# Nexus: A Data Management Mesh for Tiered Data Streams

*Raúl Gracia-Tinedo*
*Dell Technologies*

*Hossam Elghamry*
*Dell Technologies*

## Abstract

Modern event streaming systems (*e.g.*, Apache Kafka, Apache Pulsar) provide support for storage tiering: the ability to move cold stream data to an external storage service (*e.g.*, AWS S3). In this paper, we identify that there is an opportunity for exploiting advanced data management on data streams during the storage tiering process. To this end, we present `Nexus`: a data management mesh that transparently mediates storage operations between streaming systems and external storage. The main abstraction in `Nexus` is the *streamlet*: a data management function —*e.g.*, processing, routing, caching— executed inline on a chunk of tiered stream data. Administrators can deploy streamlets across heterogeneous infrastructures and orchestrate them via policies. We demonstrate `Nexus` via stateless and stateful streamlets. We show that `Nexus` achieves up to 3.9× higher data reduction than built-in compression in streaming systems, supports privacy-aware data routing (reducing metadata reads by 98%), and allows for complex health-related data management pipelines. All this is achieved transparently across multiple streaming systems (Kafka, Pulsar, Pravega) and incurring low interception overhead ($< 5\%$).

## 1 Introduction

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

In the last decade, multiple event streaming systems have emerged to expose the data stream abstraction. For example, systems like Apache Kafka [4], Apache Pulsar [6], Redpanda [18], or Pravega [16] implement streams with key guarantees, such as durability, event order, and high performance, among others. Event streaming systems also provide connectors for analytics engines —*e.g.*, Apache Flink [3], Apache Spark [56]— to become a powerful data sink/source in production-grade stream processing pipelines [9, 23].

More recently, however, the industry raised the need of storing stream data for the long term and in a cost-effective manner (*e.g.*, batch analytics, AI training, auditing) [26–28]. Augmenting event streaming systems with long-term storage capabilities is a major challenge, with profound design implications in some cases [47]. And yet, in recent years, we have witnessed most event streaming systems delivering *storage tiering* [7, 19]: the ability to move cold stream data to an external storage service (*e.g.*, AWS S3). Storage tiering for data streams allows the system to achieve a sweet spot in the infrastructure costs and performance trade-off [36].

### 1.1 Motivation: Beyond Tiered Data Streams

In general, the storage tiering process for data streams is quite simple: it consists of moving chunks[1] of stream data to/from external storage via APIs. When a chunk —*e.g.*, a few to 100s of MBs— of stream data is considered cold, a server of the event streaming system triggers a storage operation to offload it to external storage. Once stored remotely, chunks of stream data can be safely removed from the high-performance, expensive local drives used to ingest events with low latency.

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

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

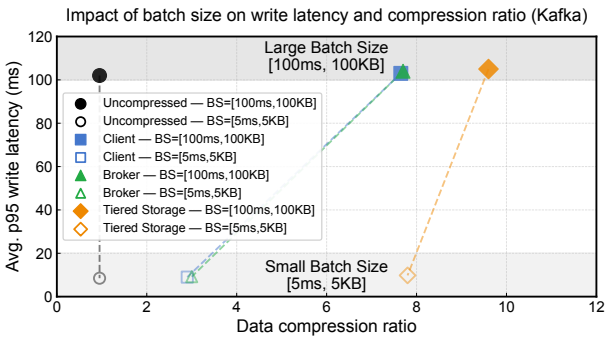


Figure 1: Impact of batch size (BS) on write latency and compression ratio enforcing data compression at different points of the Kafka write path (100e/s, 1KB events).

*data event granularity* with low latency, whereas data management would be executed offline on *stream data chunks*.

A simple example of the opportunity we explore in this paper is illustrated in Fig. 1 (details in §7). In Fig. 1, we enforce data compression at different points of a Kafka stream write path: i) *client*, ii) *broker*, and iii) *storage tiering* (broker). Concretely, we evaluate the impact of event batch size on compression ratio and write latency. Visibly, when compressing data at the event level (client, broker), there is a tension between compression ratio and write latency. This is natural, as larger event batches provide better compression opportunities at the cost of waiting for longer to accumulate more data. Interestingly, by applying compression on chunks of tiered stream data, we keep event write latency unaffected, as compression occurs outside the hot write path. We also achieve 2.6× better compression ratios for small batch sizes compared to compressing event batches at the client or broker.

A key observation of this paper is that managing chunks of tiered stream data can provide benefits beyond compression. However, in the experiments above, we used the data compression functionality built in the storage tiering binding itself [20]. As event streaming brokers are already complex systems, attempting to build a general-purpose data management framework on top of a storage tiering binding may be inflexible and hard to maintain. This calls to rethink data management for tiered data streams as an independent service.

## 1.2 Tiered Stream Management Principles

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

We aim to realize this vision guided by the following principles: i) *Decoupling*: Event streaming and data management should be decoupled concerns. A strong reason supporting this decision is that event streaming systems are performance-sensitive and complex already. Thus, adding advanced data management capabilities to them could negatively impact their maintainability and performance. ii) *Transparency*: Ideally, we should perform data management on tiered data streams without the event streaming system noticing. Addressing this challenge would also benefit not just one, but multiple event streaming systems. iii) *Extensibility*: Data management is a quite broad term [46]. As such, we should provide users with means to develop a variety of data management functions for optimizing storage flows related to tiered data streams. iv) *Heterogeneity*: As event streaming systems may be deployed at the Edge and/or the Cloud, a data management solution should also be aware of such heterogeneous infrastructures.

## 1.3 Contributions

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

- This is the first work addressing the data management gap between streaming systems and external stores.
- Design of `Nexus`, a system with streamlets, swarmlets, and policies for composable and location-aware in-transit data management of tiered data streams.
- Implementation of `Nexus` and multiple streamlets to evaluate the benefits on event streaming systems like Kafka, Pulsar, and Pravega across Edge/Cloud deployments.

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

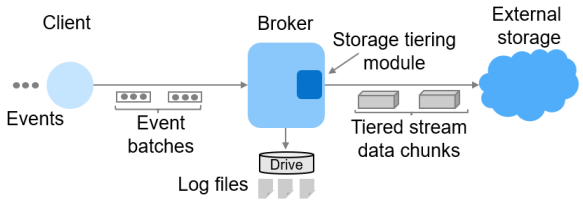


Figure 2: Architecture of an event streaming system.

## 2 Background

### 2.1 Event Streaming Systems

Event streaming systems, such as Apache Kafka [4, 43], Apache Pulsar [6], Redpanda [18], and Pravega [16], have emerged as a popular technology for ingesting and processing data in a streaming fashion. These systems expose the *data stream* (*a.k.a.*, topic) abstraction for managing data events. Internally, data streams are composed of *partitions* or *segments*, which can be parallelized for higher throughput. Most systems provide support for durability and exactly-once semantics in data streams, as well as low-latency data ingestion.

The general architecture of event streaming systems consists of producers that write data events to streams, and consumers that read and process these events (see Fig. 2). Brokers act as intermediaries, managing the ingestion, storage, and delivery of events. Each broker persists stream data to local log files on high-performance storage devices to ensure durability and low-latency access [41]. Data is typically organized into append-only logs, segmented by partitions, and replicated across brokers for fault tolerance.

### 2.2 The Shift towards Streaming Storage

Due to the growing storage demands of streaming use cases, event streaming systems have adopted *tiered storage* as a key feature [7, 13, 19] (see Fig. 2). Tiered storage consists of offloading cold stream data from high-performance drives (*e.g.*, SSDs, NVMes) to scale-out storage (*e.g.*, object storage). This results in a sweet spot in the latency-throughput trade-off while reducing storage costs, making it a crucial feature for organizations to efficiently manage and store stream data [36]. Recently, storage tiering is also relevant in diskless event streaming systems (*e.g., AutoMQ [8], WarpStream [25]*), which largely rely on external storage instead of local drives.

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

## 3 Nexus Design

This section presents the design of `Nexus`.

### 3.1 Design Principles and Insights

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

First, `Nexus` is designed considering that event streaming and data management should be clearly *decoupled* concerns. A key observation supporting this design decision is that event streaming systems work at the *data event* level, whereas tiered data streams are managed at the *chunk* level. Such data granularities are totally different both in terms of size (*i.e.*, 10-1000x size difference) and processing latency expectations (*i.e.*, milliseconds vs minutes). In this sense, we realize that most event streaming systems adopt standard APIs for managing tiered stream data from external storage services (*e.g.*, AWS S3, Azure) [7, 24]. `Nexus` design exploits this insight by transparently *intercepting* storage operations of tiered data streams. In addition to favoring a decoupled data management layer, storage API interception makes `Nexus` broadly applicable to event streaming systems with tiered storage.

Second, data management is a broad term with many types of applications [46]. As such, `Nexus` is designed to deploy and execute *user-defined data management functions* for optimizing storage operations related to tiered data streams [37, 49] (see §4.2). While some data management functions are expected to be stateless, `Nexus` design also considers supporting stateful functions. Function state may be required for storing data required for an algorithm or passing a partial event stored in previous chunk across executions, for example. In contrast to prior work [32], we do not attempt to keep globally consistent metadata shared across functions. As a novel contribution, `Nexus` exploits the sequential structure of data streams for routing data across stateful functions and linearizing function state management via consistent hashing. This relaxes the consistency requirements on shared metadata [48] (see §4.3).

Finally, event streaming systems are increasingly deployed at the Edge, or in Edge/Cloud environments [29, 33], for latency, cost, and privacy reasons. However, such a heterogeneous infrastructure also needs to be considered at the data management level. `Nexus` allows users to deploy data management functions on *specific locations* and *compose* their execution across the Cloud/Edge infrastructures via simple

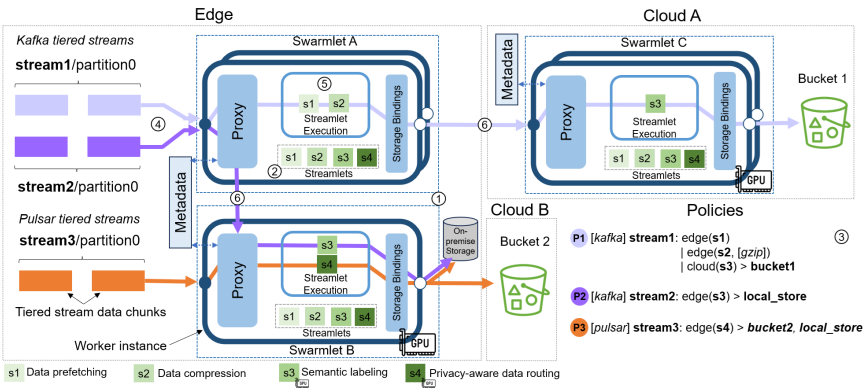


Figure 3: An architecture overview and operation of Nexus.

*policies*. Note that some functions may require specialized hardware for their execution (*e.g.*, GPU, TEE). Nexus tracks the specialized hardware available in worker instances, as well as execution requirements for transparently executing functions and routing data to the right place (see §4.4).

## 3.2 Abstractions

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

**Streamlet**: A streamlet is a function executed inline on a storage request related to a chunk of tiered stream data. Streamlets are reactive: they are invoked upon the interception of storage operations. The streamlet concept is broad enough to address diverse data management requirements on data streams. We classify the streamlets into four categories: *transformers*, which modify data contents (*e.g.*, data compression in §7.3); *performance*, which aim to improve the IO performance of the event streaming system (*e.g.*, data buffering in §7.4); *routing*, which change the default destination of stream data (*e.g.*, data routing in §7.3); and *semantic*, which analyze the contents of stream data chunks and react to them (*e.g.*, AI-based metadata labeling in §7.4).

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

**Policy**: Administrators use policies for configuring the execution of streamlets in Nexus. Policies can define the execution of a single streamlet or multiple ones composed in an execution pipeline [45]. Moreover, policies allow administrators to determine the location for a streamlet execution, as streamlets may need to be executed at different locations across a Cloud/Edge environment. In Nexus, a policy has the following elements: i) *system type*, which helps Nexus to identify actual chunks of stream data from other system-dependent metadata files also stored externally; ii) *target*, which defines the relevant stream(s) for the policy at hand; iii) *streamlet pipeline*, which defines the streamlet(s) to be executed, as well as the location and order of execution; iv) *output storage*, which is the eventual location of stream data chunks once executed the streamlet pipeline. Each streamlet may contain parameters for configuring its runtime behavior.

# 4 Nexus Architecture

Nexus runs on one or multiple clusters. Fig. 3 shows an overview of its architecture and operation. Nexus worker instances are deployed as swarmlets (step ①, Fig. 3) based on their infrastructure location (Edge, Cloud) and hardware resources (*e.g.*, GPU, TEE). Administrators can then deploy streamlets (step ②, Fig. 3) by loading binaries and setting up their metadata descriptor that describes key aspects for their execution, like hardware requirements. With swarmlets and streamlets in place, an administrator defines policies to orchestrate the execution of streamlets on data streams (step ③, Fig. 3). The metadata of streamlets, swarmlets, and policies is stored in the metadata store (see §4.1).

Event streaming systems can be configured to offload chunks of tiered stream data against a swarmlet service endpoint. Nexus worker instances act as a proxy by implementing standard APIs (*e.g.*, AWS S3) for transparently intercepting storage operations from the event streaming system viewpoint (step ④, Fig. 3). Moreover, worker instances take care of the execution of streamlets (step ⑤, Fig. 3). As visible in Fig. 3, Nexus executes streamlet pipelines enforcing the streamlet location specified by the policy. This is possible thanks to Nexus' mesh-like data routing (step ⑥, Fig. 3).

## 4.1 System Metadata

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

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

**Streamlet metadata**. Streamlets interact with metadata at multiple levels. First, an administrator installing a streamlet provides a *descriptor* specifying its main features. This includes information about if the streamlet should be executed on writes and/or reads (on=[PUT, GET, ALL]), the streamlet type (*e.g.*, type=transformer), if it is stateful (stateful=[true, false]), or if it has any hardware requirements to be executed (*e.g.*, hardware=gpu). This descriptor helps Nexus to execute streamlets correctly and in the right location via data routing (see §4.4).

## 4.2 Streamlet Execution

Streamlets are runnable function binaries that are executed in the execution engine of worker instances. Streamlet binaries can be dynamically distributed and loaded onto swarmlets. The container image of worker instances also allows packaging third-party libraries that could be invoked by streamlets [50]. Next, we provide details on streamlet execution:

**Containerized execution**. Nexus provides container level isolation for streamlets running on the same worker instance. Streamlets scheduled on the same worker instance implies that they run in the same container and can access the shared system metadata. This approach is reasonable as Nexus can be administered by a single organization, which differs from public FaaS platforms [55]. The execution engine allows to pipeline several streamlets on a single storage request, based on the policy definition. To illustrate this, policy P1 in Fig. 3 shows the pipelined execution of streamlets s1 (prefetching) and s2 (compression) in a worker instance at the Edge.

**Storing outputs**. Once its execution completes, a terminal streamlet in a pipeline stores the processed data chunk on the external storage defined in the policy. Nexus provides a storage binding module for abstracting the streamlets from the different storage API implementations. Normally, a streamlet pipeline ends by storing stream data chunks in a single storage service (P1 and P2 in Fig. 3). Still, streamlets that route data may receive as input several storage service endpoints to store data based on the computation's output. For instance, P3 in Fig. 3 routes data to a local object store or to a S3 bucket depending on whether it infers some potential privacy issue in the contents of stream data chunks.

## 4.3 Streamlet State

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

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

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

However, in this setting, managing the streamlet state can be expensive if done naively. For the sake of illustration, let us assume a data routing streamlet that analyzes stream data chunks and stores them in an alternative bucket based on the computation outcome. Moreover, it keeps track of chunks stored in the alternative bucket via a persistent map to serve GET requests. As Nexus assumes no control over load balancers, data chunks are randomly distributed and processed by streamlet instances across different worker nodes for the same stream. Therefore, streamlet instances concurrently manage the same shared map. To keep updates to the shared map consistent, we could resort to existing synchronization primitives in metadata services. But, as visible in a set of micro benchmarks on Redis (Fig. 4), both optimistic concurrency and locking mechanisms exhibit significant contention as the number of concurrent updates grow. Similarly, consistent metadata stores proposed in the serverless literature also incur overhead in keeping shared data structures consistent [32].

Instead, to enable efficient streamlet state management, we build on the observation that data stream partitions are *parallel* (*i.e.*, there is no dependency among them) and *sequential* (*i.e.*, both the data arrival and tiering process keep order). Therefore, Nexus can linearize metadata updates on a per-partition basis to avoid metadata synchronization on stateful streamlet data structures working on a stream [38]. First, within the same worker node, storage requests for a given

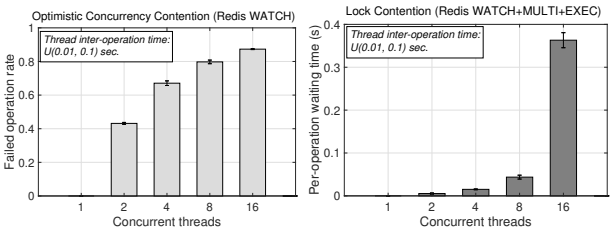


Figure 4: Redis micro-benchmarks to measure contention in concurrent updates on shared data structures (optimistic concurrency, locks). The time between checking the server state and performing the update is random $\in [0.01, 0.1]$ secs.

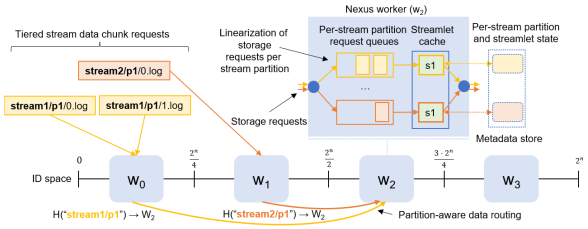


Figure 5: Example of partition-aware data routing.

stream partition are processed in order of arrival. Second, for a given stream partition, `Nexus` creates individual streamlet data structures in the metadata store that are managed by the same streamlet instance. As depicted in Fig. 5, this has two benefits: i) streamlet metadata updates do not require synchronization, and ii) on `GET`s, streamlets have a fresh view of their metadata, which reduces accesses to the metadata store.

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

## 4.4 Mesh-like Data Routing

Event streaming systems have limited flexibility for configuring the storage tiering endpoint [5, 7]. For this reason, `Nexus` handles the complexity of routing stream data chunks for building streamlet pipelines. Moreover, `Nexus` assumes no control over the service load balancer. Thus, the proxy component in the `Nexus` worker instance is responsible for intercepting and routing storage requests across the system. In `Nexus`, data routing considers direct data transfers across swarmlets, which differs from typical indirect communication in serverless pipelines [50, 55]. `Nexus` offers two routing types:

**Partition-aware data routing**: `Nexus` employs a deterministic data routing protocol to assign stream partitions to specific worker instances, thereby ensuring consistent execution of stateful streamlets. Let $S$ be a stream composed of parallel partitions $P = \{p_1, p_2, \ldots, p_i\}$ and $W = \{w_1, w_2, \ldots, w_j\}$ be the set of available worker instances within a swarmlet. `Nexus` partitions the ID space $[0, 2^n - 1]$ into $m$ contiguous intervals ($m \geq j$), each assigned to a worker instance. Given a stream partition identifier $p_x$, a consistent hash function $H : P \rightarrow [0, 2^n - 1]$ is used to compute its position in the ID space [42]. The storage request is forwarded to worker $w_z$ responsible for the interval containing $H(p_x)$. Once in $w_z$, the storage request will be processed by the expected streamlets. An example of such a process can be seen in Fig. 5.

**Location and hardware-based data routing**: `Nexus` also routes stream data chunks based on policy location and hardware. First, an administrator explicitly defines in a policy where a streamlet should be executed (Edge, Cloud). The reason is that some streamlets are more effective when executed on a specific infrastructure (*e.g.*, caching data at the Edge). Based on policy metadata, `Nexus` routes storage operations to the swarmlets available by the policy location. Moreover, `Nexus` implicitly re-routes operations within a given location to meet a streamlet's hardware requirements. To illustrate this, `P2` in Fig. 3 defines a policy in which streamlet `s3` is set for execution at the `edge` and requires a GPU. For a Kafka cluster, `swarmlet A` is defined as the storage tiering endpoint. However, `swarmlet A` does not have GPU-powered worker instances. In this situation, workers in `swarmlet A` route storage operations to `swarmlet B`, which has worker instances with GPUs. This provides flexibility to `Nexus` deployments without impacting event streaming systems configuration.

## 4.5 Fault Tolerance and Correctness

`Nexus` executes streamlets synchronously and inline as part of the storage request path. A client request is acknowledged only after all streamlets specified by the active policy complete successfully and the object is durably stored. Execution is stream-oriented and chained across swarmlets: each `Nexus` instance processes its local streamlets and forwards output to the next swarmlet, forming a sequence of open requests that completes only at the final storage write.

`Nexus` assumes fail-stop failures (crashes, execution errors, network timeouts). Byzantine failures are out of scope. `Nexus` handles: i) Worker failures are detected via partition ownership validation and failed requests trigger client retries routed to the new owner (§4.4); ii) Network, streamlet, metadata service, or object store failures break the request chain, causing immediate failure and retries; and iii) Policy mismatches across swarmlets detected via version tracking.

`Nexus` provides *at-least-once execution semantics* for both data processing and metadata updates. No additional correctness requirements are imposed beyond this execution model. While a request is acknowledged only after full pipeline completion, acknowledgment-loss windows may occur [39]: a downstream streamlet may successfully store the object and update metadata, but the upstream acknowledgment fails, causing the pipeline to re-execute on retry. Common stateful streamlets (*e.g.*, data routing and labeling) naturally fit this

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

Table 1: Summary of Nexus Streamlet APIs

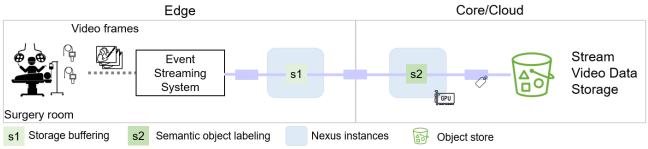


Figure 6: Surgiomics challenges addressed with Nexus.

execution model. Streamlets that require duplicate-free state updates can implement idempotent streamlet logic (*e.g.*, using object identifiers to deduplicate retries). Nexus avoids distributed transactions across streamlets to preserve scalability.

## 5 Nexus in Action

Next, we overview the Nexus API from a developer viewpoint for addressing health-related data management pipelines.

### 5.1 Streamlet API

Developers extend Nexus by implementing streamlets that process stream data chunks. Nexus supports two streamlet types: ByteStreamlet and EventStreamlet (see Table 1).

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

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

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

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

### 5.2 Health Stream Management Use-cases

We demonstrate Nexus via two use cases: genomics and surgiomics. First, the UK Health Security Agency (UKHSA) leverages genomics to enhance public health by identifying and monitoring pathogens. Second, the National Center for Tumor diseases (NCT) focuses on endoscopic multimedia analytics to improve surgical outcomes (see Fig. 6).

**Genomics streaming data ingestion**. Emerging use cases like real-time pathogen genomics monitoring during outbreaks motivate the exploration of stream processing in genomics [53]. UKHSA is exploring Pravega's byte-oriented streams [35] for ingesting FASTQ files, aiming to enable streaming analytics while maintaining efficient, tiered storage for batch processing. The goal is to allow FASTQ data to be accessed both via Pravega and directly from object storage.

We developed a ByteStreamlet that performs GZip data compression on subsets of DNA sequences of an input stream data chunk (*e.g.*, 100K DNA sequences). Each compressed subset of sequences is written as an individual GZip member. Moreover, the streamlet stores the start offset of each GZip member as an object tag. This allows batch jobs to perform parallel access on compressed chunks of tiered stream FASTQ data [31], while regular streaming reads remain unaffected.

**Buffering under storage unavailability**. Computer-assisted surgery is an increasingly important domain for improving the success of surgical procedures. To store multimedia in a streaming fashion and perform real-time analytics, NCT is using Pravega streams to ingest video and image data. Unfortunately, Pravega exhibits limitations upon unavailability of long-term storage. To wit, it can only keep ingesting data events until its in-memory cache is full of data waiting to be offloaded [36]. During a surgery, this may imply that video ingestion and data analytics are impacted until, which is not acceptable. To address external storage unavailability, we implemented a byte streamlet that durably buffers incoming data chunks. This allows Pravega to continue ingesting data uninterrupted. The streamlet implements the DataSourceStreamlet API, serving data from local storage or redirecting GET requests to the object store as needed.

**Semantic stream data annotations**. To help NCT scientists locate relevant fragments of video/image streams, some type of semantic indexing is needed. Hence, we developed an EventStreamlet that integrates NCT AI models and leverages a custom Deserializer to extract JPEG images from Pravega stream chunks. Upon processing all or a subset of images within a chunk, the streamlet annotates the corresponding data object with semantic tags (*e.g.*, surgical phase, instruments). This metadata facilitates efficient retrieval of stream fragments containing relevant visual content.

## 6 Implementation

We have implemented Nexus as a middleware for the S3Proxy project [21], an extensible S3-compatible proxy. The middleware intercepts S3 requests and processes them based on system metadata (*e.g.*, swarmlets, streamlets, and policies). From the client's perspective, Nexus behaves as a standard S3 endpoint: the client receives a response only if the request is successfully stored or fails. Internally, however, Nexus uses asynchronous programming to concurrently process requests in a streaming fashion. The implementation spans over 7K lines of Java code, including a CLI for managing metadata and scripts for deployment and benchmarking. Redis [17] serves as the metadata backend, with local caching for read-only access. Upon updates (*e.g.*, new policies), Redis notifies the Nexus metadata service to apply changes. Nexus also supports dynamic compilation and runtime loading of streamlets and deserializers. The code is available at [14].

## 7 Validation

We evaluate Nexus focusing on: i) Can Nexus provide high performance storage request interception and routing (§7.2)? ii) What benefits may Nexus transparently bring to existing event streaming systems (§7.3)? iii) Can Nexus handle complex, real-world data management pipelines (§7.4)?

### 7.1 Experimental Setup

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

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

*Apache Kafka*: Apache Kafka [4, 43] is a distributed event streaming platform optimized for high-throughput, low-latency data processing via a publish-subscribe model. It structures data into topics, partitions, and offsets, ensuring scalability and fault tolerance through replication. For tiered storage, introduced in KPI-405 [13], we use Aiven's Kafka storage tiering plugin [20].

*Apache Pulsar*: Apache Pulsar [6] is a scalable, low-latency event streaming system. Its tiered storage offloads sealed log segments from high-performance storage (*e.g.*, Apache BookKeeper [2, 41]) to cost-efficient long-term storage, reducing costs while preserving seamless access to historical data.

*CNCF Pravega*: Pravega [16, 36] is a distributed stream storage system for unbounded, high-throughput data. Pravega streams support dynamic scaling, exactly-once semantics, and long-term retention. Its tiered storage integrates low-latency short-term storage via Apache BookKeeper with cost-efficient long-term storage like cloud object stores.

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

*FIO*: FIO [11] is a powerful benchmarking tool used to evaluate the performance of storage systems, including object storage services. It supports various workload types (*e.g.*, sequential and random read/write operations, mixed I/O patterns). We use FIO S3 support to evaluate Nexus interception performance in isolation (*i.e.*, no streaming system involved).

*OpenMessaging Benchmark*: OpenMessaging Benchmark [15] is a comprehensive toolset designed for benchmarking messaging systems in the cloud. It supports systems like Apache Kafka, Apache Pulsar, Pravega, and more.

**Datasets**. We used the following datasets for building stream data payloads in our experiments[2]: i) *Data compression*: For data compression experiments, we used traces from HDFS from a collection of system logs available at [54, 59]. ii) *Semantic data routing*: For image payloads, we used images from ImageNet [34] and Cholec80 [52]. iii) *Genomic data*: For the genomics data management experiment, we use FASTQ files available from SRA Toolkit [22].

### 7.2 Interception Performance

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

**IO interception performance**. We execute a r/w workload of objects against S3 via FIO as a performance baseline. Then, we reproduce the same workload when a single Nexus worker intercepts storage requests without processing and when it executes a no-op streamlet. Fig. 7 shows the results for various levels of benchmark parallelism and object sizes.

First, the main observation in Fig. 7 is that Nexus does not induce a significant performance penalty when intercepting object storage requests. Generally, the throughput reduction that FIO reports when Nexus intercepts requests falls below 5%. Nexus worst performance cases compared to the baseline relate to one benchmark thread and extreme object sizes (1MB and 1000MB). Overall, a single Nexus instance can execute a no-op streamlet from multiple client requests at $\approx$ 730MBps, demonstrating the feasibility of the current implementation.

To better understand Nexus interception overheads, we provide a latency breakdown for 1MB chunks via local microbenchmarks (excluding network effects) comparing executing PUTs against Nexus and the object store directly. We found that the Nexus HTTP proxy and policy metadata checks add 1.51ms on average (1.54ms at p95) to a request. Then,

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

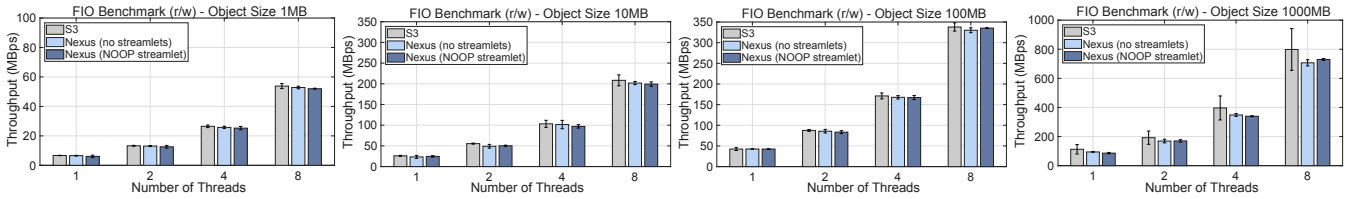


Figure 7: Interception performance of 1 Nexus instance under FIO read/write workloads on AWS.

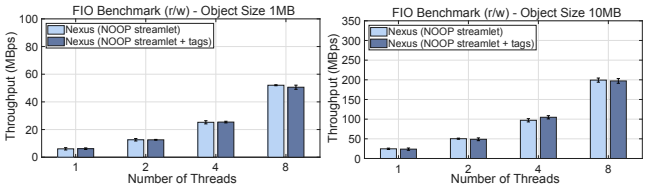


Figure 8: Performance impact of leveraging object tags using 1 Nexus instance under FIO read/write workloads on AWS.

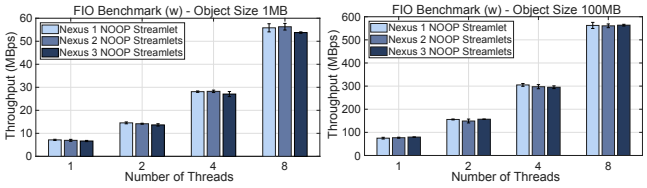


Figure 9: Processing performance depending on pipeline size of 1 Nexus instance under FIO write workloads on AWS.

building the streamlet pipeline and processing the request via a no-op streamlet takes 1.04ms on average (1.50ms at p95). Experiments with small objects (1B), where CPU costs are negligible, show similar results. In these experiments, a Nexus worker used on average ≈ 560MB of RAM and 0.011 vCPUs when idle. For sequential executions of a no-op streamlet (1MB chunks), 0.69 vCPUs were used with the same RAM usage. These results pave the way for future optimizations.

**Object metadata**. Nexus exploits object metadata to allow streamlets annotating objects, as well as to store specific system information (*e.g.*, if the object was processed by a transformer streamlet). To inspect the impact of using object tags, Fig. 8 shows the throughput comparison of a no-op streamlet versus a no-op streamlet that stores and retrieves an object tag for PUTs and GETs, respectively.

Fig. 8 shows that the throughput reduction related to managing object tags is generally minor. Specifically, the worst-case throughput reduction observed is 2.7% and 3.2% for 1MB and 10MB objects, respectively. A key reason for minimizing the performance impact of managing object tags is that these are asynchronous operations. For instance, upon an object PUT, object tags are stored asynchronously once the actual object has been correctly stored (see §4.3), thus not blocking the original request. This experiment demonstrates that Nexus can effectively manage object metadata, which can be

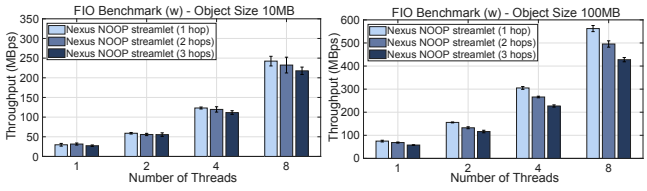


Figure 10: Performance impact of data routing hops under FIO write workloads on AWS.

| Failure | Failure point | Object stored | Metadata stored | Observed semantics |
|---|---|---|---|---|
| Worker crash | Swarmlet S1 (pre-store) | No | No | Retry, no side effects |
| Worker crash | Swarmlet S2 (pre-store) | No | No | Retry, no side effects |
| Streamlet error | Swarmlet S2 | No | No | Retry, no side effects |
| Ack loss | S1 ack fails (post-store) | Yes | Yes (default) | At-least-once |

Table 2: Summary of Nexus failure tests.

leveraged for both system and user purposes.

**Streamlet pipelining overhead**. Next, we evaluate the performance of Nexus intercepting object storage requests with a different number of pipelined functions in the same worker instance. To this end, we create policies pipelining up to 3 no-op streamlets for PUT requests (see Fig. 9).

Fig. 9 shows that Nexus can successfully pipeline multiple functions as a chain. Visibly, each additional no-op streamlet can induce an additional 1% to 6% of throughput reduction, depending on the case. In line with prior observations, the 3-streamlet case with the lowest levels of parallelism exhibits the most pronounced performance drop compared to the baseline (up to 6.8%). Overall, this experiment demonstrates the practicality of building complex streamlet pipelines in Nexus.

**Data routing overhead**. Nexus can route object requests transparently to event streaming systems based on the policies and streamlets configured (see §4.4). We evaluate the performance impact of executing no-op streamlets across 1 to 3 Nexus worker instances (see Fig. 10).

In the more typical 2-hop pipeline scenario (*e.g.*, Cloud-Edge deployments), Fig. 10 shows that the throughput reductions due to routing across swarmlets ranges from 6.2% to 14.8%, depending on object size and benchmark parallelism. We also assess a more demanding 3-hop configuration in Nexus. In this case, we observe a higher performance penalty, with throughput reductions reaching up to 25.6%, particularly for larger objects. Despite the added routing complexity, these results confirm that Nexus maintains acceptable performance levels and preserves transparency from the event streaming

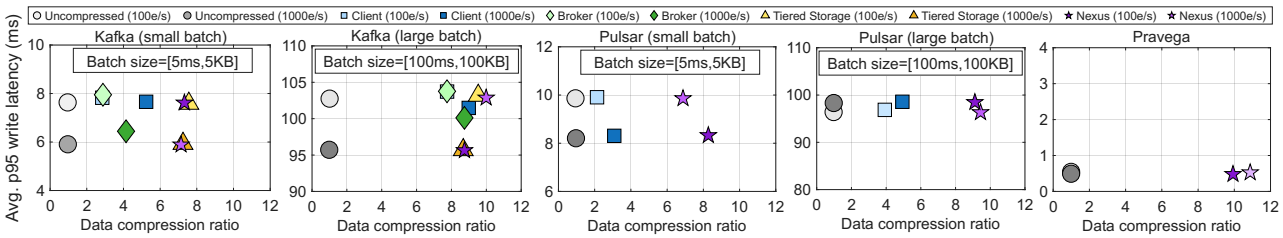


Figure 11: Data compression vs write latency of event streaming systems data compression and Nexus (1KB events).

system's perspective, which is a key design goal.

**Failure tolerance**. To validate the failure model of §4.5, we inject fail-stop failures at different points in a two-swarmlet pipeline and summarize the results in Table 2. Visibly, failures before the terminal storage stage (*e.g.*, worker crashes or pre-store streamlet errors) do not materialize any persistent state. In contrast, failures after durable storage (*e.g.*, acknowledgment loss) may lead to at-least-once semantics due to retries. This behavior stems from Nexus's fail-stop assumption and its reliance on client retries to recover from broken execution chains. Still, as state and metadata are only committed after successful local execution, idempotency can be enforced in the streamlet logic (*e.g.*, use identifiers to prevent duplicates).

## 7.3 Enhancing Event Streaming Systems

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

**Infrastructure abstraction**. We perform two experiments in this section: *data compression* and *semantic data routing* (see Fig. 12). For the first experiment, we deploy two Nexus swarmlets logically associated to Edge and Cloud regions, where data compression takes place at the Edge. Second, as may happen in real environments, the Edge cluster is expanded with GPU-enabled nodes. Then, we create a new swarmlet in the Edge region and implemented a semantic data routing streamlet pipeline that requires GPU. The crucial insight is that such infrastructure expansion is transparent to the event streaming system, as Nexus re-routes storage requests across swarmlets based on hardware requirements. Therefore, Nexus allows administrators to manage the infrastructure and policies without requiring changing the (limited) tiered storage configuration of event streaming systems.

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

Fig. 11 shows how different compression models —client-side, broker-side, tiered storage (broker), and Nexus— affect

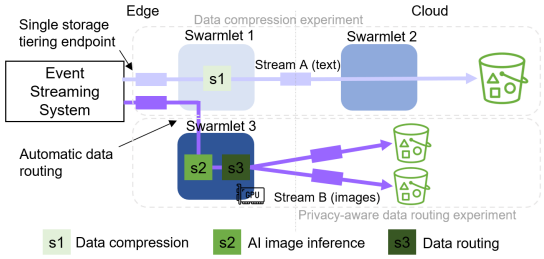


Figure 12: Experiment setup showing Nexus capabilities to abstract infrastructure from event streaming systems.

compression ratio and write latency across event streaming systems. Nexus achieves the highest compression ratios (*e.g.*, up to 10.4× in Pravega, 9.9× in Kafka, and 8.3× in Pulsar) while maintaining low write latencies (similar to compressing data at the tiered storage module in the Kafka broker). Nexus achieves compression ratios up to 3.9× higher than the best client- or broker-side compression in Kafka, and over 3.8× higher in Pulsar (Pravega has no compression implemented).

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

**Semantic data routing**. We instantiated a new swarmlet (swarmlet 3) in our on-premises infrastructure deployed on nodes equipped with GPUs (see Fig. 12). We run a workload with OpenMessaging Benchmark in which the benchmark writes images to Kafka containing humans or not, switching randomly every 2MBs. Then, we configured a two-streamlet pipeline at the Edge: i) a semantic human detection streamlet via Yolov5 [40] model (s2), and ii) a data routing streamlet (s3) that stores a data chunk on a private bucket if any of the processed images belongs to a human. Note that s2 is a stateful streamlet (see §4.3), meaning that it keeps a @Persistent map of the data chunks stored in the private bucket.

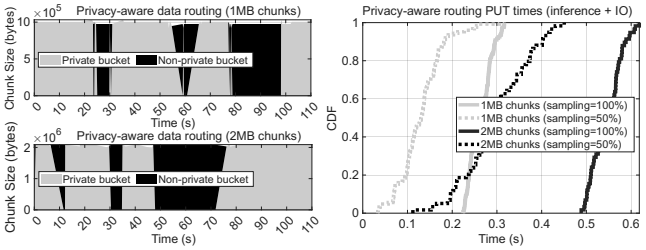


Figure 13: Storage bucket for stream data chunks decided by the semantic streamlet (left) and inference times based on chunk size and event sampling rate (right).

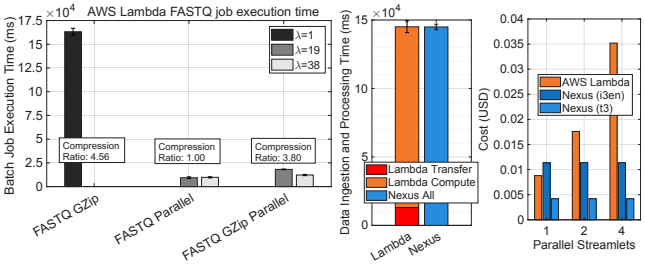


Figure 14: Compression ratio and execution time of an AWS Lambda batch job on FASTQ data stored via our `FASTQGzipParallel` streamlet vs common baselines (left). We also compare our `FASTQGzipParallel` streamlet logic upon ingestion executed in `Nexus` and AWS Lambda (right).

Fig. 13 (left) shows a time-series with the semantic decisions made by the streamlet pipeline. Visibly, `Nexus` stores stream data chunks to the right bucket based on the result of AI inference (with 1MB and 2MB chunks). Furthermore, while `s2` is an `EventStreamlet` that processes the internal content of a chunk event by event (see §5), its implementation allows the user to perform AI inference on a sample of chunk events. Fig. 13 (right) shows `s2`'s processing time depending on the image sampling rate and the chunk size. Finally, we tested `s2` by using the proposed partitioned approach vs a shared approach in which any streamlet instance can manage the streamlet state. In this sense, while the number of metadata writes is similar as any update needs to be persisted in metadata, we found that the partition-aware data routing reduced by $\approx 98\%$ metadata read requests in our experiment. The reason is that the same streamlet instance was in charge of managing metadata for a given stream partition thanks to `Nexus` partition-aware data routing (§4.4), thus avoiding having to load the most recent metadata on every request.

## 7.4 Real-World Use Cases

Finally, we demonstrate `Nexus` by addressing data management challenges in health-related use cases (§5.2).

**Genomics use case**. We compare a simple AWS Lambda job (*i.e.*, DNA sequence count) working on FASTQ files ingested as Pravega streams and processed via our streamlet with two common baselines: FASTQ Gzip and plain FASTQ. We also compare data ingestion in AWS Lambda and `Nexus`.

Fig. 14 (left) shows that the compression ratio ($4.56\times$) for the FASTQ Gzip file is the highest, as the whole file is processed by the GZip compressor. However, compressing a FASTQ file as a single unit prevents multiple lambdas from accessing it in parallel (*i.e.*, $\lambda = 1$). Besides, the plain FASTQ file can be processed much faster in parallel ($\approx 9.4$ seconds), but with no data compression. Our streamlet provides a compelling trade-off between compression and processing time compared to the tested counterparts ($3.8\times$ compression ratio, 12.1 seconds of processing time for $\lambda = 38$), while being totally transparent to the event streaming system. This allows UKHSA to access FASTQ data either through Pravega streams or directly as chunks of tiered stream data.

For data ingestion, we executed the same code both in AWS Lambda triggered by events and in `Nexus` with the same resources in both cases (2.5CPUs, 4GB of RAM). Visibly, AWS Lambda needs to transfer the FASTQ file first (*e.g.*, via `curl`) and then run the code, whereas `Nexus` does it inline. In terms of performance, the `FASTQGzipParallel` runs similarly in both systems ($\approx$ 146s). To compare costs (Fig. 14, right), we account for the cost of a `Nexus` worker instance proportionally to the resources used during the streamlet execution. Visibly, `Nexus` can be either cheaper or more expensive than AWS Lambda depending on the underlying VM flavor (*e.g.*, $2\times$ lower cost on a `t3` instance, higher on `i3en`). Importantly, `Nexus` workers are long-lived services whose cost can be amortized across concurrent streamlets, whereas AWS Lambda follows a pay-per-invocation model. Fig. 14 (right) also shows the cost amortization of running various streamlets on a `Nexus` worker. Finally, executing the streamlet at the Edge reduces transfers by $3.8\times$ compared to AWS Lambda. We conclude that `Nexus` provides a suitable deployment and cost model given the continuous and parallel nature of data management workloads generated by tiered data streams.

**Surgiomics use case**. Next, we evaluate the streamlets developed for NCT, starting with the buffering streamlet designed to provide temporary Edge-side storage and mask long-term storage unavailability in Pravega. Fig. 15 (top) compares Pravega's behavior with and without our streamlet during induced unavailability of its backing object store (MinIO). Pravega is configured with only 1GB of cache to accelerate the issue manifestation. Visibly, ingestion throughput (10MBps) halts after $\approx$ 110s when MinIO becomes unavailable. In contrast, our buffering streamlet absorbs the data during the outage and asynchronously uploads it once MinIO is restored, effectively masking the disruption. This streamlet adds value when relying on Pravega streams to ingest and perform AI on real-time surgical multimedia.

Moreover, the NCT pipeline runs AI models on streamlets to annotate data with semantic tags. Fig. 15 (bottom) shows

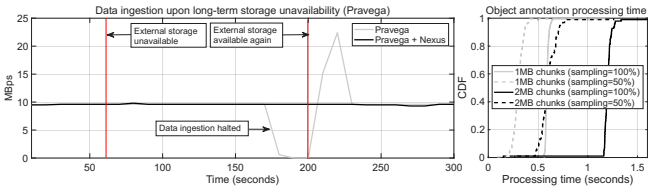


Figure 15: NCT storage buffering and annotation (instrument detection AI model) streamlets on Pravega streams.

the performance of an AI model that detects surgical instruments in JPEG images and tags corresponding stream chunks. `PUT` request latency includes interception, deserialization, inference, and tagging. Chunk size and image sampling rate significantly impact tagging performance (*e.g.*, p50 from 0.3 to 1.2 seconds). These annotations help NCT data scientists quickly locate relevant video fragments within object buckets.

## 8 Related Work

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

**Data streaming and data management**. Event streaming systems such as Apache Kafka [4], Apache Pulsar [6], Redpanda [18], and Pravega [36] have evolved to support high-throughput, low-latency ingestion and durable storage of data streams [47]. Recent advancements have introduced tiered storage mechanisms that offload cold data to external object stores [7, 13, 19], enabling unified access for both streaming and batch analytics. Unfortunately, these systems provide limited support for in-transit data management during tiering. To our knowledge, only Aiven's storage tiering module for Kafka [20] offers built-in support for data compression and caching on chunks of tiered stream data. However, extending such tiering modules, which are deployed on top of event streaming brokers, introduces additional complexity and operational burden. Instead, `Nexus` proposes a programmable data management layer that operates on tiered stream data independently of the streaming system's hot path. `Nexus` may also be applicable to diskless streaming systems [8], as it operates at the object storage interface rather than being coupled with broker internals. A natural integration point in systems like AutoMQ is the background compaction stage that persists data to object storage, mirroring today's `Nexus` interception.

**Software-defined storage**. `Nexus` builds on software-defined storage (SDS) concepts, such as policies and processing stages that shape storage behavior. IOFlow [51] pioneered SDS with end-to-end I/O policies via queuing abstractions. Crystal [37] extended SDS to object stores using filters and a centralized controller for dynamic policy enforcement. Retro [44] generalized SDS for multi-tenant systems with resource-agnostic policies. While these systems focus on internal control and resource management, `Nexus` targets programmable data management over tiered stream data and operates by intercepting and processing storage operations across Edge/Cloud infrastructures.

**Data-intensive serverless systems**. Recent serverless computing research explores data-aware function orchestration, akin to `Nexus` streamlets. Sonic [45] enables application-aware data passing across chained functions, while Sion [58] introduces elasticity and locality-aware scheduling for cloud storage. Middleware systems [49] and bucket-trigger models [55] propose data-driven execution for object stores. AWS Lambda (+ Events Notifications) [30] and S3 Object Lambda [1] enable serverless data transformations on object storage: the former asynchronously upon object PUT events, the latter at GET read time. Both lack native stateful orchestration and are confined to Cloud execution. In contrast, `Nexus` executes streamlets synchronously inline during the tiering process, supports stateful pipelines via partition-aware routing, and can span across Edge/Cloud infrastructures.

**Data and service meshes**. `Nexus` borrows ideas from data and service mesh systems to route storage requests to the right swarmlets for enforcing policy execution. While industry platforms such as Confluent's data mesh framework [10] adopt data mesh principles for decentralized governance and scalable data sharing, they operate primarily at the metadata and access control layer. In contrast, service meshes such as Istio [12] focus on network traffic routing and observability across heterogeneous infrastructures. Emerging data mesh platforms such as NebulaStream [57] aim to unify data management across heterogeneous infrastructures, especially in IoT and Edge/Cloud environments. None of these systems support programmable inline execution on tiered stream data, nor partition-aware stateful routing that leverages the sequential structure of data streams to eliminate metadata synchronization (both are novel contributions of `Nexus`).

## 9 Conclusions

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

## Acknowledgments

We thank the Cloud Native Computing Foundation (CNCF) for sponsoring the Pravega project. We thank Alan Cueva and Gerard Finol for their help in this work. We also thank the reviewers for their insightful comments on this paper. This work has received funding from the European Union Horizon Europe research and innovation programme under grants 101092644 (NEARDATA), 101092646 (CLOUDSKIN), and SIXG (101291424) under the Smart Networks and Services Joint Undertaking (SNS JU).

## References

[1] Amazon s3 object lambda. https://aws.amazon.com/s3/features/object-lambda. Accessed: July 2026.

[2] Apache bookkeeper. https://bookkeeper.apache.org. Accessed: July 2026.

[3] Apache flink. https://flink.apache.org. Accessed: July 2026.

[4] Apache kafka. https://kafka.apache.org. Accessed: July 2026.

[5] Apache kafka - documentation. https://kafka.apache.org/documentation. Accessed: July 2026.

[6] Apache pulsar. https://pulsar.apache.org. Accessed: July 2026.

[7] Apache pulsar - overview of tiered storage. https://pulsar.apache.org/docs/3.3.x/tiered-storage-overview/. Accessed: July 2026.

[8] Automq. https://www.automq.com. Accessed: July 2026.

[9] Confluent. https://www.confluent.io/. Accessed: July 2026.

[10] Confluent blog - an introduction to data mesh. https://www.confluent.io/blog/benefits-of-data-mesh-and-how-to-get-started. Accessed: July 2026.

[11] fio - flexible i/o tester. https://fio.readthedocs.io. Accessed: July 2026.

[12] Istio.io. https://istio.io/. Accessed: July 2026.

[13] Kip-405: Kafka tiered storage. https://cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage. Accessed: July 2026.

[14] Nexus github repository. https://github.com/neardata-eu/nexus. Accessed: July 2026.

[15] Openmessaging benchmark. https://github.com/openmessaging/benchmark. Accessed: July 2026.

[16] Pravega. https://cncf.pravega.io. Accessed: July 2026.

[17] Redis. https://redis.io. Accessed: July 2026.

[18] Redpanda. https://redpanda.com. Accessed: July 2026.

[19] Redpanda - tiered storage. https://docs.redpanda.com/current/manage/tiered-storage/. Accessed: July 2026.

[20] Remotestoragemanager for apache kafka tiered storage. https://github.com/Aiven-Open/tiered-storage-for-apache-kafka. Accessed: July 2026.

[21] S3proxy. https://github.com/gaul/s3proxy. Accessed: July 2026.

[22] Sra toolkit. https://hpc.nih.gov/apps/sratoolkit.html. Accessed: July 2026.

[23] Streamnative. https://streamnative.io/. Accessed: July 2026.

[24] Tiered storage in confluent platform. https://docs.confluent.io/platform/current/clusters/tiered-storage.html. Accessed: July 2026.

[25] Warpstream. https://www.warpstream.com. Accessed: July 2026.

[26] Storage reimagined for a streaming world. https://cncf.pravega.io/blog/2017/04/09/storage-reimagined-for-a-streaming-world/, 2017.

[27] Infinite storage in confluent platform. https://www.confluent.io/blog/infinite-kafka-storage-in-confluent-platform/, 2020.

[28] Streaming lakehouse: Introducing pulsar's lakehouse tiered storage, 2023. https://streamnative.io/blog/streaming-lakehouse-introducing-pulsars%-lakehouse-tiered-storage.

[29] Unlocking the edge: Data streaming goes where you go with confluent. https://www.confluent.io/blog/data-streaming-at-the-edge/, 2024.

[30] Amazon Web Services. Amazon S3 event notifications. https://docs.aws.amazon.com/AmazonS3/latest/userguide/EventNotifications.html. Accessed: July 2026.

[31] Aitor Arjona, Pedro García-López, and Daniel Barcelona-Pons. Dataplug: Unlocking extreme data analytics with on-the-fly dynamic partitioning of unstructured data. In *IEEE CCGrid'24*, pages 567–576, 2024.

[32] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. Stateful serverless computing with crucial. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–38, 2022.

[33] Alexandre da Silva Veith, Marcos Dias de Assuncao, and Laurent Lefevre. Latency-aware strategies for deploying data stream processing applications on large cloud-edge infrastructure. *IEEE Transactions on Cloud Computing*, 2021.

[34] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE CVPR'09*, pages 248–255, 2009.

[35] Raúl Gracia-Tinedo, Flavio Junqueira, and Tom Kaitchuck. "Back to the byte": Towards byte-oriented semantics for streaming storage. In *ACM/IFIP Middleware'24 (Industrial Track)*, pages 43–49, 2024.

[36] Raúl Gracia-Tinedo, Flavio Junqueira, Tom Kaitchuck, and Sachin Joshi. Pravega: A tiered storage system for data streams. In *ACM/IFIP Middleware'23*, pages 165–177, 2023.

[37] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-defined storage for multi-tenant object stores. In *USENIX FAST'17*, pages 243–256, 2017.

[38] Raluca Halalai, Pierre Sutra, Étienne Rivière, and Pascal Felber. Zoofence: Principled service partitioning and application to the zookeeper coordination service. In *IEEE SRDS'14*, pages 67–78, 2014.

[39] Pat Helland. Life beyond distributed transactions. *Communications of the ACM*, 60(2):46–54, 2017.

[40] Glenn Jocher, Ayush Chaurasia, Alex Stoken, Jirka Borovec, Yonghye Kwon, Kalen Michael, Jiacong Fang, Zeng Yifu, Colin Wong, Diego Montes, et al. ultralytics/yolov5: v7.0-yolov5 sota realtime instance segmentation. *Zenodo*, 2022.

[41] Flavio P Junqueira, Ivan Kelly, and Benjamin Reed. Durability with bookkeeper. *ACM SIGOPS operating systems review*, 47(1):9–15, 2013.

[42] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM symposium on Theory of computing*, pages 654–663, 1997.

[43] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *NetDB'11*, volume 11, pages 1–7, 2011.

[44] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *USENIX NSDI'15*, pages 589–603, 2015.

[45] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. Sonic: Application-aware data passing for chained serverless applications. In *USENIX ATC'21*, pages 285–301, 2021.

[46] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. Data storage management in cloud environments: Taxonomy, survey, and future directions. *ACM Computing Surveys (CSUR)*, 50(6):1–51, 2017.

[47] Anna Povzner, Prince Mahajan, Jason Gustafson, Jun Rao, Ismael Juma, Feng Min, Shriram Sridharan, Nikhil Bhatia, Gopi Attaluri, Adithya Chandra, et al. Kora: A cloud-native event streaming platform for kafka. *VLDB Endowment*, 16(12):3822–3834, 2023.

[48] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *USENIX NSDI'19*, pages 193–206, 2019.

[49] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. Data-driven serverless functions for object storage. In *ACM/IFIP/USENIX Middleware'17*, pages 121–133, 2017.

[50] Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López. Outsourcing data processing jobs with lithops. *IEEE Transactions on Cloud Computing*, 11(1):1026–1037, 2021.

[51] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *ACM SOSP'13*, pages 182–196, 2013.

[52] Andru P Twinanda, Sherif Shehata, Didier Mutter, Jacques Marescaux, Michel De Mathelin, and Nicolas Padoy. Endonet: a deep architecture for recognition tasks on laparoscopic videos. *IEEE transactions on medical imaging*, 36(1):86–97, 2016.

[53] Francesco Versaci, Luca Pireddu, and Gianluigi Zanetti. Kafka interfaces for composable streaming genomics pipelines. In *IEEE EMBS International Conference on Biomedical & Health Informatics (BHI'18)*, pages 259–262, 2018.

[54] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *ACM SOSP'09*, page 117–132, 2009.

[55] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *USENIX NSDI'23*, pages 1489–1504, 2023.

[56] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *USENIX HotCloud'10*, page 95, 2010.

[57] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. The nebulastream platform for data and application management in the internet of things. In *CIDR'20*, 2020.

[58] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas Newman, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, et al. Sion: Elastic serverless cloud storage. In *VLDB'23*, 2023.

[59] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. In *IEEE ISSRE'23*, pages 355–366, 2023.