



CEPH RGW MULTISITE CONSISTENCY MONITORING SERVICE

September 2025

AUTHOR(S):

Dawid Grabowski

CERN, Storage and Data Management Group, IT Department

SUPERVISOR(S):

Enrico Bocchi

Abhishek Lekshmanan





ABSTRACT

The long-term storage and availability of vast datasets, such as those generated by the Large Hadron Collider (LHC), are critical to CERN's scientific mission. The Ceph distributed storage system, with its RADOS Gateway (RGW) S3-compatible object storage interface, provides a scalable and resilient solution. To ensure high availability and disaster recovery, RGW can be deployed in a multisite replication configuration, for instance, between the Meyrin and Prévessin data centers. However, maintaining perfect data consistency across geographically distributed sites presents a significant challenge. Latency, network partitions, or software bugs can lead to replication inconsistencies, where data exists at one site but is missing or outdated at another.

This project addresses this challenge through the development of the Ceph RGW Multisite Consistency Monitor, a comprehensive tool designed to detect and diagnose replication discrepancies. The tool operates in two distinct modes: a non-intrusive Passive Monitoring Mode that listens to real-time S3 operations via Ceph's Kafka-based bucket notifications, and an Active Testing Mode that generates controlled S3 workload (PUT/DELETE operations) to stress-test the replication pipeline and validate consistency under load. The system leverages the AWS S3 command-line interface for object manipulation and a high-performance C++ component for real-time Kafka event processing. By comparing the ground truth of performed S3 operations with the stream of replication notifications, the monitor can pinpoint specific inconsistencies, such as missing notifications, extra notifications, and orphaned synchronization events. The tool produces detailed JSON reports and human-readable summaries, providing storage administrators with the necessary diagnostics to maintain the integrity of CERN's distributed storage infrastructure.



TABLE OF CONTENTS



1	INTRODUCTION	4
1.1	The Role of Distributed Storage at CERN	4
1.2	Ceph and the RADOS Gateway (RGW)	4
1.3	The Challenge of Multisite Replication Consistency	4
1.4	Project Objectives	4
2	Project Specification	5
2.1	Core Requirements	5
2.2	Functional Specifications	5
2.3	Non-Functional Specifications	6
2.4	Scope and Limitations	6
3	Background and Core Technologies	6
3.1	Ceph RGW Multisite Architecture	6
3.2	The Role of Bucket Notifications with Kafka	6
3.3	Interacting with Ceph via the S3 API	7
4	System Architecture and Design	7
4.1	High-Level Architectural Overview	7
4.2	Modular Structure	7
4.3	Configuration Management System	8
4.4	C++ Event Watcher Integration	8
5	Implementation Details: Operational Modes	9
5.1	Active Testing Mode	9
5.1.1	Workflow	9
5.1.2	Configuration	9
5.1.3	Usage Example	10
5.2	Passive Monitoring Mode	10
5.2.1	Workflow	10
5.2.2	Real-time Inconsistency Detection	11
5.2.3	Usage Example	11
6	Inconsistency Detection Logic	11
6.1	The Analysis Pipeline	11
6.2	Types of Inconsistencies Detected	12
6.3	Data Structures and Analysis	12
7	Ancillary Tooling: S3 Event Generator	12
7.1	Purpose and Features	12
7.2	Operational Modes	13
7.3	Integration with Passive Monitoring	13
8	Output, Reporting, and Results	13
8.1	Directory Structure	13
8.2	Active Mode Reports	13
8.3	Passive Mode Reports	14





8.4 Interpreting Results 14

9 Conclusion and Future Work 14

9.1 Development Challenges and Lessons Learned 14

9.1.1 S3 Notification Incompatibility 14

9.1.2 Monolithic Code Growth 14

9.1.3 Regression and Lack of Automated Testing 15

9.1.4 Python Concurrency and Performance Bottlenecks 15

9.1.5 Uncontrolled Scope Creep 15

9.2 Project Summary and Achievements 16

9.3 Future Enhancements 16

10 References 16



1 INTRODUCTION

1.1 The Role of Distributed Storage at CERN

The European Organization for Nuclear Research (CERN) operates the world's largest particle physics laboratory. The experiments conducted, particularly at the Large Hadron Collider (LHC), generate petabytes of data annually[4]. The reliable storage, rapid access, and long-term preservation of this data are paramount. Distributed storage systems are essential to meet these demands, offering scalability, fault tolerance, and high performance that monolithic systems cannot provide.

1.2 Ceph and the RADOS Gateway (RGW)

Ceph is an open-source, software-defined storage platform that provides object, block, and file storage from a single unified cluster. Its foundation is the Reliable Autonomic Distributed Object Store (RADOS)[6]. For object storage, Ceph provides the RADOS Gateway (RGW), which presents a RESTful API compatible with Amazon S3 and OpenStack Swift. This allows applications and services at CERN to interact with Ceph using a widely adopted and standardized interface.[3][2]

1.3 The Challenge of Multisite Replication Consistency

To ensure business continuity and for disaster recovery, critical data must be replicated across multiple physical locations. Ceph RGW supports a multisite configuration[3], where data written to a primary site is asynchronously replicated to one or more secondary sites. For CERN, this could involve replicating data between the primary data center in Meyrin, Switzerland, and a secondary site in Prévessin, France.

While asynchronous replication enhances write performance, it introduces the risk of data inconsistency. Potential issues include:

- **Replication Lag:** Events from the primary site are delayed in reaching the secondary site.
- **Lost Updates:** A network partition or service failure could cause a notification to be lost entirely, resulting in an object not being replicated.
- **Orphaned Events:** A replication event might be received at a secondary site for an operation that never completed or was never initiated at the primary site.

These inconsistencies can be silent and difficult to detect, potentially leading to data loss or access to stale data. A robust monitoring tool is therefore not just beneficial but essential for operational confidence.

1.4 Project Objectives

The primary objective of this project was to design, implement, and document a tool to monitor and validate the consistency of Ceph RGW multisite replication. The tool needed to be capable of both passively observing production traffic and actively generating synthetic workloads to probe the system for weaknesses.





2 Project Specification

2.1 Core Requirements

The project was defined by the following core requirements:

- **Detect Replication Inconsistencies:** The primary goal is to identify discrepancies between S3 operations performed on a Ceph cluster and the resulting replication state across sites.
- **Produce Actionable Reports:** The output must be clear, detailed, and provide administrators with the information needed to diagnose and resolve any detected inconsistencies.
- **Provide Dual Operational Modes:** The tool must support two fundamental modes of operation:
 - **Active Testing:** To generate a controlled S3 workload and verify that all operations are correctly replicated.
 - **Passive Monitoring:** To listen to S3 event notifications from an existing, live environment without generating its own traffic.

2.2 Functional Specifications

- **Active Mode:**
 - Generate a configurable number of S3 objects.
 - Perform concurrent PUT operations to upload these objects.
 - Wait for a configurable period to allow for replication (`object_lifetime_seconds`).
 - Perform concurrent DELETE operations to clean up the objects.
 - Capture all corresponding Kafka notifications.
 - Compare the set of S3 operations with the set of Kafka notifications, additionally, compare master site and secondary site notifications to generate a consistency report.
- **Passive Mode:**
 - Connect to a specified Kafka topic.
 - Continuously consume and parse S3 event notifications.
 - Provide a real-time console display of event statistics (e.g., event rate, object count).
 - Detect and log inconsistencies in real-time, such as orphaned synchronization events.
 - Run indefinitely until manually stopped.
- **Configuration:** All parameters (S3 endpoints, credentials, Kafka servers, object counts, etc.) must be configurable via a JSON file.
 - Command-line arguments must override settings from the configuration file.





2.3 Non-Functional Specifications

- **Performance:** The passive monitoring mode must be lightweight and efficient, capable of handling a high rate of events without impacting the monitored systems. A high-performance C++ component should be used for the most performance-critical task of event parsing.
- **Utilize Ceph Bucket Notifications:** The detection mechanism must be built upon the native Ceph RGW feature that publishes object operation events to a Kafka topic.
- **Usability:** The tool should be easy to set up and run, with clear command-line options and comprehensive documentation.
- **Modularity:** The codebase must be well-structured, separating concerns such as S3 management, Kafka interaction, and consistency analysis into distinct modules.

2.4 Scope and Limitations

- The tool focuses on detecting inconsistencies at the notification level. It confirms that create and delete notifications are successfully propagated for each S3 operation. It does not perform byte-by-byte data integrity checks on the object content itself.
- The tool relies on the `aws-cli` being correctly installed and configured for S3 operations.
- Automatic remediation of inconsistencies is outside the scope of this project. The tool's role is detection and reporting.

3 Background and Core Technologies

3.1 Ceph RGW Multisite Architecture

A Ceph multisite configuration consists of a **master zonegroup** and one or more **secondary zonegroups**. Every zonegroup comprises one or more zones. Data replication occurs within zonegroups, while metadata is transferred from the master zone of a zonegroup to the other zonegroups. This replication is asynchronous and relies on RadosGW daemon processes running at each site to read the logs and apply the changes.[3]

3.2 The Role of Bucket Notifications with Kafka

A significant aspect of contemporary Ceph RGW is its capability to publish notifications regarding bucket and object events to external endpoints, which can include HTTP, AMQP, and Kafka. This project takes advantage of the integration with Apache Kafka. When an object in a certain bucket is created, deleted, or modified, RGW publish a formatted notification to a designated Kafka topic.

This mechanism is central to the monitor's design. The stream of Kafka messages provides an authoritative log of operations as seen by Ceph itself. The event messages distinguish between primary operations (e.g., **ObjectCreated:Put**) and replication events (**ObjectSynced:Create**), which is crucial for detailed analysis.





3.3 Interacting with Ceph via the S3 API

The RADOS Gateway provides an S3-compatible API, which has become the de facto standard for object storage.[1] This allows the use of a vast ecosystem of existing tools. This project uses the official AWS Command Line Interface (**aws-cli**) to perform S3 operations.[5] As seen in `src/core/s3_manager.py`, operations are executed via asyncio subprocess calls to `aws s3 cp` and `aws s3 rm`, which provides a robust and well-tested method for interacting with the cluster.

4 System Architecture and Design

4.1 High-Level Architectural Overview

The monitor is designed with a modular, service-oriented architecture. A central entry point (`src/monitor.py`) parses user input and delegates control to one of two main service classes: `ActiveTestingService` or `PassiveListeningService`. These services utilize a shared set of core modules for handling configuration, S3 operations, Kafka communication, and reporting.

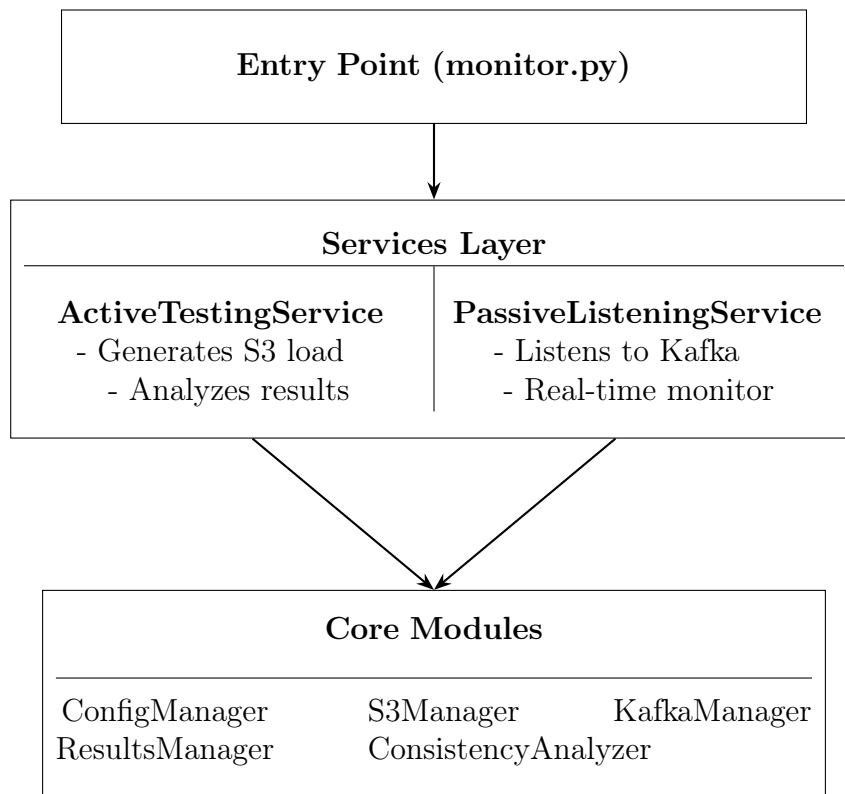


Figure 1: System Architecture Diagram

4.2 Modular Structure

- `src/monitor.py`: The main CLI entry point. Handles argument parsing and service instantiation.
- `src/services/`: Contains the primary business logic for each operational mode.
 - `active_testing_service.py`: Orchestrates the entire active test lifecycle.





- `passive_listening_service.py`: Manages the continuous Kafka monitoring loop and C++ watcher integration.
- `src/core/`: Provides shared, reusable functionality.
 - `config.py`: Manages loading and validating configuration.
 - `s3_manager.py`: An abstraction layer for performing S3 operations using aws-cli.
 - `kafka_manager.py`: Handles starting/stopping local Kafka services and consuming messages from topics.
 - `consistency_analyzer.py`: The core engine that compares S3 and Kafka data to find inconsistencies.
 - `reporting.py`: Manages the creation of output directories and report files.

4.3 Configuration Management System

To provide flexibility, the monitor uses a three-tier priority system for configuration, ensuring that users can easily override defaults for specific runs:

1. **Built-in Defaults** (Lowest Priority): Sensible default values are hard-coded.
2. **JSON Configuration File** (Medium Priority): A user-provided file (`--config`) specifies the bulk of the environment settings.
3. **CLI Arguments** (Highest Priority): Any parameter provided on the command line (e.g., `--s3-objects 50`) overrides values from the config file and defaults.

This hierarchy is managed by the `ConfigManager` class in `src/core/config.py`.

4.4 C++ Event Watcher Integration

Python, while excellent for orchestration, can be a bottleneck for high-throughput, low-latency tasks like parsing millions of log lines. In passive mode, where the monitor might process thousands of events per second, performance is critical.

To address this, the `PassiveListeningService` integrates a dedicated C++ component. This component is automatically compiled and launched by the monitor. Its responsibilities are:

- Connecting directly to the Kafka log file being written by the consumer.
- Parsing event lines in real-time with high efficiency.
- Performing immediate inconsistency detection (e.g., finding an `ObjectSynced` event without a preceding primary event).
- Writing structured output to `realtime_events.jsonl` and `inconsistencies.jsonl` with a configurable flush interval (`watcher_flush_interval_ms`).

If the C++ component fails to compile or run, the system gracefully falls back to a pure Python implementation, ensuring functionality at the cost of performance.





5 Implementation Details: Operational Modes

5.1 Active Testing Mode

5.1.1 Workflow

The active mode, implemented in `src/services/active_testing_service.py`, follows a precise, automated sequence:

1. **Dependency Check:** Verifies that `aws-cli` is installed and the S3 endpoint is reachable.
2. **Infrastructure Setup:** Starts a local Kafka instance if an external one is not available and starts the Kafka consumer process to log notifications.
3. **S3 Operations (PUT):** Generates a test file and performs a configurable number of concurrent PUT operations to create test objects. A progress bar tracks the operations.
4. **Synchronization Wait:** Pauses for a configurable duration (`object_lifetime_seconds`) to allow the multisite replication to complete.
5. **S3 Operations (DELETE):** Performs concurrent DELETE operations to clean up the test objects created in step 3.
6. **Notification Wait:** Pauses for a final, shorter duration (`notification_wait_time`) to catch any delayed Kafka notifications.
7. **Analysis:** The `ConsistencyAnalyzer` is invoked. It compares the list of successfully uploaded/deleted objects from the `S3ObjectManager` with the events parsed by the `KafkaConsumer`.
8. **Reporting:** Generates a full JSON report (`test_results.json`), a summary text file (`test_summary.txt`), and logs all captured Kafka events and debug messages.

5.1.2 Configuration

Key settings for active mode include:

- `s3_host`, `s3_access_key`, `s3_secret_key`, `s3_bucket`: S3 connection details.
- `s3_objects`: The number of objects to create and delete.
- `s3_concurrent`: The number of parallel S3 operations.
- `object_lifetime_seconds`: The crucial delay between PUTs and DELETES to allow for replication.
- `notification_wait_time`: The final grace period for late notifications.





5.1.3 Usage Example

```
python3 src/monitor.py --mode active --config config/local_multisite.json
--s3-objects 50
```

The console output provides real-time feedback on the test's progress:

```
# S3 operations with progress
Starting S3 operations test...
S3 Operations: 100%|| 100/100 [00:15<00:00, 6.5 ops/s]
Waiting 30 seconds for multisite synchronization...
```

```
# Analysis and results
Analyzing results...
```

```
FINAL CONSISTENCY REPORT
```

```
=====
```

```
S3 Operations Performed:
```

```
  PUT operations:    50
  DELETE operations: 50
  Total operations:  100
```

```
Kafka Notifications Received:
```

```
  PUT notifications:    50
  DELETE notifications: 50
  Total notifications:  100
```

```
PERFECT CONSISTENCY - No inconsistencies detected
```

5.2 Passive Monitoring Mode

5.2.1 Workflow

The passive mode, implemented in `src/services/passive_listening_service.py`, is designed for continuous, long-term observation:

1. **Infrastructure Setup:** Connects to the target Kafka cluster. It can start a local instance if needed, but typically connects to an existing production broker.
2. **C++ Watcher Launch:** Compiles (if necessary) and starts the *kafka_event_watcher* C++ process, passing it the output directory.
3. **Kafka Consumer Start:** Starts a Kafka consumer that writes raw messages to a log file, which the C++ watcher tails.
4. **Monitoring Loop:** The service enters an indefinite loop.
5. **Event Processing:** While the C++ watcher handles real-time file output, the Python service periodically processes events to update its internal state and statistics.
6. **Live Reporting:** The console displays a continuously updating status line with key metrics.





7. **Cleanup on Exit:** When the user presses Ctrl+C, the service shuts down the C++ watcher and Kafka consumer gracefully and generates a final summary report.

5.2.2 Real-time Inconsistency Detection

The primary advantage of passive mode is its ability to spot issues as they happen. The C++ watcher is designed to immediately flag problems like:

- An **ObjectSynced:Create** event appearing for an object for which no **ObjectCreated:Put** event was ever seen.
- Multiple **ObjectRemoved>Delete** events for the same object without an intervening **ObjectCreated:Put**.

These are logged instantly to `inconsistencies.jsonl`.

5.2.3 Usage Example

To start the passive monitor with a status update every 10 seconds:

```
python3 src/monitor.py --mode passive --config config/production.json --status-update 10
```

The console shows a live status dashboard:

```
STARTING PASSIVE MONITORING SERVICE
Started C++ event watcher (PID: 12345)
Real-time events will be output to: results/run_20250904_101927/realtime_events.jsonl
Press Ctrl+C to stop the service
=====

LIVE | Runtime: 212s | Events: 207 | Rate: 0.98/s | Objects: 57 | Types: 4
| Last: 10:23:13
```

6 Inconsistency Detection Logic

The heart of the monitor is the **ConsistencyAnalyzer** class from `src/core/consistency_analyzer.py`. This module provides the logic for identifying discrepancies, primarily for the active testing mode's post-run analysis.

6.1 The Analysis Pipeline

The analysis is a comparative process:

1. **Input S3 Operations:** The analyzer receives a set of object keys that were successfully uploaded and a set that were successfully deleted from the **S3ObjectManager**. This is the "ground truth."
2. **Input Kafka Events:** It receives dictionaries of all captured Kafka events, categorized by type (e.g., `put_events`, `delete_events`, `sync_put_events`).
3. **Comparison:** It compares these two datasets by object key.





6.2 Types of Inconsistencies Detected

The analyzer is programmed to detect several specific failure scenarios:

1. Missing PUT Notification: An S3 PUT operation completed successfully, but no corresponding **ObjectCreated:Put** notification was found in Kafka. This implies the event was lost between RGW and Kafka.
2. Missing DELETE Notification: An S3 DELETE operation completed successfully, but no **ObjectRemoved:Delete** notification was found.
3. Extra PUT Notification: An **ObjectCreated:Put** notification was found for an object that was never uploaded by the test client. This could indicate an issue with another client or a "ghost" event.
4. Extra DELETE Notification: A **ObjectRemoved:Delete** notification was observed for an object the client did not delete.
5. Orphaned Sync Event: An **ObjectSynced:*** event was detected without a corresponding primary S3 operation. This is a clear sign of a replication state machine error.

6.3 Data Structures and Analysis

The core of the analysis uses Python sets for efficient comparison. For example, to find missing PUT notifications, the analyzer performs a set difference:

```
missing_puts = s3_uploaded_objects - kafka_primary_put_objects
```

The final output is a **ConsistencyReport** data class, which neatly encapsulates all metrics, including consistency percentages, counts of expected vs. actual events, and detailed lists of every identified inconsistency.

7 Ancillary Tooling: S3 Event Generator

During development, it became clear that a standalone, lightweight S3 traffic generator would be valuable for testing the passive monitor and for general-purpose cluster load testing. This led to the creation of `s3_event_generator.py`.

7.1 Purpose and Features

The S3 Event Generator is a CLI tool that focuses exclusively on creating S3 load. It shares the same core **S3ObjectManager** and configuration system as the main monitor, but has no Kafka-related functionality. Its key features are:

- High-throughput, concurrent S3 operations.
- Configurable timing and object counts.
- Real-time statistics on operations per second and success rates.





7.2 Operational Modes

It supports two modes, detailed in `docs/s3-event-generator.md`:

- **Batch Mode** (`--batch`): Generates a fixed number of objects, waits, deletes them, and exits.
- **Continuous Mode** (`--continuous`): Runs indefinitely, performing PUT/DELETE cycles with a configurable delay between each cycle.

7.3 Integration with Passive Monitoring

The generator and the passive monitor form a powerful testing pair. An administrator can run the generator in one terminal to create a controlled, repeatable workload while observing the system's behavior in real-time using the passive monitor in another terminal.

Terminal 1: Generate Events

```
python3 s3_event_generator.py --continuous --objects-per-cycle 10 --cycle-interval 60
```

Terminal 2: Monitor System Response

```
python3 src/monitor.py --mode passive --debug
```

This workflow allows for precise testing of how the multisite replication holds up under specific load patterns.

8 Output, Reporting, and Results

A key project requirement was to produce clear and useful reports. Both modes generate a comprehensive set of artifacts in a timestamped run directory.

8.1 Directory Structure

Each execution creates a unique directory, e.g., `results/run_YYYYMMDD_HHMMSS/`, ensuring that results from different runs are never overwritten.

8.2 Active Mode Reports

- `test_results.json`: A machine-readable JSON file containing all configuration parameters, S3 and Kafka statistics, and a detailed breakdown of every inconsistency found. This is ideal for automated processing.
- `test_summary.txt`: A human-readable summary of the test run, highlighting the final consistency verdict.
- `inconsistencies.jsonl`: A line-delimited JSON file where each line is a single detected inconsistency, useful for easy parsing.
- `kafka_events.log`: A raw log of every Kafka message captured during the test. Essential for manual debugging.
- `debug.log`: Verbose logging of the monitor's internal operations (if run with `--debug`).





8.3 Passive Mode Reports

- `monitoring_report.json`: A summary report generated upon exit, containing aggregate statistics for the entire monitoring session (total events, event rate, unique objects, etc.).
- `realtime_events.jsonl`: The live stream of processed events, generated by the C++ watcher.
- `inconsistencies.jsonl`: The live stream of detected inconsistencies, also from the C++ watcher.
- `kafka_events.log`: Raw Kafka messages.

8.4 Interpreting Results

When an inconsistency is found, the JSON report provides rich context. For a "Missing PUT Notification", the report will include the object key and the S3 operation details, allowing an administrator to immediately investigate why RGW failed to publish the event for that specific object.

9 Conclusion and Future Work

9.1 Development Challenges and Lessons Learned

The development of the Ceph RGW Multisite Consistency Monitor was an iterative process that presented several technical and project management challenges. Overcoming these hurdles provided valuable insights that are crucial for future projects of a similar nature.

9.1.1 S3 Notification Incompatibility

- **Problem:** Initial tests revealed that certain S3 operations did not generate the expected Kafka notifications. The root cause was that some S3 event types were not fully supported in the older Ceph versions being used for development (Reef v18 and older).
- **Solution:** The test environment was upgraded to a newer Ceph version (Squid v19.2.3), which features enhanced S3 API compatibility and a more complete implementation of the bucket notification system.
- **Lesson Learned:** Always verify software version compatibility during the initial design phase. Using the latest stable versions of core dependencies like Ceph can prevent unforeseen issues related to incomplete feature implementations in older releases.

9.1.2 Monolithic Code Growth

- **Problem:** The initial prototype was developed in a single Python file which quickly exceeded 1,000 lines. This monolithic structure became difficult to navigate, debug, and maintain as complexity grew.
- **Solution:** The entire codebase was refactored into a modular, component-based architecture with clear separation of concerns (e.g., S3Manager, KafkaManager, ConsistencyAnalyzer).





- **Lesson Learned:** Establishing clear component boundaries from the project's outset is critical for scalability. Adopting a component-based architecture, even for initial prototypes, mitigates significant refactoring efforts and reduces errors during development. A more planned approach to architectural design can prove highly beneficial.

9.1.3 Regression and Lack of Automated Testing

- **Problem:** During rapid development, new feature additions frequently broke existing functionality. These regressions were time-consuming to trace manually and slowed down progress.
- **Solution:** A comprehensive testing framework was implemented, featuring unit tests for core logic, component tests, and end-to-end (E2E) tests that validate the entire workflow with live Kafka streams. The full test suite was integrated into a GitLab CI/CD pipeline.
- **Lesson Learned:** A robust CI/CD pipeline is essential for maintaining code stability. Adopting a Test-Driven Development (TDD) approach, where tests are written before or alongside new features, is the most effective strategy for preventing regressions and ensuring software quality.

9.1.4 Python Concurrency and Performance Bottlenecks

- **Problem:** The pure Python implementation struggled to process high-frequency event streams in the passive monitoring mode. Due to performance limitations and the Global Interpreter Lock (GIL), events were sometimes processed out of order or missed entirely during high-load scenarios.
- **Solution:** The performance-critical, real-time event parsing logic was migrated from Python to a dedicated, high-performance C++ component (`kafka_event_watcher`).
- **Lesson Learned:** For applications where performance, low latency, and precise timing are crucial, Python may not be the optimal tool for all tasks. C++ is better suited for high-throughput processing, and a hybrid architecture—using Python for high-level orchestration and C++ for critical components—can provide the best of both worlds.

9.1.5 Uncontrolled Scope Creep

- **Problem:** The project's scope expanded significantly beyond the original requirements. What was initially conceived as a simple monitoring tool evolved into a complex platform.
 - **Original Goal:** A service to read a Kafka stream, parse events, analyze inconsistencies, and log the status.
 - **Final Implementation:** A full platform that could auto-install and configure Kafka, manage test infrastructure, perform active stress tests (upload, sync wait, delete), and build its own components, in addition to the original passive monitoring goal.
- **Lesson Learned:** Defining and adhering to clear scope boundaries is paramount for successful project management. Using a prioritization framework like MoSCoW (Must have, Should have, Could have, Won't have) helps resist the urge to "just add one more feature" and ensures that core objectives are met on time and within the specified constraints.





9.2 Project Summary and Achievements

This project successfully delivered the Ceph RGW Multisite Consistency Monitor, a tool that meets all the core requirements outlined in the project specification. It provides robust, flexible, and performant capabilities for detecting replication inconsistencies in a Ceph multisite environment. The dual-mode approach allows it to serve both as a forensic tool for production environments and as a validation tool for staging and development. The modular architecture and documentation ensure its maintainability and extensibility. The tool represents a significant step forward in ensuring the operational reliability of CERN's distributed object storage infrastructure. And it will certainly serve as a proof of concept for collecting data from cluster operations using the Ceph feature of bucket notifications

9.3 Future Enhancements

While the current tool is fully functional, several enhancements could further increase its value:

- **Multiple buckets notification collection:** The system should be adapted to support multiple buckets, currently it is adjusted to one, additionally it must be named the same on two clusters, which is a simplification used in the project, but is easy to expand.
- **Object-lifetime based inconsistency marking:** In passive mode, the “inconsistency” flag should occur after time n . In program, an object is inconsistent immediately after being added because it hasn't synced yet, which is true. However, a synchronization could occur within 10 seconds, and if this information would be sent as an alert; after a short while, it becomes outdated. The time for defining something as inconsistent should be, for example, 20 times the synchronization time in the configuration, e.g., 100 seconds.
- **Web-based UI:** A graphical dashboard could provide a more intuitive way to view real-time statistics and browse inconsistency reports.
- **Prometheus Integration:** Exposing key metrics (e.g., inconsistency counts, event rate) in a Prometheus-compatible format would allow for integration into CERN's standard monitoring and alerting systems.
- **Automated Remediation Scripts:** The tool could be extended to generate suggested remediation commands (e.g., commands to manually sync or delete an object) based on the inconsistencies it finds.
- **Broader Protocol Support:** While focused on Kafka, the architecture could be adapted to support other notification endpoints like AMQP or HTTP.

10 References

- [1] Amazon Web Services, Inc. *AWS CLI Command Reference - s3*. 2025. URL: <https://awscli.amazonaws.com/v2/documentation/api/latest/reference/s3/index.html> (visited on 09/04/2025).
- [2] Ceph Development Team. *Ceph RADOS Gateway Bucket Notifications*. 2025. URL: <https://docs.ceph.com/en/latest/radosgw/bucket-notifications/> (visited on 09/04/2025).
- [3] Ceph Development Team. *Ceph RADOS Gateway Multisite Replication*. 2025. URL: <https://docs.ceph.com/en/latest/radosgw/multisite/> (visited on 09/04/2025).





- [4] CERN. *Cern Copmuting*. URL: <https://www.home.cern/science/computing>.
- [5] Dawid Grabowski. *Ceph S3 Bucket Notifications Service*. CERN, Sept. 2025. URL: <https://gitlab.cern.ch/ceph/s3-notifications> (visited on 09/05/2025).
- [6] Sage A. Weil et al. “RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters”. In: *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW '07)*. New York, NY, USA: ACM, 2007. DOI: [10.1145/1374596.1374606](https://doi.org/10.1145/1374596.1374606).

