-scaling of the streaming pipeline considering just two main parameters.

Storage-compute auto-scaling. Next, we aim to evaluate the behavior of our storage-compute auto-scaling mechanism for streaming processing pipelines. Fig. 24 shows a 6-hour experiment in which a fluctuating ingestion workload influences both the Pravega stream parallelism and the number of Flink task managers.

Visibly, Fig. 24 shows how the number of stream segments changes dynamically according to the workload intensity. As mentioned previously, the scaling policy for the stream is set to 0.8K events/second. Note that during stream scaling, the Pravega control plane still guarantees event ordering per routing key as it changes the segment configuration of the stream. On the reader side, we also observe that the number of Flink task managers auto-scales according to the number of

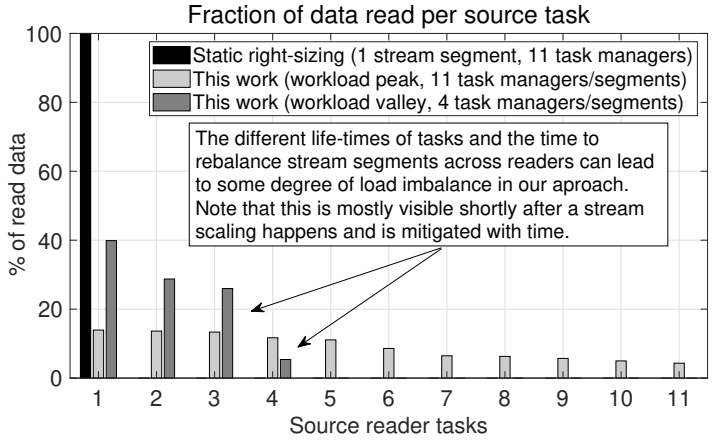

Figure 25: Impact of incurring in mismatch between data stream and processing parallelism.

segments in the stream. The job scales because we have set a one-to-one relationship between task managers and segments in the orchestrator policy. As expected, the auto-scaling process for the Flink job has a direct effect on its processing throughput. Furthermore, despite the dynamic changes in stream parallelism, our connector and the underlying reader group semantics guarantee that all events are read only once.

KPI-3 (Resource Auto-scaling): The orchestrator component allows us to auto-scaling streaming analytics jobs automatically based on elastic Pravega stream parallelism.

Implications of over-provisioning task managers. Finally, we want to evaluate the implications in our scenario of not aligning the number of data source stream segments with the number of task managers. Concretely, Fig. 25 shows a case of over-provisioning on the data processing side, as we set up a 1-segment data stream and 11 Flink task managers executing the job. We observe that only 1 Flink task is actually reading data from the Pravega stream, whereas the other 10 do not interact with Pravega. This represents a potential IO bottleneck. Furthermore, Flink attempts to use all the available task managers for executing computations in this scenario. This means that the only task reading data from the Pravega stream also needs to redistribute read events across the other task managers for balancing compute load, which yields further shuffling and network overhead. Conversely, our auto-scaling mechanism keeps a much more balanced load across task managers related to data source reading by keeping the number of task managers and segments in sync. The observable imbalance across tasks in terms of read data percentage is due to the different lifetimes of tasks and the time it takes for our connector (and the reader group) to redistribute segments across a dynamic number of task managers. This imbalance is mainly visible right after a change in the number of stream segments and gets corrected with time. Overall, this experiment supports the relevance of coordinating both data stream and processing parallelism for fully utilizing available resources.

KPI-1 (Performance Improvements), KPI-3 (Resource Auto-scaling): Our storage-compute auto-scaling mechanism improves load balancing and reduces shuffling transfers across task managers automatically, whereas static provisioning may lead to problems under dynamic workloads.

5.3 Byte Streaming for Video Analytics

Event streaming systems, such as Apache Kafka, are being increasingly used in data-intensive, storage-heavy workloads. These systems center their internal design and external APIs around the concept of *event*. However, we position that this fundamental design decision may be limiting for some complex, storage-centric use cases. Instead, we propose that event streaming systems build on top of a *byte-oriented streaming primitive* supporting key properties such as atomicity, conditional writes, and durability. Such a primitive would allow us building multiple APIs via data streams (*e.g.*, event, K/V,

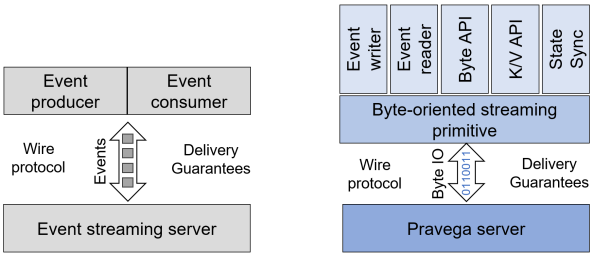

Figure 26: Event versus byte-oriented streaming.

synchronization), as well as to directly exposing byte-level APIs to cope with workloads requiring large data transfers or low-latency byte buffer IO. We implement this concept in Pravega: a tiered storage system for data streams. Via experiments on AWS, we show the practicality of exposing byte APIs for storage workloads compared to events.

5.3.1 Event vs Byte Semantics

In this section, we discuss key differences in the semantics of streaming events and bytes. This will help the reader to better understand the underlying abstractions that distinguish Pravega from other event streaming systems (see Fig. 26).

Discrete vs continuous data streaming. Inherently, using the event as the main data abstraction implies the discretization of the data stream. In many cases, this approach works well, like in publish/subscribe scenarios where events are small by nature. However, discretizing a data stream into events implies that the client should be able to write an event as a whole unit. If we think of users writing chunks of data with moderate size at once (*e.g.*, high resolution satellite image), it may be hard for the client to internally manage write operations with large payloads (see Section 5.3.3). This is specially true if we consider the configuration changes required for some systems to ingest events larger than the default maximum size. Instead, the Pravega byte-oriented streaming primitive enables clients to perform large data writes in an efficient manner and as a single logical unit.

Interpreted vs raw data streaming. Using events as the unit for performing data stream IO implies encapsulating and storing event data in a protocol-related envelope (*e.g.*, event headers, metadata). This is acceptable in applications integrating clients that implement the event streaming protocol for managing data. However, there are frameworks for which this may not be required. For instance, multimedia frameworks like GStreamer [28] work with byte buffers instead of events. For such use cases, providing a byte-level API for streaming data is the most natural approach (see Section 5.3.3). Further, using data events for large data transfers would lead to artificially partition a logical piece of data (*e.g.*, file, object) across the stream. Conversely, the Pravega byte-oriented streaming primitive allows clients to write a continuous stream of bytes addressable by offset.

Event-only vs multi-API data streams. Clients in event streaming systems mainly provide event producer and consumer APIs. This is reasonable given that the internal protocol between clients and servers, as well as the data delivery guarantees focus on data events. Instead, Pravega is the first system offering event streaming APIs on top of a byte-oriented streaming primitive. More importantly, the data reliability and durability guarantees in Pravega are provided at the byte level, not at the event level. This is a key departure from other systems that has an important benefit: a byte-oriented streaming primitive is the substrate for other APIs beyond event streaming. A good analogy is TCP, as it is implemented on top of a packet oriented protocol and encapsulates complex logic to guarantee correct in-order delivery that higher level applications rely on. As visible in Fig. 26, Pravega exposes byte, key/value, and state synchronization APIs on top of the byte-oriented streaming primitive.

5.3.2 Pravega Byte Streams

The objective of the Pravega byte-oriented streaming primitive is to enable the streaming of bytes in and out of a segment without imposing any framing or event demarcation. This primitive facilitates

the writing of data to Pravega without organizing it into events. Data written using this approach remains unframed and uninterpreted by Pravega, meaning there are no length headers or event boundaries. Consequently, byte offsets within the stream are meaningful and directly accessible. However, for this primitive to become a building block for more sophisticated APIs it is required to support: i) *atomicity*, ii) *conditional writes*, and iii) *durability*.

Atomicity: Pravega clients write (or *append*) data to stream *segments* in an atomic manner. At the server side, Pravega has a control plane —in charge of stream lifecycle operations and metadata– and a data plane —made of segment store instances handling segment IO [29]. In addition to its own data, each stream segment has an associated set of *attributes* (*i.e.*, key-value pairs with 16-byte keys and 8-byte values) that can be used either independently or in combination with various segment operations. For example, we can choose to only update some attributes, or we can choose to *atomically* append to a segment and update some of its attributes. The segment store's ingestion pipeline processes all appends in the order in which they were received, and each segment write and attribute update is atomically committed, thus ensuring the consistency of the segment data and metadata [30].

Conditional writes: As part of the segment API, segment attributes can be updated using one of the following verbs: `replace`, `replace-if-greater` (an attribute's value is updated to v if the client provides a new value v' and $v > v'$), `replace-if-equals` (an attribute's value is updated to v'' only if the current value v matches a client provided value v'), and `accumulate`. It is worth noting that the `replace-if-equals` verb embodies the classical *compare-and-set* (CAS) primitive, which provides strong consistency guarantees [31, 32]. One of the main advantages from a having a CAS primitive on segment appends is that we can implement *conditional writes on attribute values*.

Durability: Pravega is a tiered storage system for data streams. Therefore, the segment store first writes segment operations on a durable write-ahead log (WAL) [2, 1]. In parallel, it aggregates and moves segment data to long-term storage (LTS). Segment appends, including segment attribute updates, are *durably stored in WAL* before acknowledging the client and then moved to LTS. In the case of a crash, the segment store reads the tail of the WAL and recovers its internal state. This includes the latest data successfully appended to each active segment and the latest persisted state of their attributes. It is worth mentioning that segment attributes are stored in LTS separately from segment data (in a special B+Tree storage data structure [33]).

Byte Streams for a Reliable Event API. Next, we illustrate how the Pravega event API uses the byte-oriented streaming semantics for preventing event loss and duplication. A `EventStreamWriter` has a unique *writer id* and can write to multiple segments at once. Its internal state is made up of a map of *event numbers* for each segment it interacts with, that is updated every time it needs to process a new event. Every event is sent to the segment store as a *conditional append* on the event number for its writer id on that segment to match what it should be. Similarly, each segment in the segment store has an *attribute map* of writer ids to event numbers, which is atomically checked-and-updated with every append. As depicted in Fig. 27, there may be the cases (e.g., network issue, server crash) that could force the `EventStreamWriter` to retry an append that has not been acknowledged. When an append is retried, it will only be written if it is the next to write according to the server's segment attributes state. Otherwise, a conditional check failure will be returned to the client. Combining retries and conditional writes helps preventing data loss (*i.e.*, event was never persisted) and duplication (*i.e.*, event was persisted but not acknowledged) in the event API. Conditional appends also ensure that it is impossible for an earlier write to succeed and a later one to fail. Hence, appends can be *safely pipelined*, as the client does not have to wait for the acknowledgment of prior data to write more.

Pravega Byte API Use Cases. Pravega defines a layered approach to build APIs on top of stream segments (see Fig. 26). The Pravega Byte API can be seen as the lowest-level, user-facing interface for users to manage data in Pravega. Next, we describe two storage-centric use cases that are founded on the Pravega Byte API:

Large file transfers: The Byte API client interfaces, namely `ByteStreamWriter` and `ByteStreamReader`, implement the `InputStream` and `OutputStream` (Java) APIs, respectively. These APIs align well with

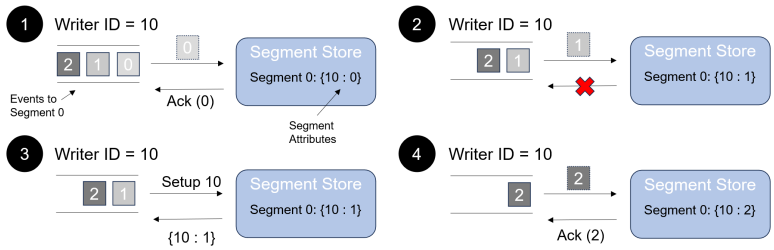

Figure 27: Event API usage of conditional writes and retries to avoid event loss or duplication.

	Pravega	**Kafka**
Version	Pravega 0.13.0, Bookkeeper 4.14.1	Kafka 3.6.1
Replication	ensemble=1, writeQuorum=1, ackQuorum=1	replication=1, acks=all, min.insync.replicas=1
Durability	Yes (default)	Yes (via configuration)
Resources	Controller=1CPU/2GB, Segment Store + Bookie=8CPU/48GB, Zookeeper=2CPU/4GB, Benchmark=4CPU/8GB	Broker=8CPU/48GB, Zookeeper=2CPU/4GB, Benchmark=4CPU/8GB
Journal Drives	1 NVMe	1 NVMe

Table 2: Default experiment configuration.

numerous existing use cases, particularly those involving the transfer of large files.

GStreamer pipelines: Streaming byte buffers is an essential feature for multimedia frameworks like GStreamer [28]. With GStreamer, users can build multimedia pipelines and streaming applications. We have developed a GStreamer connector for efficiently supporting video analytics [34]. Via the Pravega Byte API, the GStreamer connector exposes the ability to read and write byte buffers corresponding to video frames in GStreamer pipelines. Thus, Pravega is a durable `sink` and a low-latency `source` for GStreamer applications.

5.3.3 Experimental Evaluation

Our experiments focus on the performance of the Pravega Byte API from two angles: i) large data transfers, and ii) low-latency byte buffer writes. We summarize in Table 2 the configuration used in our experiments on AWS EKS [35]. For the large data transfer experiments, we compare the performance of Pravega against Apache Kafka [5], as the most widely adopted messaging system. In this case, the workloads are executed with OpenMessaging Benchmark [7], which has been extended with the Pravega Byte API. For the experiments with video byte buffers, we use the Pravega GStreamer connector [34] to perform video stream IO.

Setup. Our cluster consists of 3 `i3en.2xlarge` instances that act as Kubernetes nodes in our AWS EKS deployment. This type of instances contain local NVMe drives for fast local storage [36]. In our deployment, we use one of such NVMe drives for journaling in both Kafka and Bookkeeper (WAL for Pravega). We configured a second drive for Bookkeeper ledger volume, but it is not relevant for the presented results. As we want to evaluate the interactions between clients and the server, we configured the deployments with one Kafka broker and one Pravega segment store/bookie pair (co-located). Therefore, the replication layout is set to 1 in both systems. For data durability, we evaluate Kafka with strict durability (`log.flush.interval.ms=0` and `log.flush.interval.messages=1` to force `fsync` upon write acknowledgement). This is the equivalent to the durability configuration in Pravega (and Bookkeeper) by default, which is a key property for large byte streams. Conversely to Kafka, in our experiments Pravega has an additional cost of tiering data to LTS (AWS S3 bucket).

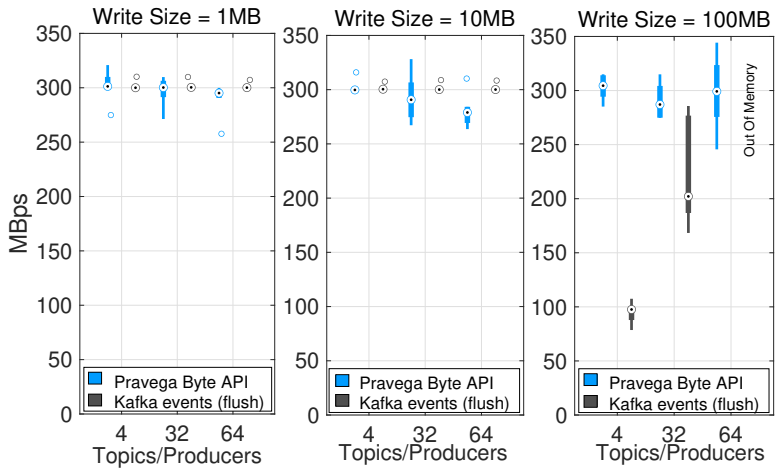

Figure 28: Pravega and Kafka throughput for transfers of multiple write sizes and number of topics/producers.

Large Data Transfers. Next, we evaluate Pravega and Kafka for large data transfers. As Kafka does not have a byte API, we evaluate the performance of the system transferring large events. Note that for enabling Kafka transferring events larger than 1MB we had to modify several parameters in both the producer (*e.g.*, `message.max.bytes`, `max.request.size`) and the broker (*e.g.*, `message.max.bytes`, `buffer.memory`) sides. Pravega does not need configuration changes, which favors usability.

To set a target ingestion rate, we have measured the performance of `i3.2xlarge` drives available in both Kafka and Pravega. Via `fio` [37] microbenchmarks, we observed that local drives can deliver up to $\approx$ 330MBps using large ($>$ 16KB), synchronous writes (`direct=1` in `fio`). Note that using synchronous writes is a requirement for data durability in byte streams. Our goal is to measure the throughput of both Kafka event API and Pravega Byte API for different write sizes and degrees of parallelism, while targeting 300MBps.

Fig. 28 shows the results of executing write workloads via 2 benchmark workers running on distinct nodes to the ones running the streaming services. Each worker instance runs producer threads writing to a 1-partition/segment topic/stream. Visibly, for 1MB and 10MB write sizes, both Pravega and Kafka can sustain the targeted 300MBps ingestion rate. This is expected as the size of writes themselves are large enough to efficiently use local drives, as in the case of large event batches. Compared to Kafka, Pravega shows greater variability in throughput. This is mainly because, in addition to ingesting data, Pravega is continuously tiering data to an S3 bucket at the ingestion rate. This induces an extra network and CPU cost unique to Pravega.

We observe that Pravega handles better than Kafka large data writes (100MB). For instance, in the 4 topics/producers case, Pravega achieves a median throughput 3x higher than Kafka. Even more, in the 64 topics/producers case for Kafka, the benchmark workers crashed due to lack of heap memory. The reason may be related to an eager behavior in the Kafka producer when allocate memory for event payloads (`buffer.memory`=150MB). To wit, the maximum JVM heap memory defined per benchmark worker (4GB) is lower than 32 producers allocating `buffer.memory` heap memory each. This indicates better efficiency in the Pravega writer.

KPI-1 (Throughput Improvements): Pravega Byte API can achieve higher throughput for large data transfers compared to the Kafka event API.

Byte Buffer Streaming. In Fig. 29, we show the end-to-end latency for a GStreamer test application consisting of a video writer and a video reader process running on individual pods. The end-to-end latency is defined as the time taken for a video frame to be written to Pravega and read by the reader. The measurement is done by tagging each video frame in GStreamer with the writer timestamp and calculating the delta at the reader end using its local time (the cluster runs NTP for

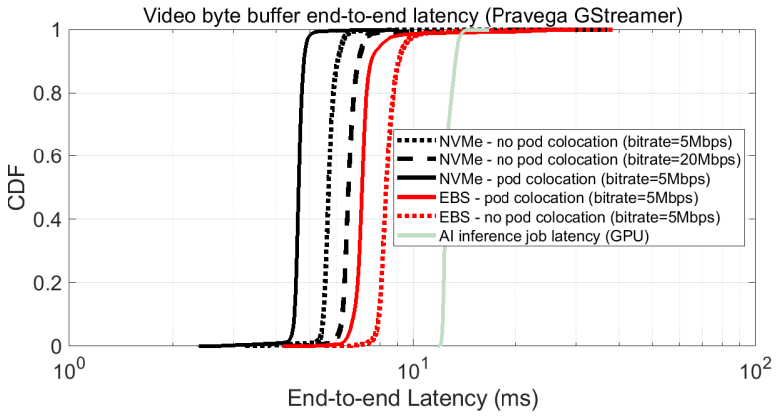

Figure 29: End-to-end latency of GStreamer byte buffers in Pravega.

time synchronization). We evaluate the impact of aspects like network location and storage type on video frame end-to-end latency.

Fig. 29 illustrates the end-to-end latency that the Pravega Byte API achieves in a GStreamer pipeline (1 writer/1 reader). First, irrespective of the storage used in the Pravega deployment (NVMe, EBS volumes), we observe that the end-to-end latency at p95 falls below 10ms. This yields that the Pravega Byte API is a reliable and efficient substrate for GStreamer video analytics pipelines. To support this observation, Fig. 29 also shows the tame taken by an AI inference job provided by the National Center for Tumor Disease (Germany) [38]. Such an inference job is intended to identify surgery instruments in a real-time video stream from surgery cameras. At the p95, the job's inference model exhibits a latency between 42.3% and 176.1% higher than the Pravega end-to-end latency, depending on the deployment model.

Still, it is important to understand the impact on end-to-end latency related to both the storage and network in a video pipeline. First, we notice that the usage of local NVMe drives in our experiments exhibits an end-to-end latency improvement of around 39.4% at p95 compared to AWS EBS volumes (gp2). This can be explained due to the frequent `fsync` calls achieve per-write data durability that may entail higher latency for network volumes. Similarly, we also point out that co-locating video writer and reader pods with the Pravega segment store improves latency by 14.9% to 21.3%.

KPI-2 (Data Speed Improvements): Pravega Byte API achieves low end-to-end latency for GStreamer byte buffer streaming compared to AI inference latency, specially with local storage and network co-location.

5.4 Streaming for Data-intensive Serverless Functions

In the first half of the NEARDATA project, we targeted an initial integration of serverless functions with Pravega streams. We believe this integration is important, as it is increasingly common to find advanced Function-as-a-Service (FaaS) serverless use cases consisting of complex function pipelines. While prior art has mainly focused on using object storage or in-memory stores to support FaaS computations, we find that data stream semantics are naturally aligned with the requirements of FaaS pipelines. Next, we perform an analysis of how Pravega streams can be a storage substrate for batch and streaming serverless analytics pipelines. Moreover, we identify that Pravega has specific features and semantics (*e.g.*, storage tiering, event key routing, exactly-once semantics, K/V interface) that can be of great use for building serverless pipelines. The outcomes of this work can pave the way to offering simplified serverless analytics engines on top of Pravega streams with advanced capabilities.

5.4.1 Storage in Serverless Analytics Pipelines

Data storage is key for serverless analytics pipelines given that functions do not only have to ingest the initial input and write the final output to persistent storage, but also to store intermediate results reliable and share them across functions [39]. However, it has been identified that storage is a major performance bottleneck for handling many parallel function interactions [40, 41]. Arguably, object storage is the main storage substrate in FaaS services. For instance, AWS Lambda [42], Google Cloud Functions [43], and Azure Functions [44] can be easily integrated with Amazon S3 [45], Google Cloud Storage [46], and Azure Blob Storage [47], respectively. We can also find multiple research works focusing on object storage as the main storage substrate [48, 49, 50]. While object storage offers high parallelism rates for both reads and writes, it presents some known weak points. First, single object writes are slow [51], which in some cases may imply a lower performance. Second, reads from object storage must wait until the writing has finished. This limitation means that the usage of object storage hinders the possibility to apply function pipelining (*i.e.*, the output of one function is the input for the next one).

On the other hand, multiple works have resorted to in-memory stores for storing intermediate results in serverless pipelines. Like in the case of object storage, it is possible to connect services like Amazon Lambda with Redis or Memcached [52]. In this sense, there are multiple research works that explore in-memory or ephemeral storage in serverless analytics [53, 54]. However, while Redis is known for its high performance and scalability, scaling Redis clusters in a serverless environment can be challenging. Besides, Redis stores data in memory, which makes it fast but also means that data is volatile and can be lost if the server crashes or restarts. While Redis offers options for persistence (such as snapshots or AOF logs), configuring and managing data persistence in a serverless environment adds complexity and potential performance overhead.

More recently, some research works turned their attention to work with durable logs as a storage substrate for serverless pipelines. For instance, Boki [55] is a seminal work that exploits the "shared log" abstraction as a storage substrate for data analytics. Moreover, event streaming systems like Apache Kafka are used in serverless analytics, but generally for notification and message passing rather than for data-intensive data transfers.

We believe that the "data stream" is the natural evolution of using queues or logs for storing data in serverless pipelines. Specifically, we refer to Pravega streams [29], as they provide multiple features and semantics which may represent a "sweet spot" for serverless pipelines:

- *Byte-oriented Streaming*: Pravega streams provide fast even IO, which can facilitate pipelining in serverless analytics. Furthermore, conversely to messaging systems like Apache Kafka and Apache Pulsar, Pravega is designed to transfer bytes, not just events. This means that we could build serverless a nalytics pipelines of large data transfers (e.g., large files, images) and still exploit pipelining. This may be inefficient if large data chunks were managed using event interfaces.

- *Storage tiering*: Pravega unifies streaming and batch data access to data streams at the infrastructure level, as it tiers cold stream data to a long-term storage system (*e.g.*, NFS, S3). This means that the same data stream API is capable to transparently access data in streaming and batch fashion. As we demonstrate in the next sections, using Pravega streams is an interesting approach to simplify the underlying storage infrastructure for serverless analytics pipelines (*i.e.*, only use data stream as the unifying data abstractions, as opposed to exposing users to a mix of objects, files, in-memory stores and other services, depending on the workload at hand).

- *Event routing*: Pravega provides consistent ordering of data events based on routing keys. We identified that we could take advantage of the event routing protocol inherent in streaming storage systems to provide and efficient shuffling operator. Concretely, we present a technique named adaptive event routing that enables the in-streaming shuffling operator. The adaptive event routing uses different hashing algorithm for the event routing keys depending on

```python
def my_map_function(obj):
    counter = {}
    data = obj.data_stream.read()

    # Store on dict "counter" the local counts of words.
    for line in data.splitlines():
        for word in line.decode('utf-8').split():
            if word not in counter:
                counter[word] = 1
            else:
                counter[word] += 1

    # Write the local word counts in a Pravega stream.
    sm = pravega_client.StreamManager(PRAVEGA_ENDPOINT)
    writer = sm.create_writer(PRAVEGA_SCOPE, PRAVEGA_STREAM, max_inflight_events=10000)
    for word in counter.keys():
        # Write local counts of words using the word as "routing key".
        # This guarantees reducers to read all occurrences of a word.
        writer.write_event(''.join([word, ',', str(counter[word])]), routing_key=word)

    writer.flush()
    return 0
```

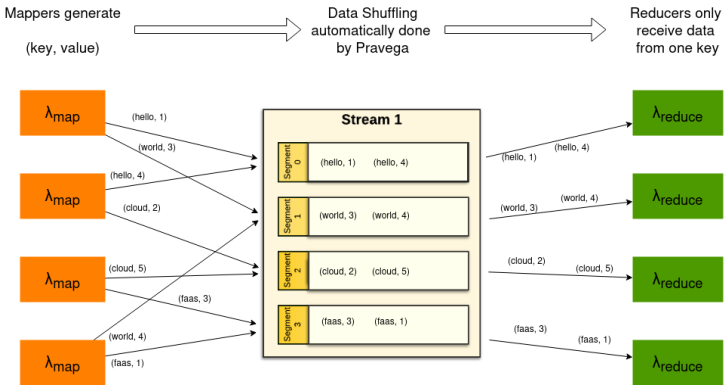

Figure 30: `Wordcount` mapper code executed with Lithops on top of Pravega streams (left). Visibly, the job exploits event routing in Pravega to transfer local word counts to the right reducer lambda without inducing extra shuffling traffic (right).

the properties that the shuffling operation must fulfill. This enables the usage of performant shuffling-dependent operations such as reduce, group-by or sort.

- *Exactly-once semantics*: Serverless analytics pipelines need a "checkpointing" mechanism to recover from failures while recovering the previous state and not missing/duplicating data. Pravega already offers transactions and checkpoints, which are the substrate to build exactly-once in stream data processing (e.g., the Pravega Flink connector already does this [14]).
- *Synchronization primitives*: A typical problem in serverless analytics is to share mutable state for synchronization purposes (e.g., counters). Pravega provides mechanisms for synchronizing state across multiple processes, such as the State Synchronizer API [56].
- *K/V Store*: Storing small, temporal state in a key/value fashion is also a valuable feature for serverless analytics pipelines. Pravega provides a K/V API on top of data streams that may satisfy the needs of many serverless analytics pipelines as an auxiliar service.

For all these properties, we identify Pravega streams as a powerful substrate to simplify storage for serverless analytics pipelines compared to state-of-the-art technologies.

5.4.2 Lithops and Pravega Integration

An example of the synergy between streaming storage and serverless analytics is the integration we have developed of Lithops and Pravega. To carry out this integration, we have used the Pravega Python client[1], given that Lithops is written in Python. The Pravega Python client allows us to package a Lithops container image with the Pravega client available, so we could use when running lambda functions in various cloud providers. This provides us a substrate to exploit the unique features of streaming storage systems for serverless analytics. An example of that is the `Wordcount` job presented in Fig. 30 and executed on AWS with Lithops and Pravega. As visible in the code, the Lithops mapper tasks get data from S3 objects containing text and perform local counts of words that are then written to a Pravega stream. One important point to notice is the way these tasks are writing data to Pravega; to wit, they are using the "routing key" parameter of the Pravega writer for writing local counts of words to the stream, and more concretely, the routing key parameter is the actual word being counted. This approach exploits the routing semantics of Pravega streams to guarantee that local words counted by mappers are consistently routed to the same stream segments. With this guarantee, we can perform reduce tasks on segments that provide the right global results for the entire dataset without incurring extra shuffling traffic, which is a well-known problem when doing shuffling in object stores [51]. In addition to that, integrating Pravega with Lithops provides

[1] `https://github.com/pravega/pravega-client-python`

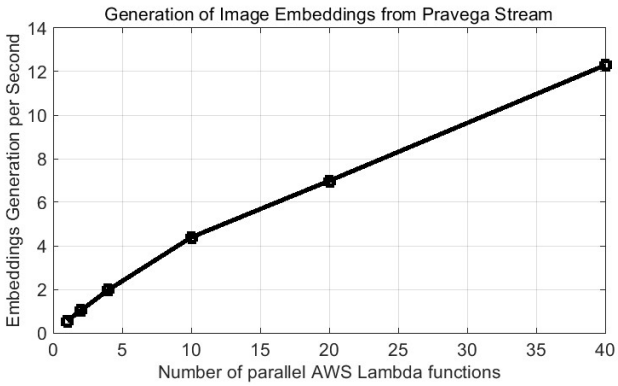

Figure 31: Generation of image embeddings on AWS in a streaming fashion with AWS Lambda and Pravega streams.

other advantages, such as having (writer) transactions and (reader) checkpoints to build exactly-once semantics on serverless processing pipelines. Moreover, the tiered architecture of Pravega, which has a low-latency write-ahead log and a scale-out long-term storage for stream data, provide a sweet spot for running both streaming and batch serverless jobs.

Next, we demonstrate some of these features via experiments on AWS.

5.4.3 Experimental Evaluation

Streaming Serverless Pipelines: Generation of Image Embeddings. In this experiment, we use AWS Lambda to read images from a Pravega stream and generate image embeddings in a streaming fashion. The sort is performed using AWS Lambda functions with 1769MB of memory (according to AWS Lambda documentation, this is equivalent to 1 vCPU). The AWS Lambda timeout has been set to 3 minutes. The throughput numbers are collected from the Lambda functions and the experiment completes when all the images of the stream are processed. The Pravega cluster consisted of 4VM with local NVMe drives (`i3.2xlarge`) and stores data on AWS S3. Each drive had a measured (sync) write throughput of 0.35GB/s. The workload consists of a benchmark VM writing 1MB images to a Pravega stream. The rate of image writes is equivalent to the processing capacity of a single AWS Lambda function running the neural model that generates embeddings (1 every 2 seconds) multiplied by the number of functions. When executing an experiment, the number of segments of the Pravega stream is equal to the number of parallel AWS Lambda functions being executed. Ideally, each AWS Lambda function will acquire a single Pravega stream segment to read data from. Once the lambdas complete the processing of image embeddings, the experiment completes. Note that we do not account for the first execution results due to the lambda cold start.

Fig. 33 shows the throughput of image embeddings generation depending on the number of AWS Lambda functions executed. As can be observed, the throughput follows a near lineal trend compared to the number of parallel lambdas. The reason for the throughput to do not grow strictly linearly with the number of lambda functions is that the distribution of stream segments across lambdas was not ideal in some cases, specially at the beginning of the experiment. Overall, this experiments demonstrates that Pravega can be used for data-intensive serverless streaming pipelines.

KPI-2 (Data Speed Improvements), KPI-3 (Resource Auto-scaling): We observe that the throughput of lambdas consuming stream data increases in a near-linear fashion with more functions and resources added to the system, which perfectly aligns with the stream auto-scaling capabilities of Pravega.

Batch Serverless Pipelines: Terasort. In this experiment, we use Lithops [57] for executing a batch Terasort job on top of Pravega streams on AWS. The sort is performed using AWS Lambda functions with 1769 MB of memory (according to AWS Lambda documentation, this is equivalent to 1 vCPU). The execution time is measured from the moment before the AWS lambdas are invoked until the sorted dataset is written to S3 and all the lambdas have finished. The S3 bucket and the AWS Lambda

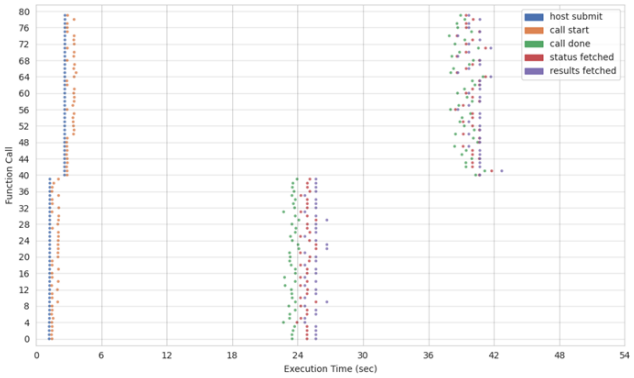

Figure 32: Generation of image embeddings on AWS in a streaming fashion with AWS Lambda and Pravega streams.

functions are in the same region (`us-east-1`). The Pravega cluster consisted of 4VM with local NVMe drives (`i3.2xlarge`). Each drive had a measured (sync) write throughput of 0.35GB/s. Terasort is a benchmark that measures the performances of shuffling operators. The benchmark consists of sorting a large dataset of randomly generated records. Each record is 100 bytes long and consists of a 10-byte key and a 90-byte value. With this benchmark we measure the time and costs it takes to sort the dataset. We used a 20 GB dataset for this benchmark. The dataset is stored unsorted in a unique file in AWS S3 and the sorted dataset must be written to a unique file in AWS S3. The sort is done with a MapReduce job using Lithops using Pravega streams for sharing data. We only use a single map and a single reduce phase. We use a Pravega stream with r segments to automatically shuffle the data. A job with m mappers and r reducers is executed as follows: i) Each AWS Lambda function of the map phase write the records to the Pravega stream using a custom routing hash algorithm. The custom routing hash algorithm is used route the records maintaining the lexicographically order. Note that each mapper reads a partition of the input file from S3; ii) Each AWS Lambda function of the reduce phase reads the records from the Pravega stream, loads the records into memory, sorts them, and writes the sorted records to the final file in S3 using the multipart upload service. We execute the experiments with different values of mappers (m) and reducers (r) to understand the impact on performance and cost.

Figure 32 shows a fine-grained view of the Terasort execution. We observe that there are 80 function calls (40 mappers, 40 reducers) to execute the Terasort with Lithops using Pravega streams. One interesting observation is that related to function pipelining. To wit, both mapper and reducer functions can start working in parallel, as the data being written by the mappers can be processed in stream fashion by the reducers. This differs with other approaches, like regular object storage access, in which mappers and reducers need to work on serial processing stages (*i.e.*, reducers need the input object resulting from mappers' computations). This is a first interesting observation of using data streams as a substrate for serverless analytics pipelines.

Another important aspect that can be highlighted in this experiment is that of data shuffling. To wit, in traditional map/reduce computing, the output of mappers should be shuffled and re-organized to serve as input for reducers, which should compute on a specific partition of the dataset. This process can be expensive in terms of object generation and associated data transfers. We realized that an inherent feature in Pravega (and other event streaming systems) called event routing can be extended to solve data shuffling with no additional data transfer overhead. For example, as this job is related to sorting elements, we modified the standard routing key algorithm in Pravega by a lexicographically ordering algorithm. This means that, considering a stream with an arbitrary number of segments, the routing algorithm will distribute elements lexicographically distributed across the stream segments. That is, a local mapper is writing to segments already with the correct global order expected in the result file. In this experiment, this enables reducers to perform a local

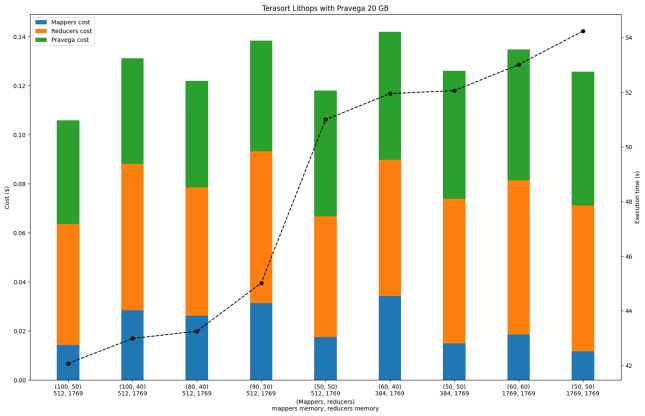

Figure 33: Cost and execution time results from running Terasort job with different combinations of mappers and reducers.

sorting task with the global section of the dataset ready to be written as output. We believe that this model can be applied to other problems using serverless analytics and Pravega streams.

It is also important to consider the abstraction of the infrastructure that Pravega streams provide to analytics job. Previously, we have described an experiment in which multiple AWS Lambda functions read images ingested into a Pravega stream and generated embeddings out of it in a streaming fashion. In this experiment, we show batch job processing data in parallel that Pravega fetches transparently from S3. For the former job Pravega provides low latency, whereas for this job it provides high throughput. And this sweet spot in the latency vs throughput trade-off is transparent to serverless analytics developers, as they only work with the Pravega Stream API.

KPI-5 (Simplicity and Productivity): Pravega can seamlessly handle streaming and batch serverless workloads with the same API, thus making it easier for developers to manage data in serverless functions compared to having to use a different storage system per workload (e.g., S3, Kinesis, etc.).

6 XtremeHub Security: Scone

SCONE can secure the XtremeHub stack of applications (CPU, memory, disk and network), by exploiting the TEE (Trusted Execution Environment) and creating an enclave in the operating system. The systems that are eligible to confidential computing will reside on SCONE prepared Docker images. Presently, Lithops and Flower ML are ported to a prepared SCONE Python image. Next steps will focus on porting other applications and establishing the attestation policies for them.

6.1 Strengthening the Security

The SCONE framework already had support for concurrent executions and sub-processes, however, applications like Lithops take a step further, by using complex programming constructs, such as using futures-like libraries. These libraries are employed in synchronization of concurrent subprocesses.

SCONE had to be improved to support it, while also strengthening the care to avoid "*race condition*", arising from two or more processes accessing the same resource, e.g. a shared variable residing in memory. This is trivial for applications that rely solely on the operating system, but require additional control mechanisms when we talk about confidential executions.

6.1.1 Enforcing the Security

A complete stack of confidential computing requires more than just running the application in an enclave protected by the TEE. In spite of that, they can be subject to malicious interference and injection of spurious data from outside.

The enforcement of confidential computing is made via CAS (Configuration and Attestation Service), a key decision maker that will only deliver secrets and configurations for trustworthy applications. The attestation workflow will measure the enclave's hash and will provide configurations related to XtremeHub interactions, storage encryption, and other configurations that will only be available to authorized applications or users, i.e., the enclave's hash has to be the same valid one configured in advance.

After correctly attested, applications will receive from CAS the corresponding "policies" of secrets and configurations they need. Policies are descriptive easily comprehensible manifests that are written (in YAML format) and uploaded onto CAS in advance. Attested applications will only run if the complete attestation workflow is finished. And this is what guarantees that the XtremeHub stack will only be contain applications and secrets allowed within its boundaries.

6.2 Lithops Experimental Execution

Non-confidential execution versus confidential execution is crucial to determine whether adversaries (say, a malicious user with superuser privileges) can extract secrets from the systems as they operate from the outside. For example, the demonstration below shows that an adversary seating inside the cloud provider and with privileged access to the server can try to eavesdrop the confidential execution, but will not obtain anything. A simple program using Lithops libraries (running in "***localhost mode***") was employed to do the assertion. Here is what an attacker can do.

- **hellolithops.py**

```
from lithops import FunctionExecutor
import time
inspected="mysecret"
def hello(name):
    return 'Hello {}!'.format(name)
while True:
    with FunctionExecutor() as fexec:
       fut = fexec.call_async(hello, 'World')
       print(fut.result())
       print(f"will sleep.  inspected="+inspected)
       time.sleep(3)
```

- **Inspect running processes in the server.** a simple `ps -ef |grep python` will bring whatever is running from the hosted containers. Then the attacker will inspect the listed processes details in */proc* and eventually find the target. For this experiment, the attacker found the eligible target to be the proccess ID **543785**. See the list below:

  - ```
    root 694 1 0 2023 ?  00:00:03 /usr/bin/python3 /usr/bin/networkd...
    root 742 1 0 2023 ?  00:00:00 /usr/bin/python3 /usr/share/unattend...
    ...
    root 403106 661223 0 13:02 ?  00:00:02 python3 backup_c...
    root 403107 2966184 0 13:02 ?  00:00:02 python3
    ...
    root 537381 537089 0 14:13 ?  00:00:01 python3
    root 542497 2966000 0 14:16 ?  00:00:00 python3 backup_c...
    root 543785 537089 95 14:17 ?  00:00:21 python3
    ```
  - An attested application will receive all the command-line arguments from the policies. The targeted process will not give out such detail in the operating system, but the actual full command-line was `python3 /hellolithops.py`. PID **543785** only has `python3`.

- **Dumping memory contents.** Linux uses a file representation to access the memory, being `/proc/$PID/maps`, for memory mapping, and `/proc/$PID/mem`, for the memory content. The attacker will get the mappings and extract the addresses contents using the `dd` command, which is available in every Unix/Linux system.

  - Example: `dd if=/proc/543785/mem bs=4096 iflag=skip_bytes,count_bytes skip=$(( 0x$ADDR_INI )) count=$(( 0x$ADDR_END - 0x$ADDR_INI )) of="543785_mem_$ADDR_INI.bin"`
  - Next, the attacker will try to extract any useful information from the dumps. In this experiment, we already know what to look for: the pieces of text "World", "inspected" and "mysecret".
  - The attacker will use a simple command `strings`, often available in Unix/Linux systems. Example: `cat 543785_mem_*.bin |strings >543785_mem_grep` and `egrep -e World -e inspected -e mysecret 543785_mem_grep`. This returns nothing. Fig. 34 depicts the actions above.

- **Inspect running processes in the server.** the same mehod for non-confidential: `ps -ef |grep python` and so on. For this experiment, the attacker found the eligible target to be the proccess ID **494873**.

  - `root 494873 473581 2 13:50 ?  00:00:00 python3 hellolithops.py`
  - One difference is that non-confidential executions have to tell the operating system the command and parameters altogether.

- **Dumping memory contents.** again, the same steps, dump using `dd`.

  - Example: `dd if=/proc/494873/mem bs=4096 iflag=skip_bytes,count_bytes skip=$(( 0x$ADDR_INI )) count=$(( 0x$ADDR_END - 0x$ADDR_INI )) of="494873_mem_$ADDR_INI.bin"`
  - Next, the extraction of useful information.
  - Example: `cat 494873_mem_*.bin |strings >494873_mem_grep` and `egrep -e World -e inspected -e mysecret 494873_mem_grep`. This time it returns a lot of content (71 lines):
  - ```
    inspected
    "mysecret"
    ```

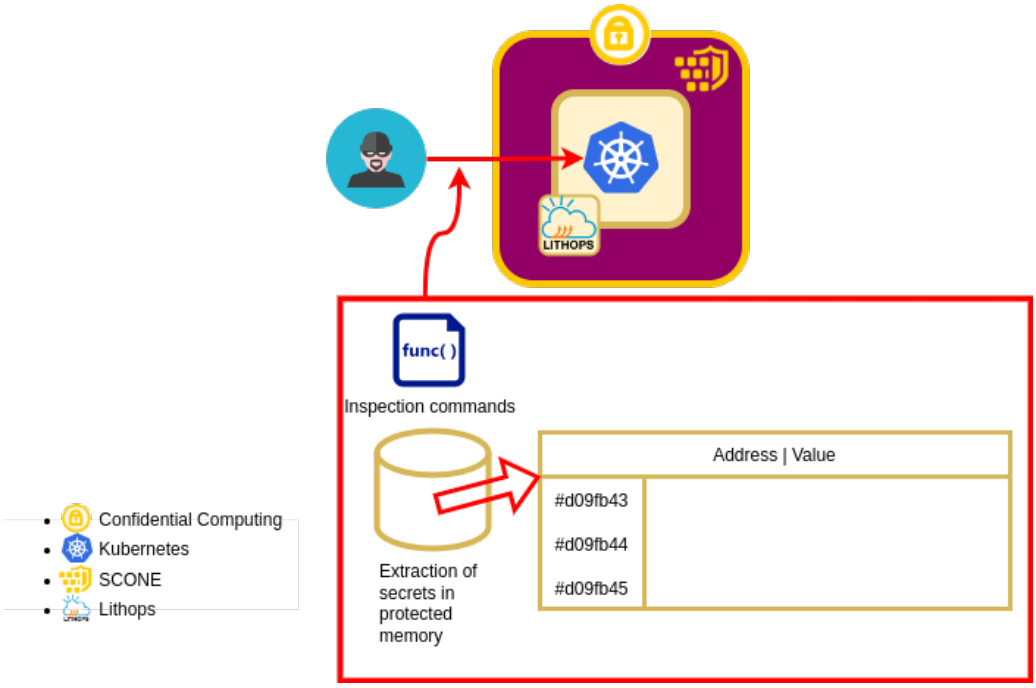

Figure 34: Attacker with high privileged access to the hosting cluster.

  - `Hello World!`
  - `b'\x80\x04\x95\x10\x00\x00\x00\x00\x00\x00\x00\x8c\x0cHello World!\x94.'`

- **In summary,** SCONE will ensure confidentiality for the execution, CAS will enforce confidential computing via attestation and policies provisioning, despite a "`root`" user residing within the premises of the cloud provider tries to inspect the XtremeHub stack. Lithops is supported in SCONE and additional work is expected to improve the integration (attestation for its "***serverless mode***" dispatching jobs to Kubernetes is under development). The porting of Pravega is expected to occur next.

7 XtremeHub Connectors

In this section, we specifically focus on a couple of Data Connectors for XtremeHub that target to optimize use case workloads. On the one hand, we describe Dataplug and evaluate its integration with the Lithops serverless data processing platform. On the other hand, we overview the progress of leveraging High-Performance Computing connectors onto our targeted use-cases.

7.1 Lithops as a compute engine for Dataplug

First of all, we will give a brief overview of the serverless data connector that we present in NEARDATA known as Dataplug. In deliverable *D2.2 NEARDATA Architecture Specs and Early Prototypes* the reader will find an extensive description of this component.

Dataplug Design Recap. Dataplug is an extensible Python framework that implements the on-the-fly data partitioning model. The goal of Dataplug is to provide extensible tools to deal with extreme data that presented major challenges to ingest and partition. Dataplug hides the complexities of unstructured scientific data pre-processing and partitioning, offering researchers a data-driven pre-processing and dynamic on-the-fly partitioning strategies for diverse unstructured scientific data formats such as, FASTA, FASTQGZIP, and VCF for genomic data, LIDAR for geospatial data, and imzML for metabolomics data. Dataplug enables efficient parallel access to existing unstructured data blobs in their original scientific format. Dataplug is serverless and is portable for use with different distributed data analysis clusters or serverless framework (Dask, Ray, PySpark, Lithops, ...) on any Cloud (AWS, Google Cloud, ...) or on-premise, without the need to manage servers for pre-processing jobs.

The data model offered by Dataplug is divided into two phases:

- Cloud-aware pre-processing and indexing: This pre-processing method is format-specific, and extracts metadata from the raw data blob, such as internal content structure, indices, and attributes.
- Dynamic on-the-fly partitioning: This phase is responsible for generating Data Slices that encapsulate partition metadata (such as byte ranges) and code. These are distributed to remote workers to be evaluated from HTTP GET requests of byte ranges to obtain the desired partition.

Lithops serverless analytical data processing platform offers the ability to run massively parallel jobs on any of the major cloud providers. The simplicity of connection between the object storage and the computing backends allows Dataplug to execute its data model phases on the different cloud services. In NEARDATA we observe a potential integration between both components due to the simplicity of connection between them. There is no need to implement a new architecture design on Dataplug since this framework includes a specific library to use Lithops as a compute backend.

In the following, we will present a couple of Dataplug benchmarks using Lithops as computational backend.

7.1.1 Experimental evaluation

This experiment focuses on the evaluation of the dynamic partitioning offered by Dataplug on the FASTQGZip compressed genomic data format.

For this purpose, the two phases that form the Dataplug data model will be analyzed in detail. The two experiments can also be differentiated:

Pre-processing comparison between Cloud-aware and static partitioing approaches: In this experiment, we want to compare the time required to generate static partitions and the time required to apply Cloud-aware indexing for different data volumes of FASTQGZip data. To generate static partitions, we use a script that employs the aws CLI, zcat, and split commands. This script reads a compressed file from S3, splits it into partitions with a specified number of lines, and writes the partitions back to S3 in a pipe stream. For the Cloud-aware pre-processing approach with Dataplug, we employ the FASTQGZip pre-processing method predefined in the framework. Our pre-processing reads the input file from S3 in a stream and generates a GZip index using gztool. This index can be

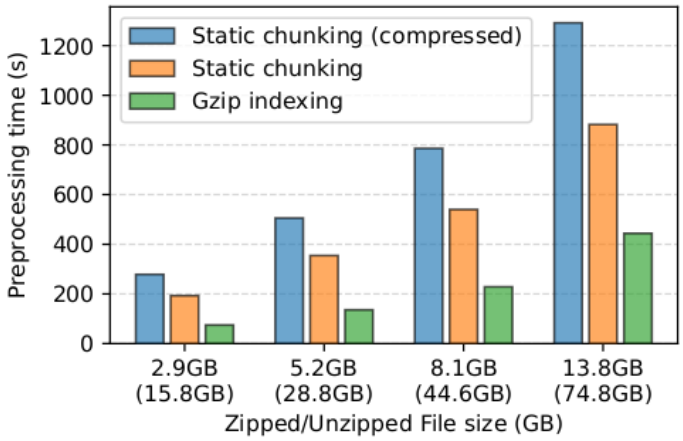

Figure 35: Comparison of pre-processing time to generate static partitions and Cloud-aware indexing for different input FASTQGZip data sizes.

then used to query the GZip file to retrieve specific chunks and decompress them on-the-fly. For both experiments, we used a t2.xlarge instance on AWS EC2.

Figure 35 shows the execution times for pre-processing different input FASTQGZip file sizes using both static partitioning and Cloud-aware indexing approaches. The results demonstrate that Cloud-aware indexing of GZip files is up to ×2.9 faster than generating static partitions. While both approaches involve reading and decompressing the entire GZip stream, generating static partitions requires an additional step of writing the chunked data back to storage, which increases the pre-processing time. This effect is further amplified if the partitions are compressed before being uploaded to storage. However, employing Cloud-ware FASTQGZip implies an extra cost in metadata storage, as we have to account for the storage of the GZip index file. Even so, on-the-fly partitioning reduces transfer by approximately 200% by avoiding completely downloading the data to be partitioned and writing the resulting partitions back to storage.

On-the-fly partitioning overhead: We measure the time taken to fetch a partition of 850MB using different methods: static FASTQGZip compressed chunks, static FASTQ decompressed chunks, and on-the-fly partitioning with Cloud-aware FASTQGZip. The partitions are retrieved from serverless functions in AWS Lambda using Lithops, with a memory configuration of 2048MB each.

Figure 36 shows the results of the experiment. We can observe that fetching a partition with Cloud-aware on-the-fly partitioning and decompressing it significantly lowers the average fetch time (5.81s) compared to obtaining the static decompressed partition (21.72s). We can also say that obtaining a partition with on-the-fly partitioning is x3.7 faster than with static partitioning. On the other hand, the fetch time for static compressed partitions is similar in both cases. This difference in fetch time can be attributed to the fact that it is faster to download a smaller compressed payload from object storage and decompress it in memory compared to downloading a larger volume of already decompressed data.

KPI-1 (Throughput Improvements and data transfer reduction): From the two experimental evaluations we can validate that Dataplug offers improvements in data throughput when preprocessing data, avoiding static data preprocessing from metadata extraction and thus ensuring data transfer reduction.

7.2 HPC Data Connectors

This section describes the progress on an HPC-compute layer for our XtremeHub. We describe an HPC data connector to leverage High-Performance Computing platforms onto our targeted use-cases. That is, to allow use-cases to use supercomputing facilities, and most particularly MareNostrum supercomputer hosted by BSC-CNS. This is a challenging activity given the use-cases of the

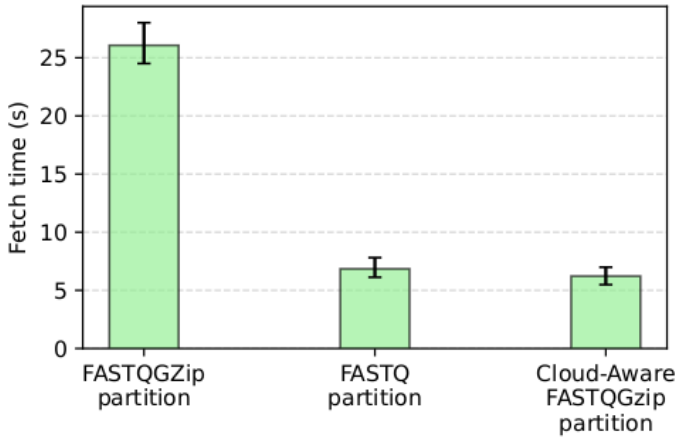

Figure 36: Comparison between fetch time for statically generated compressed and uncompressed FASTQ partitions and Cloud-aware FASTQGZip on-the-fly partitioning.

project are implemented utilizing software typically found in cloud environments (Apache Spark, Lithops, Pravega) and not typically available in a supercomputer. While this grants flexibility, cloud environments computing capabilities are less-performance oriented than those found in supercomputers. Cloud is designed with flexibility in mind and has more relaxed quality of service. While supercomputers are designed in terms of compute power so they require static resources allocations while providing high-performant network, compute and storage capabilities to guarantee a high amount of compute power.

In this section we describe the work done so far regarding enabling our MDR use-case into MPI and current initial results found.

7.2.1 HPC Data Connector for the MDR use-case

The human genome reference sequence comprises a chain of over 3 billion nucleotide pairs. Comparing DNA sequences between individuals reveals up to 3.78 million differences, termed genomics variants. Understanding genomics variation is vital for comprehending disease predisposition. Thus, a primary aim of Computational Genomics is identifying disease-associated variants. Complex diseases like Type 2 Diabetes, asthma, and Alzheimer's disease result from the combined effects of multiple genomics variants and environmental factors. Despite tools facilitating variant interaction discovery and impact assessment, analyzing variant interactions remains a computational and methodological challenge. Even in the simplest approach, analyzing pairwise interactions yields over 10^{12} combinations, posing significant computational demands.

Leveraging High-Performance Computing facilities (i.e, MareNostrum IV and V) we can address this problem using a combination of Machine Learning methods and parallel computing. This combination is what we define as an HPC data connector. Initially is being developed for the Multifactor Dimensionality Reduction (MDR) method to detect variant pairs associated with Type 2 Diabetes (T2D). However, we envision this data connector as something that can be expanded to other use-cases in the omics sciences.

KPI-1 (Data Throughput Improvements): figure 37 demonstrates the computational performance of our approach. Initial results show that our use-case MDR implemented with MPI achieves a processing rate of 1311 combinations per second per core. In this scenario, leveraging all available cores in the MareNostrum 4 cluster (totaling 165.888 CPUs), it could compute the entire set of combinations (6^{13}) within 3.2 days. Furthermore, preliminary testing indicates that utilizing the enhanced architecture offered by Marenostrum 5 enables a processing rate of 3057 combinations per second per core. Consequently, with the same number of CPUs as in MareNostrum 4, all combinations could be computed in approximately 1.4 days. Compared to our previous cloud-based solution with Apache

Spark, using MPI improves performance 5x times using the MareNostrum 4 supercomputer and the new supercomputer improves it even further.

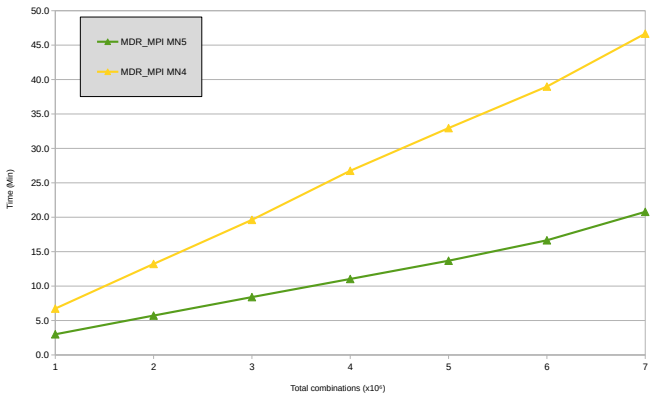

Figure 37: Computational Performance Comparison of MDR with MPI on MareNostrum 4 and MareNostrum 5 Clusters.

Due to patient information confidentiality, public datasets for such experiments are unavailable, and most private datasets require licensing. Therefore, we developed a custom script to simulate a dataset with similar complexity to real datasets, like those presented by the 70KforT2D project. This dataset adheres to the Variant Call Format (VCF) standard and so does our HPC data connector.

We have described further details of both the described use-case and the HPC data connector in *Deliverable 5.1: First release of KPI benchmarks in all use cases and data connector libraries*. In the mentioned deliverable there are further details on how we are adapting as well this HPC data connector to leverage GPU resources, which can particularly accelerate machine learning models and allow us to perform faster.

The outcomes outlined above underscore the potential of the MareNostrum architecture and MPI model in handling substantial computational workloads. Building upon the initial iteration of our use-cases, we are in the process of designing a standardized data connector. This innovative connector enables the seamless distribution of large databases across multiple machines, optimizing efficiency and scalability.

Currently the benchmarks have been performed on a rather small amount of resources. This is due to access a broader amount of resources available in the supercomputer special access requests must be made through the grant of a RES [58] or even a PRACE [59] proposal. The difference among them resides on the amount of resources that can be granted. The PRACE grants the most, however, typically such large grants are made with splitting granted resources across supercomputers of the network, instead of having them all in a single one. Thus, requiring to adapt the workload to run in different environments and then combine the results. An application for a RES has been already made and awaiting acceptance to carry on with large-scale experiments. Depending on the outcome of this initial large-scale experiments we may apply for a PRACE to expand the testbed even further. Nonetheless, current results are already representative as one can observe the extreme linearity of the experiments. Consequently we can extrapolate how those results will look like on a larger set of computational nodes.

KPI-3 (Demonstrated resource auto-scaling for batch processing): in line with our ongoing efforts, we are dedicated to improving the computational performance of NEARDATA use-cases, and most particularly for our specific study case. This entails optimizing data structures and streamlining certain functions to increase the number of combinations computed per core per second. Additionally, we aim to evaluate the scalability of our approach by examining its performance with a broader scope, such as analyzing $3x3$ combinations, which presents a significant increase in computational complexity as well as in terms of scientific validation, both of them are in progress.

8 Conclusions and Next Steps

In this deliverable, we have presented the data plane component of NEARDATA: the XtremeHub. First, we have provided an overview of Lithops and a performance evaluation of serverless workloads on AWS (T3.1). Second, we have described Pravega and compared its performance with popular messaging systems, such as Apache Kafka and Apache Pulsar. We have also introduced a new notion of storage-compute auto-scaling, in which Pravega elastic streams are scaled in coordination with processing engines (T3.2). In this sense, we have also evaluated the byte API of Pravega for storage-oriented workloads (*e.g.*, GStreamer, large transfers) and its integration with Lithops for running serverless workloads on top of Pravega streams (T3.1). In third place, we have briefly introduced Scone and how it integrates with Lithops to perform confidential serverless computations. We conclude the technical content of this deliverable by focusing on some Data Connectors: i) we introduced Dataplug, which provides advanced indexing and partitioning capabilities to Lithops (T3.5); ii) we reviewed the current progress on HPC Data Connectors (T3.4). A summary of the KPIs for XtremeHub components is illustrated in Table 3.

Component	KPI	Results
Lithops	KPI-3	Lithops incorporates an extensible storage and compute backend architecture that enables elastic and scalable cloud solutions to be designed according to the resources needed to run the workload.
Pravega	KPI-1	The Pravega writer achieves good write performance compared to the Kafka producer (*e.g.*, 73% higher for 1 segment) while guaranteeing data durability.
	KPI-2, KPI-3	Dynamic batching allows writers to achieve an excellent balance between latency and throughput without forcing the user to decide the performance configuration of the writer ahead of time.
	KPI-1	Pravega shows the highest throughput (350MBps) compared to Kafka (330MBps) and Pulsar (250MBps) when using multiple segments.
	KPI-2	The Pravega reader achieves both low end-to-end latency and high throughput compared to Kafka and Pulsar for the cases tested. Pravega is virtually insensitive to the distribution of routing keys, as opposed to the other systems.
	KPI-1	Pravega is the only system that achieves consistent throughput for the number of producers and segments tested, while guaranteeing event ordering and data durability. Also, it efficiently exploits drive throughput irrespective of the degree of parallelism.
	KPI-1	Pravega achieves much higher historical read throughput compared to Pulsar (up to 7x) without user intervention or complicated configuration settings.
	KPI-3, KPI-5	Pravega is the first streaming storage system that provides elastic streams: data streams that are automatically re-partitioned according to the ingestion load and the scaling policy.
	KPI-5	Administrators can reason about the storage-compute auto-scaling of the streaming pipeline considering just two main parameters (*i.e.*, CPU threshold, stream auto-scaling rate).
	KPI-3	The orchestrator component allows us to auto-scaling streaming analytics jobs automatically based on elastic Pravega stream parallelism.
	KPI-1, KPI-3	Our storage-compute auto-scaling mechanism improves load balancing and reduces shuffling transfers across task managers automatically, whereas static provisioning may lead to problems under dynamic workloads.
	KPI-1	Pravega Byte API can achieve higher throughput for large data transfers compared to the Kafka event API (*e.g.*, 3x improvement for 100MB events).
	KPI-2	Pravega Byte API achieves lower end-to-end latency (p95 between 42.3% and 176.1% lower) for GStreamer byte buffer streaming compared to AI inference latency.
	KPI-2, KPI-3	The throughput of lambdas consuming Pravega stream data increases in a near-linear fashion with more functions and resources added to the system.
	KPI-5	Pravega can seamlessly handle streaming and batch serverless workloads with the same API, thus making it easier for developers to manage data in serverless functions compared to having to use a different storage system per workload (*e.g.*, S3, Kinesis, etc.).
Scone	KPI-4	SCONE supports confidential execution aided by TEE (hardware enabler) of both Lithops and Flower ML, two systems developed in Python. Flower ML can run attested as a standalone application. Lithops attestation for its serverless mode is a work in progress.
Connectors	KPI-1	Introducing Dataplug as a dynamic data partitioner offers data transfer reduction (200%) and data throughput improvements in preprocessing tasks (x2,9 faster in FASTQGZip indexing and x3,7 faster in FASTQGZip fetching partitions). MDR use-case has applied the HPC Data Connector and shows an speed-up improve of 5x times respect cloud-based version (Apache Spark).
	KPI-3	We intend to explore the integration of the HPC Data Connector into the Lithops framework and allow for maximum scalability enhancing the connector's capabilities.

Table 3: Highlights of main KPIs achieved by XtremeHub components.

The consortium has achieved great progress so far, not only in the execution of work-package tasks, but also in the integration of XtremeHub components. But we remain ambitions in what is

yet to be done in the near future. For instance, we expect further integration among XtremeHub components and exploit them on our use cases. We also expect to develop a wider variety of data connectors that allow us to address different goals (*e.g.*, special formats, data reduction, etc.). Last but not least, we will also work on making XtremeHub more adaptive to workload and usage patterns.

References

[1] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," ACM SIGOPS operating systems review, vol. 47, no. 1, pp. 9–15, 2013.

[2] "Apache bookkeeper." https://bookkeeper.apache.org, 2023.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in USENIX ATC'10, vol. 8, 2010.

[4] "Apache zookeeper." https://zookeeper.apache.org, 2023.

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

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

[7] "Openmessaging benchmark." https://github.com/openmessaging/benchmark, 2024.

[8] "Pravega." https://cncf.pravega.io, 2023.

[9] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck, "Sdgen: Mimicking datasets for content generation in storage benchmarks," in USENIX FAST'15, pp. 317–330, 2015.

[10] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. Epema, "Analysis and modeling of time-correlated failures in large-scale distributed systems," in IEEE/ACM International Conference on Grid Computing, pp. 65–72, 2010.

[11] G. H. Gonnet, "Expected length of the longest probe sequence in hash code searching," Journal of the ACM, vol. 28, no. 2, pp. 289–304, 1981.

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

[13] 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.

[14] "Pravega - flink connector." https://github.com/pravega/flink-connector, 2023.

[15] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," arXiv preprint arXiv:1506.08603, 2015.

[16] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," ACM Transactions on Computer Systems (TOCS), vol. 3, no. 1, pp. 63–75, 1985.

[17] "Apache flink jira - flink-6390." https://issues.apache.org/jira/browse/FLINK-6390, 2023.

[18] "Pravega - working with pravega: Transactions." https://cncf.pravega.io/docs/latest/transactions, 2023.

[19] "Wikipedia - two-phase commit protocol." https://en.wikipedia.org/wiki/Two-phase_commit_protocol, 2023.

[20] "Apache flink jira - flink-7210." https://issues.apache.org/jira/browse/FLINK-7210, 2023.

[21] Apache Flink, "Apache flink javadoc - twophasecommitsinkfunction." https://nightlies.apache.org/flink/flink-docs-release-1.17/api/java/org/apache/flink/streaming/api/functions/sink/TwoPhaseCommitSinkFunction.html, 2023.

[22] "Kubernetes - horizontal pod autoscaling." https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale, 2023.

[23] "Prometheus." https://prometheus.io/, 2023.

[24] Dell Technologies, "Dell emc isilon h600 hybrid nas storage." https://www.dell.com/en-uk/dt/storage/isilon/isilon-h600-hybrid-nas-storage.ht, 2023.

[25] R. Metzger, "Flink blog - scaling flink automatically with reactive mode." https://flink.apache.org/2021/05/06/scaling-flink-automatically-with-reactive-mode/, 2021.

[26] "Uk health state agency data." https://www.ncbi.nlm.nih.gov/bioproject/248064, 2023.

[27] K. Gontarska, M. Geldenhuys, D. Scheinert, P. Wiesner, A. Polze, and L. Thamsen, "Evaluation of load prediction techniques for distributed stream processing," in IEEE IC2E'21, pp. 91–98, 2021.

[28] "Gstreamer." https://www.nct-heidelberg.de/en/the-nct.html, 2024.

[29] R. Gracia-Tinedo, F. Junqueira, T. Kaitchuck, and S. Joshi, "Pravega: A tiered storage system for data streams," in ACM/IFIP Middleware'23, pp. 165–177, 2023.

[30] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 12, no. 3, pp. 463–492, 1990.

[31] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in Distributed Computing: 16th International Conference, DISC 2002 Toulouse, France, October 28–30, 2002 Proceedings 16, pp. 265–279, Springer, 2002.

[32] J. D. Valois, "Lock-free linked lists using compare-and-swap," in Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pp. 214–222, 1995.

[33] "Pravega - segment attributes." https://cncf.pravega.io/blog/2019/11/21/segment-attributes/, 2024.

[34] "Pravega - gstreamer connector." https://github.com/pravega/gstreamer-pravega, 2024.

[35] "Amazon elastic kubernetes service." https://aws.amazon.com/en/eks/, 2024.

[36] "Amazon ec2 i3en instances." https://aws.amazon.com/es/ec2/instance-types/i3en, 2024.

[37] "Fio's documentation." https://fio.readthedocs.io/en/latest/, 2024.

[38] "National centre for tumor diseases (nct) in heidelberg." https://gstreamer.freedesktop.org/, 2024.

[39] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," ACM Computing Surveys (CSUR), vol. 54, no. 10s, pp. 1–34, 2022.

[40] 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, pp. 445–451, 2017.

[41] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al., "Cloud programming simplified: A berkeley view on serverless computing," arXiv preprint arXiv:1902.03383, 2019.

[42] "Amazon lambda." https://aws.amazon.com/en/lambda/, 2024.

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

[44] "Azure functions." https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview?pivots=programming-language-csharp, 2024.

[45] "Amazon s3." https://aws.amazon.com/eN/s3/, 2024.

[46] "Google cloud storage." https://cloud.google.com/storage, 2024.

[47] "Azure blob storage." https://azure.microsoft.com/en-us/products/storage/blobs, 2024.

[48] 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, pp. 263–274, 2018.

[49] 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), pp. 363–376, 2017.

[50] J. Sampé, M. Sánchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez, "Outsourcing data processing jobs with lithops," IEEE Transactions on Cloud Computing, vol. 11, no. 1, pp. 1026–1037, 2021.

[51] M. Sánchez-Artigas and G. T. Eizaguirre, "A seer knows best: Optimized object storage shuffling for serverless analytics," in Proceedings of the 23rd ACM/IFIP International Middleware Conference, pp. 148–160, 2022.

[52] "Serverless development with aws lambda and redis enterprise cloud." https://redis.com/blog/serverless-development-with-aws-lambda-and-redis-enterprise-cloud/, 2024.

[53] 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), pp. 427–444, 2018.

[54] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding ephemeral storage for serverless analytics," in 2018 USENIX annual technical conference (USENIX ATC 18), pp. 789–794, 2018.

[55] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp. 691–707, 2021.

[56] "Pravega - state synchronizer javadoc." https://cncf.pravega.io/docs/latest/javadoc/clients/io/pravega/client/state/StateSynchronizer.html, 2023.

[57] "Lithops." https://lithops-cloud.github.io/, 2024.

[58] "Red española de supercomputación (res)." https://www.res.es/es/acceso-a-la-res/, 2024.

[59] "Partnership for advanced computing in europe (prace)." https://prace-ri.eu/hpc-access/calls-for-proposals/, 2024.