CTAARCHS: Cloud-based Technologies for Archival Astronomical Research Contents and Handling Systems

Stefano Gallozzi *, Georgios Zacharis, Federico Fiordoliva, Fabrizio Lucarelli

Posted Date: 11 June 2025

doi: 10.20944/preprints202506.0847.v1

Keywords: CTAARCHS; cloud and edge storage; astronomical archives; big-data in astronomy; distributed archives; distributed databases; distributed storage

 doi:10.20944/preprints202506.0847.v1

Article

CTAARCHS: Cloud-based Technologies for Archival Astronomical Research Contents and Handling Systems

Stefano Gallozzi [1,*], Georgios Zacharis [1], Federico Fiordoliva [1] and Fabrizio Lucarelli [1,2]

[1] INAF-OAR, Istituto Nazionale di Astrofisica – Osservatorio Astronomico di Roma

[2] INAF- SSDC, Science Space Data Center

* Correspondence: stefano.gallozzi@inaf.it; Tel.: +39 0694286453 – ORCID: 0000-0003-4456-9875

Abstract: This paper presents a flexible approach to a multipurpose, heterogeneous archive model that merges the robustness of legacy Grid-based technologies with modern Cloud and Edge computing paradigms. It leverages innovations driven by Big Data, IoT, AI, and Machine Learning to create an adaptive data storage and processing framework. In today's digital age, where data is the new intangible gold, the "gold rush" lies in managing and storing massive datasets effectively—especially when these data serve governmental or commercial purposes, raising concerns about privacy and the misuse by third-party aggregators. Astronomical data, in particular, require this same thoughtful approach. Scientific discovery increasingly depends on efficient extraction and processing of large datasets. Distributed archival models, unlike centralized warehouses, offer scalability by allowing data to be accessed and processed across locations via cloud services. Incorporating edge computing further enables real-time access with reduced latency. Major astronomical projects must also avoid common Single Points of Failure (SPOFs), often resulting from suboptimal technological choices driven by collaboration politics or In-Kind Contributions (IKCs). These missteps can hinder innovation and long-term project success. This paper outlines best practices in archive project management—from policy development and task planning to use-case definition and implementation. Only after these steps can a coherent selection of hardware, software, or virtual environments be made. The proposed model—CTAARCHS (Cloud-based Technologies for Astronomical Archiving Research Contents & Handling Systems)—is an open-source, multidisciplinary platform supporting big data needs in astronomy. It promotes broad institutional collaboration, offering code repositories and sample data for immediate use.

Keywords: CTAARCHS; cloud and edge storage; astronomical archives; big-data in astronomy; distributed archives; distributed databases; distributed storage

1. Good and Bad Practices in Data Management Projects

This paper introduces a flexible archival model that integrates recent developments across Data-Grid, Cloud, Edge, and Fog computing technologies. Designed to meet the requirements of large-scale astronomical projects, the model emphasizes resilience, performance, and sustainability while avoiding typical Single Points of Failure (SPOFs), which often arise from short-sighted political management decisions and suboptimal In-Kind Contribution (IKC) allocations.

A recurring issue in large scientific collaborations is the allocation of leadership roles based not on technical expertise, but on political convenience or financial leverage. This leads to fragmented and inefficient work organization, particularly in core areas such as data handling and archiving. It is common to see simple tasks unnecessarily divided among multiple groups, each with distinct visions and leadership, making coordination and integration difficult. In response, project leaders often "descope" activities, reducing group autonomy in favor of hierarchical control. While this may streamline decision-making, it suppresses innovation and undermines project agility.

A particularly harmful trend is the political fragmentation of archive design, where medium- to long-term data management is split across loosely defined entities without real architectural

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 June 2025 doi:10.20944/preprints202506.0847.v1

boundaries. Such divisions introduce complexity and delay, especially when multiple groups interact with a shared infrastructure. Leadership may be assigned to individuals with little or no technical background, and the final decision-making authority may reside with administrative bodies rather than developers. This practice results in systems driven by political compromise rather than technological soundness.

Effective archive development must begin with robust planning. As outlined in Figure 1, project management strategies should reflect the project's timeline and goals. For short-term implementations, use-case generalization and rapid prototyping are essential to test technological feasibility. For long-term projects, more detailed planning, including thorough documentation of use cases, requirements, and interfaces, should be established early on. However, premature commitment to specific technologies should be avoided, as rapid technological evolution can render early choices obsolete.

The system design phase consolidates all use cases and validated requirements into an integrated solution based on proven technologies. This is followed by code development, pre-production testing, and final deployment. A major constraint, particularly in scientific archiving, is budgetary: long-term maintenance costs are often underestimated or ignored. As a result, hardware acquisition frequently follows funding availability rather than design logic. To overcome this, a virtualized, service-based model is adopted, allowing for the decoupling of hardware from software layers.

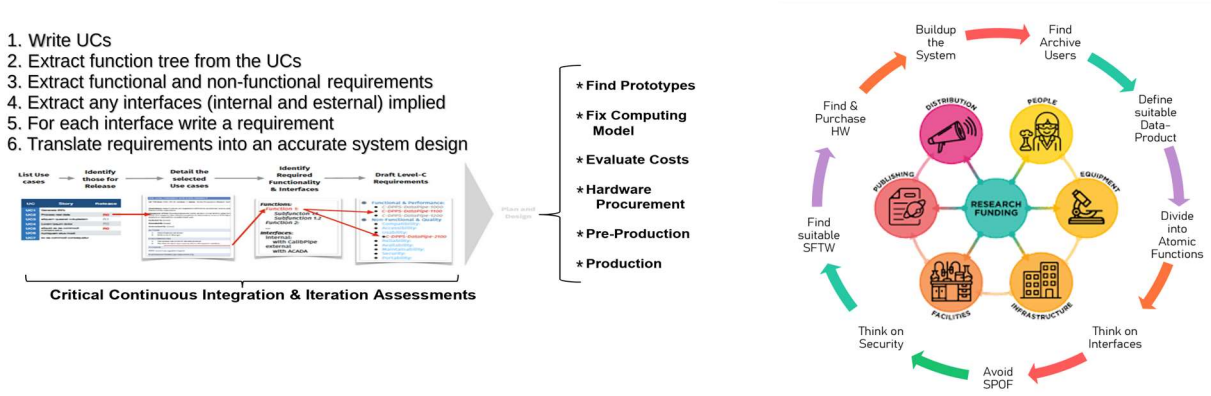


Figure 1. Good-policies in Data Management Projects.

This approach enables the implementation of Archive as a Service (AaaS), which builds upon the established paradigms of Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), as illustrated in Figure 2.

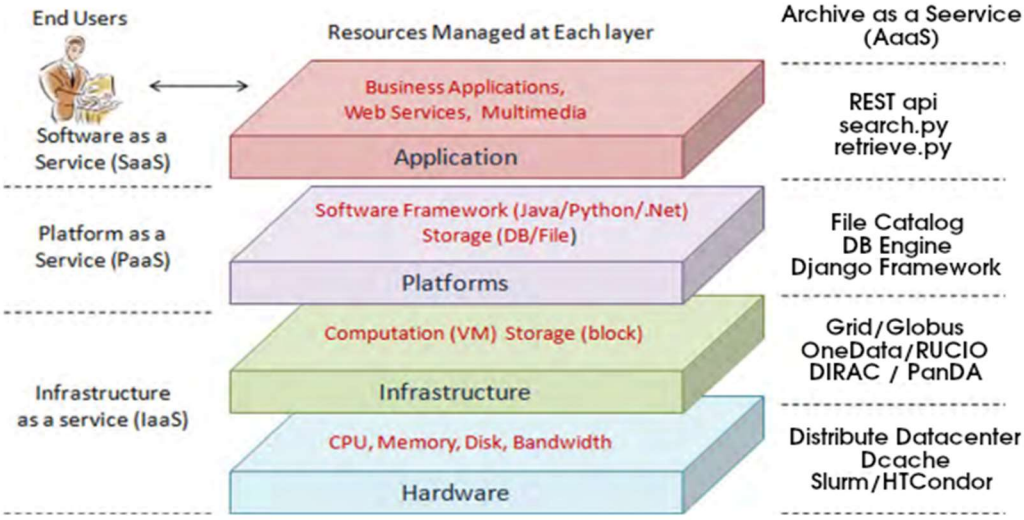


Figure 2. Archive as a Service using DataCloud paradigms IaaS, PaaS and SaaS.

 doi:10.20944/preprints202506.0847.v1

The archive model distinguishes two main user roles: Data Producers, who supply content at various levels, and Data Consumers, who access and possibly process the data (Figure 3). While users may act in both roles, each must interact with the system through standardized, role-specific interfaces.

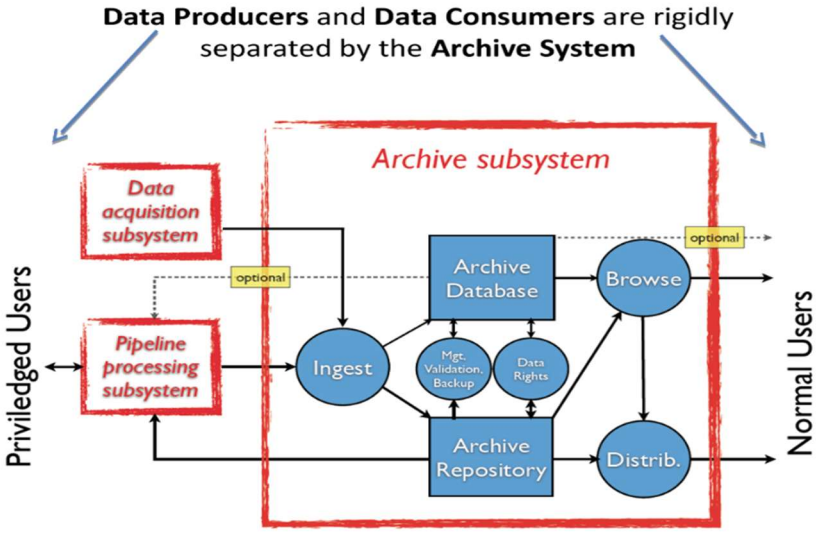


Figure 3. OAIS Standard explanation.

The proposed architecture aligns with the **Open Archival Information System (OAIS)** reference model, which logically separates user interaction from core system operations. The core functions of the archive—Ingest, Search/Browse, and Retrieve/Distribute—are built upon two foundational components: a repository and a database. The choice of technologies in these areas is dictated by the archive's system topology and performance goals. Ultimately, an effective archive is not just a structured data store but an intelligent platform facilitating efficient data access and long-term preservation. In the following sections, we discuss database selection as a critical factor in the design of scalable, user-centered archive systems.

2. Storage Architecture in Archival Systems: Centralized vs Distributed Approaches

One of the key design challenges in developing an astronomical archive system is selecting the appropriate storage architecture. The decision between a centralized or distributed model depends on the archive's use cases, particularly when access is required across geographically distributed locations. There is no universal solution—each approach has strengths and limitations based on scalability, resilience, access latency, cost, and administrative complexity.

As summarized in Table 1, centralized architectures offer simplicity, streamlined security, and ease of management, making them suitable for small-scale or local deployments. However, they pose greater risks of failure and limited scalability. In contrast, distributed architectures support high availability (HA), redundancy, and better performance across dispersed users, though they require more sophisticated orchestration and monitoring.

Table 1. Pros and Cons of distributed and centralized archive solution approaches.

	Pros		Cons	
	Distributed Archive	Centralized Archive	Distributed Archive	Centralized Archive

 doi:10.20944/preprints202506.0847.v1

Scalability	Easily scalable by adding nodes or storage across locations.	Simpler infrastructure for small-scale systems.	More complex to manage coordination between multiple nodes.	Harder and costlier to scale once capacity is reached.
Resilience & Redundancy	High availability; failure of one node doesn't compromise access.	–	Requires sophisticated monitoring and synchronization tools.	Vulnerable to outages if no redundancy or failover is in place.
Performance & Speed	Improved access times via geographic proximity; enables load balancing.	Fast access for users close to the central server.	Latency may increase if not optimized for global access.	Performance can degrade under heavy load or traffic congestion.
Flexibility & Cost	Potentially cheaper to grow incrementally (e.g., via cloud or P2P).	More cost-effective for small/medium deployments.	Higher operational overhead for maintaining distributed nodes.	Expensive upgrades required as demands increase.
Fault Tolerance	Built-in disaster recovery ensures data integrity.	–	Ensuring data consistency across all nodes is challenging.	Greater risk of data loss if no proper backup or disaster plan exists.
Security	–	Easier to manage access and enforce security centrally.	Harder to enforce consistent security policies across locations.	Centralized point may be a larger attack surface if not properly secured.
Data Consistency	–	Strong consistency due to single control point.	Data may be temporarily inconsistent due to network delays or partitions.	–
Ease of Management	–	Easier setup, backup, and management from a single location.	More complexity in setup and maintenance.	–
Geographic Access	Efficient access from multiple locations.	–	Latency if nodes are not well-distributed or if networks are slow.	Slower access for users located far from the central server.

The choice ultimately depends on system scale, geographic distribution, and acceptable complexity. The model presented here allows flexible configuration—from a single-node centralized instance to a distributed system with multiple nodes in MASTER+SLAVE or fully redundant HA configurations, ensuring no single point of failure (SPOF).

Historically, Data-Grid computing was the dominant model in research environments, where computing and storage were distributed across tiered datacenters connected by middleware for data orchestration. While effective in some contexts, its hierarchical structure limited scalability and flexibility. Over the past decade, this model has largely been replaced by Cloud Computing, which enables horizontal scaling, service-based architecture, and global accessibility. Cloud systems offer improved resource outsourcing, built-in redundancy, and disaster recovery, making them better suited to handle complex, large-scale datasets with minimal management overhead.

 doi:10.20944/preprints202506.0847.v1

However, widespread data sharing via cloud platforms raises serious security and privacy concerns, making robust access control and encryption critical challenges.

More recently, computing paradigms have shifted toward Edge Computing, where data processing occurs closer to data sources—often at the sensor or device level. This reduces network congestion, minimizes latency, and enables real-time applications. Edge computing is particularly valuable in time-sensitive use cases, where immediate processing and decision-making are required. Enhancing this model with Edge Intelligence—that is, applying AI and machine learning algorithms locally—enables automated decisions based on complex, use-case-specific criteria. This adds significant value where human intervention must be minimized.

At a broader level, this leads to Fog Computing, a form of fine-grained distributed processing that extends computing and storage further toward the network edge. By integrating IoT devices and localized data sources, fog architectures process large volumes of unstructured data near their origin, which is essential for real-time analytics.

Given the limited computational capacity typical at the edge, adaptive AI algorithms play a critical role in optimizing performance. These systems can identify semantic patterns, adapt compression techniques, and reduce computational loads, enabling efficient data analysis and visualization. The use of optimized low-latency databases becomes essential in transforming raw data into science-ready outputs quickly and interactively.

Note: Although these models raise legitimate concerns about environmental impact—particularly related to the power demands of AI training and edge infrastructure—this paper does not address sustainability. It is misleading to discuss energy use without a comprehensive life-cycle analysis of the hardware and algorithms involved. The sustainability of AI and edge computing should not be reduced to superficial claims but rather evaluated within a systemic framework, which is beyond the scope of this discussion.

3. Selecting the Appropriate Database Architecture for Archival Systems

The database lies at the heart of any archive system, making its selection a critical component of the overall design. However, there is no universally optimal solution—the appropriate database choice depends on multiple factors, including the storage use case, system topology, data access patterns, and geographic distribution of users.

In distributed storage environments, relying on a centralized database for file cataloging introduces significant risks. It creates a Single Point of Failure (SPOF) and becomes a performance bottleneck under concurrent, geographically dispersed queries. This undermines the redundancy and resilience typically sought in distributed systems.

Conversely, centralized database architectures are well-suited for smaller or geographically constrained archives, where high availability can be ensured through network and service redundancy. These systems benefit from ACID-compliant transactions—Atomicity, Consistency, Isolation, and Durability—which are essential in contexts requiring strong data integrity, such as financial systems.

However, distributed databases cannot fully guarantee ACID properties and instead operate under the CAP Theorem (Brewer's Theorem), which states that a distributed system can only simultaneously satisfy two of the following: Consistency, Availability, and Partition Tolerance. Trade-offs among these properties must be carefully evaluated depending on the archive's performance and reliability needs (see Table 2 and Figure 4).

In summary, the choice between centralized and distributed database architectures must align with the system's scale, access requirements, and fault-tolerance goals. The database model must not only support efficient data access but also integrate seamlessly into the broader storage and computing infrastructure.

Table 2. C.A.P. Theorem, summary properties.

Definition	Key Characteristics

 doi:10.20944/preprints202506.0847.v1

Consistency	All nodes return the most recent write for any read request.	- Guarantees up-to-date data across the system - All parts of the system see updates immediately
Availability	Every request gets a response, even if some nodes are down.	- System remains responsive at all times - May not always return the latest data
Partition Tolerance	System continues to work despite network failures or communication breakdowns between nodes.	- Handles network partitions gracefully - Ensures continued operation despite node isolation or failure

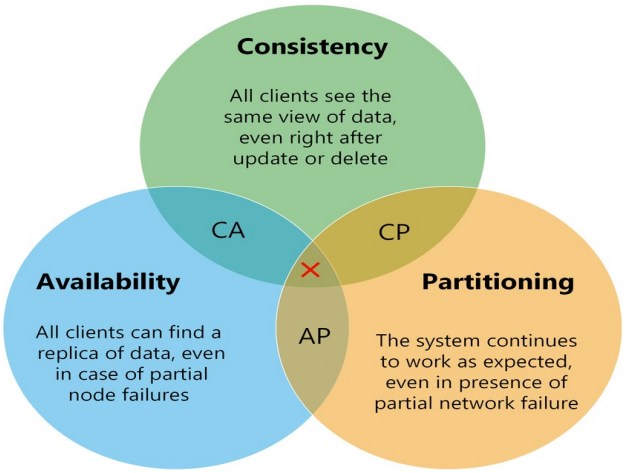


Figure 4. The CAP Theorem. Of the three properties of database you can pick only two.

In distributed databases, data is replicated across multiple nodes. When network partitions occur—isolating one or more nodes—the system must prioritize among **Consistency**, **Availability**, and **Partition Tolerance** (the CAP Theorem).

- Prioritizing **Consistency** may halt reads or writes to prevent divergence, sacrificing availability.
- Prioritizing **Availability** ensures responsiveness, but may serve outdated or inconsistent data.
- Prioritizing **Partition Tolerance** allows continued operation despite communication failures, though it may compromise either consistency or availability.

Many systems dynamically balance these trade-offs based on application needs. For archival systems, using pre-assigned physical file names and a Write Once, Read Many (WORM) model minimizes consistency concerns. Once written, immutable data simplifies coherence across nodes. This permits a focus on Availability and Partition Tolerance (AP), ensuring the system remains operational and responsive—even if some nodes are unreachable.

Partition tolerance is often the most critical factor in large-scale or globally distributed environments, as network disruptions are inevitable. Ensuring only a single version of any file exists and is replicated guarantees that if a file is accessible, it is valid and consistent system-wide.

Another key factor in choosing a database system is balancing data scalability with the complexity of the data model and queries. As illustrated in Fig. 5, certain database families are inherently unsuited to large-scale data. For instance, relational databases (SQL), while efficient for smaller datasets and simpler queries, struggle when dealing with high-complexity joins or terabyte-scale tables. At this point, only three options remain:

1. Simplify the data model or queries.
2. Scale up the hardware infrastructure.
3. Migrate to a different database family—such as a document-oriented (NoSQL) system.

 doi:10.20944/preprints202506.0847.v1

In practice, restructuring or hardware upgrades often cause service interruptions, particularly when the database was not properly designed from the outset. This underscores the importance of selecting the appropriate architecture early in the project lifecycle.

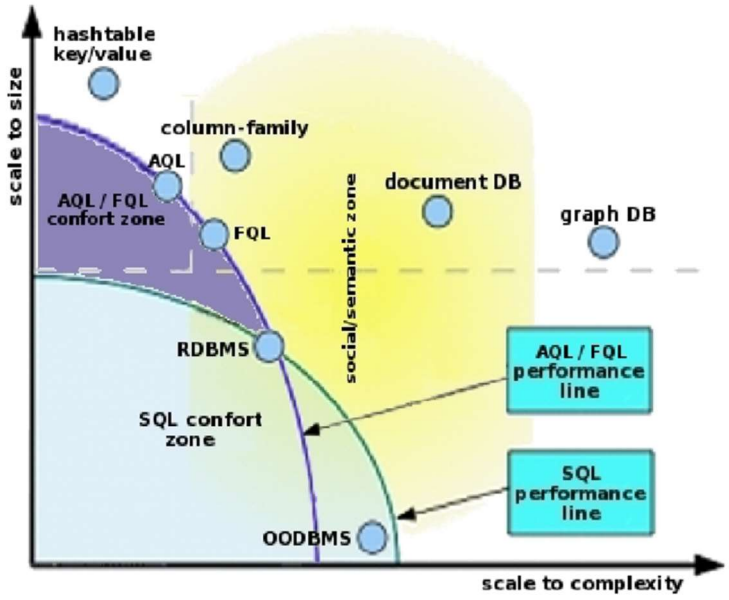


Figure 5. Performance Scale-up to size _vs_ to complexity in different database families.

Databases can broadly be categorized into two groups: Relational DBMS (RDBMS) and Not Only SQL (NoSQL) systems. A comparative summary is provided in Table 3.

Table 3. Different databases technologies.

	Subtype / Model	Key Characteristics	Use Cases
Relational Databases (RDBMS)	**Traditional RDBMS**	• Structured schema (tables with rows and columns) • Uses SQL • Strong consistency with ACID (Atomicity, Consistency, Isolation, Durability) properties	Banking systems, ERP, CRM, enterprise apps
	NewSQL	• Combines ACID consistency of RDBMS with horizontal scalability • Maintains SQL interface • Built for modern, high-scale applications	High-performance apps requiring strong consistency (e.g., fintech, gaming)
	OLAP/MOLAP	• Optimized for analytical and BI queries • Pre-aggregated data cubes • High performance for historical data • Supports complex analytical calculations	Business Intelligence (BI), data warehousing, reporting tools
NoSQL Databases	**Key-Value Stores**	• Simple key-value pairs • Excellent read/write performance • Easy to scale horizontally • Flexible schema	Caching, session data, real-time systems

 doi:10.20944/preprints202506.0847.v1

Column-Family Stores	• Stores data by columns, not rows • Ideal for distributed large datasets • High availability and fault tolerance • Schema-less rows with flexible structure	Analytics, time-series data, telemetry, log storage
Document-Oriented DBs	• Stores semi-structured data in documents (JSON, BSON, XML) • Schema-less and flexible • Supports nested data structures • Good for modern app development	Content management, product catalogs, APIs, evolving schema applications
Graph Databases	• Data represented as nodes and relationships • Efficient for traversing complex relationships • Schema flexibility • Optimized for relationship-based queries	Social networks, recommendation systems, fraud detection

4. Polyglot Persistence in Modern Archive Systems

For this archival model, we focus on the versatility, schemaless nature, and aggregation capabilities of document-oriented databases. Their architecture supports scalability through replication, sharding, and clustering, depending on performance demands and availability requirements. Strategies for scaling read/write capacity and ensuring high availability are summarized in Table 4.

If data size exceeds single-server capacity, two strategies are available: scaling up infrastructure or scaling out via clustering. Similarly, read performance can be improved through replication and caching, while write scalability benefits from partitioning and sharding. To mitigate SPOFs and ensure service resilience, especially in geographically distributed collaborations, combining clustering with cross-site replication is essential. Inter-datacenter distances of several hundred kilometers are generally sufficient to safeguard against regional failures and enable disaster recovery.

A key principle here is polyglot persistence, which leverages multiple database types, each tailored to a specific data class. For example:

- **Relational databases** (e.g., PostgreSQL, MariaDB) for structured data like observation proposals.
- **Document-oriented databases** (e.g., MongoDB) for semi-structured metadata.
- **Column stores** (e.g., Cassandra) for streaming telemetry.
- **Key-value stores** (e.g., Voldemort) for fast-access logs.
- **Graph databases** (e.g., Neo4j, Cosmos DB) for user interaction mapping.
- **Array or Functional query languages** for analytical pipelines.

This modular approach allows independent scaling of archive components and optimization of performance and cost. The main drawback lies in the complexity of managing diverse technologies and the associated manpower and training costs.

Table 4. Common database problems and common solutions.

Problem	Limits & What to do	Solutions
Scale Data Size	Approaching the maximum server capacity ▹ Distribute tables and	**Clustering**

 doi:10.20944/preprints202506.0847.v1

	databases across multiple machines (nodes)	
Scale Read Requests	Approaching the maximum number of DB server requests ▹ Reduce the number of requests made ▹ Distribute request traffic among different replicas	**Chaching Layer and Replication**
Scale Write Requests	Approaching the maximum number of write requests handled by a DB server ▹ Split writes among multiple instances ▹ Split table records across multiple shards/containers	**Data Partitioning and Sharding**
Provide High Availability	Avoind SPOF ▹ Make services independent by crashes	**Data Replication**

5. Polyglot Persistence in a Data Lake Scenario

In modern observatories, archives manage more than just raw scientific data. A **Data Lake** approach is adopted to incorporate a wide range of heterogeneous data products—proposals, schedules, weather station outputs, logs, alarms, analytics, and system monitoring.

Different database systems are better suited for handling different types:

- Relational databases for structured data.
- Object storage for unstructured or large datasets (e.g., images, videos, documents).
- NOSQL databases for semi-structured data that doesn't fit into a rigid schema.
- Graph databases for analyzing complex relationships and social semantic analytics.

Polyglot persistence ensures that each data type is managed by the most appropriate database and storage technology, enabling long-term flexibility and integration across services. In Figure 6 it is reported a generic case-study of the archives commonly managed within an Astronomical Observatory Facility.

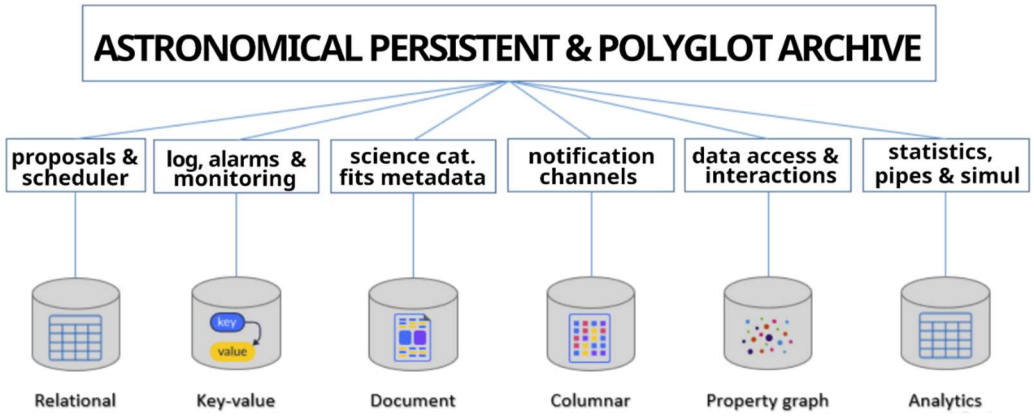


Figure 6. Descriptive use of Polyglot Persistence to different kind of data.

Polyglot persistence relies on a Unified Access Layer—a middleware abstraction that enables seamless querying, handling, and processing of heterogeneous datasets across diverse storage backends. This layer simplifies interaction with various database systems and protocols within a distributed archive.

Different data types are best served by specialized database technologies:

 doi:10.20944/preprints202506.0847.v1

- **Structured Proposal Data** can be easily managed by a Relational DBMS (e.g., MariaDB, PostgreSQL)
- **Logs and Alarms** require high-throughput so a key-value stores (e.g., Voldemort) can well fit.
- **JSON-based Scientific Metadata** can rely on a Document-oriented DBs (e.g., MongoDB)
- **Streaming Telemetry and Event Data** may need a Column-family databases (e.g., Cassandra) approach
- **Tracking Accesses and Users Interactions** could be managed by a Graph databases (e.g., Neo4j, Azure Cosmos DB)
- **Data Analytics/Pipelines** can be easily stored by an Array or functional query systems approach

By matching each data type to the most suitable database family, this model enables independent scaling of archive components and optimized performance. Object storage handles large unstructured datasets efficiently, while NoSQL systems provide high responsiveness for semi-structured content. However, this flexibility comes at the cost of increased operational complexity and a steep learning curve for diverse technologies.

Extending this model, a **multi-observatory abstraction layer** can integrate science-ready data products from multiple facilities into a unified archive, enabling **MOLAP-based multiwavelength research** with consistent access to distributed, heterogeneous datasets, optimized and standardizen by Virtual Observatory standards, see Figure 7.

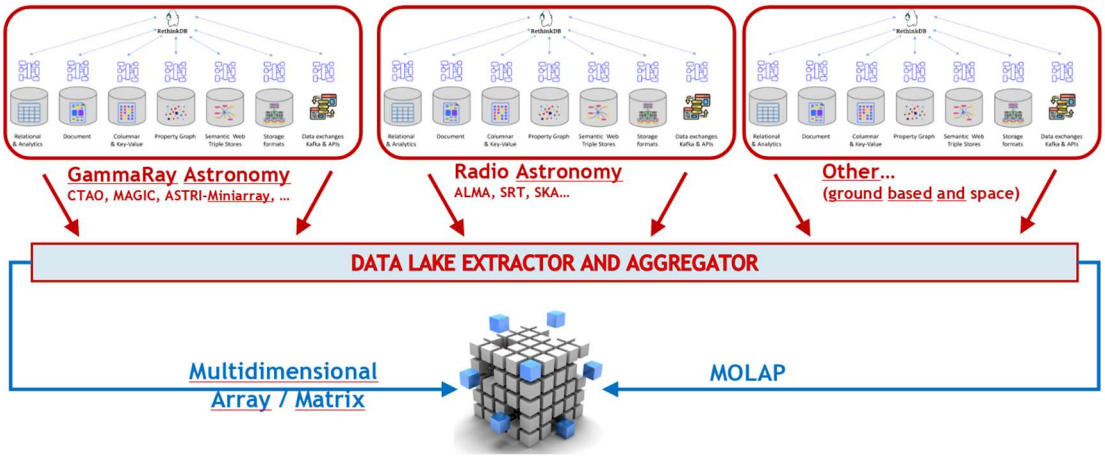


Figure 7. Datalake Extractor and Aggregator for Archive Middleware.

6. Distributed Strategy for a Petascale Astronomical Observatory

Consider a distributed observatory composed of mountaintop telescope arrays, multiple observing sites, and geographically dispersed data centers. Managing tens of petabytes of data annually and enabling broad scientific access—potentially to proprietary datasets—requires an archive system that is scalable, efficient, and responsive, see Figure 8.

 doi:10.20944/preprints202506.0847.v1

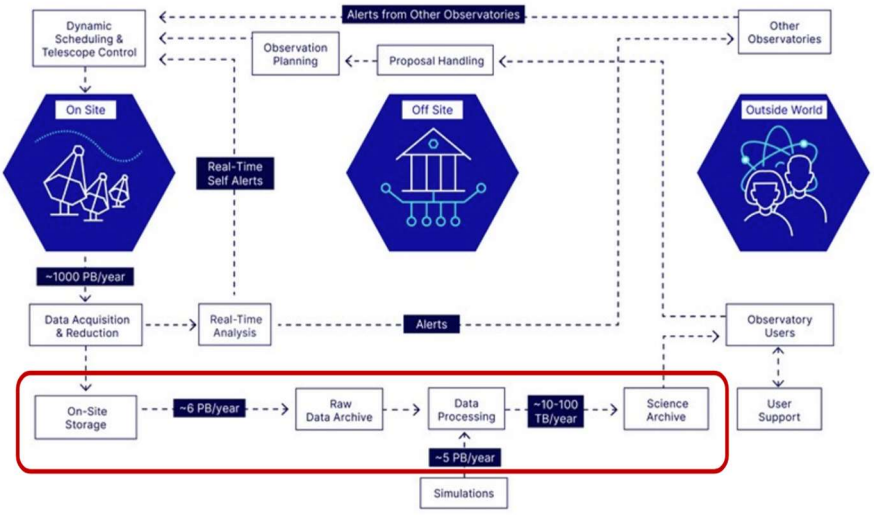


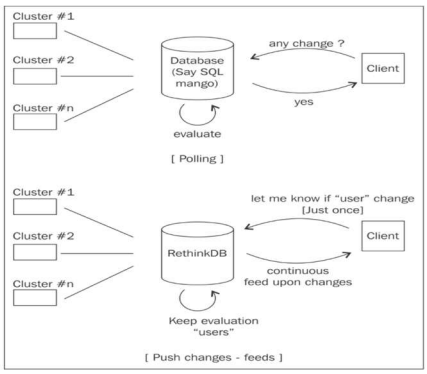
Figure 8. The CTAARCHS is referred to a common approach for the data management, archival and handling of data in the red box. .

As widely described the Database is central in an Archive Solution so taking into account a such distributed scenario, where data is generated on-site and transferred to off-site facilities for long-term storage and processing, the database architecture must mirror the data topology. A **document-oriented, schemaless database** is optimal, given its flexibility and scalability.

For CTAARCHS, several open-source databases were evaluated. While MongoDB and Couchbase were considered, RethinkDB was selected due to its native **change-feed** mechanism, which enables real-time triggers for any database event. This functionality supports near-automated archive operations, reducing human intervention and eliminating the need for resource-heavy polling systems (Fig. 9).

Only Azure Cosmos DB offers similar changefeed support, but RethinkDB provided a more lightweight, open-source alternative with low complexity and ease of deployment.

All other possible database solutions including the proprietary relational ones, do not have this functionality integrated and to develop similar functions it is necessary to imprint a standard polling mechanism, see Fig. 9, that consumes a lot of resources and performs several "not-needed" queries and consequent I/O traffic.



On the CLIENT side:

```
feed = r.table('DLFITS').filter(r.DLFITS['datatype'] ==
"event").changes().run(conn)
for change in feed:
    //print change
    //do action
    try:
      exec_pipeline (change["archive"]["paths"]["PFNP"] + "/" +
change["archive"]["paths"]["PFN"] , PIPE_INIT)
    except:
      print ("ERROR pipelines")
```

Figure 9. Differences between standard polling and changefeed. With the changefeed the client is triggered to execute something for each change in the result of pre-defined query. The polling strategy means to execute the query several times compare with the result of a previous query and if there are differences then trigger an action (+ sleep + redo!).

The recommended configuration involves deploying at least two RethinkDB instances per data center, ensuring local availability, distributed processing, and high resilience (Fig. 10).

 doi:10.20944/preprints202506.0847.v1

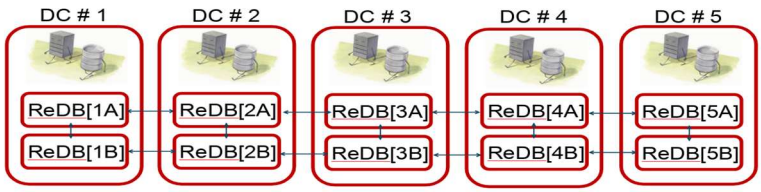


Figure 10. The Database topology cluster of 2 nodes for each datacenter clusterred together. In the figure are shown 5 different datacenters.

7. FAIR Principles and VO Integration in Polyglot Persistence

In modern Polyglot Persistence / Data Lake environments handling heterogeneous data types, the **FAIR principles**—Findable, Accessible, Interoperable, and Reusable—serve as foundational guidelines for enabling data discoverability and reuse. These principles, combined with the **Open Archives Initiative (OAI)**, support metadata standardization and cross-repository interoperability.

To ensure scientific data is interoperable and accessible at the final stage, adherence to **Virtual Observatory (VO)** standards is essential. These standards, defined by the International **Virtual Observatory Alliance (IVOA)**, require metadata to be exposed via TAP services and formatted as VO-Tables. This enables seamless integration with VO tools for accessing and analyzing high-level science products such as multi-wavelength catalogs, spectra, and images. Execution workflows are brokered via standardized APIs (e.g., OpenAPI, REST) and submitted to local resource managers such as Slurm, as shown in Figure 11.

Note: In this paper, depending by the context we use as VO notation both to the **Virtual Observatory** (for public data access) and **Virtual Organization** (for managing access rights and group policies).

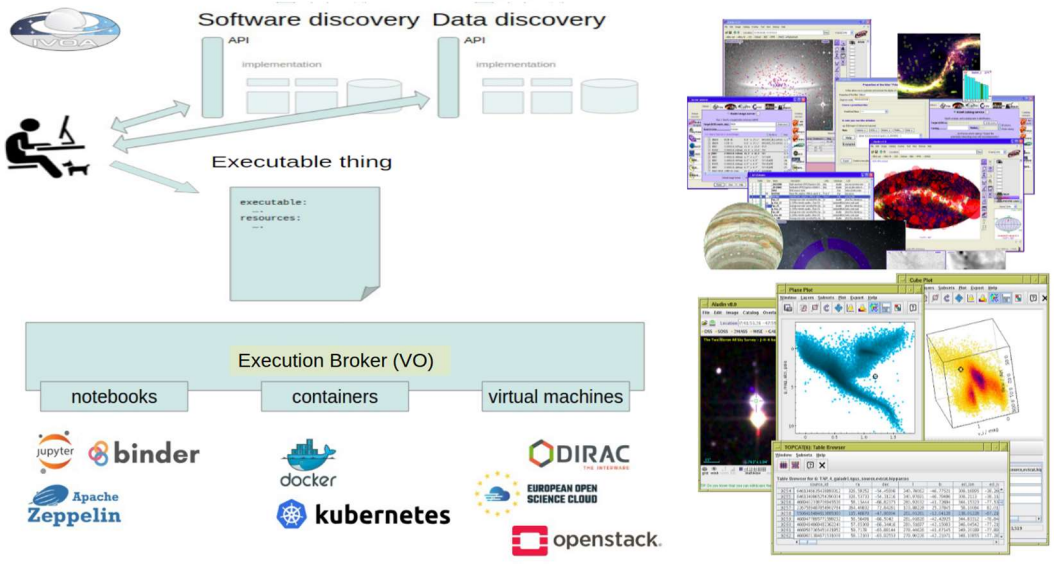


Figure 11. IVOA standards and implementation schema.

8. CTAARCHS Implementation

8.1. Modular Design and Data Transfer Workflow

CTAARCHS provides flexible access to its archive functionalities through multiple modular access interfaces:

- **Command-Line Interface (CLI)**: Executable Python scripts with standardized input/output.
- **Python Library**: Core actions encapsulated in run_action() functions, enabling seamless integration into external applications.

 doi:10.20944/preprints202506.0847.v1

- **REST API**: Web-based access via HTTP methods (POST, GET, PUT/PATCH, DELETE), allowing CRUD operations through scripts or clients (e.g., CURL, Requests).
- **Containerized Deployment**: Distributed as a Docker container (AMASLIB_IO) to ensure platform compatibility and ease of deployment in Kubernetes (K8s) environments.

8.2. On-Site-Off-Site Data Transfer System

In typical observatory setups, raw data is generated on-site and archived off-site. To facilitate this, CTAARCHS implements a dedicated **Data Transfer System (DTS)** with optimized bandwidth, error handling, and transfer resumption via client-server architecture and RPC communication.

The **on-site storage** is treated as a passive element, exposed only to authorized services via secure authentication protocols. This avoids performance bottlenecks and long-term maintenance overhead. Data management and archiving responsibilities reside with **off-site data centers**, integrated into a broader Grid/Cloud/Edge/Fog infrastructure, each with its own Virtual Organization (VO). See Fig. 12 for architecture.

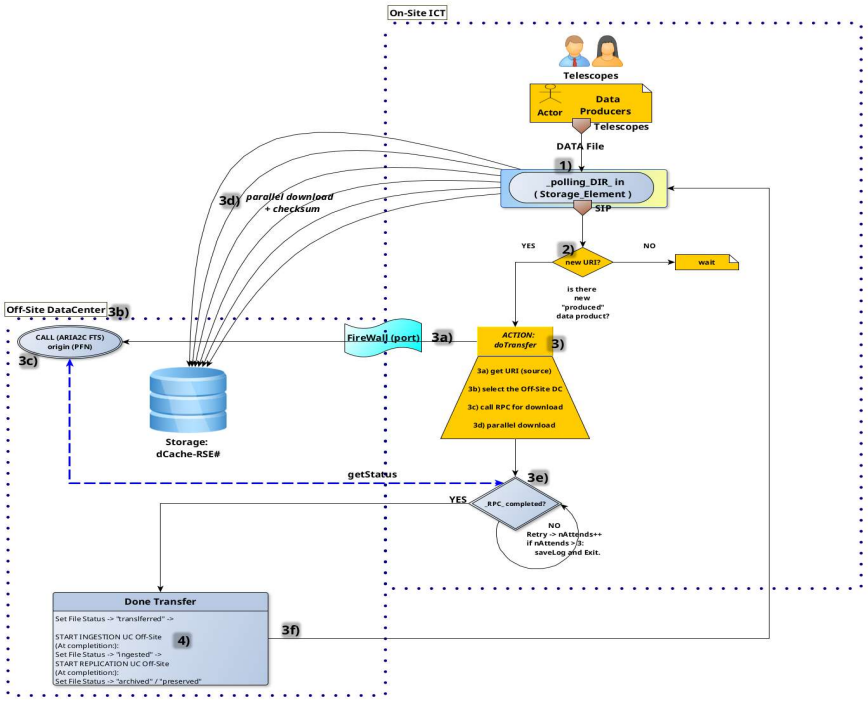


Figure 12. The generic diagram of the file-Transfer from On-Site to Off-Site.

8.2.1. Prerequisites

To enable automated data transfer from observatory sites to archival facilities, the following prerequisites must be met:

A. **Remote Access to On-Site Storage:** On-site storage must be remotely accessible via secure, standardized protocols (e.g., HTTPS or XRootD), with appropriate ports opened between datacenters. This can be achieved through object storage systems or secure web-accessible file directories.

B. File Monitoring and Triggering: On-site storage must monitor a designated _new_data/ directory to detect new files and trigger transfer actions. A lightweight Python watchdog script can monitor for symbolic links—created upon file completion—and initiate transfer, then remove or relocate the link upon success.

C. **Off-Site Download Mechanism:** Off-site datacenters must run an RPC service hosting the Aria2c downloader. Aria2c supports high-throughput parallel downloads, chunking, resume capability, and integrity verification via checksums. A web UI provides real-time monitoring and automatic retries.

 doi:10.20944/preprints202506.0847.v1

Note: While tools like GridFTP or GFAL2 are alternatives, CERN FTS is discouraged due to its complexity, dependency on on-site RSE configuration, and notable failure rates (~10%). Aria2c provides superior control, reliability, and easier integration for this use case.

8.2.2. Typical Workflow

1) **Data Generation:** Telescope systems write data to local storage; upon completion, a symbolic link is placed in _totransfer/.

2) **Trigger Detection:** A local Python client monitors the directory and detects new links.

3) Transfer Initialization:

a) The symbolic link is resolved to a URI.

b) The target off-site datacenter is selected based on policy rules (e.g., time-based, data level, or project ID).

c) The client invokes an RPC command to the off-site **Aria2c service**, initiating parallel downloads.

d) Transfer progress is tracked, and completion is confirmed via RPC status queries.

e) Upon success, the symbolic link is removed.

4) **Post-Transfer Actions:** Additional use cases, such as replication or data ingestion, can be triggered automatically on the off-site side.

8.3. Dataset Ingestion

The ingestion process must adhere to the **Open Archival Information System (OAIS)** model, which requires that only verified and validated data products be archived. This mandates a structured, pre-ingestion validation phase, where data integrity and metadata completeness are confirmed before registration and for ingesting datasets minimal **Data Product Acceptable Requirements (DPAR)** are applicable (i.e. checksum, fits header format and content vieriefied). These verification steps can not be postponed to an on the fly registration since the file catalog can be affected only when the data-product is ready to be registered/stored, even for temporary data, see Figure 13.

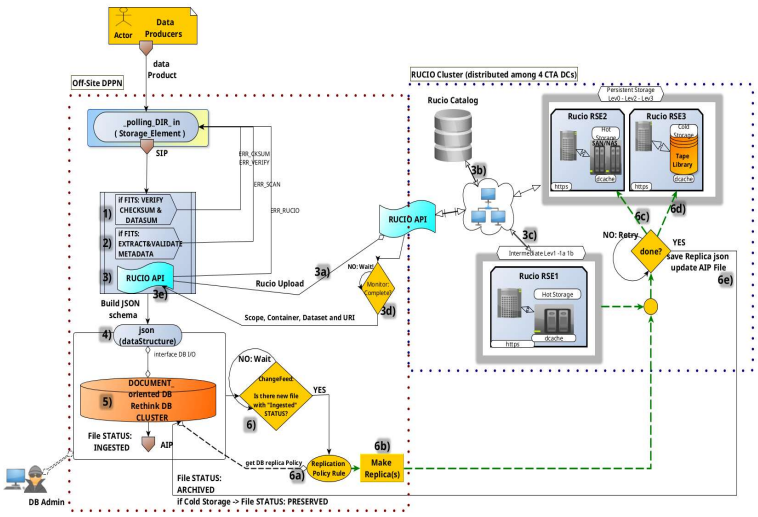


Figure 13. Generic Ingestion UC and Replica if ingestion is successful and data policy needs a replica for the corresponding data product.

8.3.1. Prerequisites

A. The _toingest/ **storage-pool** directory must be POSIX-accessible, even if hosted on object storage.

B. Python environment must include fitsio (or astropy), json, rucio, and rethinkdb libraries.

C. The external storage endpoints called **Remote Storage Elements (RSEs)** must be accessible via standard A&A protocols (e.g., IAM tokens or legacy credentials).

 doi:10.20944/preprints202506.0847.v1

D. A write-enabled **RethinkDB node** must be reachable on the local network.

8.3.2. Typical Workflow

1) D**ata Staging:** Data products from Data Producers (pipelines, simulations, or DTS) are placed in _toingest/.

2) **SIP Creation:** A Software Information Package (SIP) is generated, including checksums to verify file integrity.

3) **Metadata Validation:** FITS headers are parsed and validated to ensure required metadata fields are present, correctly typed, and semantically consistent.

4) 1 Storage Upload:

a) Files are uploaded to an Object Storage path (e.g., dCache FS) using RUCIO or equivalent tools.

b) If already present on the storage, only a move to a final archive path is needed.

c) Upload status is monitored; once confirmed, metadata (e.g., scope, dataset, RSE) is added to a corresponding JSON record.

Alternative: Use gfal2 to upload directly, guided by storage protocol settings in the ReThinkDB StoragePool collection.

5) **Database Registration:** Finalized JSON is ingested into the RethinkDB archive, changing file status to "ingested" and completing the Archive Information Package (AIP) creation.

6) **Trigger Replication:** Upon new entry detection (via RethinkDB's changefeed), the MAKE_REPLICA process is automatically launched.

8.4. Replica Management in CTAARCHS: Automation and Policy Enforcement

As part of the data ingestion process (point n.6), **automated replication** ensures compliance with redundancy and long-term preservation policies. Triggered via a **change-feed** from the ReThinkDB file catalog, the replication logic references a DATA_POLICY_REPLICATION table to determine the required number of copies per data type and storage level.

If no policy rule is found, the data product is assumed to be for temporary processing only. Policies define replication support types (e.g., hot, cold, or hot+cold) and preservation intent. This mechanism fulfills key archival use cases such as **tracking preservation state** and **monitoring physical data locations** across distributed storage resources.

8.4.1. Replication Status Levels

- **Ingested**: One off-site catalog record exists.
- **Archived**: At least one replica stored across another RSE.
- **Preserved**: Includes a backup on cold storage.

Each replication rule specifies the data type, number of required replicas, and preferred storage configuration. Example:

Any record of the DATA_POLICY_REPLICATION table is called **"Replication Rule"**, here is an example:

{ "ruleid": "1" , "rulename": "AMAS_dl0-raw" , "datatype": "dl0.raw" , "replica_lev": "2", "rule": "preserve" , "supports": "hot+cold", "timeseries": [{"RSE1": "jan-mar" }, {"RSE2": "apr-jun" },{"RSE3": "jul-sep" },{"RSE4": "oct-dec" }] }

{ "ruleid": "1" , "rulename": "AMAS_dl0-fits" , "datatype": "dl0.fits" , "replica_lev": "3", "rule": "preserve" , "supports": "any"}

{ "ruleid": "2" , "rulename": "AMAS_dl1-fits" , "datatype": "dl1[a-c].fits" , "replica_lev": "1", "rule": "ingest" , "supports": "any"}

{ "ruleid": "3" , "rulename": "AMAS_dl3-fits" , "datatype": "dl3.fits" , "replica_lev": "3", "rule": "ingest" , "supports": "any"}

A generic UML of the Make_Replica is shown in the Fig. 14.

8.4.2. Prerequisites

 doi:10.20944/preprints202506.0847.v1

A. All target RSEs must be reachable over secure protocols (e.g., HTTPS, xrootd), and relevant ports must be open across data centers.

B. The ReThinkDB cluster must support read/write access from local clients.

C. Each off-site RSE must run an **ARIA2c RPC daemon** for parallel downloads and transfer monitoring.

8.4.3. Typical Workflow

0) Data coming from Data Producers generates a change in the DB cat.

1) Ingestion completion updates the file catalog, triggering the replication process via the changefeed.

2) The client fetches the file's URI (2a), matches it against the replication policy (2b), and evaluates eligible RSEs based on latency, throughput, and availability (2c).

3) It initiates **parallel data transfers** using ARIA2c RPC (3a) and monitors each transfer (3c).

4) On success, the checksum is verified, a new replica record is added to the file's JSON metadata, and the replica count is updated.

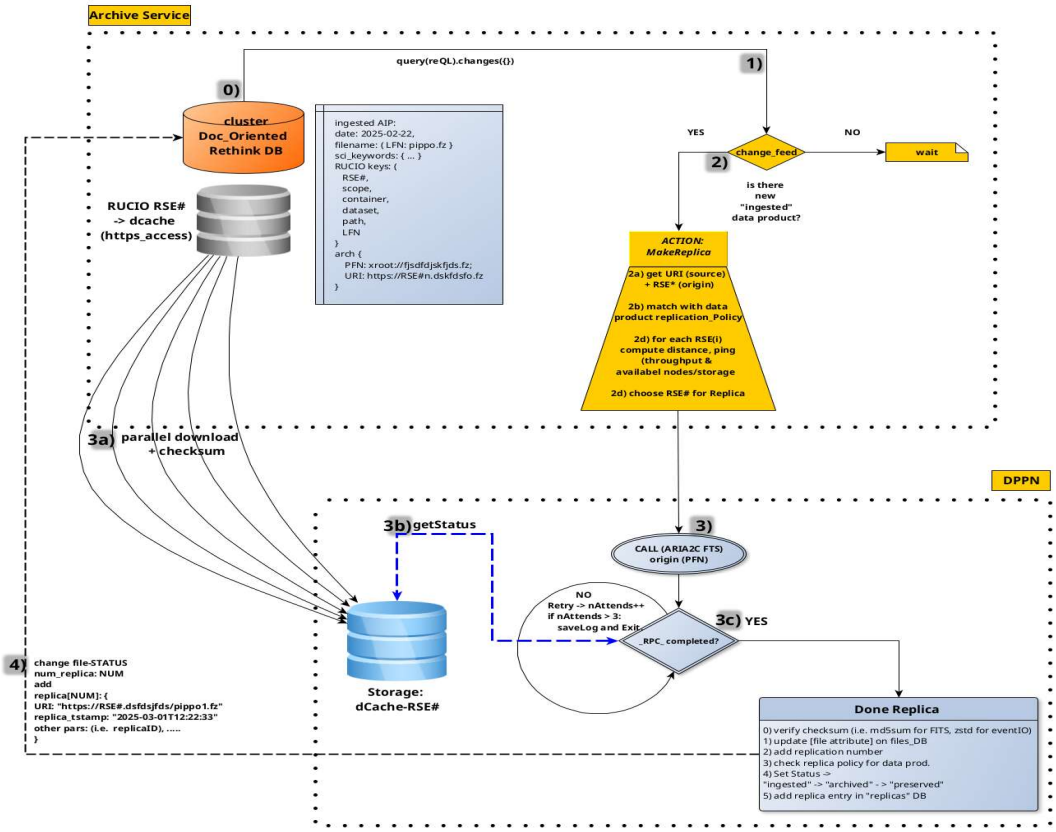


Figure 14. The detailed operation of Make Replica task.

8.5. Dataset Search

Once a data product is ingested—regardless of its archival status ("ingested", "archived", or "preserved")—its metadata becomes searchable through the **ReThinkDB catalog**. This enables external users to retrieve dataset identifiers and associated replica information, see Figure 15.

 doi:10.20944/preprints202506.0847.v1

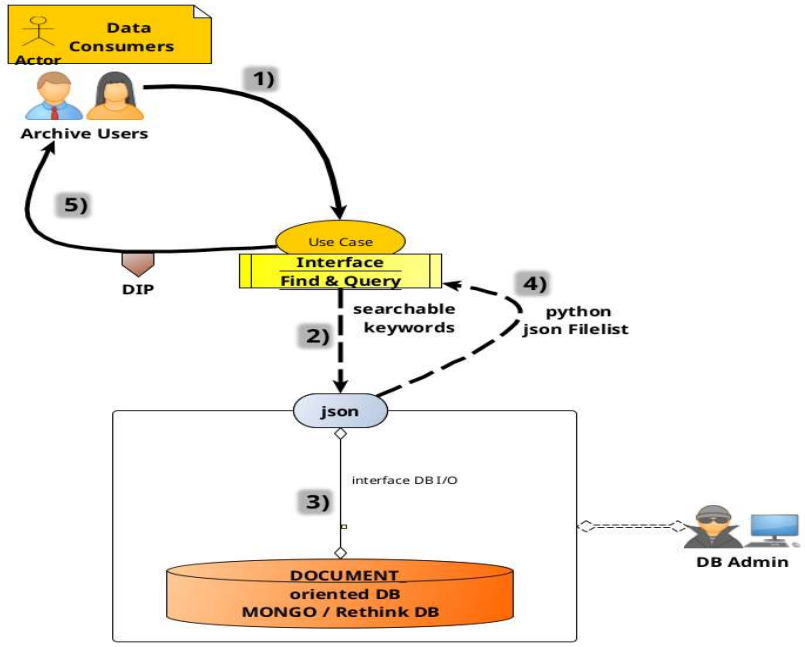


Figure 15. Generic search for a Dataset use case UML.

8.5.1. Prerequisite

A. Read-only access to the ReThinkDB cluster must be available from at least one node in the local network.

8.5.2. Typical Workflow

1) A user submits a query via the archive interface, specifying metadata fields of interest.
2) The interface maps the request to searchable metadata intervals.
3) It then queries the ReThinkDB cluster through a local node.
4) The database returns a list of matching data products in JSON format, including URIs and identifiers.
5) This list is delivered to the user for potential retrieval.

Note: This process is typically followed by the "Retrieve" use case.

8.6. Dataset Retrieval

Once a dataset is ingested into the archive, regardless of its status (ingested, archived, or preserved), external users can query the RethinkDB metadata catalog to retrieve corresponding datasets and their available replicas. This process involves querying the catalog for metadata, translating the request into predefined searchable metadata intervals, and executing the query via a local node connection. The database returns a JSON file list containing URIs and identifiers of data products matching the query criteria, which are then provided to the user, see Fig. 16 for a generic workflow.

 doi:10.20944/preprints202506.0847.v1

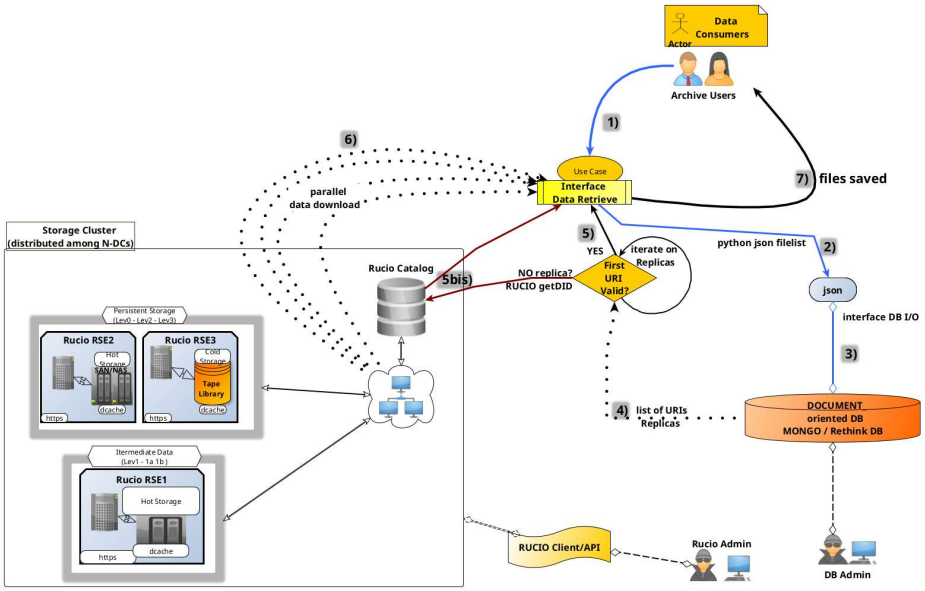


Figure 16. Generic UML for local data product retrieval.

8.6.1. Prerequisites

A. **Remote Storage elements (RSEs)** must be accessible across data centers via secure protocols (e.g., HTTPS, XRootD), with required ports open. Resources may be object storage pools or directories exposed via HTTPS with encryption and authentication.

B. The **RethinkDB cluster** must be accessible in read-write mode from at least one node within the local network.

8.6.2. Typical Workflow

1) A **Data Consumer** provides a JSON list of requested data products to the retrieval interface.
2) The system queries the local RethinkDB node
3) The database returns a list of replica URIs for each product
4) The interface verifies the existence of each replica
5) Valid URIs are downloaded in parallel

5bis) if no URI from the replica list is available the system calls the RUCIO catalog to get DID (filename+scope) and find in the RUCIO catalog

6) The parallel download starts for any available URI
7) Retrieved files are stored in a user-specified local or remote directory.

8.7. Search and Retrieve Integration/Concatenation

Search and Retrieve are often combined as a single use case, chaining Python methods to locate metadata and then download the associated data products efficiently, see fig. n.17.

```
cedadm@amas:/var/www/amas/amas/static/rethinkdb$ ./search.py MINIARRAYDB DQC -k RUN -v 900 | ./retrieve.py STDIN -o testdir/ --force
2025-04-14 14:35:41 - INFO - 1 (0.13 MB)]: -> copy 20250127_MA01_Crab_W2.50p000_00000900_R_001828_dqc_plot.lv0.pdf  -> testdir//
2025-04-14 14:35:41 - INFO - 2 (0.03 MB)]: -> copy 20250127_MA01_Crab_W2.50p000_00000900_R_001828_dqc_plot.lv1c.pdf  -> testdir//
2025-04-14 14:35:41 - INFO - 3 (0.0 MB)]: -> copy 20250127_MA01_Crab_W2.50p000_00000900_R_001828_0601.lv1.fits.gz  -> testdir//
2025-04-14 14:35:41 - INFO - 4 (0.06 MB)]: -> copy 20250127_MA01_Crab_W2.50p000_00000900_R_001828_dqc_superplot.pdf  -> testdir//
2025-04-14 14:35:41 - INFO - 5 (0.04 MB)]: -> copy 20250127_MA01_Crab_W2.50p000_00000900_R_001828_dqc_plot.lv1b.pdf  -> testdir//
2025-04-14 14:35:41 - INFO - 6 (0.05 MB)]: -> copy 20250127_MA01_Crab_W2.50p000_00000900_R_001828_dqc_hist.lv1b.pdf  -> testdir//
cedadm@amas:/var/www/amas/amas/static/rethinkdb$
```

Figure 17. Find & Retrieve concatenation, as expressend by the pipe-concatenation of two python functions exported by CTAARCHS py-library.

The search.py utility interfaces with the RethinkDB cluster to locate data products based on metadata queries. Depending on the execution context, results may point to internal POSIX paths, external URIs, or RUCIO-based identifiers (RSE + LFN + SCOPE).

 doi:10.20944/preprints202506.0847.v1

The advanced AMAS Search Interface exposes a REST API via a dedicated web server, supporting fast and complex metadata-based queries across distributed data centers. Users can execute searches from any location or pipeline stage, provided they have network access.

A typical query can be executed with a simple curl command, specifying key-value filters such as date, run number, or filename, see Figure 18.

```
LIST=$(curl "https://amas-rest/search?key=DATE&val=2025-01-15:2025-02-
26&key=RUN&val=846:895&key=FILE&val=20250120_MA01_OffFixed")
echo $LIST
{
  "files": [
    "https://amas.oa-roma.inaf.it/static/data/Miniarray/.../20250120_MA01_OffFixed-60-
015_Fixed_00000849_I_001761_1001.lv0.fits.gz",
    "... more URIs ..."],
  "nfiles": 9
}
```

Figure 18. AMAS REST API in action.

The dataset search returns a JSON-formatted file list containing URIs pointing to RSE storage locations. Access typically requires user authentication.

The retrieve.py interface reads this list (e.g., from STDIN), then downloads the corresponding files to a user-specified directory. It connects to the local RethinkDB node using read-only credentials to fetch replica metadata.

To optimize performance, the system dynamically selects the most efficient replica for each file using a "**down-cost**" algorithm. This decision is based on several site-specific parameters:

- **Cost(i)**: Estimated retrieval cost from site i
- **Latency(i)**: Time to initiate transfer
- **FileSize**: Total size of the file
- **Throughput(i)**: Nominal data rate
- **Workload(i)**: Current system load (0 = idle, 1 = saturated)
- **Distance(i)**: Network or geographic distance

These parameters are used to minimize download time and network usage. Workload reflects real-time system strain, while throughput, latency, and distance help assess the optimal retrieval path—especially important in geographically distributed storage systems or under regulatory constraints. Distance could be affected by latency or used explicitly if needed for geo-pinning or regulatory concerns.

$$\text{Cost}_i = \alpha \cdot \left(\text{Latency}_i + \frac{\text{FileDimension}}{\text{Throughput}_i \cdot (1 - \text{Workload}_i)} \right) + \beta \cdot \text{Distance}_i$$

The optimal replica for download is dynamically selected by computing the retrieval cost (Cost_i) in real time. The replica with the lowest cost is chosen, and its URI is returned. Final access requires authentication and authorization.

Latency_i is easily measured via network ping; Throughput_i and Distance_i are typically available from infrastructure documentation. Estimating Workload_i, however, is more complex and can be approximated by comparing the $\text{MeasuredThroughput}_i$—from a small test download—to the $\text{NominalThroughput}_i$.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 June 2025 doi:10.20944/preprints202506.0847.v1

$$\text{Workload}_i = 1 - \left(\frac{\text{MeasuredThroughput}_i}{\text{NominalThroughput}_i}\right)$$

```
if latency > threshold or throughput < expected * 0.5:
      Workload_i = 0.8            # heavy
elif throughput < expected * 0.8:
      Workload_i = 0.5            # moderate
else:
      Workload_i = 0.1            # low
```

```
#! /usr/bin/python3.10

from rethinkdb import r
########CONNECT TO DB##########
r.connect(host='IP_or_HOST_ReDBNODE', port=28015, db=database, user='guest', password='guest').repl()
#select RUN_v
prink=r.db(database).table(table).filter((r.row['runid'].eq(int(value)))).pluck({'archive': {'PFNP': True, 'PFN': True, 'paths': {'uripath': True }}})
for pippo in prink.run():
    try :
        filetto=(str(pippo["archive"]["PFNP"]) + "/" + str(pippo["archive"]["PFN"]) )
        if not os.path.isfile(filetto):
            filetto=str(pippo["archive"]["paths"]["uripath"])
        print(filetto)
    except:
        print(f"Error database!")
        sys.exit(1)

sys.exit(0)
```

Figure 19. Few lines of code to connect and retrieve data from a RethinDB node using a simple filtering search.

The interfaces are readily accessible via the amas-api_1.0.2 **docker image**.
With Docker installed, users can deploy the environment using the following minimal setup:

```
docker load -i amas-api_1.0.2.tar;
docker run -it amas-environment bash;
./venv/bin/python ./search.py
```

8.8. Monitor Integrity, Reports and Alarms

RethinkDB enables usage statistics, logging, and failover reporting. Entry-level metrics integrate easily with monitoring tools such as Grafana or NetData via customizable dashboards. Data transfer performance, tracked through the aria2c WebUI, can be logged in the replicas collection. These records support straightforward analytics and visualization, see Figure 20.

Figure 20. Different Monitoring and Alarm systems.

9. Deployment of CTAARCHS at CIDC and AMAS

Deploying a data center requires careful planning to ensure efficiency, scalability, and security. At the CTA Italy Data Center (CIDC), deployment of the ASTRI and Miniarray Archive System followed a structured strategy aligned with observatory goals and technical constraints. Emphasis was placed on building a secure, scalable infrastructure, minimizing risks while supporting

 doi:10.20944/preprints202506.0847.v1

operational demands. Initial phases included logical and physical design—rack layouts, network topology, cooling, and power—supported by the Tier-2 facility at INFN Frascati, where CIDC is currently hosted, see Figure 21.

Figure 21. The INFN-LNF LHC tier2 where CIDC is located.

9.1. Hardware Resources

The CTAARCHS implementation is based on the AMAS archive system, supporting the ASTRI-Horn prototype and the nine-telescope ASTRI Miniarray at Teide Observatory, Tenerife. AMAS represents the complete off-site infrastructure for these projects and serves as the technical deployment of CTAARCHS.

Built on the CTAARCHS/AMAS IaaS, the CTA Italy Data Center (CIDC) forms one of four designated off-site data centers for the CTAO Project (see Fig. 22). Hardware requirements for computing and storage are defined annually by each project office and reflected in a procurement plan for 2025–2026.

Software services follow a Continuous Integration/Delivery (CI/CD) model, with the exception of the archive system, which must be accessible from project initiation. Archive deployment is coordinated with collaboration partners and adapted through a virtualized abstraction layer.

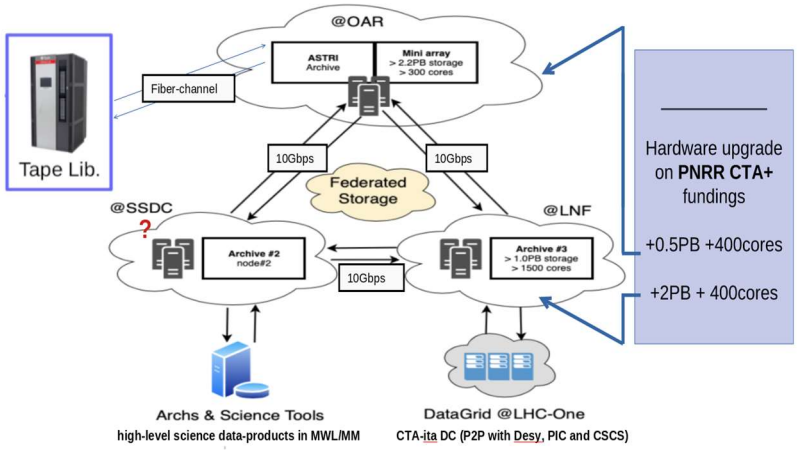


Figure 22. The Hardware topology of AMAS.

The AMAS implementation of CTAARCHS relies on shared hardware located in mainly three sites:

1. INAF – OAR, Astronomical Observatory of Rome
2. INAF – SSDC, ASI Science Data Center

 doi:10.20944/preprints202506.0847.v1

3. INFN – LNF, National Laboratoies of Frascati

In total AMAS hardware list consists in a federated distributed "hot" storage of **6PB** (directly upgradable to 10PB), around 10PB (directly upgradable up to 100PB) of "cold" storage (Fiber Channel Tape Library); an HPC@OAR consisting in about **800**-cores (8.8 kHS06) with ~1TB RAM and a grid HTC@LNF consisting of about **1400-**cores (15.4 kHS06) with ~2.5TB of RAM. In SSDC are foreseen only minimal services and resources not listed here for sharing MWL data.

9.2. The Setup

Datacenters can join the CTAARCHS environment by registering to access repositories of Docker containers, virtual machines, and Kubernetes (K8s) orchestration for various services.

The K8s clusters at INAF-OAR and INFN-LNF sites share resources within the ReDB "resource_pools" collection, managing storage, computing, services, and user registrations. The distributed RethinkDB cluster spans multiple sites—OAR (DC1), LNF (DC2), and SSDC (DC3, pending activation)—as illustrated in Fig. 10.

9.3. Users Interfaces

Main common archive users are basically:

- **Pipeline/Simul** (for low level data products)
- **Science User** (for higher level data products)
- **BDMS-user** and **admin** (for high level operation on archives)

9.4. Pipeline / Simulation Users Access and Interface

Users access data via different tools and workflows. Simulation and Pipeline users employ **Workload Management Systems (WMS)** like DIRAC or PANDA to run DAGs on grid computing or HPC queues (e.g., Condor, Slurm), interacting with off-site Object Storage. The latest approach envisions a Kubernetes-based **Computing Element Service (CES)** to orchestrate queues and manage virtual organizations and authorization. However, current WMS like DIRAC and PANDA are not yet adapted for Kubernetes.

Simulation users typically write output directly to Object Storage for asynchronous ingestion, while Pipeline users first query the archive for input datasets using metadata searches, then process data close to storage locations to minimize transfers (this task is describein in the "Search" use case). All I/O operations must strictly follow use cases (UC) without customization; if a WMS cannot comply, it must be adapted or replaced, rather than altering the archive design.

9.5. Unconventionl Challenges

International collaborations face challenges due to political mandates to use pre-existing systems or software deloped by IKC and used for other datamodels and/or scientific scenarios. For instance RUCIO Data Management System and/or DIRAC for Workload Management System impose to CTAARCHS several limitations. These software often become single points of failure (SPOF) in a no-SPOF infrastructure, forcing inefficient archive adaptations and violating OAIS principles that mandate strict separation between data producers, consumers, and archive submodules through standard interfaces. Modify the Archive requirements to adapt to these limitations becomes detrimental to the continuation of a good collaboration.

For example, RUCIO suffers from SPOF in its centralized PostgreSQL catalog and is complex for multi-institutional sharing due to its fixed CERN-centric data model, leading to storage overhead and high operational costs. A natural antagonist of RUCIO is the **OneData**, which is a distributed data management system too, designed to integrate diverse storage resources, facilitating seamless data access and sharing across institutions. Differently by RUCIO, OneData offers a storage federation model based on a distribute document-oriented database model, being based on distributed, document-oriented DB cluster (i.e. CouchBase) it offers a storage federation that better supports metadata management, open data, and collaboration, aligning with Open Science goals (see Table 5). Choosing storage federation technology to serve an astronomical observatory community should prioritize technical effectiveness and use case fit over political or economic pressures.

 doi:10.20944/preprints202506.0847.v1

Table 5. Comparison of RUCIO and OneData storage federation softwares.

Feature	RUCIO	OneData
Main Use Case	Scientific data management of CERN experiments (e.g., ATLAS)	Distributed data access and sharing.
SPOF?	Yes (Centralized Relational Catalog)	No (Distributed DB Catalog)
Data Sharing	Requires data duplication for cross-institution sharing	Supports federated access without data duplication
Integration Flexibility	Limited: Specialized for scientific workflows and fixed Data models	Advanced: Designed for integration with different workflows and Data models
Metadata Management	Basic Support	Advanced metadata handling with multiple formats
Open-Data Support	Limited	Strong support with integration to open data standards, IVOA, etc

Finally because of the several points of failure involved with RUCIO environment it is clear that NO PERSISTENT ARCHIVAL SERVICE can be dependent by a potentially unstable archival software without the possibility to have a "plan-B" ready and usable.

So we need to deprecate the wide use of RUCIO as central storage system for a good archive and we auspicate to relegate it only as marginal common interface because it is optimized for different storage elements protocols.

Throughout this work, the term **RSE (Remote Storage Element)** is used generically to denote any remote storage resource accessible via standard protocols, independent of the RUCIO framework.

9.6. Database and DataModel Interfaces

Intermediate and end users may require direct access to metadata for scientific analysis or simulation output. To support this, a dedicated read-only user role enables querying across all data levels. For FITS files, primary headers are indexed within the data model, allowing advanced search capabilities. A sample data model and query interface are provided in the appendix (see Fig. 23), with customizable code available for tailored use cases.

The code sample is similar to those used for the Find & Query interface Client but can be espressly customized on demand. See Fig. n.23.

 doi:10.20944/preprints202506.0847.v1


```
#! /usr/bin/python3.10

from rethinkdb import r
########CONNECT TO DB###########
# SSL context setup
ssl_opts = {
    "ca_certs": "/path/to/ca_cert.pem",  # optional
    "certfile": "/path/to/client_cert.pem",
    "keyfile": "/path/to/client_key.pem",
    "cert_reqs": None  # Can be ssl.CERT_REQUIRED if you're validating CA
}
# Connect with SSL
conn = r.connect(
    host='your.rethinkdb.server.node',
    port=28015,
    ssl=ssl_opts, #if SSL connection is needed
    user='guest',
    passwd='guest',
    db='MINIARRAYDB'
)
# Run a query SELECTING A FILENAME
try:
    cursor = r.table('DLFITS').filter({'filename': "20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz"}).run(conn)
except:
    print(f"Error database!")
    sys.exit(1)
#####ALL META-DATA MODEL in "cursor"#####
for data in cursor:
    print(data["archive"]["paths"]["uripath"])  #print file URI path
    print(data["header"]["Primary"]["RA_OBJ"])  #print RA Target
    print(data["dateobs"])                      #print DateObs

conn.close()

sys.exit(0)
```

Figure 23. Sample code to obtain the FULL metadata content of an DB entry once selected and filtered the dataset in the "data" structure are stored ALL metadata coming from the json record. Using any kind of python function it is possible to scan and filter again the "data" json-structure.

RethinkDB supports the creation of secondary indexes on metadata fields, enabling faster queries as datasets grow. This feature is simple to implement, with no strict limits on the number of indexes, making it highly effective for optimizing search performance over time.

```
r.table("DLFITS").index_create("dateobs").run(conn) #CREATE INDEX
r.table("DLFITS").index_wait("dateobs").run(conn)    #WAIT COMPLETITION
# Query using the index
r.table("DLFITS").get_all("2024-12-06", index="dateobs").run(conn)
```

9.7. Web Archive Portal for the End-user and Other Interfaces

Science users—primarily researchers accessing high-level data products—interact with the archive via a dedicated web portal. These users are considered as Data Consumer and are planned to retrieve level-3 datasets in read-only mode to conduct analyses or run customizable pipelines. Data dissemination relies on the distributed database, with pipeline execution triggered by change-feed mechanisms monitoring the level-3 collection (see Fig. 24).

 doi:10.20944/preprints202506.0847.v1

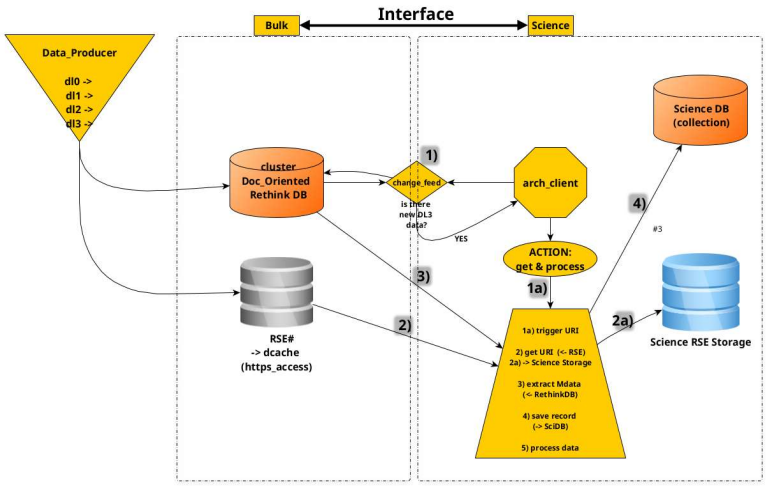


Figure 24. Standard interface from Bulk (<=dl3) Archival to Science (>=dl3) archival. This interface triggers the ingestion/process up to dl3.

9.7.1. Prerequisites

A. Python environment must include fitsio (via Astropy), json, rethinkdb, and rucio libraries.

B. **Bulk and Science RSEs** must be accessible via supported authentication methods: IAM tokens (preferred), legacy grid certificates (deprecated), or credentials.

C. The **ReThinkDB** cluster must be accessible in read-only mode via at least one local node.

D. The Science Database may reside within ReThinkDB or any compatible RDBMS.

9.7.2. Typical Workflow

0) A pipeline processes data and ingests new DL3 products into the archive.

1) Detection of new DL3 entries triggers the get&process action.

2) The associated URI is fetched from the source RSE and transferred to the Science RSE.

3) DL3 metadata are extracted from ReThinkDB and written to the Science DB.

4) Optional automated workflows convert DL3 to DL4 and DL5 products.

Note: Since higher-level science data (DL3–DL5) involve smaller volumes, they may be handled via lightweight solutions such as local Airflow DAGs and executed on dedicated clusters (see Fig. 25).

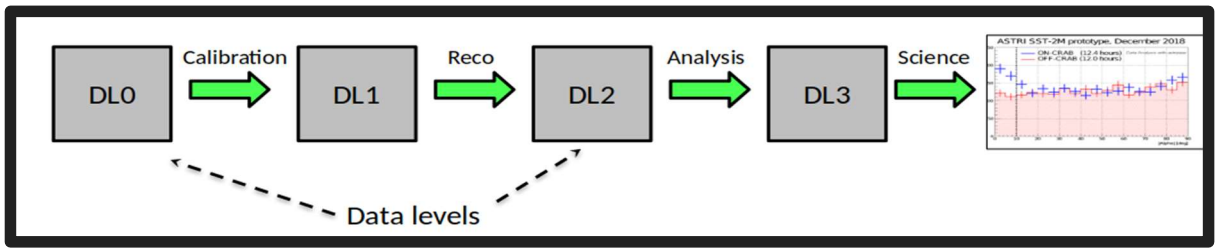


Figure 25. Simple processing to pass from DL0 to Science Data.

Community LDAP or VPN access enables shared resource usage and supports defining Airflow pipeline steps. The Search and Retrieve Python APIs remain functional but require read-only access to the ReThinkDB cluster. Alternatively, REST-API endpoints can be used to bypass direct database access (see Fig. 26).

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 June 2025 doi:10.20944/preprints202506.0847.v1

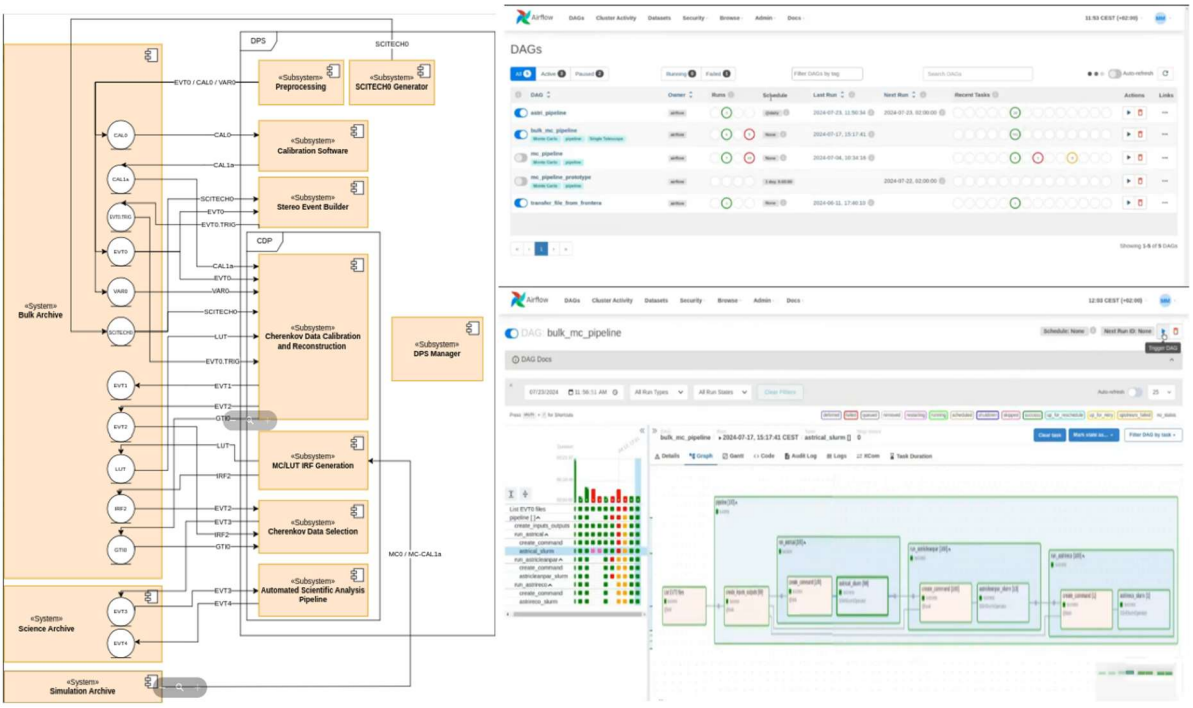


Figure 26. A simple Airflow implementation for higher level processing.

The low level processing, for huge amount of data can be easily shared and distributed among DPPN datacenters, while the science processing can be easily concentrated in one site using a dedicated slurm queue and an AIRFLOW DAG authomatic processing.

The output Science RSE can benefit of a localized access dedicated only to scientific end-users passing through Web Portal to browse and access proprietary "proposals" data or trhough a web - Gateway facility sharing a user-defined policy repository buckets on Cloud-based Storage utility like Min-IO.

Note: A scientific end-user data access can not rely on complicated grid-based data I/O access like IAM (grid-based certificates/tokens for authentication) required for low-level big-data processing. So a cloud based approach like amazon-AWS (i.e. A customized MinIO facility) gives the end user a very simple access customized on a common LDAP authenticaiton (login+password) and permits access to proprietary data products using standard posix and REST api access, as well as mount and share local storage areas for analysis and collaboration within research groups.

A simple implementation for High Energy Astronomical Archives has been realized for the **ASTRI Project** in the **AMAS, ASTRI and Miniarry Archive System**, containing the Proposal Hangling System, a Observing Scheduler and Planner a Data Web-base Access for any scientific data-levels and also for logs/alarms, housekeeping, data quality checks and quicklook visualizator. The common usage foresee the end-user to access through a collaboration VPN access and share local-cluster facility, using Find/Search queries and data-Retrieval and using a user/defined namespace local bucket shared on a MinIO insfrastructure Interface.

The bulk data processing is distributed among 3 different nodes and the archive system is distributed too; the science processin instead is principally driven through an AIRFLOW facility running on the top of a SLURM HPC queue in the OAR cluster.

As shown storage and computing resources are build on **AAAS (Astronomical Archive As a Service)** paradigm applied to Astronomical use-case; the resulting infrastructure is easily horizontally scalable as well upgradable without out-of services when needed. See Fig. n. 27 for current implementations.

 doi:10.20944/preprints202506.0847.v1

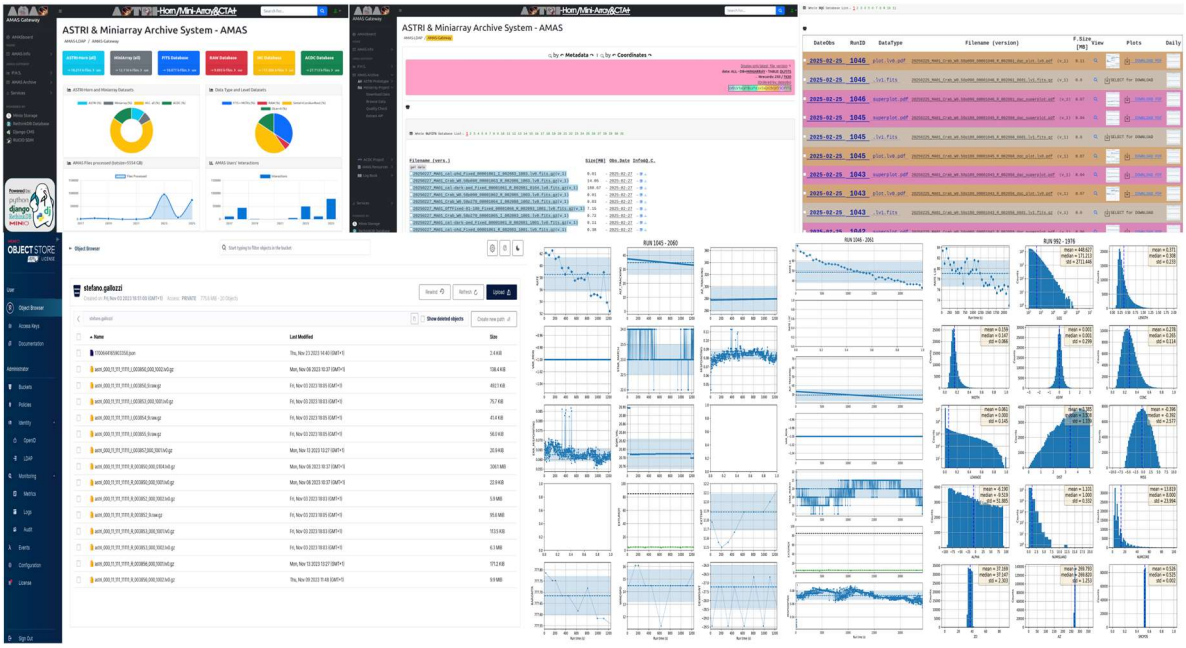


Figure 27. AMAS web Portal implementation Database Browsing, Web Interfaces and AMAS implementation of MinIO - Amazon AWS Cloud Storage System and plots of AMAS Data-Quality Check.

10. Conclusions and Recommendations

The definition of Archive Solutions to be adopted in wide-range Scientific Collaborations is a crucial step for the success of a project, especially in the astronomical fields where, differently by nuclear and sub-nuclear particle experiments, the number of end-users is several orders of magnitude greater.

Although political choices can be made and pushed on the basis of pre-existing economic and technological contributions, choosing the best technologies to assemble the most efficient system for the project's use cases is the most important obstacle to overcome and depends extremely on the project management capabilities of the different teams (working groups) identified to assemble the different modules/packages.

In our CTAARCHS we present a feasible and versatile implementation of all Archival and Data Management ecosystem needed for an astronomical observatory use-case, with a set of possibilities or alternate scenarios and equally valid technological choices.

10.1. Software Resources & Repositories

In this section are summarized a small list of CTAARCH Software Packages, Modules, Resources and Repositories included third parites Packages.

10.2. CTAARCH & AMAS

- Homepage: https://amas.oa-roma.inaf.it
- Repo: https://www.ict.inaf.it/gitlab/AMAS/
- Docker API: https://www.ict.inaf.it/gitlab/AMAS/dockerAPI/
- Py-API: https://www.ict.inaf.it/gitlab/AMAS/pyAPI
- REST-API: https://amas-rest.oa-roma.inaf.it

10.3. RethinkDB

- Homepage: https://rethinkdb.com
- Repo: https://hub.docker.com/_/rethinkdb/
- Py-API: https://rethinkdb.com/docs/install-drivers/python/

10.4. Minio

 doi:10.20944/preprints202506.0847.v1

- Homepage: https://min.io/
- Repo: https://min.io/download?view=aistor
- Client: https://min.io/docs/minio/linux/reference/minio-mc.html

10.5. Airflow

- Homepage: https://airflow.apache.org/
- Repo: https://airflow.apache.org/docs/apache-airflow/stable/installation/index.html
- REST-API: https://airflow.apache.org/docs/apache-airflow/stable/stable-rest-api-ref.html
-

HTCondor

- Homepage: https://htcondor.org/
- Repo: https://github.com/htcondor
- REST: https://htcondor.readthedocs.io/en/latest/apis/python-bindings/api/htcondor.html and https://github.com/htcondor/htcondor-restd

OneData

- Homepage: https://onedata.org/#/home
- Repo: https://onedata.org/#/home/api/stable/onezone and https://github.com/onedata/getting-started
- RESTful: https://docs.webmethods.io/on-premises/webmethods-onedata/en/10.7.0/onedata-webhelp/index.html#page/onedata-webhelp/to-rest_services_13.html

10.6. Rucio

- Homepage: https://rucio.cern.ch/
- Repo: https://github.com/rucio/rucio
- REST-API: https://rucio.cern.ch/documentation/html/rest_api_doc.html

10.7. Django

- Homepage: https://www.djangoproject.com/
- Repo: https://github.com/django/django
- REST-API: https://www.django-rest-framework.org/

PanDA

- Homepage: https://panda-wms.readthedocs.io/en/latest/
- Repo: https://github.com/PanDAWMS
- PyAPI/REST: https://panda-wms.readthedocs.io/en/latest/client/rest.html

10.8. Dirac

- Homepage: https://dirac.readthedocs.io/en/latest/AdministratorGuide/Systems/WorkloadManagement/
- Repo: https://github.com/DIRACGrid
- Py-API: https://dirac.readthedocs.io/en/latest/UserGuide/GettingStarted/UserJobs/DiracAPI/

Acknowledgments: I would like to thank my colleagues and collaborators of the various national and international projects with which I have had the privilege to work during the last decades and which have allowed me to develop a know-how in the field of data archiving, databases, data handling processing and mining. The projects in question are the Large Binocular Telescope in Arizona (USA); link: https://www.lbto.org/; ASTRI-Horn and ASTRI Miniarray projects; link: http://www.astri.inaf.it/; the Cherenkov Telescope Array Observatory; link: https://www.ctao.org/

Conflicts of Interest: No conflict of interest has been declared by the authors. I finally declare that the content of this manuscript is a human intellectual work and that no artificial intelligence program has been adopted.

Acronyms and Abbreviations

The following abbreviations are used in this manuscript:

ACID	Atomicity, Consistency, Isolation and Durability

 doi:10.20944/preprints202506.0847.v1

AMAS	ASTRI and Miniarray Archive System
ASTRI	Astrofisica con Specchi a Tecnologia Replicante Italiana
CAP	Theorem: Consistency (data Consistency), Availability (data Accessibility) and Partitioning (partition Tollerance)
CLI	Command Line Interface
CTAARCHS	Cloud-Based Technology for Archiving Astronomical Reseach Contents & Handling System
CTAO	Cherenkov Telescope Array Observatory
DBMS	DataBase Management System
DPAR	Data Product Acceptable Requirements
DPPN	Datacenters, like DC
FAIR	Findable, Accessible, Shared and Reusable
IKC	In-Kind Contribution
MWL	Multi WaveLenght
NOSQL	Not Only SQL
RDBMS	Relational DBMS
REST API	REpresentional State Transfer Application Programming Interface
RSE	Remote Storage Element
SQL	Structured Query Language
WMS	Workload Management System

Appendix

Data Model Example (DL0 Fits-Data)

The datamodel can be searchable and filtered in any keyword. The selection/filtering can be easily speedup by sorting keyword indexes. In principle each data-product and data-level has its own Data Model defined by a dedicated JSON Schema File. In this appendix an example of a DL0 Fits JSON entry.


```
{
    "aaid": "17447020150367122",
    "aipvers": "AMAS_1.0.0",
    "archive": {
        "PFN": "20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz",
        "PFNP": "/archive/MINIARRAY/PHYSICAL/pass_0.0.1/20250227/00001063/dl0/varlg/v2",
        "archdate": "2025-04-15 07:26:55.036728",
        "archtime": 1744702015,
        "checksum": "3ac92bdc",
        "container": "20250415",
        "dataset": "fits-data",
        "filesize": 14747749,
        "paths": {
            "RSE": "astriFS",
            "replicaflag": 0,
            "type": "web-https",
            "uid": "17447020150367122",
            "uripath": "https://amas.oa-roma.inaf.it/static/data/Miniarray/pass_0.0.1/20250227/00001063/dl0/varlg/v2/20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz"
        },
        "replicas": {
            "replica": [
                {
                    "number": "0",
                    "rdata": "2025-04-15 07:26:55.036728",
                    "rid": "17447020150367122",
```

 doi:10.20944/preprints202506.0847.v1


```
                "rtype": "web-https",
                "spool": "AMAS",
                "uri": "https://amas.oa-roma.inaf.it/static/data/Miniarray/pass_0.0.1/20250227/00001063/dl0/varlg/v2/20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz"
            }
        ]
    },
    "scope": "MA01"
},
"author": "Stefano Gallozzi",
"camera": "{'name': 'ASTRIMA', 'origin': 'ASTRIDPS', 'creator': 'adas preprocessing v1.1', 'npdm': 37, 'modeid': 'R', 'datatype': 'fits-data'}",
"daqmode": "R",
"datadesc": "lv0_var_lg",
"datatype": "1003",
"dateobs": "2025-02-27",
"event": {
    "obsdate": "2025-02-27"
},
"file_version": 1,
"filename": "20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz",
"fsize": 14747749,
"header": {
    "Primary": {
        "ALT_PNT": -999,
        "AZ_PNT": -999,
        "BITPIX": 16,
        "CHECKSUM": "g5XEj3XBg3XBg3XB",
        "COMMENT": [
            "= 'FITS (Flexible Image Transport System) format is defined in ''Astron'''
        ],
        "CREATOR": "adas preprocessing v1.1.1",
        "DAQ_ID": "002086",
        "DAQ_MODE": "R",
        "DATAFORMAT": "v1.0",
        "DATALEVEL": "lv0",
        "DATAMODE": "10",
        "DATASUM": "0",
        "DATE": "2025-02-28T07:34:30",
        "DATE-END": "2025-02-27T21:10:49",
        "DATE-OBS": "2025-02-27T20:29:41",
        "DEC_OBJ": 22.0174,
        "DEC_PNT": 21.517,
        "EQUINOX": "2000.0",
        "EXTEND": true,
        "FILENAME": "20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits",
        "FILEVERS": 1,
        "INSTRUME": "CAMERA",
        "MJDREFF": 0.00080074,
        "MJDREFI": 58849,
        "NAXIS": 0,
        "NTEL": 1,
        "OBJECT": "Crab",
        "OBS_DATE": "20250227",
```

 doi:10.20944/preprints202506.0847.v1


```
            "OBS_MODE": "W0.50p090",
            "ORIGIN": "ASTRIDPS",
            "ORIG_ID": "00",
            "PROG_ID": "001",
            "RADECSYS": "FK5",
            "RA_OBJ": 83.6324,
            "RA_PNT": 83.632,
            "RUN_ID": "00001063",
            "SBL_ID": "002",
            "SIMPLE": true,
            "SUBMODE": "02",
            "TELAPSE": "2468",
            "TELESCOP": "ASTRI-MA",
            "TEL_ID": "01",
            "TIMEOFFS": 0,
            "TIMESYS": "TT",
            "TIMEUNIT": "s",
            "TSTART": "162851381",
            "TSTOP": "162853849"
        }
    },
    "id": "f9c92d8e-e2a8-4c8b-bba9-7d6209317676",
    "infomail": "stefano.gallozzi@inaf.it",
    "latest_version": 1,
    "object": "Crab",
    "obsid": 2086,
    "obsmode": "W0.50p090",
    "packtype": "fits-data",
    "programid": 111,
    "proposal": {
        "carryover": "Y",
        "category": "EXT/RP",
        "cycle": {
            "name": "cycle2024/1",
            "period": "[ 2023-05-01,2023-05-05 ]",
            "type": "semester"
        },
        "obsprog": {
            "arrayconf": {
                "acq_mode": "wobble",
                "acq_submode": "2.5",
                "telmatrix": {
                    "confname": "fulla_ma",
                    "on_off": "[ 1,0,0,0,0,0,0,0,0 ]",
                    "type": "array"
                },
                "trigger": "S2"
            },
            "constraints": {
                "maxMIF": "0.7",
                "maxZA": "60.0 [deg]",
                "minAT": "0.7",
                "minET": "100 [hrs]",
                "minMD": "100 [deg]",
                "minZA": "0.0 [deg]"
```

 doi:10.20944/preprints202506.0847.v1


```
            },
            "progid": "1",
            "target": {
                "coord": "[ 83.6329, 22.014 ]",
                "dec": 22.014,
                "diameter": "7.0 [arsec]",
                "epoch": "J2000",
                "magnitude": "8.4 [ABmag]",
                "name": "Crab_Nebula",
                "rad": 83.6329,
                "tooflag": "0",
                "type": "wcs"
            }
        },
        "piname": "Giovanni Pareschi",
        "propdate": "2023-11-21",
        "propid": "2",
        "proplink": "https://amas.oa-roma.inaf.it/proposals/2/",
        "proptype": "SCI",
        "reqtime": "300 [hrs]"
    },
    "runid": 1063,
    "schema": "https://amas.oa-roma.inaf.it/static/aipMADLFITSschema",
    "schemavers": "FITS_v0.0.1",
    "telescope": {
        "altitude": "2390 [m]",
        "diamS1": "4.6 [m]",
        "diamS2": "1.5 [m]",
        "geo": {
            "coord": [
                28.30015,
                -16.50965
            ],
            "type": "gps"
        },
        "optconf": "DM",
        "telid": "01",
        "telname": "MA01",
        "type": "AIC"
    },
    "timestamp": "2025-02-28 09:31:54+00:00"
}
```


Json Object Schemas

Here is an example of a JSON Schema for the **"archive"** Object:


```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Archive Object Schema",
  "type": "object",
  "properties": {
    "archive": {
      "type": "object",
      "properties": {
        "PFN": { "type": "string" },
        "PFNP": { "type": "string" },
```

 doi:10.20944/preprints202506.0847.v1


```
        "archdate": { "type": "string", "format": "date-time" },
        "archtime": { "type": "integer" },
        "checksum": { "type": "string" },
        "container": { "type": "string" },
        "dataset": { "type": "string" },
        "filesize": { "type": "integer" },
        "paths": {
          "type": "object",
          "properties": {
            "RSE": { "type": "string" },
            "replicaflag": { "type": "integer" },
            "type": { "type": "string" },
            "uid": { "type": "string" },
            "uripath": { "type": "string", "format": "uri" }
          },
          "required": ["RSE", "replicaflag", "type", "uid", "uripath"]
        },
        "replicas": {
          "type": "object",
          "properties": {
            "replica": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "number": { "type": "string" },
                  "rdata": { "type": "string", "format": "date-time" },
                  "rid": { "type": "string" },
                  "rtype": { "type": "string" },
                  "spool": { "type": "string" },
                  "uri": { "type": "string", "format": "uri" }
                },
                "required": ["number", "rdata", "rid", "rtype", "spool", "uri"]
              }
            }
          },
          "required": ["replica"]
        },
        "scope": { "type": "string" }
      },
      "required": [
        "PFN", "PFNP", "archdate", "archtime", "checksum", "container",
        "dataset", "filesize", "paths", "replicas", "scope"
      ]
    }
  },
  "required": ["archive"]
}
```


And a valid **"archive"** property for that schema:


```
  "archive": {
      "PFN": "20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz",
      "PFNP": "/archive/MINIARRAY/PHYSICAL/pass_0.0.1/20250227/00001063/dl0/varlg/v2",
      "archdate": "2025-04-15 07:26:55.036728",
      "archtime": 1744702015,
      "checksum": "3ac92bdc",
```

 doi:10.20944/preprints202506.0847.v1


```
            "container": "20250415",
            "dataset": "fits-data",
            "filesize": 14747749,
            "paths": {
                "RSE": "astriFS",
                "replicaflag": 0,
                "type": "web-https",
                "uid": "17447020150367122",
                "uripath": "https://amas.oa-roma.inaf.it/static/data/Miniarray/pass_0.0.1/20250227/00001063/dl0/varlg/v2/20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz"
            },
            "replicas": {
                "replica": [
                    {
                        "number": "0",
                        "rdata": "2025-04-15 07:26:55.036728",
                        "rid": "17447020150367122",
                        "rtype": "web-https",
                        "spool": "AMAS",
                        "uri": "https://amas.oa-roma.inaf.it/static/data/Miniarray/pass_0.0.1/20250227/00001063/dl0/varlg/v2/20250227_MA01_Crab_W0.50p090_00001063_R_002086_1003.lv0.fits.gz"
                    }
                ]
            },
            "scope": "MA01"
        },
```


References

1. Giaretta, D. (2011). Introduction to OAIS Concepts and Terminology. Advanced Digital Preservation. Springer Berlin Heidelberg. doi: https://doi.org/10.1007%2F978-3-642-16809-3_3
2. O.J. Akindote, O.J., Adegbite, A.O, Dawodu, S.O., Omotosho, A., Anyanwu, A. (2023.). Innovation in Data Storage Technologies: from Cloud Computing to Edge Computing, Computer Science & IT Research Journal P-ISSN: 2709-0043, E-ISSN: 2709-0051 Volume 4, Issue 3, P.273-299, December 2023 DOI: 10.51594/csitrj.v4i3.661
3. Bajaj, K., Sharma, B., & Singh, R. (2022). Implementation analysis of IoT-based offloading frameworks on cloud/edge computing for sensor generated big data. Complex & Intelligent Systems, 8(5), 3641-3658. DOI: 10.1007/s40747-021-00434-6
4. Bhargavi, P., & Jyothi, S. (2020). Object detection in Fog computing using machine learning algorithms. In Architecture and Security Issues in Fog Computing Applications (pp. 90-107). IGI Global. DOI: 10.4018/978-1-7998-0194-8.ch006
5. Chen, S., & Rodero, I. (2017). Understanding behavior trends of big data frameworks in ongoing software-defined cyber-infrastructure. In Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (pp. 199-208). DOI: 10.1145/3148055.3148079
6. Gąbka, J. (2019). Edge computing technologies as a crucial factor of successful industry 4.0 growth. The case of live video data streaming. In Advances in Manufacturing II: Volume 1-Solutions for Industry 4.0 (pp. 25-37). Springer International Publishing. DOI: 10.1007/978-3-030-18715-6_3.
7. Kumar, D., Singh, R.K., Mishra, R., & Vlachos, I. (2023). Big data analytics in supply chain decarbonisation: a systematic literature review and future research directions. International Journal of Production Research, 1-21. DOI: 10.1080/00207543.2023.2179346

 doi:10.20944/preprints202506.0847.v1

8. Lathar, P., Srinivasa, K.G., Kumar, A., & Siddiqui, N. (2018). Comparison study of different NoSQL and cloud paradigm for better data storage technology. Handbook of Research on Cloud and Fog Computing Infrastructures for Data Science, 312-343. DOI: 10.4018/978-1-5225-5972-6.CH015
9. Manoranjini, J., & Anbuchelian, S. (2021). Data security and privacy-preserving in edge computing: cryptography and trust management systems. In Cases on Edge Computing and Analytics (pp. 188-202). IGI Global. DOI: 10.4018/978-1-7998-4873-8.CH010
10. Padhy, R.P., & Patra, M.R. (2012). Evolution of cloud computing and enabling technologies. International Journal of Cloud Computing and Services Science, 1(4), 182. DOI: 10.11591/CLOSER.V1I4.1216
11. Patel, S., & Patel, R. (2023). A comprehensive analysis of computing paradigms leading to fog computing: simulation tools, applications, and use cases. Journal of Computer Information Systems, 63(6), 1495-1516. DOI: 10.1080/08874417.2022.2121782
12. Shiau, W.L. (2015). An evolution, present, and future changes of cloud computing services. Journal of Electronic Science and Technology, 13(1), 54-59.
13. Strohbach, M., Daubert, J., Ravkin, H., & Lischka, M. (2016). Big data storage. New Horizons for a Data-Driven Economy: A Roadmap for Usage and Exploitation of Big Data in Europe, 119-141. DOI: 10.1007/978-3-319-21569-3_7
14. Zhang, J., Chen, B., Zhao, Y., Cheng, X., & Hu, F. (2018). Data security and privacy-preserving in edge computing paradigm: Survey and open issues. IEEE Access, 6, 18209-18237. DOI: 10.1109/ACCESS.2018.2820162
15. Svoboda T., Raček T., Handl J., Sabo J., Rošinec A., Opioła L., Jesionek W., Ešner M., Pernisová M., Valasevich N.M., Křenek A., Svobodová R., (2923) Onedata4Sci: Life science data management solution based on Onedata, https://doi.org/10.48550/arXiv.2311.16712
16. Gadban F.. (2021) Analyzing the Performance of the S3 Object Storage API for HPC Workloads. Appl. Sci. 2021, 11, 8540. https://doi.org/10.3390/app11188540.
17. Malhotra S., Yashu F., Saqib N., Mehta D., Jangid J., Dixit S.. (2025) Evaluating Fault Tolerance and Scalability in Distributed File Systems: A Case Study of GFS, HDFS, and MinIO, https://doi.org/10.48550/arXiv.2502.01981