CloudSkin

Funded by the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CloudSkin

(grant agreement No 101092646)

Adaptive virtualization for AI-enabled Cloud-edge Continuum

D3.2 Ephemeral Data Store Release Candidate and Specifications

Due date of deliverable: 31-12-2023
Actual submission date: 29-12-2023

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	12
WP/Task related to this document	WP3 / T3.2
WP/Task responsible	IBM
Leader	Bernard Metzler (IBM)
Technical Manager	Raúl Gracia (DELL)
Quality Manager	Marc Sanchez-Artigas (URV)
Author(s)	Bernard Metzler + Pascal Spörri (IBM)
Partner(s) Contributing	IBM, DELL, URV
Document ID	CloudSkin_D3.2_Public.pdf
Abstract	Interim report with details on data store design including performance tier, disaggregated storage tier and integration with persistent data store for checkpointing/recovery. Early access prototypes and initial results from benchmarking framework and experiments.
Keywords	Key-value store, ephemeral data, persistency, elasticity, checkpointing.

History of changes

Version	Date	Author	Summary of changes
0.1	11-12-2023	Bernard Metzler, Pascal Spörri	First draft.
0.2	14-12-2023	Bernard Metzler, Pascal Spörri	Benchmarks results added.
0.3	15-12-2023	Bernard Metzler, Pascal Spörri	Text enhancements and reordering.
0.4	18-12-2023	Bernard Metzler, Pascal Spörri	Grafana monitoring added.
0.5	20-12-2023	Raúl Gracia	Pravega integration added.
1.0	28-12-2023	Marc Sanchez	Text and content enhancenments, text polishing and final version.

Table of Contents

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 computational workload. Compared to available high performance application specific storage solutions and generic key-value stores, GEDS strives to provide an ephemeral data storage service, which balances universality and performance. The data store comprises multiple storage tiers. Efficiency is achieved by direct integration of application buffers with the lowest tier (Tier 0). Subsequent storage tiers: the Disaggregated NVM Tier and the Persistency Tier, have been introduced to provide node-independent storage resources and object persistency, respectively. GEDS can be seen as a successor of the *Crail* [1] data store, which was designed around the assumption of system wide availability of high-speed RDMA networking, allowing a radical performance tailored design, but missed general applicability in today's cloud and edge computing environments.

We aim at using GEDS as a generic ephemeral data store within the CloudSkin project, with a distinct focus on the specific requirements of a data store for serverless workload execution environments in a cloud core-edge continuum.

As of the time of this M12 CloudSkin report, a stable prototype of the GEDS data store as available for M6 was extended with: (1) The completion of the S3 Tier integration; (2) Fixes to the TCP transport subsystem; (3) Configurable object spilling; and (4) A basic solution for supporting data persistency for checkpointing/recovery using the integrated Persistence Tier. A stable open source Version 1 is available at Github [2].

Currently, GEDS is undergoing a redesign aimed at enhancing resource efficiency in an environment featuring multiple storage clients per node. In the initial GEDS design, a distinct storage client with an integrated Tier 0 was instantiated for each application process utilizing GEDS services. However, the proposed redesign consolidates the Tier 0 instances to one per node. Applications utilizing GEDS services will establish connections to this single instance through a lightweight client library. This approach eliminates the need for full-fledged RPC communication among GEDS clients, especially when they are localized to the same node. Importantly, the redesign does not impact the established GEDS API. For integration and testing, the latest stable version as available on [2] shall be used.

GEDS exports application bindings for both Python applications via `smart_open`[3] and Java applications via HDFS interface.

2 Status of the GEDS Data Store Design and Implementation

2.1 Functional Overview

This section summarizes the development status of GEDS. Main changes compared to the status reported in [4] are improvements to features not completely implemented 6 months ago. Table 1 depicts the overall implementation state of GEDS. Resulting advanced state is marked in *italic*. The current implementation state is tagged as Version 1.0 in [2].

2.2 Configurable Object Spilling

The combination of multi-tiering and object relocation features of GEDS enables elastic per-node storage resource provisioning. In cases where the available local DRAM memory for GEDS to store objects in Tier 0 is limited, any excess object storage requirements seamlessly result in the relocation of objects from Tier 0 to the S3 Tier, also known as the Persistency Tier. In the current version of GEDS, an LRU (Least Recently Used) policy is implemented to transfer the least recently used objects to S3.

Running a complex workload with potentially multiple stages and task execution parallelism can benefit from this automated object spilling mechanism for two reasons:

- In case of limited local node storage resources, workloads with excessive resource requirements can be run to completion without failure.
- Imbalanced workloads (due to, e.g. workload skewing), or single intermediate execution stages with extreme storage requirements would require the provisioning of that maximum expected storage to all workload execution entities from the beginning. Left side of Figure 1 illustrates resource usage fluctuations over time for a TPC-DS [7] query run with Apache Spark. In case a computational task cannot be run to completion due to missing storage resources, Apache Spark would retry its execution, but finally fail (Figure 1, right side).

Elastic spilling of data objects to the Persistency Tier avoids application failure and limits resource costs.

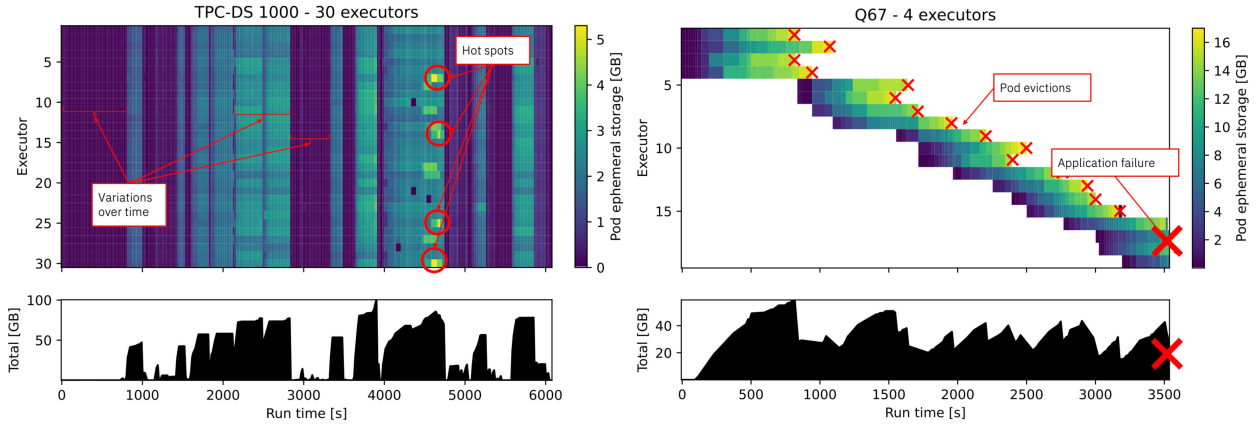

Figure 1: Spark with TPC-DS workload – Query 67. Resource utilization and final application crash.

2.3 Monitoring

GEDS exposes performance statistics on each client. These statistics can be monitored by Prometheus [6] and visualized in Grafana [8].

In Figure 2, we provide a screenshot of GEDS running TPC-DS [7] Query 67 (q67) with Apache Spark deployed on 12 Kubernetes pods. q67 generates 67GB of intermediate shuffle data in total. In order to trigger spilling, we artificially limited the available storage on each pod to only 2GB. The screenshot shows how GEDS spills data to a local object store and automatically fetches data from remote locations.

Table 1: State of GEDS at M12

Feature	State	Comment
Basic Features		
Create buckets	Done	
List buckets	Done	
Create objects	Done	
Update objects	Done	
Delete objects	Done	
List objects	Done	
Read from object	Done	POSIX-like read interface.
Write to object	Done	POSIX-like write interface.
Move objects	*Done*	In its current implementation objects are first copied and then finally deleted.
Copy objects	Needs work	Objects currently copied. Copy-on-write (COW) semantics desirable.
Advanced Features		
File metadata support	Needs work	Allows attaching metadata to objects.
Memory-mapped file access	Needs work	Compile-time flag. Only works for local objects. Requires additional semantics and design work.
Object store integration	*Done*	Object stores can be configured on a per-bucket basis. Objects can be read and written.
Object spilling	*Done*	Requires a configured object store.
Pub/Sub Support	Needs work	Pull request available but not yet integrated. Integration deferred to new client architecture.
Object Caching	*Done*	Only objects stored on S3 are cached in GEDS.
Object Replication	Not started	Required for memory-mapping of remote objects.
Configurable resiliency	Not started	E.g., force write-back to S3 if a file is written.
Multi-process support	*Started*	One GEDS instance created per process. Release of functionality deferred to new major GEDS software release.
NVMeOF Support	Not started	
Library Integration		
Java integration	Done	HDFS [5] input streams do not support ByteBuffers.
Python integration	Needs work	Not scalable since a new instance is created for each fork. No data sharing between individual processes.
Simplified wrapper	Not started	`GEDS.h` needs to be wrapped and re-exported with no external dependencies for seamless linking with C#, Go or Rust.
NVIDIA CUDA integration	Not started	Expose GEDS objects in CUDA.
Fuse Filesystem	Not started	Local GEDS object access via file system abstraction.
Prometheus integration	Prototype	GEDS exposes a web-server to export performance statistics. that can be ingested by Prometheus [6].

2.4 Benchmarks

We re-evaluated GEDS performance in a multi-tiered setup. Here, we evaluate the impact of GEDS's automated spilling in application performance in cases where Tier 0 resources are exhausted. We

Figure 2: Grafana visualization of GEDS metrics. GEDS is running in a local Kubernetes cluster with a Prometheus deployment.

assume a virtually infinite Persistence Tier to focus on the impact of automated object spilling without side effects.

Experiment. We ran four TPC-DS queries (q49, q5, q67, q75) in Apache Spark, with a particular focus on the highly I/O-intensive queries q67 and q75. The experimental setup involved 12 Kubernetes pods, each equipped with 4 cores and 32GB of DRAM. Additionally, we allocated 16GB of tmpfs (a file system for creating memory devices) per pod. Each pod hosts one Spark executor. We compare the outcomes obtained from two configurations: the first involves vanilla Spark with sufficient local storage, while the second employs our Spark Shuffle plugin [9] to manage all shuffle I/O data[1]. This shuffle plugin is connected either directly to IBM COS [11] using its S3-compatible API or to GEDS. GEDS in turn is connected to the same IBM COS instance for spilling excessive data. Additionally, we extend our comparison to include the utilization of AWS's Spark Shuffle plugin (referred to as *Chopper*, see [10]) serving as the Spark frontend to IBM COS. To simulate local resource constraints for GEDS, we limit the GEDS Tier 0 resources to 16GB and 2GB, respectively. All experiments were conducted in IBM Cloud Code Engine serverless environment [12].

Results. Figure 3 shows the results of this experiment. Overall, it becomes evident that the utilization of GEDS has a negligible impact on application performance, even when operating under significant local resource constraints. Directly spilling to the Persistency Tier (IBM COS) always comes with a performance penalty, which of course depends on the amount of data spilled (with a slight advantage for our Spark spilling plugin compared to AWS's solution). For I/O intensive TPC-DS queries, GEDS is able to spill a varying amount of data to S3, with virtually no performance impact, even under the imposed 2GB size limit for local Tier 0 (see q67 and q75 results).

[1]Shuffling refers to the operation of transferring data between partitions, allowing data rows to move between executors when their source partition and the target partition reside on different machines.

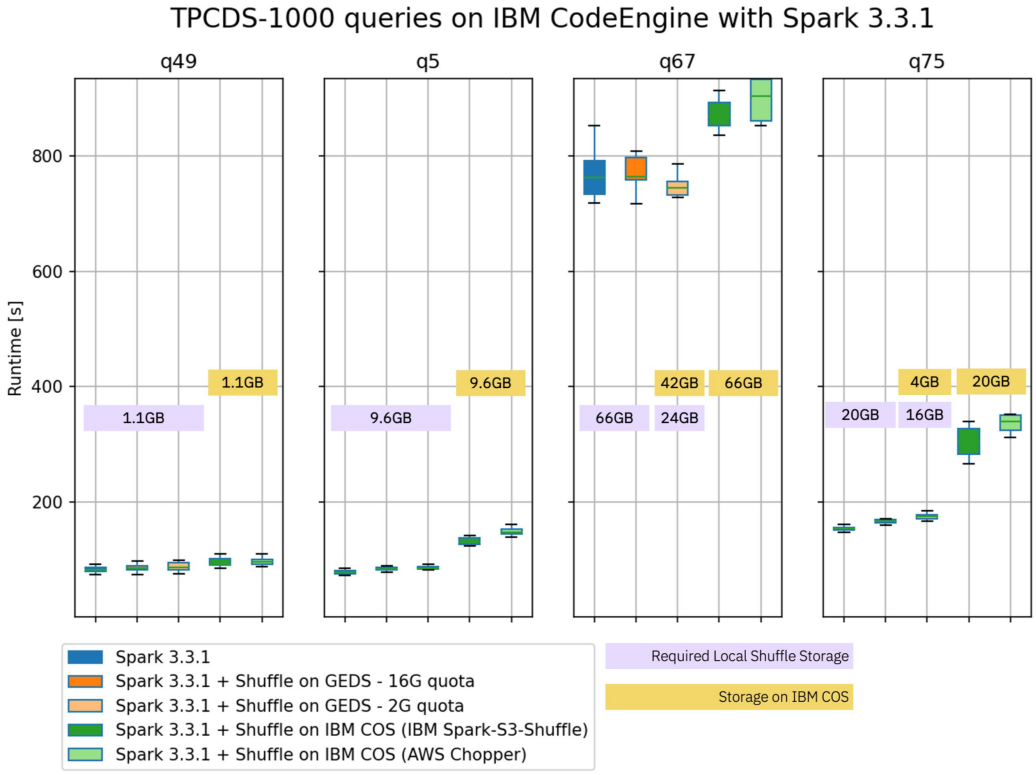

Figure 3: Apache Spark TPC-DS query execution with different resource constraints and data shuffle storage systems: 1. GEDS; 2. IBM COS with IBM Spark S3 shuffle plugin [9]; and 3. IBM COS with AWS Chopper [10].

2.5 Architectural Redesign

The existing architecture of GEDS does not assume any specific mechanism for handling the sharing and exchange of data objects among GEDS clients when they are co-located on the same physical node. As of today, access to data objects is consistently orchestrated through the Metadata Server and performed through TCP/IP communication, irrespective of whether the clients are located locally or remotely. While this approach simplifies the overall design, it foregoes important opportunities for performance and resource optimization that could be leveraged from client co-location.

GEDS shall be redesigned to support multiple local instances more efficiently. A central daemon will be responsible for managing files and sharing files between other daemons.

Goals:

- Enable data-sharing of GEDS objects between multiple Python processes.
- Enable data-sharing of GEDS objects across Kubernetes pods using a CSI Hostpath[2].
- Data should survive crashes.
- Facilitate encryption.
- Enable reading/writing data to/from GPU and/or NVMe directly.
- Native integration with Kubernetes.

[2]CSI Hostpath is a Container Storage Interface (CSI) driver that allows containers to use a directory on the host machine as a volume within a pod.

2.5.1 Motivation

As of today, GEDS clients need to communicate over the GEDS TCP stack in order to copy the content of a single file from one client to another. This approach becomes highly inefficient when processes running on the same physical machine need to read and write the same data.

2.5.2 New GEDS Client Architecture

GEDS runs as a daemon on a separate process or in a dedicated Kubernetes pod (see Figure 4 below). GEDS clients write and read to a mounted filesystem or a Kubernetes CSI Hostpath used to share the data between pods co-located in the same Kubernetes node. In the diagram below, each gray box represents a physical Kubernetes node. We also want to signal that the new design has been crafted with extensibility in mind, e.g., to support NVMeoF[3] and GPU Direct RDMA in the future, so that, for instance, multiple NVMe SSDs can be remotely accessed and operated independently (NVMe JBOD in Figure 4 stands for "Non-Volatile Memory Express Just a Bunch Of Disks).

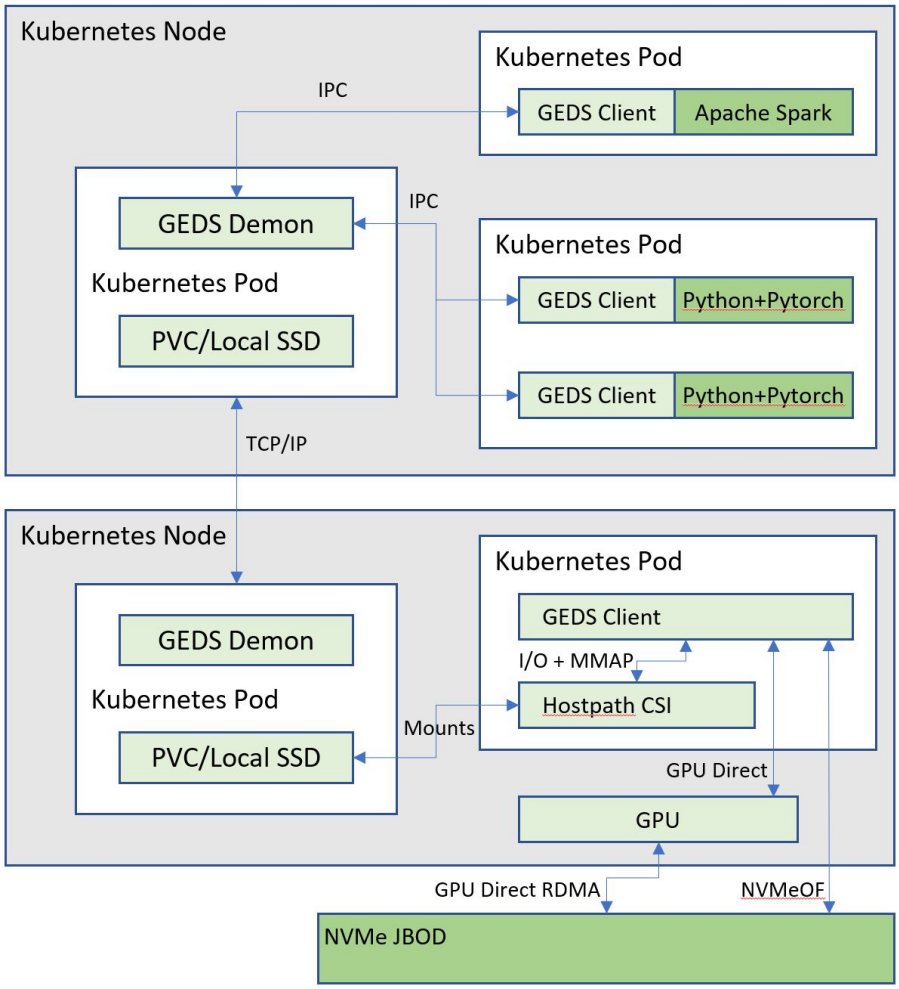

Figure 4: GEDS deployment in envisioned Daemon Mode.

Objects created in GEDS are represented as files directly. GEDS Clients will transfer the ownership of the created files to the GEDS Daemon. The GEDS Daemon will then be responsible for managing the lifecycle of the created and opened objects. Put in a nutshell, the GEDS Daemon will a reference counter for each open file. When a file is no longer open, the GEDS Daemon will initiate its relocation to the Persistency Tier, built upon IBM COS or AWS S3, or another designated object store. This

[3]NVMe over Fabrics (NVMeoF) is a storage protocol that extends the NVMe (Non-Volatile Memory Express) storage interface from a local storage system to a storage network, allowing remote access to NVMe storage devices.

process will ensure efficient management and storage of objects within the GEDS system.

Update semantics. If a client wants to modify a file, a copy will be first created by the GEDS Daemon. The client then will modify the copy. That is, changes made to this file will only become visible when a file is reopened. This will ensure consistency for clients reading a file, implementing a simple CoW (Copy-on-Write) mechanism.

This simple design fulfills most of our needs. For instance, by exploring analytics tools such as Spark and Apache Ray[4] storage APIs we found out that file re-writing does not happen very often. Some formats, like Parquet, put metadata at the end of the file. Immediately broadcasting changes to a file would lead to inconsistencies when concurrently reading the file by multiple clients. Thus, a synchronization point in the consumer application is required anyway. Forcing the clients to reopen ensures that applications can complete their tasks without encountering invalid or inconsistent data.

2.6 Data Confidentiality and Encryption in GEDS

We started initial design work to support data confidentiality and encryption in GEDS. At this point in time GEDS communicates over plaintext protocols and requires the underlying infrastructure (e.g., Kubernetes) to handle the confidential processes. Files are not encrypted, and additionally, GEDS relies on the encryption-at-rest features of AWS S3 or IBM COS to protect data confidentiality.

While single-user/customer clusters can be considered secure-enough, shared clusters and multi-cloud clusters require additional protection. In this context, a key objective of GEDS in a near future is to harness Confidential Computing technologies based on Trusted Execution Environments (TEEs), such as SCONE [13] by TU Dresden. Concretely, TU Dresden collaborates within the WP4 (Work Package 4) of the project, with the aim of enhancing data security by implementing these additional protective measures.

2.6.1 SCONE Integration

The CloudSkin project aims at integration of the SCONE Confidential Computing solution [13] with its compute infrastructure and use cases. Adding support for SCONE would allow GEDS to provide additional security features in enhanced deployments:

- SCONE will be responsible to provide the secrets for accessing object stores such as AWS S3. Secrets will be either provided as environment variables or as part of a configuration file.
- SCONE technology would be responsible for encrypting the data at rest. As SCONE already supports the encryption of files written to a folder, integrating it into GEDS would enable GEDS to support encryption-at-rest features, such as those offered by AWS S3.
- The SCONE Network shield protects unencrypted communication for applications. Integrating this technology into GEDS would enhance the communication security between the various components within the GEDS software stack. Essentially, it provides a means to safeguard the exchange of information, contributing to the overall security posture of GEDS.

2.6.2 Next steps

Related to the above goals, the tasks to performed in the next months will be the following:

Securing access credentials. GEDS needs to integrate support for reading SCONE credentials and Kubernetes secrets. This way object store credentials can be shared without explicitly storing them on the GEDS Metadata service.

Encrypting data at rest. GEDS needs to add support for encrypting data moved from the protected enclave such as Intel SGX to an external object store. This can be addressed by leveraging the client-side encryption using the AWS S3 encryption client[5]. An additional benefit will be the possibility

[4] Apache Ray, `https://www.ray.io/`

[5] What is the Amazon S3 Encryption Client?, `https://docs.aws.amazon.com/amazon-s3-encryption-client/latest/developerguide/what-is-s3-encryption-client.html`

of downloading the created objects with any other S3 client implementation of the same encryption mechanisms.

Securing communication between GEDS Clients and Metadata Service. While communication between GEDS components will remain protected with SCONE, protocol encryption is not available for environments without SCONE support. Securing communication between the GEDS Clients and the Metada Service should be accomplished irrespective of SCONE. We see this as an optional second tier feature.

2.7 Pravega Integration

Pravega is a tiered storage system for data streams [14, 15]. Within the WP3 (Work Package 3) of the CloudSkin project, Pravega is the main technology to provide high-performance and durable data stream storage for streaming workloads, such as in the NCT computer-assisted surgery use case. One of the main design characteristics of Pravega is storage tiering. That is, data events in Pravega are first written to the write-ahead log (WAL), and then the system automatically groups events belonging to the same stream segment to offload them to long-term storage (LTS). Storage tiering is part of the data ingestion path in Pravega. This means that if either WAL or LTS are not reachable or too slow, Pravega would eventually throttle writers to prevent an unbounded backlog of incoming data.

While this mechanism is good for protecting the system, it is not hard to image that in some cases, such as the computer-assisted Edge video analytics of NCT, could entail some risks. To clarify, in the context of the NCT use-case, the network connection of the Edge server in the surgery room could be unstable for a long period. This instability could result in Pravega being unable to persistently offload data to Long-Term Storage (LTS). In this situation, Pravega writers would be throttled, and this would prevent an ongoing surgery from using video analytics. One of the planned mitigations for this problem is to use GEDS as an LTS option for Pravega. Via the HDFS interface, Pravega could offload data to GEDS. At this point, GEDS would be offloading that data to the actual LTS system that is supposed to store historical video data from surgeries (e.g., object store, file system). In the case of network unstability, GEDS would take care of buffering video data from Pravega in different local storage tiers until the network connection becomes stable again and the storage offload resumes.

2.8 Stabilization and Bug Fixes

During the course of the last 6 months, GEDS code stability was constantly improved. For instance, one major issue affecting the efficiency of TCP-based object transfer subsystem was solved: During the simultaneous initialization of GEDS clients, we made sure that a potential cross-connect between both entities does not result in excessive TCP connections.

3 GEDS Persistency Tier

By default, GEDS provides no **persistency**. GEDS relies on external object stores and filesystems to provide persistency for objects. As a default behavior, objects created using GEDS stay within the GEDS system. To provide persistency to object, GEDS can be configured to always spill created files to an attached object store. This mode asynchronously uploads files to the associated object store in the background. Furthermore, GEDS can be configured to persist files when GEDS is shut down.

Integrating a native persistency and snap-shot & restore functionality is desirable to enable use-cases such as training machine learning (ML) models, where the current state of the model, including its parameters and other relevant metadata, may need to be saved at specific intervals during model training. With this functionality, GEDS will allow developers to later resume training from the saved checkpoint rather than starting from scratch. From the AI-enabled continuum perspective, snap-shot & restore functionality may be also useful to the Learning Plane developed in W5 (Work Package 5) to recover the state of a given model and continue training from the last saved point. Moreover, an application leveraging GEDS might want to explicitly tell GEDS to create a snapshot of an object.

3.1 Overview

GEDS exposes programming language filesystem wrappers like HDFS [5], `smart_open` [3] or `fsspec` [16] that emulate file-system access for efficient integration into existing applications and pipelines. For instance, `smart_open` is a Python 3 library for efficient streaming of very large files from and to object stores such as AWS S3. In order to effectively integrate persistency into GEDS, we have to translate filesystem wrapper concepts to GEDS.

3.2 Properly supporting persistency

The GEDS configuration and API has to be extended in such a way that objects are mirrored to the underlying object store by default. The upload of a file should then tied to the `close` operation of the filesystem wrapper.

3.3 Snap-Shot & Restore

In contrast to persistency, "Snap-Shot & Restore" cannot be directly mapped to a filesystem wrapper concept. In its most simplest form a "Snap-Shot" could be triggered by calling `close` on the opened file on GEDS. However, we don't believe that this is a good solution: A `close` can trigger all sort of side-effects, such as the object automatically being relocated to the associated object store.

The next obvious step would be to integrate an extra API that creates the snapshot. This would allow applications leveraging the native GEDS API to create and restore snapshots. To this aim, the snapshot functionality of the associated object store will be leveraged as much as possible.

4 Conclusion and Outlook

At the current stage of the project, the GEDS store is prepared for integration with the majority of workloads. A stable open source Version 1 can be downloaded from GitHub [2]. While keeping the interfaces of this prototype stable for application integration, we are also enhancing its functionality and implementing some re-design to achieve higher resource efficiency, as detailed in Section 2.5.

As shown in Section 2.4, one large potential benefit of using GEDS as a data store is its seamless management of available local memory resources. If the GEDS Persistence Tier is integrated with AWS S3 (or other, filesytem based) backup store, applications can execute workloads whose storage requirements exceed the capacity of the used compute systems, while not necessarily encountering a performance penalty.

GEDS maintains an elastic design very appropriate for the CloudSkin use cases, where individual data store clients can be added and removed dynamically. Also, the inclusion of the Persistency Tier is key for transparently and dynamically managing transparent data node local resource shortages.

We will extend the GEDS Ephemeral Data Store with persistency functionality as required by CloudSkin applications. Persistency will be mapped to locating data objects within the Persistency Tier. Section 3 discusses the envisioned approach.

We envision to enhance GEDS with data encryption and confidentiality. The integration of GEDS with the SCONE Confidential Computing solution has just started. We are working with our project partners at TU Dresden to develop an appropriate design.

References

[1] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler, "Unification of temporary storage in the NodeKernel architecture.," in 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019.

[2] "Hadoop filesystem implementation for GEDS." https://github.com/IBM/GEDS.

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

[4] "Early release of Ephemeral Data Store." Deliverable D3.1 of CloudSkin project. https://cloudskin.eu/assets/deliverables/CloudSkin_D3.1_Public.pdf.

[5] "The Hadoop FileSystem API definition." https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/filesystem/index.html.

[6] "Prometheus, an open-source systems monitoring and alerting toolkit." https://www.prometheus.io/, 2012.

[7] R. O. Nambiar and M. Poess, "The making of tpc-ds," in Proceedings of the 32nd International Conference on Very Large Data Bases, 2006.

[8] T. Ödegaard, "Grafana, an open-source analytics and interactive visualization web-application." https://www.grafana.com/, 2014.

[9] P. Spörri, "Shuffle Plugin for Apache Spark and S3 compatible storage services." https://github.com/IBM/spark-s3-shuffle, 2022.

[10] "Cloud Shuffle Storage Plugin for Apache Spark." https://docs.aws.amazon.com/glue/latest/dg/cloud-shuffle-storage-plugin.html, 2022.

[11] "IBM Cloud Object Store." https://www.ibm.com/products/cloud-object-storage.

[12] "Serverless on IBM Cloud." https://www.ibm.com/products/code-engine.

[13] "SCONE - Confidential Computing." https://sconedocs.github.io.

[14] 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 on Middleware, pp. 165–177, 2023.

[15] R. Gracia-Tinedo, F. Junqueira, B. Zhou, Y. Xiong, and L. Liu, "Practical storage-compute elasticity for stream data processing," in Proceedings of the 24th International Middleware Conference Industrial Track, pp. 1–7, 2023.

[16] M. Durant, "fsspec: Filesystem interfaces for Python." https://filesystem-spec.readthedocs.io/en/latest/, 2018.