HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D3.3 Active Ephemeral Data Store Release Candidate and Specification

Due date of deliverable: 30-06-2024
Actual submission date: 28-06-2024

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	11
WP/Task related to this document	WP3 / T3.1
WP/Task responsible	IBM
Leader	Bernard Metzler (IBM)
Technical Manager	Raúl Gracia-Tinedo (DELL), Pascal Spoerri (IBM)
Quality Manager	Raúl Gracia-Tinedo (DELL)
Author(s)	Bernard Metzler (IBM), Pascal Spoerri (IBM), Raúl Gracia-Tinedo (DELL), Sean Ahearne (DELL), Omar Jundi (DELL)
Partner(s) Contributing	IBM, DELL
Document ID	CloudSkin_D3.3_Public.pdf
Abstract	Interim report on functional and performance improvements of the GEDS data store, a possible application for efficient ML checkpointing, and the detailed presentation of the successful integration of GEDS with the projects NCT use case.
Keywords	Object store, tiering, ephemeral data, persistency, long-term storage, elasticity, ML training and checkpointing

History of changes

Version	Date	Author	Summary of changes
0.1	17-05-2024	Bernard Metzler	First draft.
0.5	29-05-2024	Raúl Gracia	Pravega integration.
0.6	18-06-2024	Raúl Gracia and Bernard Metzler	Text improvement and cleanup.
1.0	27-06-2024	Raúl Gracia and Bernard Metzler	Final version.

Table of Contents

1 Executive summary **2**

2 Status of the GEDS Data Store Implementation **3**
2.1 Functional Improvements . . . 3
2.2 Performance Enhancements . . . 3
2.2.1 Boost Integration . . . 3
2.2.2 Object Relocation/Spilling and Python performance . . . 4
2.3 Additional Test Cases . . . 4
2.3.1 Accelerating AI model training . . . 4
2.4 Important Bug Fixes . . . 5
2.5 Adjustments to the Code Development Infrastructure . . . 5

3 Pravega and GEDS Integration **5**
3.1 Problem Statement: Streaming Storage Tiering under Network Unreliability . . . 5
3.2 Solution Design: Pravega and GEDS Integration . . . 6
3.2.1 Implementation . . . 7
3.3 Validation . . . 8

4 Conclusions **10**

List of Abbreviations and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CC	Creative Commons
CoW	Copy-on-Write
CSI	Container Storage Interface
CSV	Comma-separated values
CUDA	Compute Unified Device Architecture
DOI	Digital Object Identifier
GEDS	Generic Ephemeral Data Store
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
IO	Input/Output
JNI	Java Native Interface
KV	Key/Value
LTS	Long-Term Storage
MDS	Metadata Service
ML	Machine Learning
NVMe	NonVolatile Memory express
NVMeoF	NVMe over Fabrics
POSIX	Portable Operating System Interface
RDMA	Remote Direct Memory Access
RPC	Remote Procedure Call
S3	Amazon Simple Storage Service
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TPC-DS	Transaction Processing Performance Council Benchmark - Decision Support
WAL	Write-ahead log

1 Executive summary

The **G**eneric **E**phemeral **D**ata **S**tore (GEDS) aims at the efficient handling of temporary data as being created, exchanged and consumed by compute tasks of a complex, potentially multi-staged computational workload. Particular attention was paid to the efficient execution of workloads in a serverless context, where the number of compute elements may vary greatly during runtime. Efficiency is achieved by direct integration of application buffer management with the lowest tier (Tier 0) of the muti-tiered GEDS. With the exception of a namenode maintaining object metadata, the data store is implemented as an application loadable library, which avoids running any extra storage service when deploying GEDS.

At the current implementation status, besides Tier 0, a Persistency Tier provides node-independent, disaggregated storage resources and object persistency, respectively. GEDS configuration allows automated object spilling from Tier 0 into the Persistency Tier. It has been integrated into GEDS via S3 API.

GEDS is positioned within the CloudSkin project as a flexible service to access and exchange working set data in an efficient manner and to allow for seamless integration with a persistent, S3 based data store. With its lightwight architecture, we aim at fulfilling the specific requirements of a data store for serverless workload execution environments in a cloud core-edge continuum. GEDS exports application bindings for both Python applications via `smart_open` [1] and Java applications via HDFS interface.

For the milestone M18 of the project, we report on new functionality, performance improvements, bug fixes, adjustments to the code development infrastructure, and successful integration with the Computer Assisted Surgery use case, a draft architecture for GPU memory integration.

Main accomplishments to be reported are: (1) GEDS integration with Pravega, (2) AI workload experiments where GEDS enables asynchronous training checkpointing. (3) Significant performance improvements achieving full 100Gb/s link speed for object sharing among GEDS nodes.

A stable open source Version `v1.0.5` is available at Github [2].

Since the last report at D3.2, main focus was on progressing the CloudSkin project use cases integration. The previously envisioned redesign of GEDS towards efficient resource sharing in an environment featuring multiple storage clients per node would not contribute to this effort and was therefore put on hold.

2 Status of the GEDS Data Store Implementation

2.1 Functional Improvements

Since M12, the following functional improvements to the GEDS data store were applied:

- Allow configuration of object stores within GEDS and GEDS itself via environment configuration variables (to support the `Pravega` use-case).
- The GEDS `smart_open` integration has been extended so that additional GEDS configuration options are supported:
  - `GEDS_AVAILABLE_STORAGE`: Configures the storage threshold for GEDS.
  - `GEDS_AVAILABLE_MEMORY`: Configures the memory threshold for GEDS.
  - `GEDS_RELOCATE_WHEN_STOPPING`: Relocate files in GEDS to a configured object store when GEDS ist stopped.
- Changes to the Python integration via `smart_open`:
  - GEDS is now stopped when the application exists.
  - Relocation is exposed as an API.
  - `open(file, mode='wb')` now always creates a new file in GEDS
  - `open(file, mode='ab')` does an explicit copy of the file if the file is not writable (e.g. on a remote node).
  - A GEDS read call directly allocates a writable Python buffer.

2.2 Performance Enhancements

2.2.1 Boost Integration

The proprietary, TCP based object transfer subsystem within Tier 0 was replaced by network services from the `boost::asio` C++ library [3]. This resulted in better performance for large object transfers and allows for easier code maintenance and extension.

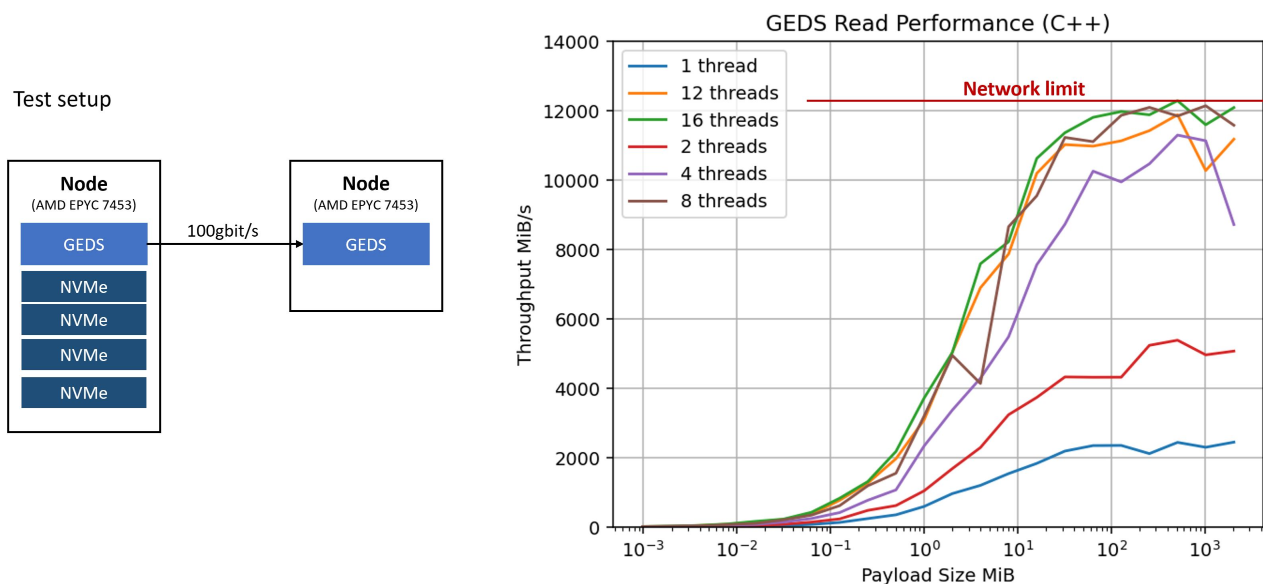

Figure 1: Remote object read performance

Figure 1 shows the results of a simple test which illustrates the increased object transfer performance between two nodes in a 100 Gb/s network setup. The node originally hosting the object to

be transferred is equipped with local NVMe drivers, which serve the local Tier 0 storage. The peer node fetches objects of different size to read access its content. For large objects and a sufficient (configurable) number of IO threads, access bandwidth is only limited by the network technology.

2.2.2 Object Relocation/Spilling and Python performance

In general, asynchronous object relocation/spilling is implemented via a threadpool. The threadpool size can be controlled via `NUM_THREAD_POOL` environment variable. The Python interop performance was significantly improved by explicitly releasing the GIL during Python I/O operations, which allows for multiple asynchronous parallel I/O operations.

2.3 Additional Test Cases

New use cases and examples were developed, including:

- A simple test case to test the GEDS HDFS integration. This test case is available in [4] at `src/test/scala/com/ibm/geds/hdfs/TestGEDSHDFS.scala`.
- A simple AI workload experiment based on a Python application, which emulates epochs of consecutive model training and asynchronous checkpointing. This test case illustrates a potential use case of GEDS for accelerating AI training workloads (see subsection below for details).

2.3.1 Accelerating AI model training

The execution time for training complex AI models typically spans long time periods, from hours to even days or weeks. Therefore, a sudden, unexpected stop of the model training due to software or infrastructure failure must not result in loosing the current stage of the model adaptation, forcing a complete re-compute to the current status. Model checkpointing to persistent storage after each training epoch allows to resume an interruped training at any point. Since it is often not possible to completely interleave training and checkpointing, the time spent in checkpointing is affecting the overall efficiency of training infrastructure usage, especially due to stalled GPUs.

We implemented a simple test scenario, which emulates the training/checkpointing cycles as a Python program. This test case is available in [2] as `examples/ai-workload`. After some emulated training time, we write checkpoints synchronously to GEDS' node-local Tier 0 instance. Subsequently, data is moved to the Persistence Tier, while model training already resumes. In our experiment, a node local NVMe RAID hosts the local file system instance backing the data objects in Tier 0. The Persistency Tier is served by a remote MinIO installation, which is integrated into the GEDS namespace via 100Gb Ethernet.

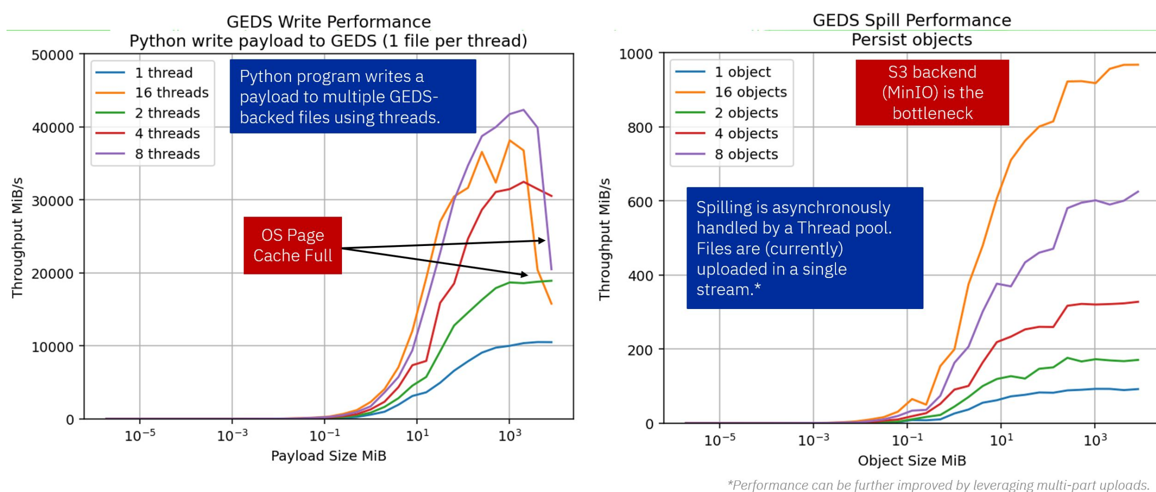

Figure 2: GEDS write and spilling performance

As a first microbenchmark, Figure 2 shows the performance of the two individual stages of writing a checkpoint. The left side shows the results for writing different sizes of data objects into Tier 0. With a OS page cache configured to 1GB, objects of similar size are written at very high speed exceeding 40GB/s. Larger objects are written at lower speeds, since the NVMe RAID's raw write performance becomes the bottleneck. On the right side, the performance of moving data from Tier 0 into the the S3 store is shown. For large objects, it is limited by the MinIO installation itself.

Translating the given numbers into a real world application as discussed in [5] describing checkpointing 43GB per GPU, we can accelerate the checkpoint time from 229 seconds to just 6 seconds. The time to persist the checkpoint would take around 500 seconds and could be completely hidden as an asynchronous GEDS spilling operation, since checkpointing happens after each 128 training iterations, or around every 1200 seconds.

Since it is planned to extend the projects Agriculture Dataspace use case towards the application of Machine Learning methods, it can potentially benefit from this approach in the future.

2.4 Important Bug Fixes

Compared to the last report at M12, the following important bug fixes have been applied.

- Fixed a bug within spilling that prevented it from working with a large number of files.
- Reduced memory pressure for large files located on S3 by downloading them in chunks.
- Improved the Java <-> HDFS interop by allowing file overwrites during create
- Fixed GEDS-HDFS files append
- Fixed a path prefix issue with GEDS-HDFS

2.5 Adjustments to the Code Development Infrastructure

Several adjustments to the development infrastructure at [2] were made, including:

- To ease potential external contributions to the project, the GEDS github resource was migrated from Travis CI to Github Actions.
- Increased the Java API version to 1.3.
- Upgraded Debian from 11.5 to 12
- Upgraded the AWS SDK

3 Pravega and GEDS Integration

In this section, we describe the integration between Pravega and GEDS. This integration enhances the capabilities of Pravega to tolerate long-term storage unavailability in the stream data tiering process.

3.1 Problem Statement: Streaming Storage Tiering under Network Unreliability

Pravega [6] is the main streaming storage infrastructure in CloudSkin. Pravega has salient performance and parallelism properties [7], and it has been a pioneer system in incorporating streaming data tiering as a core feature. Storage tiering for stream data allows us to unify the storage infrastructure for data streams, while using the right storage type for streaming (low latency write-ahead log (WAL)) and batch (high throughput long-term storage (LTS)) workloads.

Pravega was originally designed for a cluster or datacenter environment. In such environments, services are expected to be generally available and reachable. Retaking the example of storage tiering of streaming data, Pravega expects services like S3 or a network file system to be available for continuously storing stream data being written by writer applications. For the sake of clarity, Fig. 3 shows a high-level diagram of the data ingestion and tiering process in Pravega. That is, event writers append events to Pravega stream segments. Such events are immediately queued and stored in a consistent

and durable manner in WAL (*e.g.*, Apache Bookkeeper). Once the WAL acknowledges Pravega that a write has been successful, Pravega can in turn acknowledge the writer about the success of that write. In parallel, Pravega caches the event data in an in-memory cache for fast access by streaming readers. This allows a durable and high performance streaming data ingestion and reading.

The Pravega in-memory cache is also the place where data is read during the storage tiering process to LTS. To wit, the storage writer component in the Pravega Segment Store takes care of reading the operations written to WAL and applying them to LTS. In the case of appends, the payload of events is accessed from the in-memory cache, aggregated into chunks, and then written to LTS. It is important to note that data cached after an event write is marked as "unevictable" (*i.e.*, it cannot be removed from cache). The reason being is that Pravega assumes that the final destination for stream data is LTS. Therefore, until an event's data is not durably written in LTS, it cannot be removed from cache and deleted from WAL.

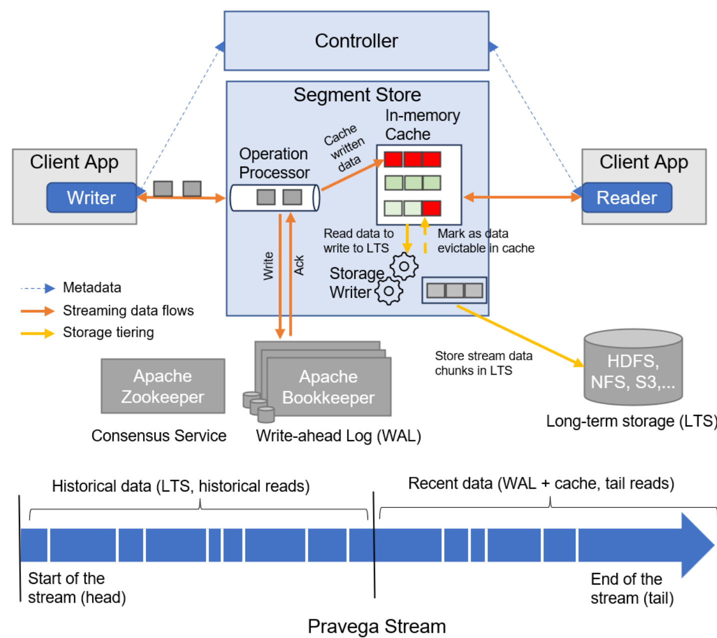

Figure 3: Overview of the data ingestion and storage tiering process in Pravega.

While the Pravega storage tiering process works well in cluster environments, the Cloud-Edge Continuum poses unique challenges for which Pravega was not initially designed for. One of such challenges is that the network connection between a Pravega instance (Edge) and the LTS service (Core/Cloud) may be unreliable. This may lead to Pravega not being able to move data to LTS during non-negligible periods of time. In some cases, this could lead to a situation in which Pravega may need to throttle the data ingestion from writers until LTS becomes reachable again. Retaking the workflow in Fig. 3, if the storage writer cannot move data to LTS, the in-memory cache will eventually become full of unevictable data. When the system reaches this situation, it starts pushing back writers by inducing delays in the ingestion pipeline. If we consider the NCT use case, a prolonged network disconnection with the remote storage to store stream data may lead Pravega to eventually stop the ingestion of video streams, and therefore interrupt the computer-assisted analytics during a surgery, which may not be acceptable.

3.2 Solution Design: Pravega and GEDS Integration

Enhancing Pravega to keep ingesting data under longer disconnections from LTS can be done in several ways. One approach is to re-design the in-memory cache with a storage spill-over mechanism that could increase the overall caching capacity of the system for buffering data in cache until data ingestion throttling kicks in. However, this approach requires an important development effort in

redesigning one complex and performance critical component of the Segment Store.

Instead, in CloudSkin we realized that storage tiering is a key aspect of the infrastructure that specialized systems already address. Concretely, IBM GEDS [2] is an ephemeral storage system with advanced storage tiering capabilities. GEDS already considers multiple storage tiers for managing the lifecycle of data (*e.g.*, memory, local storage, external storage), which could be a good fit for enhancing the buffering capabilities of Pravega upon LTS unavailability. More importantly, this integration would mitigate the targeted problem with very little development effort, thus making it an appealing alternative.

Figure 4: Overview of the integration of Pravega and GEDS.

In Fig. 4, we provide a high level overview of the integration of Pravega and GEDS. As can be seen, the main goal of the integration is to allow Pravega to offload data to GEDS, which now plays the role of LTS from Pravega's viewpoint. This means that Pravega will use the GEDS libraries to offload data to LTS, GEDS will take the control over such data storage operations from that point onwards. From Pravega's viewpoint, the workflow will be the same, meaning that the data stored in GEDS can be evicted from cache and the respective part of WAL can be deleted. On its turn, GEDS will exploit local (or remote) storage volumes as part of its tiering capabilities and it will offload data to the external storage service whenever it is available. The key point to note is that GEDS can exploit local storage volumes that may be one or two orders of magnitude larger than the available memory space, especially in Edge environments. Our expectation is that this will greatly improve the capability of Pravega to keep ingesting data when LTS is unavailable.

3.2.1 Implementation

Next, we describe the work related to implementing the Pravega and GEDS integration (see Fig. 5). As visible in Fig. 5, we decided to use the HDFS API for connecting both systems. The reason being is that GEDS already offers a Java library with the implementation of the HDFS API and Pravega already supports HDFS as LTS option. However, we had to extend Pravega's HDFS configuration (new `fs.hdfs.impl` parameter) to enable the administrator to determine the actual class to be used as HDFS binding implementation[1]. With this change, an administrator willing to use Pravega with GEDS just needs to configure the `fs.hdfs.impl` parameter of Pravega's configuration

[1] See Pull Request `https://github.com/pravega/pravega/pull/7400`

to `com.ibm.geds.hdfs.GEDSHadoopFileSystem`. During this process, we also identified some problems in the way HDFS API was implemented in GEDS, which have been addressed and merged to mainline[2] [3] [4] [5]. With all these changes in place, we created a Pravega docker container including the GEDS libraries for easy deployment in a Kubernetes cluster [6]. The instructions to deploy this integration can be found at `https://github.com/cloudskin-eu/pravega-geds`.

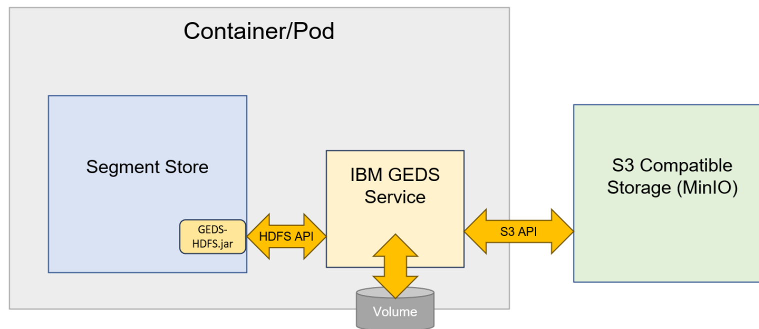

Figure 5: Implementation of the Pravega and GEDS integration.

3.3 Validation

In the following, we describe the experimental results of integrating Pravega and GEDS.

Deployment. The on-premises cluster used for testing was made of Dell PowerEdge R750 servers with Intel Xeon CPU's running VMWare Virtual Machines. A set of 4 VM's were connected using Kubernetes, with Pravega deployed within the cluster. Each VM was given 8 vCPU cores, 16GB of RAM, and 250GB of local server NVMe SSD storage, and the 3 worker VM's each had a separate 100GB drive mounted to act as dedicated MinIO drives.

Inside the cluster, we iteratively test two Pravega deployment flavors: The baseline deployment and a GEDS-integrated deployment. In both deployment flavors we use the following general structure: 1 Zookeeper instance, 3 Bookkeeper Instances, 1 Pravega Controller, 1 Pravega Segment Store, and 1 MinIO instance. In the baseline deployment, Pravega is configured to use MinIO as the LTS storage solution. In the GEDS-integrated deployment, Pravega is configured to use GEDS as LTS storage via the HDFS interface, and in turn, GEDS is configured to use MinIO as long-term storage.

Metrics. In our experiments, we use the following metrics:

- *Ingestion buffering capacity*: This metric, measured in MB, accounts for the data which Pravega can buffer while LTS is not reachable.
- *Ingestion buffering time*: This metric, measured in seconds, accounts for the time it takes for Pravega to begin throttling writers since LTS becomes unreachable.

Experimental procedure. Each experiment runs the following procedure. First, Pravega is deployed on the cluster. Then, the video reader and writer pods are started and they begin to perform IO against Pravega. After a specified duration, the MinIO stateful set is scaled to 0, making it impossible for Pravega to store data into it. After a specified duration, the MinIO stateful set will scale up to 3 and the reader and writer pods will stop. After restoring the MinIO service, Pravega is expected to resume the data ingestion process. The configuration, logs, and metrics are stored in an appropriate folder for further analysis.

Next, we focus on having a first evaluation on the potential improvement in data ingestion buffering when LTS is unavailable from integrating Pravega with GEDS. Fig. 6 shows an experiment that

[2] `https://github.com/IBM/GEDS/commit/aef8782ee1132166e9984f1762e3e8d8e00ba55b`
[3] `https://github.com/IBM/GEDS-HDFS/commit/ab6459102650828f3c9642852c59a13731fac052`
[4] `https://github.com/IBM/GEDS-HDFS/commit/73420fd31f88bdb545de5cafa373229f3839a7b1`
[5] `https://github.com/IBM/GEDS-HDFS/commit/3d6d096f159f616fde9d8cd3d6b7ec22b2547db6`
[6] `https://hub.docker.com/r/ojundi03/geds`

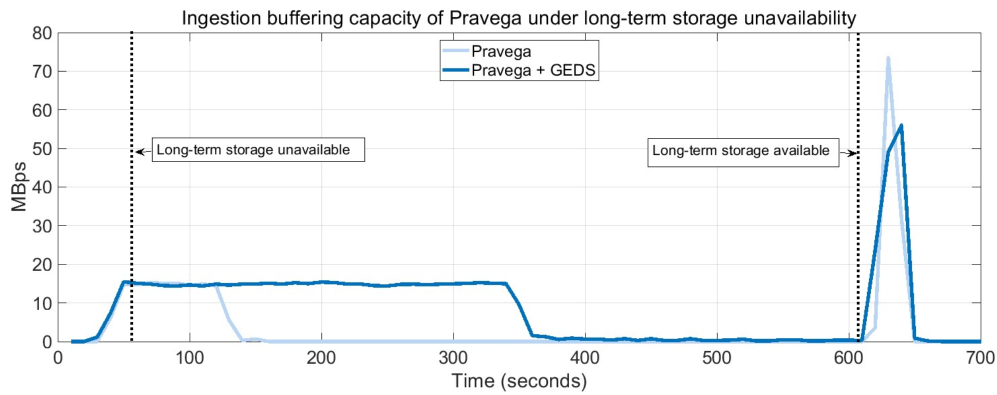

Figure 6: Improvement of Pravega ingestion buffering capabilities when integrated with GEDS.

compares the ingestion buffering capacity of Pravega with and without GEDS. In this experiment, Pravga is configured with an in-memory cache size of 1.5GB, whereas the local storage configured in GEDS is 5GB. The ingestion throughput is generated via GStreamer video writers which write at $\approx$ 15MBps. During the experiment we disconnect the long-term storage service storing stream data (MinIO) to measure the ingestion buffering capacity of the system with no long-term storage available. Visibly, Pravega can handle workload ingestion for 77 seconds, whereas Pravega with GEDS can last 292 seconds. This represents an improvement of $3.8x$ in terms of ingestion buffering in front of long-term storage outages. Note that this experiment uses a small GEDS volume; we could consider much larger GEDS volumes, as local storage may be a more abundant Edge resource compared to memory.

With GEDS, Pravega can tolerate longer unavailability of the LTS service without incurring significant code changes in the system. This is very important for the NCT computer-assisted surgery use case, as now the streaming infrastructure for the PoC (see deliverable D5.2) is more resilient to network unreliability between surgery rooms and the service storing surgery video streams.

4 Conclusions

GEDS is positioned within the CloudSkin project as a flexible object storage service to access and exchange working set data in an efficient manner and to allow for seamless integration with a persistent, S3 based data store. At the time of writing this report, a stable open-source version of GEDS is available. Since the last report, its object read performance has been further improved, as well as configurability and persistency integration. For applications using Python bindings, the I/O path was revisited to allow for parallelism during read or write operations, yielding better thoughput.

As the major achievement to be reported, in close cooperation between DELL and IBM as project partners, the current GEDS service was successfully integrated with the projects Computer Assisted Surgery use case. Using simple configuration options, Pravega can now optionally make use of GEDS to tolerate an otherwise disruptive transient unavailability of the connected long term storage services. The integration uses the Java HDFS interface of GEDS.

The report exemplifies another potential use case of GEDS. Here it is used to implement efficient ML model training snapshots, minimizing the time a training epoch is interrupted. An extension of the projects "Agricultural Dataspace" use case towards ML training can potentially benefit from this approach.

References

[1] R. Řehůřek, "smart_open – utils for streaming large files in Python." https://github.com/RaRe-Technologies/smart_open, 2015.

[2] "GEDS - a generic ephemeral data store." https://github.com/IBM/GEDS.

[3] "Boost C++ Libraries." https://www.boost.org.

[4] "Hadoop filesystem implementation for GEDS." https://github.com/IBM/GEDS-HDFS.

[5] T. Yoshimura, T. Chiba, S. Choochotkaew, S. Seelam, H. fang Wen, and J. Pfefferle, "Objcache: An elastic filesystem over external persistent storage for container clusters," 2023.

[6] "Pravega." https://cncf.pravega.io.

[7] R. Gracia-Tinedo, F. Junqueira, T. Kaitchuck, and S. Joshi, "Pravega: A tiered storage system for data streams," in Proceedings of the 24th International Middleware Conference, pp. 165–177, 2023.