

UNIVERSITY OF TRIESTE
Department of Mathematics, Informatics and Geosciences



Master's Degree in
Data Science and Scientific Computing
Research thesis in: Data Management

Designing and deploying a FAIR-by-design data
pipeline and platform for electron microscopy
laboratories

2025-09-19

Candidate
Nicola Perin

Supervisor
Dr. Federica Bazzocchi

Academic Year 2024–2025

*And what is good, Phaedrus,
and what is not good—
Need we ask anyone to tell us these things?*

- Robert M. Pirsig

Contents

Abstract, english	III
Abstract, italian	IV
Introduction	V
1 Foundations and context	1
1.1 Data management challenges in electron microscopy	1
1.2 FAIR data and national initiatives	2
1.3 Electron microscopy instrumentation	2
1.3.1 Transmission electron microscopy (TEM)	2
1.3.2 Scanning transmission electron microscopy (STEM)	2
1.4 Data standards: HDF5, NeXus, and NXem	3
1.4.1 HDF5: a general-purpose scientific container	3
1.4.2 NeXus: community conventions on top of HDF5	3
1.4.3 NXem: the electron microscopy profile	4
1.5 Institutional context: Area Science Park and RIT	4
2 ORFEO infrastructure: distributed storage and identity services	6
2.1 ORFEO datacenter and distributed storage	6
2.1.1 ORFEO datacenter: infrastructure and physical data storage	6
2.1.2 Ceph filesystem in ORFEO (CephFS)	7
2.1.3 Ceph RADOS Gateway and the S3 API	9
2.2 Identity and access management (IAM): FreeIPA with Authentik	10
3 Application platform choices: Django and PostgreSQL	12
3.1 Why a web application for lab workflows (upload + annotation together)	12
3.2 Django: Model–View–Template and project layout	12
3.3 Relational databases and the choice of PostgreSQL	14
3.3.1 Choosing PostgreSQL as the database backend	15
3.3.2 Django’s ORM (Object-Relational Mapper)	16
4 Deployment and validation in the VirtualOrfeo environment	18
4.1 VirtualOrfeo as a digital twin of the HPC environment	18
4.2 K3s cluster topology in the virtual environment	19
4.3 Identity and access management in VirtualOrfeo	20
4.4 Storage integration in VirtualOrfeo	22
4.5 Application packaging and deployment on K3s	24

4.6	Secrets and configuration management	25
5	Deep dive into the application	28
5.1	Domain model and data flow	28
5.2	Metadata path	30
5.3	NeXus construction	31
5.4	Storage gateway	35
5.5	Background tasks	38
5.6	API and UI surfaces	40
5.7	User interface walkthrough	42
5.8	Security model in-app	49
5.9	Performance and scalability	50
	Conclusion	52
A	Tools and technologies	54
	Bibliography	58

Abstract, english

Electron microscopy (EM) experiments generate very large and complex datasets. These are often stored in vendor-specific formats with little metadata, which makes them hard to share, reproduce, or reuse. To address this, this thesis applies the FAIR data principles—Findable, Accessible, Interoperable, and Reusable—to the workflows of the *Laboratorio di Microscopia Elettronica* (LAME), part of the *Ricerca e Innovazione Tecnologica* (RIT) institute at Area Science Park in Trieste, Italy.

The project develops and tests a FAIR-by-design data infrastructure that:

- captures data and metadata directly during acquisition,
- converts and stores them automatically in standardized NeXus/NXem format,
- uses a centralized data lake for durability and scalability,
- and provides a web application for structured uploads, annotations, and access.

The system separates microscope operations from heavy processing, relying on the ORFEO high-performance data center for storage and computation. A virtual test environment was used to safely develop and validate the software stack before production deployment. This ensures EM data are preserved in interoperable formats, enriched with metadata, and prepared for open access.

The work contributes a practical and innovative blueprint for reproducible FAIR data management in microscopy, with the potential to scale across research infrastructures in material science, like those affiliated to the NFFA-DI program¹, and align with European open science policies.

¹<https://nffa-di.it/en/about-us/project/>

Abstract, italian

Gli esperimenti di microscopia elettronica (EM) generano dataset molto grandi e complessi. Questi sono spesso memorizzati in formati proprietari con poche informazioni descrittive (metadata), il che rende difficile condividerli, riprodurli o riutilizzarli. Per affrontare questo problema, questa tesi applica i principi FAIR—Findable, Accessible, Interoperable, Reusable—ai flussi di lavoro del *Laboratorio di Microscopia Elettronica* (LAME), parte dell'Istituto di *Ricerca e Innovazione Tecnologica* (RIT) presso Area Science Park a Trieste.

Il progetto sviluppa e testa un'infrastruttura dati FAIR-by-design che:

- cattura dati e metadata direttamente durante l'acquisizione,
- li converte e archivia automaticamente in formato standardizzato NeXus/NXem,
- utilizza un *data lake* centralizzato per garantire durabilità e scalabilità,
- e fornisce un'applicazione web per caricamenti strutturati, annotazioni e accesso.

Il sistema separa le operazioni al microscopio dall'elaborazione intensiva, affidandosi al centro di calcolo ad alte prestazioni ORFEO per archiviazione e computazione. Un ambiente virtuale di test è stato usato per sviluppare e validare in sicurezza lo stack software prima del rilascio in produzione. Questo assicura che i dati EM vengano conservati in formati interoperabili, arricchiti con metadata e pronti per l'accesso aperto.

Il lavoro contribuisce con un modello pratico e innovativo per la gestione FAIR e riproducibile dei dati in microscopia, con il potenziale di scalare ad altre infrastrutture di ricerca in scienza dei materiali, come quelle affiliate al programma NFFA-DI², e di allinearsi alle politiche europee per la scienza aperta.

²<https://nffa-di.it/it/about-us/project/>

Introduction

Modern scientific laboratories are generating unprecedented volumes of complex data. In electron microscopy (EM), a single transmission electron microscopy (TEM) session can produce gigabytes of images, spectra, and multidimensional datasets (Poger, Yen and Braet 2023). Managing and making sense of this data deluge has become a central challenge for both scientists and facility managers. EM data are typically recorded in vendor-specific formats and analyzed with proprietary software, which complicates long-term access, interoperability, and reusability (Moore et al. 2021). Without planned and systematic data management, researchers often fall back on informal practices—renaming files, maintaining local notes—that do not scale, especially across collaborations (Korir, Kleywegt et al. 2024). As a result, valuable datasets are frequently siloed, forgotten, or irretrievable, undermining reproducibility and slowing scientific progress.

The scientific community increasingly addresses these issues through the FAIR data principles: research outputs should be **F**indable, **A**ccessible, **I**nteroperable, and **R**eusable by both humans and machines (Wilkinson et al. 2016; GO FAIR Initiative 2016). FAIR practices emphasize rich metadata³, standardized formats, and infrastructures that allow data discovery and reuse with minimal friction. They also stress machine-actionability, since data volumes are now too large to be curated manually (Wilkinson et al. 2016). Funding agencies and journals increasingly expect adherence to FAIR workflows (Hodson et al. 2018; European Commission 2021), making these practices not only desirable but necessary.

This thesis investigates how FAIR principles can be applied to the daily operations of a state-of-the-art electron microscopy laboratory. We focus on the *Laboratorio di Microscopia Elettronica* (LAME) at Area Science Park in Trieste, Italy (Area Science Park 2025a). Equipped with cutting-edge TEM⁴, STEM⁵, FIB-SEM⁶, and EELS⁷ instruments, LAME produces large and heterogeneous datasets—high-resolution images, diffraction patterns, elemental maps, EELS spectra, tomography reconstructions—that require careful management to ensure future usability. While FAIR data practices are increasingly adopted in other fields, few complete pipelines exist today for electron microscopy, making this work a step toward filling that gap.

Our project develops and validates a FAIR-by-design data infrastructure for LAME. The core strategy is to capture data at the moment of acquisition, enrich it with standardized metadata, and package it into the *NeXus* format—an HDF5-based community standard for scientific data (Könnicke et al. 2015a). In particular, we adopt the NXem application definition⁸ for electron

³Metadata are “data about data”: descriptive information such as authorship, date, experimental settings, or file format, which help interpret and reuse the primary data.

⁴Transmission Electron Microscopy

⁵Scanning Transmission Electron Microscopy

⁶Focused Ion Beam – Scanning Electron Microscopy

⁷Electron Energy Loss Spectroscopy

⁸In the NeXus standard, an *application definition* specifies a fixed schema for a given experimental technique, defining which groups and fields must be present and how metadata should be organized. This ensures that data from different instruments can be interpreted in a consistent way.

microscopy, which specifies schemas for instrument settings, detector parameters, and sample context (NeXus International Advisory Committee (NIAC) 2024; FAIRmat (NFDI) 2024). By leveraging NeXus/NXem, we ensure that microscopy datasets are self-describing, interoperable, and future-proof.

The pipeline developed at LAME automatically ingests raw microscope outputs, maps metadata into the NeXus schema, and deposits curated files in a centralized data lake. A lab-facing web application couples user uploads with structured annotation, while the heavy data processing occurs server-side at the institute’s *ORFEO* data center. The result is a reproducible and scalable workflow that preserves experimental data in FAIR form, prepares it for open access, and provides a model for adoption across other laboratories.

This thesis makes the following contributions:

- A FAIR-by-design workflow for electron microscopy at LAME, from raw TIFF acquisition to standardized *NeXus/NXem* packaging and curated storage.
- A web application that unifies data upload and structured annotation, producing both human-readable README artefacts and machine-readable NeXus files.
- A deployment pattern that separates laboratory acquisition systems from heavy processing, by centralizing conversion and storage in a robust data center.
- A blueprint for reproducible FAIR data workflows, intended for extension across the NFFA-DI program (see Section 1.2) and aligned with European open science policies.

The remainder of this thesis is organized as follows:

1. Chapter 1 introduces the broader context: challenges in EM data management, FAIR principles, related initiatives, and the institutional setting at Area Science Park.
2. Chapter 2 describes the existing infrastructure, including storage (Ceph/RGW/S3) and identity (*FreeIPA/Authentik*).
3. Chapter 3 motivates the use of the Django framework and PostgreSQL.
4. Chapter 4 presents the *virtualorfeo* test environment and the application deployment.
5. Chapter 5 provides a deep dive into the application’s inner workings (metadata extraction, NXem construction, storage flows).
6. In the [Conclusions](#), we summarize the lessons learned and outline directions for adoption across NFFA-DI.

Chapter 1

Foundations and context

This chapter situates our work within the broader landscape of data management in electron microscopy. It discusses the challenges of handling modern EM datasets, introduces the FAIR data principles and related initiatives, describes relevant standards for data and metadata, and presents the institutional setting in which our project operates.

1.1 Data management challenges in electron microscopy

Electron microscopy facilities routinely generate large, heterogeneous datasets: images, diffraction patterns, spectrum images (4D-STEM), EDS maps, and EELS hyper-spectra. A single session can yield hundreds of megabytes to tens of gigabytes of data (Poger, Yen and Braet 2023). Two challenges stand out:

1. vendor-specific formats and proprietary software, which hinder interoperability and long-term accessibility,
2. personal practices (like manual file naming or keeping local notes) that do not scale and are easily lost when staff change (Moore et al. 2021; Korir, Kleywegt et al. 2024)

These factors increase the risk of orphaned datasets, duplicated effort, and limited reproducibility (Poger, Yen and Braet 2023; Korir, Kleywegt et al. 2024).

Within LAME at Area Science Park, the concrete needs that emerge from day-to-day operations are:

- (a) a central, durable repository for raw outputs and derived products;
- (b) early, automated capture of operative metadata (e.g., acceleration voltage, pixel size, dwell times, detector configuration, stage coordinates);
- (c) a community format that couples data and metadata in a self-describing structure; and
- (d) low-friction deposit and retrieval that does not disrupt microscope workflows.

Meeting these needs requires standardized file formats and workflows that are *FAIR-by-design*, beginning at the point of data creation rather than at publication.

1.2 FAIR data and national initiatives

The FAIR principles (data should be **F**indable, **A**ccessible, **I**nteroperable, and **R**eusable) provide a framework for addressing these challenges (Wilkinson et al. 2016; GO FAIR Initiative 2016). In practice, FAIR means using persistent identifiers, standardized metadata, and open or well-documented formats, and depositing data in searchable repositories with clear access policies and explicit usage licenses (Hodson et al. 2018; European Commission 2021).

In Italy, these principles are being put into practice through national initiatives such as NFFA-DI (Nanoscience Foundries & Fine Analysis—Digital Infrastructure). Supported by the PNRR¹ program, NFFA-DI coordinates FAIR-by-design data management across eleven research institutions and 119 scientific instruments (NFFA-DI 2024).

Its Overarching FAIR Ecosystem for Data (OFED) provides S3-compatible object storage for NeXus files, metadata indexing, and visualization endpoints, containerized on the ORFEO HPC cluster (Bazzocchi 2024a). Lab-embedded data stewards ensure quality and metadata completeness before archival. LAME contributes TEM and STEM workflows to NFFA-DI.

Internationally, FAIRmat within Germany’s NFDI program extends the NOMAD computational repository to experimental data, including EM outputs (FAIRmat Consortium 2024). Its contributions to the NXem definition give laboratories concrete templates for storing images, diffraction patterns, and spectra with metadata (The HDF Group 2021).

Aligning local workflows with these initiatives increases interoperability and prepares laboratories for transparent data exchange across national and European portals, particularly through the European Open Science Cloud (EOSC).

1.3 Electron microscopy instrumentation

Electron microscopy (EM) is a family of techniques that use beams of electrons, rather than visible light, to form images of materials. Because electrons have much shorter wavelengths than photons, EM can resolve structures down to the atomic scale. Different instrument configurations extend this capability with analytical techniques such as diffraction and spectroscopy, providing not only images but also information about composition and electronic structure.

Electron microscopy supports nanoscience with imaging and spectroscopy at the atomic scale. A brief overview of the modalities available at LAME clarifies the sources and structure of the data to be managed.

1.3.1 Transmission electron microscopy (TEM)

In TEM, electrons accelerated to 80–300 keV traverse thin specimens, producing real-space images with near-atomic resolution (Metilli et al. 2020). By adjusting the lens system, selected-area or nano-beam electron diffraction (ED) patterns can be recorded, revealing lattice spacings and crystal symmetry. Modern instruments, such as those at LAME, allow rapid switching between imaging and diffraction within a single session.

1.3.2 Scanning transmission electron microscopy (STEM)

In STEM, a focused probe is raster-scanned across the sample while detectors collect signals at each position (Metilli et al. 2020). High-angle annular dark-field (HAADF) and bright-field (BF) imaging modes provide complementary contrast mechanisms. Spectroscopic attachments enable

¹PNRR: *Piano Nazionale di Ripresa e Resilienza*, Italy’s National Recovery and Resilience Plan (part of the EU’s NextGenerationEU).

acquisition of hyperspectral data cubes: two spatial dimensions combined with spectral dimensions (X-ray EDS or energy-loss EELS). Hybrid-pixel direct detectors further support 4D-STEM, where full diffraction patterns are recorded at every probe position, leading to datasets of 10–100 GB per experiment. These volumes demand disciplined metadata tracking and standardized storage.

1.4 Data standards: HDF5, NeXus, and NXem

Large-scale electron microscopy experiments produce not only very large files, but also complex combinations of raw signals and contextual information. To make such datasets usable over the long term, they need to be stored in formats that can handle both high-volume numerical arrays (such as images or spectra) and descriptive metadata (such as microscope settings or sample details). Ordinary file formats like TIFF or CSV are not sufficient: they can capture the raw numbers, but they cannot reliably encode the experimental context in a standardized, machine-readable way. For this reason, the scientific community has developed hierarchical container formats that integrate data and metadata in a single file. In the context of electron microscopy, three related standards are especially relevant: HDF5, NeXus, and NXem. Each builds on the previous one, adding additional layers of structure and semantics.

1.4.1 HDF5: a general-purpose scientific container

The *Hierarchical Data Format version 5* (HDF5) is a binary file format and associated library designed for storing and organizing large, complex datasets (NeXus International Advisory Committee 2024a). The key idea is that an HDF5 file acts like a self-contained file system: it contains *groups*, which behave like folders, and *datasets*, which behave like multidimensional arrays (e.g. images, spectra, or time series). Any group or dataset can also carry *attributes*, which are small pieces of metadata (such as labels, units, or calibration constants). This hierarchical structure allows scientists to organize both raw signals and their context in a single, portable file.

HDF5 was designed with performance in mind: features like chunked storage (breaking datasets into blocks), compression, and partial I/O make it efficient for reading and writing terabyte-scale data. It is widely used across domains such as physics, climate modeling, and genomics, and has become a de facto standard for large numerical datasets. However, HDF5 itself is agnostic about scientific meaning: it defines *how* data are stored, but not *what* the data represent. Two HDF5 files produced by different laboratories may have completely different internal layouts, even if they describe the same type of experiment. This lack of shared semantics motivates higher-level conventions such as NeXus.

1.4.2 NeXus: community conventions on top of HDF5

NeXus is an international standard that was originally introduced in the neutron scattering community and later adopted by other experimental fields, including X-ray and electron microscopy (NXcanSAS Working Group 2012; FAIRmat-NFDI 2025). It builds on HDF5 by introducing shared conventions for scientific experiments. NeXus defines a set of base classes (such as `NXinstrument`, `NXsample`, `NXdetector`) that describe common experimental entities. On top of these, it provides *application definitions*: community-agreed templates that specify which data and metadata fields are required, optional, or recommended for a given experimental technique. For example, an application definition for X-ray diffraction will specify how to record beam energy, detector geometry, and intensity maps, so that any compliant dataset has the same structure.

This approach makes NeXus files self-describing not only in a technical sense (as in HDF5), but also at the scientific level. A NeXus file tells both humans and machines what kind of experiment was performed, which instrument and sample parameters are relevant, and where the actual measurement data can be found in the hierarchy. By standardizing these semantics, NeXus greatly improves interoperability: software tools can be written once and applied across datasets from different laboratories, as long as they follow the same application definition.

1.4.3 NXem: the electron microscopy profile

For electron microscopy, the relevant NeXus extension is the NXem application definition (Bazzocchi 2024b; NeXus International Advisory Committee 2024b). NXem specifies how to organize both the raw signals (such as images, diffraction patterns, EDS maps, or EELS spectra) and the contextual metadata (beam energy, detector geometry, stage coordinates, sample description). By adopting NXem, laboratories avoid inventing their own ad-hoc layouts and instead align with a shared, community-maintained schema. This ensures that essential provenance information is captured in a consistent way, and that downstream software can locate both data and metadata without relying on vendor-specific formats or parsers.

To facilitate adoption, community-developed libraries such as `pynxtools-em` provide ready-to-use functions for writing microscopy data and metadata into the NXem structure (FAIRmat-NFDI Development Team 2025). This lowers the barrier for laboratories to transition from raw vendor files to standardized, FAIR-compliant NeXus/NXem containers.

1.5 Institutional context: Area Science Park and RIT

The implementation of a FAIR data workflow at LAME is supported by the organizational context within Area Science Park. Specifically, the project is carried out under AREA’s Institute for Research and Technological Innovation (RIT – Istituto per la Ricerca e l’Innovazione Tecnologica). RIT is a multi-disciplinary research institute within Area Science Park that conducts fundamental research while also focusing on cutting-edge R&D and providing advanced services to external partners. It is structured into several laboratories equipped with state-of-the-art technologies. The three key labs under RIT, each with a specialized domain, are:

1. **LADE – Laboratorio di Data Engineering (Data Engineering Laboratory):** A laboratory devoted to data infrastructure, data management, and artificial intelligence. LADE drives research and innovation in building AI-augmented data platforms, with a strong emphasis on FAIR-by-design approaches and open science. The LADE team designs and maintains the data systems that support RIT’s scientific work. Notably, LADE manages and analyzes the data produced by the genomics and microscopy labs (LAGE and LAME), and it oversees the IT infrastructure of RIT – specifically the ORFEO data center that powers large-scale computing and storage for the institute.
2. **LAGE – Laboratorio di Genomica ed Epigenomica (Genomics and Epigenomics Laboratory):** A genomics and epigenomics laboratory dedicated to DNA/RNA sequencing and genotyping. LAGE operates on an open-access model, integrating resources and expertise to provide services like high-throughput sequencing and genomic data analysis. The lab participates in numerous collaborations, contributing genomic insights to both fundamental research and applied projects (e.g. biomedical studies). LAGE generates massive sequencing datasets that require intensive data processing and secure storage, which in turn rely on the support of LADE’s data infrastructure.

3. **LAME – Laboratorio di Microscopia Elettronica (Electron Microscopy Laboratory):** The Electron Microscopy lab which is the focus of this thesis. As described, LAME offers advanced material characterization with its double-corrected TEM/STEM instruments and FIB-SEM. The lab’s mission is aligned with European research standards, providing not only internal research capability but also open access to external users for competitive projects in nanoscience and nanotechnology. LAME’s integration into RIT means it works closely with the data engineering lab (LADE) to handle the large volumes of microscopy data and to develop new methods for data analysis (such as applying AI to microscopy images, an area of active interest).

Underpinning these laboratories is the ORFEO Data Center (Open Research Facility for Epigenomics and Other), the cornerstone of RIT’s technological infrastructure for HPC and storage. Detailed compute, storage, and networking characteristics are presented in Chapter 2. In our context, ORFEO resources are leveraged to perform on-the-fly data conversion and ingestion as microscopy data streams from the lab, and to provide central, durable storage and downstream compute capabilities.

Within this institutional setup, our project is a collaboration between LAME (domain science) and LADE (data engineering). The objective is to tie LAME’s instruments to the ORFEO-centered data ecosystem in a seamless way. A high-level overview of the workflow (from raw TIFF and metadata capture to NeXus/NXem packaging and curated storage) appears in the Introduction; implementation and deployment details follow in Chapters 4 and 5.

Chapter 2

ORFEO infrastructure: distributed storage and identity services

2.1 ORFEO datacenter and distributed storage

This section introduces ORFEO’s computing and storage infrastructure, which serves as the backbone for research workflows at Area Science Park. We begin with the datacenter itself and its physical storage, then describe the Ceph filesystem, and finally examine the RADOS Gateway interface.

2.1.1 ORFEO datacenter: infrastructure and physical data storage

ORFEO is a cutting-edge data center managed by the Laboratory of Data Engineering (LADe) at Area Science Park. It is housed in a specialized container and provides advanced computing and data services for scientific research (ORFEO Team [2023a](#); Area Science Park [2025b](#)).

In terms of hardware, ORFEO offers:

- high-performance computing nodes (including GPU-equipped servers and large-memory nodes up to 1.5 TB RAM),
- interconnected with high-speed networks (InfiniBand 100 Gb/s) (ORFEO Team [2023b](#)).

To support data-intensive workloads, ORFEO’s storage infrastructure delivers over 5 PB of raw storage capacity, implemented as a parallel distributed filesystem. This storage is served to users via a Ceph cluster, which combines:

- high-speed flash storage tiers (NVMe and SSD drives),
- standard capacity tiers (HDD drives) (ORFEO Team [2023c](#); ORFEO Team [2024](#)).

A long-term archival storage partition of more than 3 PB is available for data retention, and a tape library is also present for backups and cold storage (ORFEO Team [2023c](#); ORFEO Team [2024](#)).

In summary, ORFEO physically stores data across a large Ceph distributed storage system augmented by dedicated archival appliances, ensuring both high-performance access and reliable long-term preservation of research data.

From a physical perspective, the Ceph storage cluster underlying ORFEO consists of numerous storage nodes (servers with many disks) acting together. ORFEO’s shared filesystem is hosted on a Ceph cluster of 23 storage nodes, providing roughly 4.8 PiB of raw space across the cluster (ORFEO Team 2023c; ORFEO Team 2024).

The Ceph cluster is configured with different tiers to meet varying performance needs:

- Four Ceph nodes with SSDs (about 279 TiB raw capacity) provide high-speed storage, used both for home directories and for fast per-user scratch areas.
- 19 Ceph nodes with HDDs provide a larger capacity pool for standard shared scratch storage (bulk data and temporary analyses) (ORFEO Team 2023c).

In practice, user data is striped and replicated across many disks and servers: Ceph’s distributed design writes pieces of each file (as objects) to multiple nodes, protecting against hardware failures. Thanks to Ceph’s redundancy (via replication or erasure coding), the data center can tolerate disk or node failures without losing data (Project 2025a; Project 2025b; Project 2025c).

The result is a highly reliable storage backbone: users see a single unified filesystem, but underneath, their data blocks are physically spread across many drives and machines in the ORFEO cluster.

It’s worth noting how different research groups utilize ORFEO’s storage. All LADE-managed projects and users have their data on the Ceph-based filesystem by default. Each user gets:

- a home directory (with quotas such as 200 GB and 1 million files enforced) on a CephFS volume backed by SSD storage (with triple replication for safety) (ORFEO Team 2023c).
- access to a shared scratch space on Ceph—a large high-capacity area on HDD-based storage for active analyses. This scratch is configured with an erasure-coded scheme (6+2 encoding, meaning data is split into 6 chunks plus 2 parity chunks) to maximize usable space while still tolerating failures (Project 2025c; Project 2025a).
- a per-user fast scratch area on SSDs (also replicated 3×) for I/O-intensive tasks requiring faster throughput (Project 2025b).

In contrast, the LAME laboratory (Microscopy lab) historically has stored its data on local storage systems separate from ORFEO’s Ceph cluster. This means LAME’s data hasn’t benefited from ORFEO’s centralized distributed storage.

A key motivation for our work is to enable transferring and integrating LAME’s locally stored data into ORFEO, so that microscopy data can reside on the Ceph-based infrastructure alongside the genomics and other datasets managed by LADE. (The genomics lab, LAGE, is already set up to use ORFEO’s storage, so we focus here on LADE and LAME integration.)

By moving LAME’s data into ORFEO, researchers will gain the advantages of Ceph’s reliability, scalability, and unified access for all their scientific data.

2.1.2 Ceph filesystem in ORFEO (CephFS)

Ceph is the primary storage technology underpinning ORFEO’s filesystem (ORFEO Team 2023c). CephFS (Ceph Filesystem) is a POSIX-compliant distributed file system built on Ceph’s object store (RADOS) (Project 2025d).

In simpler terms, CephFS allows all compute nodes in the cluster to see a shared directory tree (“/orfeo/...”) where user files and project data reside, just like a network file system, but with no single server bottleneck. Instead of storing files on a single disk or server, CephFS distributes file

data in objects across the cluster’s OSDs (Object Storage Daemons)—essentially across many disks on many nodes—and a cluster of Metadata Servers (MDS) coordinates the file system namespace and directory operations (Project 2025d; Project 2025a; Hat 2025).

This architecture provides high throughput and availability: clients (compute nodes or user login nodes) directly read/write data to the storage nodes in parallel, while metadata (filenames, directories, permissions) is handled by dedicated MDS servers for efficient lookup (Project 2025d; Hat 2025). As a result, CephFS can scale out linearly with the size of the cluster, making it ideal for HPC scenarios like ORFEO where numerous users and jobs concurrently access shared datasets (Project 2025d).

One of the strengths of CephFS in ORFEO is its built-in redundancy and data management policies. Ceph transparently manages data replication and/or erasure coding across the cluster, so users do not have to worry about manually copying data to protect against failures (Project 2025b; Project 2025c).

For instance, as noted above, some Ceph pools on ORFEO keep three replicas of each data block on different storage nodes (triple mirroring) for important or hot data (Project 2025b). Other pools use an erasure-coded scheme (6+2), which breaks each file into 6 data chunks and 2 parity chunks distributed to different nodes; any 6 of the 8 chunks are sufficient to recover the file, giving fault tolerance with only $\sim 1.33\times$ storage overhead instead of $3\times$ (Project 2025c; Project 2025a).

Ceph automatically handles re-balancing and healing: if a disk or node fails, the system detects the loss and recreates missing chunks or copies on healthy devices in the cluster. This self-healing design, along with capacity scaling by simply adding more nodes or disks, makes Ceph a very robust solution for a growing data center (Project 2025a). Moreover, CephFS supports features like quotas on directories (e.g. enforcing per-user storage limits) and snapshots, which are useful in a multi-user research environment (ORFEO Team 2023c).

All these capabilities mean that ORFEO’s Ceph-based filesystem can simultaneously serve as home directories, scratch spaces, and project archives for many users without performance bottlenecks or single points of failure.

In ORFEO’s configuration, the CephFS is mounted under the `/orfeo` directory on all cluster machines, providing a unified space for data. As outlined earlier, this space is organized into subfolders like `home`, `scratch`, `fast`, etc., each possibly backed by different storage tiers (ORFEO Team 2023c).

Users’ home directories (in `/orfeo/cephfs/home`) are intended for personal files and configurations and have a fixed quota (e.g. 200 GiB) (ORFEO Team 2023c). The scratch areas (in `/orfeo/cephfs/scratch`) are shared working spaces for active computations, visible on all nodes and divided into project-specific subfolders; these reside on high-capacity 10k RPM HDDs, and while they have no strict quota, older files may be purged after a minimum retention period (e.g. 30 days) to free space (ORFEO Team 2023c).

The fast area (accessible via `/orfeo/cephfs/fast`) is another shared scratch intended for workloads needing faster I/O, stored on SSDs; similarly un-quotaed, it’s managed with a policy to clear old data as needed (ORFEO Team 2023c).

Finally, for archival storage, ORFEO provides a Long-Term Storage (LTS) separate from CephFS: this is implemented via a Dell PowerVault SAN (Network Attached Storage using NFSv4) offering ~ 2.8 PiB raw space, allocated in volumes per group/project for long-term data retention (ORFEO Team 2023c).

In summary, CephFS handles the high-performance and working-storage tiers (`home`, `scratch`, `fast`) for LADE users on ORFEO, while a traditional NAS and tape handle archival storage. The key point is that every LADE user’s active data partition is on Ceph—giving them the benefits

of distributed storage—whereas LAME users currently keep data on local storage outside this system.

Bridging that gap is a goal of integrating LAME with ORFEO, allowing microscopy data to be moved into CephFS where it can be easily shared and processed in the HPC environment.

2.1.3 Ceph RADOS Gateway and the S3 API

In addition to providing file-system access through CephFS, the Ceph storage platform also supports an object storage interface called the RADOS Gateway (RGW) (Project 2025e).

To unpack this: Ceph is built on top of a distributed storage system called Reliable Autonomic Distributed Object Store (RADOS), which handles the actual storage of data across many servers and disks. The RADOS Gateway is a service (a daemon called `radosgw`) that runs as an HTTP server. Instead of presenting storage as files and directories, the gateway exposes a web-based API—in other words, a network endpoint that applications can talk to using standard internet protocols (Project 2025e; Project 2025f).

The most important feature of the RADOS Gateway is that its interface is designed to be compatible with the Amazon Simple Storage Service (Amazon S3) API (Project 2025f). Amazon S3 is one of the earliest and most widely adopted cloud storage services, introduced by Amazon Web Services in 2006. It popularized the concept of “object storage,” where data is managed as discrete units called objects rather than as files in a traditional hierarchical file system. Each object consists of the raw data (for example, the contents of a file) along with associated metadata (such as its size, creation date, or custom tags). Objects are organized into logical containers called buckets. Access to objects is performed through API calls sent over HTTP or HTTPS, such as “PUT” to upload a file, “GET” to download it, or “DELETE” to remove it.

By imitating this interface, the Ceph RADOS Gateway makes it possible for applications and users to interact with Ceph’s storage as if they were interacting with Amazon S3 itself (Project 2025f; Project 2025e). This compatibility is powerful: it means that existing tools, libraries, and workflows that already support Amazon S3 can be used directly with a Ceph cluster, without modification. For example, common command-line utilities, backup systems, scientific data management pipelines, or even web applications can all speak the S3 protocol and therefore work seamlessly with Ceph when the RADOS Gateway is deployed (Project 2025f).

Internally, the RADOS Gateway does not bypass Ceph’s design. All objects that arrive through the S3 interface are still stored within the underlying RADOS cluster, just like the data written via CephFS (Project 2025e). This ensures the same level of redundancy, fault tolerance, and scalability. However, the object storage model is different from a file system model: instead of directories and file paths, users think in terms of buckets and object keys. The gateway also maintains its own system of user accounts, credentials, and access control rules, separate from Linux accounts on the cluster, which allows external collaborators to interact with the storage securely without being given direct access to the cluster’s internal filesystem (Project 2025e; Project 2025f).

The RADOS Gateway supports many features of Amazon S3, such as bucket creation, object versioning, fine-grained access policies, and metadata management (Project 2025f). Moreover, Ceph’s unified design means that data stored through one interface can, in some cases, be accessed through another: for example, data written via the S3 protocol could also be read using OpenStack Swift (another popular object storage protocol) (Project 2025e). This interoperability highlights Ceph’s flexibility as a storage backend.

From the perspective of ORFEO, deploying a Ceph RADOS Gateway would allow research groups and external partners to upload, download, and manage their data through the familiar S3 protocol over standard web connections. For instance, a laboratory could transfer large microscopy datasets to ORFEO simply by targeting the S3 endpoint with existing tools, rather than having to mount a network file system or rely on manual file transfers (Project 2025f). This can greatly simplify data sharing and integration, especially in collaborative environments where not all users have direct logins to the HPC infrastructure.

Looking ahead, this S3 compatibility also creates a natural integration point with higher-level software. Many modern web frameworks and data platforms (including the Django web framework, which we will consider later) offer built-in or plug-in support for S3 storage backends. This means that a Django-based web portal running on ORFEO could treat the Ceph cluster's object storage just like Amazon S3, using standard libraries to upload, retrieve, and manage user files (Project 2025f; Project 2025e).

In summary, the Ceph RADOS Gateway transforms the Ceph cluster into a private cloud storage service by exposing an S3-compatible object storage API (Project 2025f; Project 2025e). While CephFS provides traditional file-based access within the cluster, RGW makes the same reliable storage accessible externally through web protocols, enabling secure and scriptable data exchange with collaborators, laboratories, and applications. This dual approach—file system access for HPC workloads and object storage for external and web-based workflows—broadens ORFEO's utility as a central data platform.

2.2 Identity and access management (IAM): FreeIPA with Authentik

Modern computing infrastructures typically provide many different services (web portals, storage systems, analysis tools), but users should not be forced to maintain separate usernames and passwords for each. Instead, organizations deploy an **Identity and Access Management (IAM)** system that centralizes accounts and authentication. This ensures that users have *one set of credentials* to access multiple services, while administrators retain a single point of control for enabling, disabling, or updating accounts.

In ORFEO, IAM follows a *directory* \rightarrow *identity provider* pattern. The authoritative directory is provided by **FreeIPA**¹, while the user-facing single sign-on layer is provided by **Authentik**².

In this arrangement, FreeIPA acts as the *source of truth* for identities, while Authentik synchronizes with FreeIPA via LDAP to provide a standards-based login and token service for web applications (goauthentik 2025a; FreeIPA Project 2025a). The two roles can be summarized as:

- *FreeIPA* \rightarrow manages user and group accounts, and provides the underlying directory.
- *Authentik* \rightarrow synchronizes with FreeIPA and acts as an SSO gateway, issuing tokens that client applications can validate.

¹FreeIPA is an open-source identity management platform that combines two major components: (i) an *LDAP* directory (Lightweight Directory Access Protocol) for storing structured identity data such as users and groups, and (ii) a *Kerberos* realm, a ticket-based authentication protocol originally developed at MIT for secure single sign-on in distributed systems. FreeIPA is also the upstream community version of Red Hat Identity Manager.

²Authentik is an open-source *identity provider (IdP)*, meaning it is the system that users directly log into. It supports modern web authentication standards such as OAuth2, OIDC (OpenID Connect), and SAML (Security Assertion Markup Language). Comparable commercial systems include Okta and Auth0; a popular open-source alternative is Keycloak.

Token flow and claims

When a user attempts to access a protected application (for example, a Django portal), the application redirects the user to Authentik for sign-in. Authentik validates the credentials against FreeIPA’s directory, and on success issues an **ID Token**³. The application trusts this token, as configured in its OIDC client settings, and establishes a session for the user. Importantly, the application itself never handles raw passwords: authentication is fully delegated to Authentik and FreeIPA (goauthentik 2025b; Sakimura et al. 2014). This mechanism enables **single sign-on (SSO)** across multiple services with a single set of credentials.

Group-based authorization

Beyond authentication, this setup supports centralized authorization. Because Authentik imports groups from FreeIPA during synchronization, it can embed group information as **claims**⁴ in the tokens it issues. Applications can then map these group claims to local roles or permissions. For example, all members of a FreeIPA group `app-admins` could automatically become administrators in a Django web application. Administrators only need to manage membership once in FreeIPA, and the changes propagate consistently to all relying applications (goauthentik 2025c; FreeIPA Project 2025b).

In summary, ORFEO’s IAM solution combines FreeIPA for directory and group management with Authentik for OIDC-based single sign-on. This separation of responsibilities improves both security (applications never store passwords) and usability (users log in once, then access multiple services). The same pattern is reproduced and validated in the VirtualOrfeo environment, as described in Chapter 4.

³An ID Token in OIDC is a *JSON Web Token (JWT)*, i.e. a compact, signed JSON object. A JWT can carry information (“claims”) such as username, email, or group membership, and can be cryptographically verified by the application without contacting the IdP again.

⁴In OIDC, a claim is a key–value pair inside a token that provides identity attributes, such as `email=alice@example.org` or `groups=[researchers,admins]`.

Chapter 3

Application platform choices: Django and PostgreSQL

3.1 Why a web application for lab workflows (upload + annotation together)

Electron microscopy data are most useful when the raw signals (e.g. TIFF images, spectra) and their context (who acquired them, with which settings, on which sample, and for what purpose) are captured together. In practice, however, these two streams often diverge: files accumulate on shared drives while contextual notes sit in lab books or spreadsheets. A web application brings them back into the same place and time. By presenting a form-driven interface in the browser, it allows users to *deposit* raw data and *annotate* it in a single, guided interaction. The form can validate required fields, enforce controlled vocabularies, and provide immediate feedback about missing or inconsistent entries, reducing the need for follow-up and helping the lab converge on shared conventions.

A web application also makes it straightforward to separate human–computer interaction from heavier processing. Once a deposit has been accepted, the server can take over the structured steps needed for durable storage, without holding the user’s browser open. Centralizing uploads likewise enables uniform access control, consistent record-keeping, and the provision of an *application programming interface (API)*¹ for scripted ingest from acquisition workstations. Finally, a web application can mirror the laboratory’s working practices in its interface and underlying database, so that daily activities map naturally onto durable storage structures. The concrete hierarchy used in this project is described later in Chapter 5.

3.2 Django: Model–View–Template and project layout

Django is a Python web framework designed to encourage clear separation of concerns and fast iteration (Django Software Foundation 2025a). Its architectural pattern is called *Model–View–Template (MVT)* (Django Software Foundation 2025b). This separates the application into three main components:

¹API: an interface that exposes application capabilities to software clients, allowing automation and integration.

- **Model:** Represents the data layer of the application. Models define the database structure and handle all data-related logic.
 - Implemented in Django as Python classes (subclassing `models.Model`), each mapping to a database table.
 - Responsible for creating, reading, updating, and deleting records.
 - Typically backed by relational databases such as MySQL, PostgreSQL, or SQLite (Django Software Foundation 2025c).
 - In our project, `experiment_manager/models.py` defines models such as `Project`, `Proposal`, and `Sample`.
 - Relationships are expressed through foreign keys (e.g. a `Proposal` belongs to a `Project`; a `Sample` belongs to a `Proposal`), and Django’s ORM enforces these links in the database.
- **View:** Contains the business logic and request handling. A view receives an HTTP request, processes it, and returns an HTTP response.
 - Implemented in Django as Python functions or classes (often defined in `views.py`).
 - Interacts with models to retrieve or update data.
 - Delegates presentation to a template after preparing the necessary context data (Django Software Foundation 2025d).
 - In our project, modules such as `experiment_manager/views.py`, `views_manage.py`, and `view_modals.py` handle user actions.
 - Examples include displaying dashboards and processing form submissions for new proposals or experiments.
- **Template:** Manages the presentation layer of the application. Templates define how data is displayed to the user.
 - Written as HTML files, optionally enriched with the Django Template Language for dynamic content.
 - Contain static structure and placeholders/tags where dynamic data is inserted.
 - The template engine fills these placeholders using context data from views (Django Software Foundation 2025e).
 - In our project, templates are located in `experiment_manager/templates/`, e.g. `homepage.html` or `dashboard.html`.
 - These templates determine the UI layout and are rendered by views to produce the final HTML pages shown in the browser.

In the MVT workflow, an incoming request is routed by Django’s URL dispatcher (configured in `urls.py`) to the appropriate view. The **view** may interact with **models** to fetch or update data, and then passes that data to a **template** for rendering into an HTML page. The view finally returns the rendered HTML to the client as an HTTP response (Django Software Foundation 2025f; Django Software Foundation 2025g).

This separation of concerns makes the code more maintainable and reusable: the same model can support multiple views or templates. Django’s design emphasizes the **Don’t Repeat Yourself (DRY)** principle, keeping logic (views), data access (models), and presentation (templates) decoupled for clarity and reuse (Django Software Foundation 2025h).

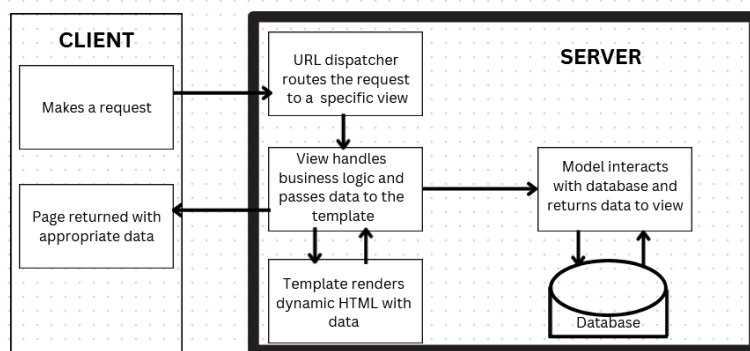


Figure 3.1: Django Model–View–Template (MVT) architecture (Collins 2021).

3.3 Relational databases and the choice of PostgreSQL

To store and manage the application’s data (such as project info, proposals, samples, etc.), we chose a **relational database** as the backend.

A *relational database* is a type of database that organizes data into tables with rows and columns, and links those tables through defined relationships (keys) (Elmasri and Navathe 2015). In other words, each entity in our system (Project, Proposal, Sample, Experiment, Measurement) can be represented as a table, and relationships (like which samples belong to which proposal) are modeled via foreign keys.

This approach ensures data is stored in a structured way with referential integrity – for example, a `Measurement` record cannot exist without its associated `Experiment`, and an `Experiment` is linked to a `Sample`, and so on, maintaining a hierarchy.

Relational databases use the SQL language for querying and manipulating data, which is a well-established standard for database operations (Elmasri and Navathe 2015).

We opted for a relational database because our data is highly structured and interrelated. The hierarchical nature of projects→proposals→samples→experiments lends itself well to the relational model – we can enforce that hierarchy through foreign key constraints and ensure consistency (e.g., preventing a user from creating an `Experiment` without specifying its parent `Sample`).

Relational databases also provide **ACID** properties (Atomicity, Consistency, Isolation, Durability), which guarantee reliable transactions and consistency of data – an important factor when lab users are uploading and annotating critical experiment data (The PostgreSQL Global Development Group 2025a).

Using a robust DBMS (Database Management System) allows multiple users to collaborate and access data concurrently with safety mechanisms for concurrent writes and reads.

Django’s database support: Django is designed to work with relational databases out of the box. It officially supports several popular SQL databases, including:

- PostgreSQL,
- MySQL (and its fork MariaDB),
- Oracle,
- SQLite (Django Software Foundation 2025c).

By default, a new Django project is often configured with SQLite (a lightweight file-based database) for convenience in development. SQLite requires no separate server and is great for testing or prototyping, but it is not ideal for multi-user production use due to its limitations in concurrency and scale (SQLite Consortium 2025).

For a production environment and for handling potentially large volumes of data, a more robust client-server database is preferable. Django makes it straightforward to switch the database engine since the model layer is abstracted – you simply change the database configuration in `settings.py` and run migrations, without having to rewrite model code (Django Software Foundation 2025i).

In our case, we use SQLite for quick local tests (because it’s simple and requires no setup), but for the deployed application we use a dedicated relational database server.

3.3.1 Choosing PostgreSQL as the database backend

After evaluating the options, we decided to use **PostgreSQL** as the database backend for our application. PostgreSQL (often simply “Postgres”) is a powerful open-source relational database system known for its reliability and rich feature set (The PostgreSQL Global Development Group 2025a). Below are the key reasons for this choice:

- **Reliability and integrity:**
 - Proven architecture with a long track record of stability (The PostgreSQL Global Development Group 2025a).
 - Fully ACID-compliant for decades, ensuring safe concurrent transactions.
 - Reduces the risk of data corruption (e.g. experiment annotations and metadata).
 - Can be trusted to faithfully record uploads and form submissions.
- **Advanced feature set:**
 - Supports complex queries and a wide range of data types (including JSON for semi-structured data).
 - Built-in full-text search capabilities (The PostgreSQL Global Development Group 2025b; The PostgreSQL Global Development Group 2025c).
 - Highly extensible through user-defined functions and extensions (e.g. PostGIS).
 - Strong adherence to SQL standards, making it a future-proof choice (The PostgreSQL Global Development Group 2025a).
- **Scalability and performance:**
 - Efficiently handles large datasets and many concurrent users (The PostgreSQL Global Development Group 2025a).
 - Scales well as the number of experiments and users grows.
 - Optimizations include indexing, query planning, and replication support.
 - Solid performance for both read-heavy and write-heavy workloads, with tools for tuning and optimization.
- **Community support and Django recommendation:**
 - Free and open-source, with decades of active community development.

- Widely recommended within the Django ecosystem for production deployments.
- Django creators highlight PostgreSQL as the most balanced backend in terms of cost, features, speed, and stability (Django Software Foundation [2025c](#); Django Software Foundation [2025j](#)).
- Many Django features (e.g. `JSONField`, GIS support) work best with PostgreSQL (Django Software Foundation [2025j](#)).
- Rich ecosystem of documentation, examples, and community plugins tailored for Django + PostgreSQL.

The configuration is straightforward – we installed the `psycopg2` adapter (which allows Django to talk to Postgres) and updated our `DATABASES` setting in Django to use the `django.db.backends.postgresql` engine with the appropriate credentials. Django handles the rest through its object–relational mapping system, so all our data models defined in Python were automatically translated into PostgreSQL tables via migrations. Overall, PostgreSQL provides a solid foundation for storing the application’s critical data, offering both peace of mind (in terms of reliability) and powerful capabilities if we need them down the road.

3.3.2 Django’s ORM (Object-Relational Mapper)

One of Django’s most powerful features is its **Object-Relational Mapper (ORM)**, which bridges the gap between our Python code and the relational database. The Django ORM is an abstraction layer that allows developers to interact with the database using Python classes and methods instead of writing raw SQL queries (Django Software Foundation [2025k](#)).

In practice, this means we define our data models as Python classes (in `models.py`), and Django will automatically generate the necessary SQL to create tables, insert or retrieve records, and so on. For example, a simplified version of a model in our app might look like:

```
class Proposal(models.Model):
    title = models.CharField(max_length=100)
    project = models.ForeignKey(Project, on_delete=models.CASCADE)
    date_created = models.DateTimeField(auto_now_add=True)
    # ... other fields
```

When we run Django’s migrations, the ORM translates this model definition into a SQL `CREATE TABLE` statement in PostgreSQL. The table will have columns for each field (`title`, `project_id`, `date_created`, etc.).

The `ForeignKey` to `Project` tells Django (and the database) that each `Proposal` is related to a `Project`. The ORM enforces this relationship at the database level (e.g., by preventing a proposal from existing without a valid project reference, thanks to PostgreSQL’s foreign key constraints).

Importantly, we did not have to write any SQL by hand; Django’s migration system and ORM handled it for us. This approach improves productivity and reduces the likelihood of errors. By eliminating the need to write manual SQL queries for most operations, the ORM helps **reduce errors and speed up development**, allowing us to focus on business logic rather than database syntax (Django Software Foundation [2025k](#)).

Typical operations with the ORM:

- Create a new object with `Proposal.objects.create(...)`.
- Query records with methods like `.filter()` or `.all()`.

- Update by modifying attributes and calling `.save()`.
- Delete with `.delete()`.

Under the hood, Django translates these high-level operations into optimized SQL queries. For instance, `Proposal.objects.filter(title__icontains="test")` becomes a `SELECT` query with a `WHERE` clause in SQL.

The developers using our codebase don't need to know SQL dialect differences or complex joins; Django ORM takes care of that. It even provides an API for traversing relationships. For example: retrieving all `Experiments` for a given `Project` can be as simple as `project.experiment_set.all()`. The ORM translates this into the appropriate SQL join.

Benefits of Django's ORM:

1. **Database independence** – since our code is not tied to a specific SQL syntax, we can switch the underlying database engine (e.g., SQLite to PostgreSQL) with minimal changes (Django Software Foundation [2025k](#)). This is useful in our cycle: tests run on SQLite, deployment uses PostgreSQL.
2. **Security** – Django automatically escapes parameters to protect against SQL injection (Django Software Foundation [2025l](#)).
3. **Integration with Django components** – once a model is defined, Django can auto-generate an admin interface and forms for it, without extra SQL work (Django Software Foundation [2025m](#)).
4. **Support for complex lookups and relationships** – we can express advanced queries in a readable way. If needed, we still have the option to write raw SQL or use lower-level cursors, but so far the ORM has been sufficient (Django Software Foundation [2025n](#)).

Chapter 4

Deployment and validation in the VirtualOrfeo environment

4.1 VirtualOrfeo as a digital twin of the HPC environment

To develop and test the data management platform in a realistic but safe setting, we use an internal system called **VirtualOrfeo**. VirtualOrfeo functions as a *digital twin* of the ORFEO HPC cluster: a virtualized replica of its key infrastructure and services. This allows us to deploy and evaluate applications under conditions that closely mimic production, without the risk of disrupting the actual cluster. Digital twins are often used in High-Performance Computing (HPC) to experiment with configuration changes or new workflows before rolling them out in production (Kammeyer et al. [2023](#)).

VirtualOrfeo was originally created by Isac Pasianotto, a PhD student at the University of Trieste, and Ruggero Lot, former technologist and system administrator of the ORFEO cluster, as a general-purpose environment for ORFEO infrastructure testing. The work in this thesis builds on their platform: we use it to deploy and integrate the data management application, while the underlying environment itself is their contribution.

The system is built from a collection of virtual machines (VMs) that replicate the main roles of the HPC cluster. They run on Linux KVM/QEMU and are managed through Vagrant and Ansible. The virtual machines represent essential services rather than the full production scale. For example:

- An authentication VM provides directory and identity services.
- One or more VMs host a lightweight Kubernetes cluster for containerized applications.
- Optional VMs simulate HPC login nodes, compute nodes with a Slurm scheduler, and storage nodes.

The goal is to preserve fidelity where it matters while keeping the environment manageable. For instance, VirtualOrfeo runs a small Ceph cluster to mirror the storage architecture of ORFEO, and it includes a simplified Kubernetes deployment for cloud-native services. Differences remain: the virtual system operates at smaller scale, lacks specialized hardware, and cannot match the performance capacity of the real cluster. Still, the software stack and APIs are close enough to ensure that if our application runs successfully on VirtualOrfeo, it will work on the production cluster with only minor adjustments.

In practice, VirtualOrfeo provides a controlled environment to connect our Django-based application with HPC components such as authentication, storage, and container orchestration. By offering a faithful but bounded replica of ORFEO’s software environment, it supports safe experimentation and rapid iteration, in line with DevOps practices in scientific computing (Kammeyer et al. 2023).

4.2 K3s cluster topology in the virtual environment

At the heart of VirtualOrfeo is a Kubernetes cluster deployed on one of the virtual machines (or across a couple of VMs) using **K3s**. K3s is a lightweight, certified Kubernetes distribution designed for resource-constrained or edge environments (Rancher Labs and CNCF 2025). Unlike a full-fledged multi-node Kubernetes deployment, K3s packages the essential components into a single, easy-to-install binary, reducing the complexity and footprint of the cluster. In VirtualOrfeo, this K3s cluster provides the platform on which our web application and related services are orchestrated as containers.

The cluster topology is relatively simple. In our setup:

- A single VM (`kube01`) acts as the Kubernetes control-plane node and also runs as a worker node. This means the API server, scheduler, and controller-manager run on that VM, and it also hosts the application pods. (The K3s design allows one to add additional worker nodes easily if needed, but one node is sufficient for testing purposes.)
- Another VM (`ipa01`) runs the identity management services (discussed later).
- Additional VMs can be spun up for storage or other auxiliary roles (e.g. Ceph OSD servers).

All these VMs are connected via a private virtual network so that services can reach each other using hostnames like `ipa01.virtualorfeo.it` or `kube01.virtualorfeo.it`.

Within the K3s cluster, we adopt a logical separation of concerns using Kubernetes **namespaces**. For example, the Authentik identity provider is deployed in its own namespace (called `authentik`), and our Django application is deployed in another namespace (e.g. `lame-fair` for our “FAIR data” app). Namespaces provide isolation for configuration and secrets, and they help mirror the structure of a real cluster where different services might be managed by different teams.

To make services in K3s reachable from outside the cluster (by default they can only be accessed inside the Kubernetes network), the VirtualOrfeo setup uses two key components: a Kubernetes **Ingress**¹ controller, and a software load-balancer called **MetalLB**².

The ingress controller (NGINX in our case) listens on a special “cluster IP address” and forwards incoming requests to the right service, based on the hostname in the URL. For example, a request to `lame-fair.k3s.virtualorfeo.it` should go to our Django application, while a request to `minio.k3s.virtualorfeo.it` should go to the object storage service.

MetalLB assigns a single virtual IP address (for example `192.168.132.100`) to the ingress controller. That IP is reserved on the local network and represents the “entry point” for all external requests.

¹In Kubernetes, an *Ingress* is a component that accepts incoming web traffic (HTTP/HTTPS) from outside the cluster and routes it to the correct service inside the cluster, based on the hostname or URL path.

²MetalLB is an open-source load balancer for Kubernetes in bare-metal or virtualized environments. It allows cluster services to be assigned real IP addresses on the local network, instead of being confined to internal-only addresses.

To make this usable from a browser, we define DNS entries³ (or, during development, simple `/etc/hosts`⁴ mappings). Example entries are:

- `auth.k3s.virtualorfeo.it`
- `lame-fair.k3s.virtualorfeo.it`
- `minio.k3s.virtualorfeo.it`

All these hostnames point to the same ingress IP (`192.168.132.100`). The NGINX ingress then looks at the hostname of the request to decide which service to send it to. In practice, if we open `https://lame-fair.k3s.virtualorfeo.it`, the request reaches the ingress controller on the cluster VM, which forwards it to the Django service inside Kubernetes.

Because these services are accessed over **HTTPS**, they also need **TLS certificates**⁵. In VirtualOrfeo, the certificates are issued by a local Certificate Authority (the FreeIPA server, acting as a CA). This gives us proper HTTPS without relying on the public internet. We configured **cert-manager** in the cluster to automatically request and refresh TLS certificates from this internal CA using the ACME protocol, so that every service hostname always has a valid certificate.

In terms of storage for the cluster, K3s provides a default storage class (built on local disk storage via the “local-path” provisioner), which we use for most purposes. For example, the PostgreSQL database backing our Django app uses a `PersistentVolumeClaim` that by default binds to local storage on `kube01`. This is acceptable in a virtual environment where data persistence is not mission-critical.

Optionally, VirtualOrfeo’s Ceph deployment could be integrated via a Container Storage Interface (CSI) driver to supply distributed block storage (RBD volumes) or shared filesystem volumes (CephFS) to the cluster, thereby simulating a more production-like storage backend. However, to keep things simple, our deployment mostly relies on the out-of-the-box storage provisioner for now. The key storage integration we do leverage is object storage (Ceph’s RADOS Gateway), which will be discussed in Section 4.4.

To summarize the topology: VirtualOrfeo’s Kubernetes is a lightweight cluster hosting multiple namespaces (for Authentik, the Django app, monitoring tools, etc.), all accessible through an NGINX ingress on a single IP. The infrastructure pieces such as load balancing and certificate management are in place so that each service behaves as if it were in a normal production cluster with distinct hostnames and HTTPS secured by TLS. This setup allows our application to be deployed with the same Helm charts and configurations that we would use on a real cluster, providing high confidence that the behavior will be consistent when transitioning to production.

4.3 Identity and access management in VirtualOrfeo

As outlined in Chapter 2, ORFEO relies on a combination of **FreeIPA** (as the authoritative directory for users and groups) and **Authentik** (as an OIDC identity provider) for identity

³DNS (Domain Name System) is the distributed naming system of the internet: it translates human-friendly names such as `lame-fair.k3s.virtualorfeo.it` into the numeric IP address (`192.168.132.100`) that computers use to connect.

⁴The `/etc/hosts` file is a local configuration file on Linux, macOS, and other Unix-like systems. It can override DNS by mapping hostnames to IP addresses directly on one machine.

⁵TLS (Transport Layer Security) is the standard protocol that secures web traffic over HTTPS. It ensures that the communication between a browser and a server is encrypted and cannot be intercepted or modified by third parties. A TLS certificate is a digital credential that proves a site’s identity and enables this secure communication.

and access management. VirtualOrfeo mirrors this arrangement so that authentication and authorization behave in the same way as in production. The focus here is on the practical steps required to deploy and integrate these services in the virtual environment, rather than reintroducing the concepts already discussed in Section 2.X.

In VirtualOrfeo, the FreeIPA server runs on the VM `ipa01`. It maintains the user directory and also acts as a Certificate Authority, issuing TLS certificates for services. For testing, dummy users can be created in FreeIPA (some are provisioned automatically by Ansible).

On the Kubernetes (K3s) side, Authentik is deployed via Helm in the `authentik` namespace. After deployment, a one-time initial configuration is required:

- **Bind account for LDAP access:** A dedicated service account in FreeIPA (e.g. `svc_authentik`) is created for Authentik. This account has read permissions on users and groups (added to the “User Administrators” role). Authentik uses it to bind to LDAP, synchronize identities, and check credentials during login.
- **LDAP source configuration in Authentik:** In the Authentik admin interface, we configure an LDAP Directory source pointing to the FreeIPA server:

- LDAPS URL (`ldaps://ipa01.virtualorfeo.it`),
- Base DN (e.g. `dc=virtualorfeo,dc=it`),
- Bind DN (`uid=svc_authentik,cn=users,cn=accounts,...`) and password,
- Attribute mapping (`username` → `uid`, `email` → `mail`, full name, groups).

User and group synchronization was enabled, and after the initial sync, all FreeIPA users and groups were imported into Authentik.

- **Federation of FreeIPA with Authentik:** Once synchronization is active, any FreeIPA user can log into Authentik. Authentication requests are validated against FreeIPA, while Authentik manages sessions and issues tokens. FreeIPA thus remains the backend store, and Authentik provides the OIDC-facing layer.
- **Registering the Django app as an OIDC client:** We registered the Django application in Authentik as an OAuth2/OIDC client (“Lame Fair”) with the following configuration:
 - **Client type:** Confidential.
 - **Redirect URIs:** `https://lame-fair.k3s.virtualorfeo.it/oidc/callback/`, `https://lame-fair.k3s.virtualorfeo.it/`.
 - **Flow:** Authorization Code flow with explicit consent.

Authentik generated a Client ID and Client Secret, which we stored in Kubernetes Secrets for injection into the Django deployment.

- **Scopes and claims:** We enabled the standard `openid` scope plus `email`, `profile`, and `offline_access`. This ensures that ID tokens include basic identity information and support refresh tokens. Claims were mapped so that tokens carry `preferred_username` and `email`. Group membership could also be added (e.g. for role-based access), though it was not required for our current application.

Once configured, the login flow works as follows:

- The Django app redirects users to Authentik’s OIDC authorization endpoint.
- Authentik presents a login form and validates credentials against FreeIPA.
- Upon success, Authentik issues an authorization code, which the app exchanges for an ID Token (JWT).
- Django validates the token using Authentik’s public key and logs the user in.

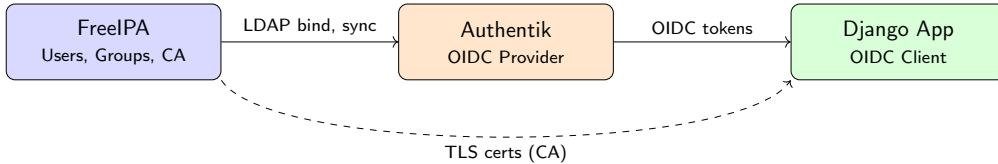


Figure 4.1: Integration of FreeIPA and Authentik in VirtualOrfeo. FreeIPA is the authoritative directory and certificate authority. Authentik synchronizes users and groups via LDAP and provides OIDC tokens to clients. The Django application is registered as an OIDC client and authenticates via Authentik.

At no point does the Django app handle raw credentials; authentication is delegated to Authentik and FreeIPA. This setup enables secure single sign-on (SSO), ensures immediate revocation when a FreeIPA account is disabled, and mirrors the IAM pattern described in Chapter 2.

In summary, the VirtualOrfeo IAM stack demonstrates the same directory → IdP integration used in production, but applied in a safe, virtualized environment. This validates that our Django application can integrate seamlessly with FreeIPA/Authentik, and confirms that the same deployment model can be carried over to ORFEO or replicated with other IdPs such as Keycloak.

With authentication and identity services in place, the next step is to ensure that applications can interact with persistent storage in a reliable and secure way. We therefore examine how VirtualOrfeo incorporates object storage to mirror the production environment.

4.4 Storage integration in VirtualOrfeo

Storage of experimental data is a major concern for our application, and VirtualOrfeo replicates the production storage environment to a large extent. In the real ORFEO facility, high-volume data (like images from microscopes) might be stored on a distributed file system or an object storage service for durability and scalability. In VirtualOrfeo, we deploy the **Ceph** storage system to provide a multi-purpose storage backend. Ceph is a widely used open-source distributed storage platform that can provide object storage, block storage, and file system interfaces within a unified cluster (Weil et al. 2006). We focus on Ceph’s object storage interface, which is provided by the **RADOS Gateway (RGW)** component. The RGW exposes an API compatible with Amazon S3, meaning our application can interact with it as if it were talking to AWS S3, but the data is stored on-premise in Ceph. A broader discussion of RESTful APIs is given later in Section 5.6.

In VirtualOrfeo, a small Ceph cluster is set up using additional VMs (for monitor and OSD daemons). Once Ceph is running, an RGW daemon is deployed (in our case, on one of the Ceph nodes, listening on a specific port and configured with a default Ceph “realm” and “zone” for

object storage). To integrate this with Kubernetes, we created a Kubernetes Service and Ingress that points to the RGW:

- The **Service** of type `ClusterIP` has a backing `Endpoints` object that contains the static IP of the RGW (e.g., 192.168.132.90, the address of the VM running RGW) and the port (7480 for plain HTTP or 8080 with SSL termination). Essentially, this registers an external service inside Kubernetes.
- The **Ingress** for RGW uses the host `rgw.k3s.virtualorfeo.it`. We added annotations to allow large uploads (disabling body size limits and request buffering, since experimental data can be quite large). It routes all paths under that host to the RGW service. The ingress is secured with TLS using a certificate issued by the FreeIPA CA.
- This setup means that any application inside the cluster can access the RGW via the internal service DNS (`http://ceph-rgw.lame-fair.svc.cluster.local:7480`) or via the external URL `https://rgw.k3s.virtualorfeo.it`. Our Django application uses the external URL because it expects an S3 endpoint with valid TLS.

As an alternative to Ceph RGW, VirtualOrfeo also provides the option of deploying **MinIO**, a standalone open-source object storage server that is S3-compatible. MinIO is easier to set up for simple tests (it can run as a single container). Indeed, our overlay repository contains manifests for a MinIO deployment in the cluster.

The Django application does not need to care whether the underlying storage is Ceph or MinIO – it only sees the S3 API. This abstraction is powerful: it means we can develop against MinIO locally, then switch to Ceph RGW in integration tests, and eventually use the same code with an Amazon S3 bucket in production.

Within the object store, our application creates and uses buckets to organize data. The data model we follow (as described in Chapter 3) involves Projects, Proposals, Samples, and Experiments. We reflect this hierarchy in the storage layout:

- Data is partitioned by project: each top-level project (e.g., `NFFA_DI`, `RIANA`) has a dedicated bucket.
- Within each project’s bucket, object keys (paths) encode the proposal, sample, and experiment.
- For example, a raw TIFF image for Experiment #123 of Sample #5 in Proposal P001 under project `NFFA_DI` would be stored as:

`NFFA_DI/PROPOSAL-P001/SAMPLE-5/EXP-123/raw/imagename.tiff`

In this scheme, the bucket might be named `lamefair-data` or `nffa-di-data`, and the directories (prefixes) group the data. The application’s code (in the `pathing.py` and `s3.py` utilities) handles constructing these S3 keys so users do not have to worry about the exact paths.

After an upload, the system generates a NeXus file (see Section 1.4) containing metadata and possibly processed data. We store these NeXus files alongside the raw data or in a parallel location. Common strategies include:

- storing NeXus files in a `processed/` subdirectory of the same experiment path,

- placing them in a separate bucket for easier access by portals, or
- using a “mirroring” strategy, where raw data remains in one bucket while NeXus files are copied to another.

In VirtualOrfeo, we adopted a simple approach: NeXus files are written to the same bucket with a distinct prefix (e.g., `nx/experiment_123.nxs`). This way, the data portal or UI can easily locate them.

Using an open, self-describing format like NeXus for processed data aligns with best practices in scientific data management, ensuring results are sharable and standardized rather than proprietary (Könnecke et al. 2015b; Korir, Kleywegt et al. 2024). NeXus (built on HDF5) can encapsulate not only the image data but also relevant metadata (instrument settings, timestamps, sample identifiers, etc.) in one file (Könnecke et al. 2015b). This ensures that years later, users still have a usable record of the experiment, and that the data remain interoperable with community platforms like NOMAD⁶.

This focus on interoperability and reuse reflects the FAIR principles (Findable, Accessible, Interoperable, Reusable). By having VirtualOrfeo’s storage mimic the final layout, we verified that our application correctly places files and that NeXus generation produces files compatible with external tools.

To facilitate testing, we also set up some dummy buckets and data. VirtualOrfeo’s Ceph RGW can be configured with an initial bucket structure or seeded with example data (via Ansible scripts). This was useful for integration tests, such as verifying that:

- listing objects shows the expected keys,
- the app detects when a bucket is empty versus containing experiments.

In conclusion, VirtualOrfeo’s storage integration provided a realistic S3-compatible endpoint. Our application could create buckets, put objects (uploads), get objects (downloads), and list contents just as it would on a real object store. Using Ceph RGW meant performance and concurrency were closer to a production scenario than with a simple local filesystem. It also allowed us to test failure modes (e.g., wrong credentials, unreachable store).

All of this contributes to confidence that the application will handle data storage reliably when deployed to the HPC environment or any other S3-compatible cloud storage.

With authentication and storage available, the environment is ready to host real applications. The following subsection describes how a representative Django-based web service is packaged and deployed on the VirtualOrfeo cluster.

4.5 Application packaging and deployment on K3s

To demonstrate how a web application can be deployed in the VirtualOrfeo cluster, we use a representative **Django** application packaged and deployed with **Helm**. The application is built into a container image and distributed as a Helm **chart**, which bundles all Kubernetes manifest templates (Deployments, Services, ConfigMaps, Secrets, etc.) along with configuration values. This approach ensures that deployments are versioned, reproducible, and portable across any Kubernetes cluster.

In our repository, the Helm chart is maintained under `charts/lame-fair/`. It defines the following Kubernetes resources, which are typical for deploying a Django-based service:

⁶<https://nomad-lab.eu/>, a FAIR data platform for materials science that supports NeXus/HDF5 ingestion

- **Deployment of the web service:** Creates pods that each run two containers:

- a container running Django (served by Gunicorn),
- an NGINX sidecar acting as a reverse proxy and serving static files.

This separation allows NGINX to handle static assets efficiently while Gunicorn processes dynamic requests. NGINX also ensures the application is exposed only on standard ports (80/443 through ingress).

- **Deployment of a worker:** Many Django deployments rely on a background task queue (e.g. Redis Queue, Celery). To reflect this pattern, the chart defines a second Deployment running a worker container that processes queued jobs, keeping the main web service responsive.
- **Redis instance:** A Redis server (via Bitnami’s sub-chart) is deployed as the queue backend for asynchronous tasks. It runs as a single pod with a PersistentVolume for durability.
- **PostgreSQL database:** A PostgreSQL instance (via Bitnami’s chart) provides the relational database backend. The chart provisions credentials in a Kubernetes Secret and PersistentVolumes for storage. While single-node here, in production one would typically rely on a managed or replicated database.
- **Jobs and hooks:** A Kubernetes Job runs Django’s database migrations on install or upgrade, ensuring the schema is synchronized with the models. This Job is triggered as a Helm hook.
- **Ingress and Service:** An Ingress exposes the service at `lame-fair.k3s.virtualorfeo.it`, with traffic routed by the cluster’s NGINX ingress controller. Certificates are provisioned by cert-manager via the FreeIPA CA.
- **Configurations:** Settings are supplied through a ConfigMap (for non-sensitive values) and a Secret (for sensitive values such as the Django SECRET_KEY, OIDC client secrets, and database password).

The container image for the Django application is built from a Dockerfile and published to a container registry. It includes all dependencies (`requirements.txt`), the application code, and pre-collected static files. This image, combined with the Helm chart, provides a portable and reproducible deployment unit.

Application packaging, however, is only part of the deployment story. Equally important is the secure and flexible management of configuration values and secrets, which is handled through Kubernetes-native mechanisms.

4.6 Secrets and configuration management

Managing configuration and secrets is a critical part of deploying the application. In the Virtual-Orfeo environment, the application requires a mix of configuration values: some are non-sensitive (e.g. URLs, feature flags), while others are highly sensitive (e.g. passwords, private keys, API secrets). Kubernetes provides native resources to handle both types: **ConfigMaps** for plain configuration and **Secrets** for sensitive values. Our deployment makes extensive use of both, ensuring that credentials remain secure and configuration stays flexible.

At deployment time, we supply several categories of information to the application:

- **Django settings:** Externalized into ConfigMaps rather than baked into the container image. Examples include:

- whether the app runs in debug mode,
- the allowed hostnames,
- the OIDC callback URL and issuer URL
(e.g. `AUTHENTIK_SERVER_URL=https://auth.k3s.virtualorfeo.it`).

These are set as environment variables (e.g. `DJANGO_SETTINGS_MODULE`, `DJANGO_DEBUG`) via the Helm chart’s `configmap-env.yaml`.

- **Secrets for OIDC and database:**

- OIDC Client ID and Client Secret for the application, generated by Authentik when the app was registered (see Section 4.3).
- PostgreSQL credentials (username, password, host, database name), provisioned by the Helm sub-chart that deploys PostgreSQL. The app reads these values from a Kubernetes Secret (e.g. `pg-postgresql`).

- **S3 storage credentials and endpoints:**

- Dedicated S3 user in Ceph RGW (e.g. “`django-app-user`”) with access and secret keys.
- Keys stored in a Kubernetes Secret (`S3_ACCESS_KEY`, `S3_SECRET_KEY`).
- Endpoint URL (e.g. `https://rgw.k3s.virtualorfeo.it`), treated as non-sensitive and stored in a ConfigMap.

Within Django, these values configure the `boto3` client.

- **Certificates and CAs:**

- FreeIPA acts as the internal Certificate Authority.
- To avoid TLS errors when connecting to Authentik or RGW, the FreeIPA CA certificate (PEM) is mounted into the Django container.
- The app is configured (via `AWS_CA_BUNDLE` or Python SSL settings) to trust this certificate.
- The CA certificate itself is stored in a Kubernetes Secret (e.g. `ipa-root-ca`) and mounted at runtime.

- **Application secrets:** The Django `SECRET_KEY`, used for cryptographic signing (sessions, CSRF tokens, etc.), is generated uniquely for each deployment and stored as a Kubernetes Secret.

The main reason for using ConfigMaps and Secrets is to separate configuration from code,⁷ following best practices for cloud-native deployment.

This avoids hardcoding environment-specific values into the image or repository and aligns with the 12-factor app principle. It also allows credentials to be rotated without changing the

⁷The “Twelve-Factor App” methodology defines twelve best practices for building scalable web applications. Factor III (“Config”) states that configuration such as database credentials or API keys should not be hardcoded in the source but instead supplied at runtime via environment variables or equivalent mechanisms (Wiggins 2011).

application deployment itself. For example, if an S3 key needs replacement, updating the Secret and restarting the pods is sufficient—no rebuild is required.

A further consideration is the use of **scoped credentials**. The S3 access keys created for the application can be restricted to particular permissions:

- In production, one might allow access only to specific buckets or restrict destructive operations (e.g. allow PUT but not DELETE).
- In our VirtualOrfeo setup, we allowed full access to a defined namespace of buckets, but finer scoping is possible if needed.

Similarly, OIDC tokens issued by Authentik can be scoped. By default, our app requests only minimal scopes (`openid`, `email`, `profile`) to retrieve basic identity information. Additional scopes (e.g. group membership) could be added, but limiting scopes to what is necessary is a good security practice.

This principle of scoped access also applies at the application level:

- A frontend designed only for browsing data could use read-only S3 keys or tokens.
- Our current application handles both uploads and browsing, but a future read-only data portal could be configured with narrower permissions.

In summary, the deployment relies on Kubernetes-native mechanisms for securely injecting configuration and secrets. By distinguishing clearly between sensitive and non-sensitive values, we lower the risk of leakage and make configuration easier to manage. Helm charts were written to ensure that secret values are never displayed in logs or upgrade diffs, following common cloud-native security practices.

Chapter 5

Deep dive into the application

This chapter gives an implementation-focused look at the Django application for experimental data management. The system follows a layered architecture that separates business rules, storage access, background processing, and presentation. Experimental data are organized in a clear hierarchy of projects, proposals, samples, and experiments, with each creation automatically writing metadata into S3 buckets alongside the raw files. Large microscope outputs are uploaded directly to object storage through presigned URLs, keeping the web server lightweight, while background workers (Redis Queue) handle checksums, NeXus conversions, and other heavier tasks. Metadata extracted from TIFFs is translated into standardized NeXus/NXem structures via JSON mapping files, making the system adaptable across instruments. A thin storage gateway hides S3 details, simplifying development and testing, while a REST API exposes the same operations available in the browser-based management board. The UI itself uses HTMX modals for interactive creation and validation. Authentication is delegated to Authentik via OIDC; group claims control authorization; presigned URLs enforce strict, time-limited access to storage. Checksums, README files, and logs together ensure traceability, while the architecture separates uploads from processing, allowing independent scaling of web and worker processes. Overall, the platform is robust, scalable, and FAIR-oriented, providing a reproducible and future-proof approach to handling growing volumes of microscopy data.

5.1 Domain model and data flow

In this section, we describe how the application structures and manages its information. We first introduce the core entities of the domain model, then explain how these entities are created and linked to storage through domain commands. Finally, we outline the typical lifecycle of data as it flows from user input to raw storage, metadata capture, and background processing.

a) Entities. The application organises information into four main building blocks, each represented by a Django model:

- **Project:** this is the top-level container, for example *NFFA_DI*. Each project has a human-readable name and a machine-friendly slug (a simplified version of the name, produced automatically by a local `slugify` helper). The slug is guaranteed to be unique so that it can safely be used in URLs and bucket names.
- **Proposal:** a proposal belongs to exactly one project and is uniquely identified by the pair (project, number). In addition to the proposal number, it stores information such as the

principal investigator (`pi_name`), the proposal date, a free-text `description`, and the user who created it. Each proposal also owns exactly one S3 bucket (`bucket`) where its raw data is stored. This bucket name is created automatically the first time the proposal is saved: it combines the project slug and proposal number, and adds an 8-character SHA-1 hash for uniqueness, prefixed with `lame-raw-` (or another prefix from settings).

- **Sample:** a sample belongs to one proposal and is uniquely identified by the pair (`proposal`, `slug`). The slug is derived from the sample's `name`. Samples may also carry optional descriptive fields such as an external `identifier`, a `preparation_date`, the list of `atom_types`, and a description of the `physical_form`. These optional details are written into README files to keep them close to the data.
- **Experiment:** an experiment belongs to one sample and is uniquely identified by the pair (`sample`, `slug`). It can have a free-text `description` and optional start-time metadata. Experiments group together the actual measurement files uploaded by users.

The database enforces these uniqueness rules through Django's Meta constraints (`UniqueConstraint` or `unique_together`), which prevents accidental duplication.

b) Domain commands. To create these entities, the application does not call the models directly. Instead, it uses command functions in `domain/commands.py`, such as `create_proposal`, `create_sample`, and `create_experiment`. These commands wrap several steps into one unit of work:

1. **Friendly validation:** before writing, they check if an entity with the same identifiers already exists, and if so, raise a `ValidationError` with a clear message (avoiding generic database errors).
2. **Safe database write:** the actual insert happens inside a transaction. If two requests collide, the code re-checks after catching an `IntegrityError` so that the user still gets a meaningful error message instead of a crash.
3. **Bucket provisioning:** when a proposal is created, the corresponding bucket is created (or confirmed to exist) using `create_bucket`. Calling this every time is safe, because the operation does nothing if the bucket already exists.
4. **README writing:** immediately after creation, a README file is generated with the entity's metadata (e.g. project name, proposal number, sample identifier, experiment start time) and written to object storage at a standard path such as `<proposal_prefix>/README.txt`, `<sample_prefix>/README.txt`, or `<experiment_prefix>/README.txt`.

In this way, each creation command leaves both the database and the storage layer in sync: a new row is added to the relational database, the bucket (if any) is guaranteed to exist, and a README appears in storage so that anyone browsing raw data can also see the associated metadata.

c) Lifecycle and flow. The typical sequence of actions is:

1. **Creation via forms or API.** Users create projects, proposals, samples, or experiments through HTMX modals (`view_modals.py`) or the REST API. The modals call the domain commands. On success, the server replies with `204 No Content` and triggers a refresh in the browser. If validation fails, the modal is re-rendered with a single error block, instead of returning a generic HTTP error.

2. **README writing.** Right after each entity is created, a README file is written into object storage through the storage gateway. The path is predictable (`<proposal_prefix>/README.txt`, `<sample_prefix>/README.txt`, `<experiment_prefix>/README.txt`), and the body is generated by `services/readme.py`. This way, descriptive metadata always sits next to the raw data.
3. **Measurement registration and raw upload.** To upload data, a user chooses the target proposal, sample, and experiment. The browser then uploads the raw files directly to S3 using presigned PUT URLs from `/s3_presign`. Once the upload finishes, the browser tells the backend which S3 keys were written, together with minimal metadata such as the instrument pair, user name, and a short description. This is handled by `register_measurement()`.
4. **Measurement README and background jobs.** `register_measurement()` creates a measurement-level README next to the uploaded files. It also schedules background jobs: one to compute a checksum for every uploaded file, and another to convert TIFFs into NeXus if the instrument pair supports it. The code makes sure that the needed buckets (RAW and mirror) exist before writing.
5. **Mirroring to derived buckets.** The system derives the name of a “mirror” bucket using `mirror_bucket_for()` and creates it if it does not exist. Derived artefacts such as NeXus files are uploaded here, keeping them separate from raw data. The `/ofed/` view lets users browse this mirrored namespace.

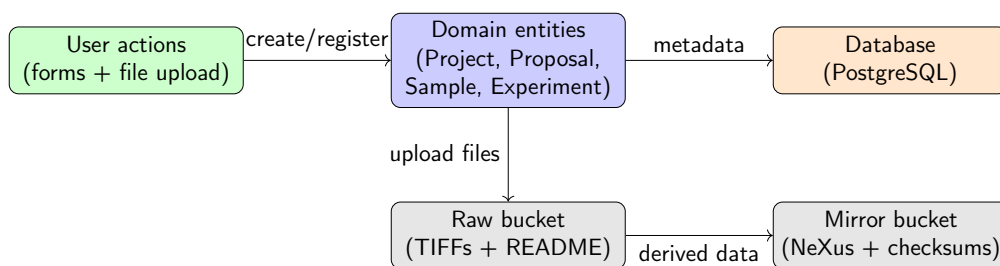


Figure 5.1: Simplified pipeline overview. Users create entities and upload raw data, which is stored alongside metadata in the database and raw buckets. Derived NeXus files and checksums are stored in mirror buckets.

Consistency and error handling. Domain commands always validate inputs before writing, so users get clear error messages instead of database errors. All writes run inside database transactions to avoid partial updates. If two users try to create the same entity at the same time, the code re-checks uniqueness after catching an integrity error. Creating buckets is always safe to call multiple times, because the storage layer ignores duplicates. Background jobs that compute checksums use `update_or_create`, so retries simply update the existing row rather than causing conflicts.

5.2 Metadata path

In this section, we describe how metadata are captured, translated, and stored when new measurements are uploaded. Starting from raw TIFF files, the system extracts instrument-specific

metadata, applies mapping rules to align them with NeXus conventions, and normalizes values into a consistent, machine-readable NXem structure.

a) TIFF ingestion. Most raw microscope outputs arrive as TIFF files. When a user uploads them, the system only stores the object keys immediately; the heavy work of reading the files is left to background workers. TIFF parsing happens inside the builder services (see §5.3). For large files, the app relies on S3's `StreamingBody` so data can be read in chunks instead of all at once. Typical chunk sizes are 8–16 MiB, which are also used in checksum calculation and ZIP streaming to avoid memory spikes.

b) Instrument metadata extraction and mapping. Each instrument family writes metadata into TIFF headers in its own way. To avoid hard-coding these differences in the code, the system uses JSON mapping files stored under `experiment_manager/data/` (for example `ED_mapping.json`, `TVIPS_mapping.json`). These files describe how to translate instrument-specific keys into NeXus fields. The chosen instrument pair (instrument + detector) is passed from the UI (PAIRS in `views.py`) through the `register_measurement()` call and finally into the builder.

c) Translation to NXem concepts. The NXem definition (NeXus for electron microscopy) expects a common structure: an entry node with subgroups for instrument, detector, and sample, each containing well-known fields such as pixel size, accelerating voltage, camera length, and dwell time. The mapping layer is responsible for filling this structure by translating the raw metadata. Concretely:

- *Source fields:* information pulled from TIFF tags (e.g. `ImageDescription`, `XResolution`, `YResolution`, `Software`), text written into measurement README files, and extra form inputs like operator or description.
- *Mapping:* each JSON file matches these source keys (sometimes with path expressions inside `ImageDescription`) to target NeXus paths (for example `/entry/instrument/detector/exposure_time`).
- *Normalization:* values can be converted to consistent units (e.g. seconds instead of milliseconds, metres instead of nanometres), cast to the right data type, and filled with sensible defaults if missing.

By moving the instrument-specific rules into data files, the builder code stays simple, reusable, and easy to test.

5.3 NeXus construction

In this section, we describe how the system constructs NeXus files from the raw inputs and metadata prepared earlier. The process separates low-level file writing from orchestration, applies common scaffolding across all instruments, and then fills in instrument-specific details using mapping rules. The result is a standardized NXem structure ready for storage and reuse.

a) Responsibilities and split. The NeXus build process separates the code that *creates* a NeXus file from the code that decides *where and when* it should be stored:

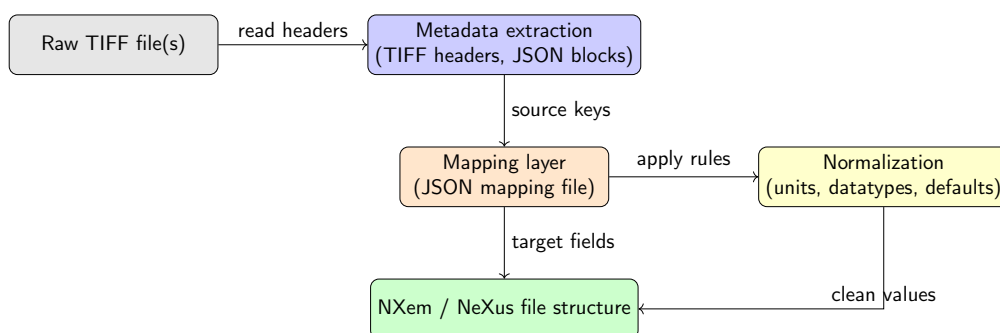


Figure 5.2: Metadata flow during ingestion. Raw TIFF headers are extracted and flattened, mapped through instrument-specific JSON files, optionally normalized, and written into the standardized NXem/NeXus structure.

- `services/nexus_builders.py`: contains the low-level writers that take one TIFF file and a JSON mapping, and turn them into a valid NXem tree (backed by HDF5). The main functions are `build_nexus_from_tiff_TEM_ED(...)` and `build_nexus_from_tiff_TVIPS(...)`.
- `services/nexus.py`: handles the orchestration around storage. It chooses the right builder for the selected instrument pair, finds or creates the mirror bucket, manages temporary files, and uploads the finished `.nxs`. Its main entry point is `build_and_upload_nexus(...)`.

b) Common scaffold (writer core). Both builders share a common function `_build_scaffold(...)` that sets up the basic NeXus file. It runs through the following steps:

1. **Read and flatten metadata.** The first TIFF page is opened with `tifffile`. All tags are flattened into a single dictionary with dot-separated keys. If a string tag contains embedded JSON (for example TVIPS puts a JSON block inside `ImageDescription`), it is parsed, flattened, and merged. Lists are indexed with keys like `foo[0].bar`.
2. **Load mapping file.** A JSON mapping file (for example `ED_mapping.json` or `TVIPS_mapping.json`) is read into a dictionary of `source_key->nexus_dotted_path`. Both `/` and `.` are accepted as separators in the paths.
3. **Create root and entry.** An `NXroot` is created with the output file name. Inside it, an `NXentry` is added and marked with `definition="NXem"`. Group names use uppercase or explicit naming to satisfy NeXus validators.
4. **Add sample block.** A required `SAMPLE` group (`NXsample`) is created with default values: `name="UNKNOWN_SAMPLE"`, `identifier_sample="unknown_id"`, `physical_form="bulk"`, and a boolean flag `is_simulation`. If extra fields from the README forms are provided (see §5.2), they replace the defaults: `sample_identifier`, `preparation_date` (normalized to full ISO date-time), `atom_types`, and `physical_form`.
5. **Add coordinate system.** An `NXcoordinate_system` group called `coordinate_system_1` is attached with fixed values: `type=cartesian`, `handedness=right`, `origin=front_top_left`.
6. **Apply mapping.** For each flattened TIFF key found in the mapping, the corresponding NeXus path is created. Intermediate groups are added as needed, and leaf values are stored as `NXfield(val)` without conversion.

7. **Normalize start time.** If the mapping produced separate fields `start_time_date` and `start_time_time`, they are combined into a single ISO8601 `start_time`. If no start time is present at all, the default `1970-01-01T00:00:00Z` is used.
8. **Embed image data.** The image is read with `tiffimage.imread` and stored under `image_2d` as an `NXdata` group with `signal="data"`.
9. **Write extras and header dump.** Any extra fields not placed under `SAMPLE` are written at the `NXentry` level. If a header dictionary is given, it is saved as a JSON `NXnote` called `hdr_metadata`.

c) Instrument-specific population. After the scaffold, a small *population* step adds instrument-facing structure in a uniform way. The builder selects a pair-specific helper from the registry and uses it to (i) ensure the expected `NXem` layout exists and (ii) attach common instrument nodes. Concretely, the helper:

- guarantees the presence of an events branch `NXentry/measurement/events/event_data` with `NX_class=NXevent_data_em`;
- creates an instrument subtree (`NXinstrument_em`) and, where relevant, children such as `ebeam_column` (`NXebeam_column` with `electron_source` `NXsource` and `filter` `NXfilter`), `detector` (`NXdetector`), and `stage` (`NXmanipulator`);
- fills missing fields with neutral defaults (e.g., "unknown" for strings, numeric zeros for stage axes) using create-if-missing semantics, so values coming from the mapping or the caller are never overwritten;
- optionally adds a lightweight microscope-identity branch under `NXentry/measurement/instrument` (e.g., `name`, `location`, `vendor/model/serial_number`).

Acquisition-specific quantities (such as exposure time, pixel size, beam voltage, trigger mode) are intentionally supplied by the JSON mapping rather than hard-coded here. Adding support for a new instrument means contributing a new population helper and its mapping file; the scaffold and orchestration remain unchanged.

d) Mapping mechanics (example). Mappings reference *flattened* TIFF metadata keys on the left, and dotted NeXus paths on the right. A (shortened) TVIPS example:

```
ImageDescription.camera.start_time
-> NXentry.measurement.events.NXevent_data_em.start_time
ImageDescription.camera.exposure_time
-> NXentry.measurement.events.NXevent_data_em.instrument.NXdetector.exposure_time
ImageDescription.camera.pixel_size
-> NXentry.measurement.events.NXevent_data_em.instrument.NXdetector.x_pixel_size
ImageDescription.stage.x
-> NXentry.measurement.events.NXevent_data_em.instrument.stage.x_position
ImageDescription.electron_gun.voltage
-> NXentry.measurement.events.NXevent_data_em.instrument.ebeam_column
.electron_source.voltage
ImageDescription.camera.simplon_parameters.trigger_mode
-> NXentry.measurement.events.NXevent_data_em.instrument.NXdetector
.simplon_parameters.trigger_mode
```

```
ImageDescription.repo_id  
-> NXentry.experiment_identifier
```

Paths are created on demand; where the code knows the NeXus class for a node (e.g. `NXdetector`, `NXbeam_column`), it stamps `NX_class` accordingly.

e) What the writer deliberately *does not* do (yet). The current implementation *does not* convert units, change data types, or check the full schema beyond combining the start time and making sure core groups exist. Values are written *as they are* from the flattened metadata. This shifts complexity into the mapping files and keeps the writer straightforward. Section 5.3 describes planned improvements.

f) Orchestration and publishing. The function `build_and_upload_nexus(...)` manages the full flow:

1. **Pair registry.** A fixed registry maps a UI pair (e.g. `TEM_JEOL_F200_TVIPS`) to a builder function and mapping file. Unregistered pairs or missing mapping files are logged and skipped.
2. **Mirror-bucket resolution.** The destination (derived) bucket is computed by rewriting the configured RAW prefix (e.g. `lame-raw-` → `lame-ofed-`); bucket creation is attempted safely.
3. **Repeat-safe existence check.** If an object with the final `.nxs` key already exists in the mirror bucket, the job exits without changes.
4. **Upload race-avoidance.** To avoid conflicts with just-written objects, the code polls `HeadObject` with a short timeout/backoff. If the low-level client is unavailable, it briefly sleeps instead.
5. **Local build.** The RAW TIFF is downloaded to a temp file; *extra fields* are collected by reading and parsing the per-sample and per-experiment README files (see §5.2). The selected builder produces a temp `.nxs`.
6. **Publish and clean up.** The `.nxs` is uploaded to the mirror bucket under the same key path (extension changed). Temp files are cleaned in a `finally` block. All steps are wrapped in broad exception handling with structured logging; failures return `None` so the queue can retry if needed.

g) Resulting file layout (as written). The produced tree reflects the code's naming and hierarchy:

- `/NXentry (NXentry)` with `definition="NXem"`, `start_time`, and optional `experiment_identifier`; the sample is mounted at `/NXentry/SAMPLE (NXsample)`.
- `/NXentry/coordinate_system_1 (NXcoordinate_system)`.
- `/NXentry/measurement/events/event_data (NXevent_data_em)` with `instrument (NXinstrument_em)` including `NXdetector`, `NXbeam_column (NXsource, NXfilter)`, and `NXmanipulator stage`.
- `/NXentry/measurement/instrument (NXinstrument_em)` as a static identity branch.

- `/NXentry/image_2d` (NXdata) with the TIFF raster attached as `data` and declared `signal`.
- Optional `/NXentry/hdr_metadata` (NXnote) with JSON-serialized headers.

Group names like `NXentry` and `SAMPLE` are capitalised/explicit rather than the conventional `entry/sample`; this matches the writer and avoids validator warnings in our environment.

h) Operational characteristics. Builders run entirely in-process and load the full TIFF into memory via `tifffile.imread`; this is acceptable for current file sizes. For very large frame stacks, the API can be extended to use memory-mapped reads or external links.

i) Limitations and planned validation The following checks are *not* yet implemented and are planned:

- **Schema validation:** verify required NXem fields, classes, and attributes post-write (e.g. via `nexusformat` traversal or `punx` validation hooks).
- **Unit/dtype normalization:** perform coercion at write-time (e.g. `ms`→`s`, `nm`→`m`) based on per-key policies in the mapping.
- **Axes semantics:** annotate NXdata with axes and units when known; link to detector geometry if provided.
- **Multi-page TIFFs:** support frame stacks with an explicit `signal` dimension and timebase metadata.

j) Traceability and repeat-safety. The target key is deterministically derived from the TIFF key by changing the extension to `.nxs`; re-runs overwrite the same object in the mirror bucket. Logs carry the pair, RAW and mirror locations, and mapping path. Combined with the checksum pipeline (§5.5), this gives a clear and reproducible conversion trail.

5.4 Storage gateway

Why a gateway layer? All files in the system—whether raw microscope outputs or derived NeXus datasets—live in an object store that speaks the Amazon S3 protocol. In practice this means either Ceph RGW or MinIO. While the Python library `boto3`¹ could be called directly from any part of the code, the application instead defines a thin *storage gateway* layer. This layer hides S3-specific details behind a common interface, so that the rest of the codebase only deals with a handful of simple methods such as “create a bucket”, “list objects”, “upload”, “download”, or “generate a presigned URL”.

This abstraction has two main benefits:

- It isolates S3 configuration (endpoints, credentials, and CORS rules²) in one place.
- It allows swapping in a fake, in-memory implementation during testing or development, so that the entire stack can be deployed without requiring a live S3 service.

¹`boto3` is Amazon’s official Python SDK for S3 and related services; it provides functions for creating buckets, uploading objects, and generating presigned URLs.

²CORS, or Cross-Origin Resource Sharing, is explained in detail later in this section.

Gateway interface and implementations. The gateway interface is declared in `storage.py` as a Python Protocol called `StorageGateway`. It defines the expected methods:

- `create_bucket(name)` to create a bucket if it does not exist.
- `list(bucket, prefix)` to list objects under a given prefix.
- `put(bucket, key, body)` to upload a new object.
- `get(bucket, key)` to fetch an object into memory.
- `presign_get(...)` and `presign_put(...)` to generate time-limited URLs for download and upload.

There are two concrete implementations:

Real S3 (`_BotoStorage`) For production, the gateway forwards each call to helper functions in `services/s3.py`, which wrap `boto3`. Here the methods correspond directly to S3 operations: `create_bucket` sends a `CreateBucket` request, `list` pages through `ListObjectsV2`, and `presign_put` uses AWS's Signature Version 4 (SigV4) algorithm to generate a secure upload URL.

Fake S3 (`InMemoryGatewayImpl`) For testing or developer deployments without Ceph/MinIO, the system provides an in-memory backend. Objects are stored in a nested Python dictionary, keyed by `bucket` and `key`. Crucially, the fake implementation also generates *same-origin presigned URLs*: instead of pointing to `https://rgw.k3s.virtualorfeo.it/...`, a PUT `presign` returns a URL such as

```
https://lame-fair.k3s.virtualorfeo.it/s3-fake-put/{bucket}/{key}
```

This URL is handled by two lightweight Django views (`s3_fake_put`, `s3_fake_get`) that accept PUT or GET requests, update the in-memory store, and return appropriate CORS headers. This avoids cross-origin requests entirely and lets the browser upload directly to the Django process.

Bucket naming and strategy. Each Proposal in the domain model owns one dedicated bucket. The bucket name is derived deterministically from the Project slug and Proposal number, plus an eight-character hash to avoid collisions. For example:

```
lame-raw-nffa-42-a1b2c3d4
```

This design has several advantages:

- Buckets clearly separate data between Proposals.
- Names are predictable and collision-resistant.
- Lifecycle management (e.g. deletion or access control) can be handled per bucket.

The system also maintains *mirror buckets* for derived data such as NeXus files. A helper `mirror_bucket_for(...)` rewrites `lame-raw-*` into `lame-ofed-*`, so the raw files and the processed results remain side by side but in separate namespaces.

Key naming conventions. Within each bucket, keys follow a structured prefix layout that reflects the domain hierarchy:

```
<proposal_prefix>(proposal)/README.txt
  <sample_prefix>(sample)/README.txt
<experiment_prefix>(experiment)/README.txt
```

Raw measurements place a `README.txt` next to the uploaded files inside their folder. This “local README” makes browsing easier: metadata is always co-located with the data it describes.

Workflow and presigned URLs. Large microscopy files can be tens or hundreds of gigabytes in size. Routing these through the Django web process would overwhelm server memory and bandwidth. Instead, the system uses *presigned URLs*. A presigned URL is a normal-looking web address that contains a cryptographic signature granting permission to perform one specific action (GET or PUT) on one object, for a limited time (usually one hour).

The workflow looks like this:

1. When a user wants to upload, the browser asks the backend at `/s3_presign/` for a presigned PUT URL.
2. The backend generates it using either `boto3` (real storage) or the in-memory gateway (fake storage).
3. In the real case, the URL points at `https://rgw.k3s.virtualorfeo.it/...`; in the fake case, it points back to Django itself at `/s3-fake-put/...`
4. The browser (via Uppy³) uploads the file to that URL with the given headers.
5. Downloads use the same mechanism: `presign_get` yields either a real S3 link or a local `/s3-fake-get/...` link.

This design cleanly separates the control plane (authorization, URL generation) from the data plane (actual byte transfer), and makes it possible to run the whole system locally with no external dependencies.

Browser constraints: CORS and preflight. When using real S3, the storage endpoint is a different origin from the web app⁴, so browsers apply the same-origin policy, which blocks requests to other origins unless explicitly allowed. *Cross-Origin Resource Sharing (CORS)* is the mechanism to declare such allowances.

In practice, when the app creates a new bucket, it also applies a CORS policy with:

- **AllowedOrigins:** exactly the site origin, e.g. `https://lame-fair.k3s.virtualorfeo.it`.
- **AllowedMethods:** GET, HEAD, PUT, POST, DELETE, OPTIONS.
- **AllowedHeaders:** *, plus authorization, content-type, x-amz-.*.
- **ExposeHeaders:** ETag, x-amz-request-id, x-amz-id-2.

³Uppy is an open-source JavaScript file uploader with drag-and-drop, resumable uploads, and progress indicators: <https://uppy.io/>.

⁴The “origin” is the combination of scheme, host, and port.

- **MaxAgeSeconds:** 3600, so that the browser can cache the positive preflight result for one hour.

Browsers enforce CORS in two steps:

1. For “non-simple” requests (e.g. PUT with custom headers), the browser first sends an OPTIONS preflight.
2. If the bucket replies with headers matching the above policy, the browser proceeds with the actual upload or download.

If the policy is wrong or missing, the browser blocks the request *before* it even reaches the application. This is why bucket creation always refreshes the CORS rules. In fake-storage mode, this problem disappears: the presigned URLs are same-origin, so no CORS negotiation is required.

Retries and resiliency. The boto3 client is configured with `max_attempts=3` and `mode=standard`, which provides limited automatic retries. Functions like `create_bucket` and `ensure_bucket` handle corner cases where Ceph RGW might respond with unusual errors such as `MalformedXML` during CORS updates. Listings use paginators to scale to large numbers of objects. Large downloads use `StreamingBody`, so data is processed chunk by chunk instead of loading everything in memory. When the user downloads a whole folder, the app streams a ZIP archive on the fly with `zipstream-ng`, so the server never holds the entire archive in RAM.

Integrity checks. Whenever a new file is uploaded, the system schedules a background job to compute its **SHA-256 checksum**⁵. A checksum is a short, fixed-length string of characters that uniquely represents the content of a file. Even a one-bit change in the file will produce a completely different checksum.

The job reads the object from S3 in fixed-size chunks (so that even very large files can be handled without exhausting memory), feeds the data into the SHA-256 algorithm, and records the resulting *digest* (the fixed-length output of the hash function) together with the file’s bucket and key in the `MeasurementFileChecksum` table.

This fingerprint can then be used to verify the file later on:

- to confirm that a mirrored copy is identical to the original,
- to compare against a checksum provided by the user at upload time,
- or to detect accidental corruption during storage or transfer.

By recording these digests systematically, the application ensures long-term data integrity and provides a trustworthy way to check that scientific data has not been altered.

5.5 Background tasks

Queueing with Redis RQ. The application does not perform heavy work inside the web request–response cycle. Instead it uses **Redis Queue (RQ)**⁶ to schedule jobs in the background.

The flow is as follows:

⁵SHA-256 is part of the SHA-2 family of cryptographic hash functions, standardized by NIST; see <https://csrc.nist.gov/projects/hash-functions>.

⁶<https://python-rq.org/>, a lightweight task queue built on Redis.

1. When a user uploads files or registers a measurement, the web process enqueues one or more jobs into Redis.
2. Worker processes (separate long-running Python processes) listen on the queues, pull jobs, and execute them.

Two main queues are configured:

- **default**: used for checksums and general tasks.
- **ofed**: optional, dedicated to NeXus builds so that these long-running jobs do not block faster ones.

The key background tasks are:

- `compute_checksum(bucket, key)`: streams the uploaded file in chunks, computes a SHA-256 digest, and writes it into the `MeasurementFileChecksum` table using `update_or_create` (so repeated runs update the same row instead of failing).
- `make_nexus_from_tiff(...)`: runs the full NeXus build pipeline (mapping, scaffold population, and publishing into the mirror bucket; see §5.3).

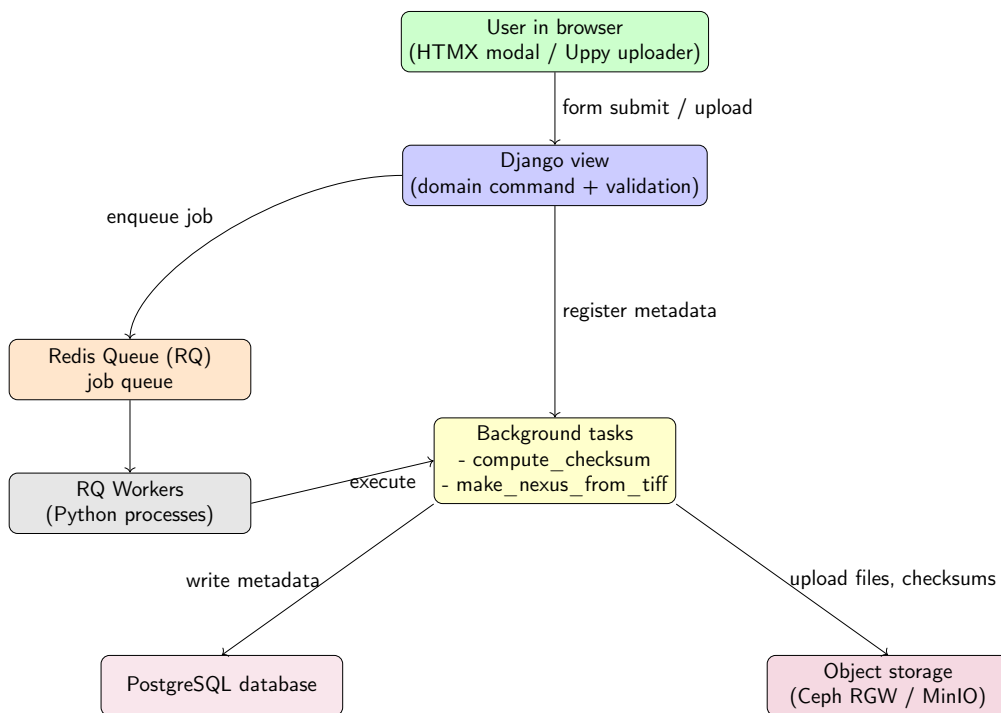


Figure 5.3: Flow of background tasks. An action in the browser (via HTMX or Uppy) calls a Django view, which validates input and enqueues jobs into Redis RQ. Worker processes execute tasks such as computing checksums or building NeXus files, writing results to the database and object storage.

Because workers are decoupled from web requests, the user interface stays responsive even if the actual job involves gigabytes of data or many minutes of processing. RQ requires that all

task functions be importable by workers; this is why they are all exposed under `experiment_manager.tasks`.

Failure modes and retries. Jobs are enqueued with an explicit retry policy (`max=3`, with backoff intervals of 10, 60, and 300 seconds). Common transient errors include:

- timeouts while reading or writing large objects to S3,
- or temporary SSL certificate / CA chain issues.

The design ensures safety under retries:

- Database writes use `update_or_create`, which means that repeating the same job simply updates the existing row.
- Storage keys are deterministic, so re-running a NeXus build will overwrite the same `.nxs` object instead of creating duplicates.
- NeXus jobs use a stable `job_id` derived from the file key and instrument pair, so that the same TIFF cannot be processed twice in parallel.

Observability. Operators and developers need visibility into what jobs are running. The application provides several tools:

- **Web UI:** `django-rq`⁷ is mounted under `/django-rq/`. It shows the state of queues, active jobs, and failed jobs.
- **Logs:** enqueueing decisions, job IDs, and completion events are logged. This makes it possible to trace a measurement from upload to checksum to NeXus conversion.
- **Presigned URLs:** when downloads are batched (e.g. with aria2 manifests⁸), the URLs include explicit expiration times. This ensures that the download window is limited and that past downloads can be reviewed if needed.

5.6 API and UI surfaces

Why an API? An **Application Programming Interface (API)** is a structured way for software to talk to software. Instead of a person clicking through the user interface, another program can send requests over the network and receive structured responses (usually JSON). This makes the system usable not only by humans in the browser but also by scripts, analysis pipelines, or external services. In practice, the project uses two complementary “surfaces”:

- a *REST API*, meant for machine-to-machine communication,
- and a *browser-based UI*, which wraps the same logic in a more user-friendly interface.

⁷<https://github.com/rq/django-rq>, the Django integration for RQ.

⁸<https://aria2.github.io/>, a multi-source download utility.

REST API (Django REST Framework). The `api/` package exposes Projects, Proposals, Samples, and Experiments via a REST⁹ interface. This is implemented using Django REST Framework (DRF)¹⁰.

The API is built from:

- **Serializers**, which translate Django model instances into JSON and back.
- **ViewSet**s, which handle the HTTP verbs (GET, POST, PATCH, DELETE) for each entity.
- **Routers**, which collect the viewsets under clear endpoints.

For example, the router registers:

- `/api/proposals/` → `ProposalViewSet`
- `/api/samples/` → `SampleViewSet`
- `/api/experiments/` → `ExperimentViewSet`

Validation rules in serializers mirror the same constraints as the domain commands, so that API clients cannot bypass the business logic enforced by the UI. Automated tests (`test_api.py`) verify not only the output shape but also permissions: unauthenticated requests get **403 Forbidden**, while authenticated users can filter samples by proposal or experiments by sample.

HTMX modals and the management board. For human operators, the primary surface is a browser-based “management board.” It is a three-pane view defined by `ManageProposalsView`:

- **Left:** list of proposals (`ProposalPane`).
- **Middle:** samples for the selected proposal (`SamplePane`).
- **Right:** experiments for the selected sample, and the form to register new measurements (`ExperimentPane`, `MeasurementPane`).

A visual walkthrough of this interface is provided in the next section.

The board is dynamic thanks to **HTMX**¹¹. When the user clicks “create proposal/sample/experiment”, an HTMX-driven modal opens. Submitting the form calls the domain command under the hood. On success, the server returns **204 No Content** and triggers an `HX-Trigger`, which closes the modal and refreshes the affected pane. If validation fails, the server responds with a normal **200 OK**, but the response body contains the form HTML again, now with error messages filled in. HTMX swaps this HTML back into the modal so the user can correct their input.

Dashboards and landing. After login, users land on dashboards tailored to their role:

- `admin_dashboard` for privileged operators (group “`lamepi`”),
- `regular_dashboard` for everyone else.

Guests who are not logged in see only a minimal landing page and are redirected to the dashboard once authenticated.

⁹<https://restfulapi.net/>, a widely adopted architectural style for web APIs.

¹⁰<https://www.django-rest-framework.org/>, the de-facto standard toolkit for APIs in Django

¹¹<https://htmx.org/>, a small JavaScript library that allows declarative AJAX interactions.

Bucket and file views. The application also provides a direct window into S3 buckets, a feature not available in the native Ceph RGW interface. `/buckets/` lists proposals with their RAW or mirrored (`/ofed/`) buckets. `/buckets/<bucket>/?prefix=...` renders a file-browser-like view by scanning keys and grouping them into “folders.”

For each file or prefix, the UI provides tools:

- **Presigned GET links** for single-file downloads.
- **ZIP streaming** for folders: the server streams a ZIP of the whole subtree (without compression, to save CPU).
- **aria2 manifests**¹² for high-throughput bulk downloads. Each manifest (`.aria2.txt`) contains per-file presigned URLs and output paths.

If a bucket is missing, the view shows an actionable error and allows an administrator to re-create it safely.

Static assets and client behaviour. The client-side code is deliberately kept small:

- custom CSS/JS is minimal,
- the Uppy¹³ library handles drag-and-drop uploads, querying `/s3_presign` to get presigned PUT URLs,
- `htmx-modals.js` manages modal lifecycle (open, close, escape key).

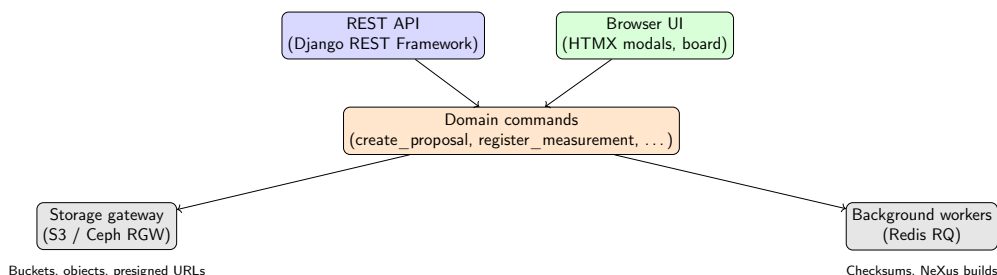


Figure 5.4: Two faces of the system: REST API for programmatic clients and HTMX UI for human operators. Both call into the same domain commands, which in turn use the storage gateway and background workers.

5.7 User interface walkthrough

This section illustrates how a user interacts with the application, step by step, from logging in to browsing stored experimental data. The screenshots show the main screens a researcher would encounter during routine use.

The workflow begins with login. Users are first greeted by a minimal login page (Figure 5.6). Authentication is then delegated to Authentik (Figure 5.7), which provides single sign-on using OpenID Connect.

¹²<https://aria2.github.io/>, a command-line download tool supporting parallel and segmented transfers.

¹³<https://uppy.io/>, a JavaScript file uploader with progress tracking and retries.

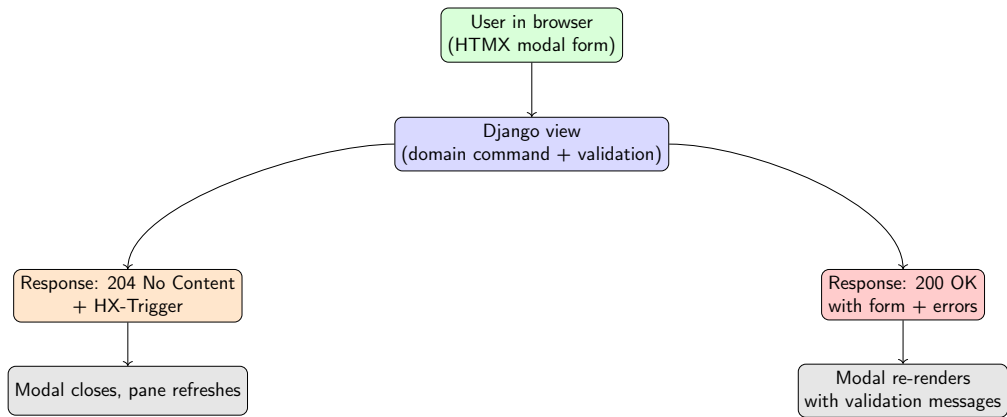


Figure 5.5: HTMX modal flow. After form submission, the server either replies with **204 No Content** (success: close modal, refresh pane) or with **200 OK** and the form HTML filled with errors (failure: modal re-renders).

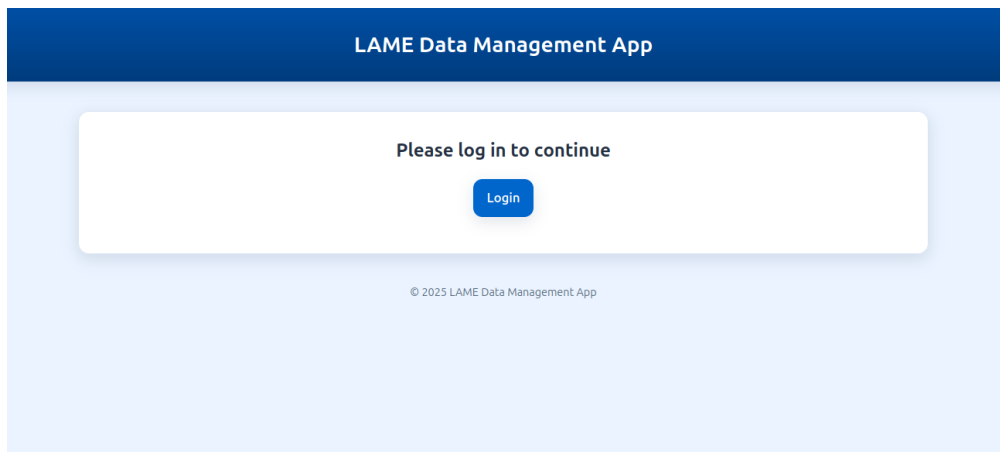


Figure 5.6: Initial login screen of the LAME Data Management App.

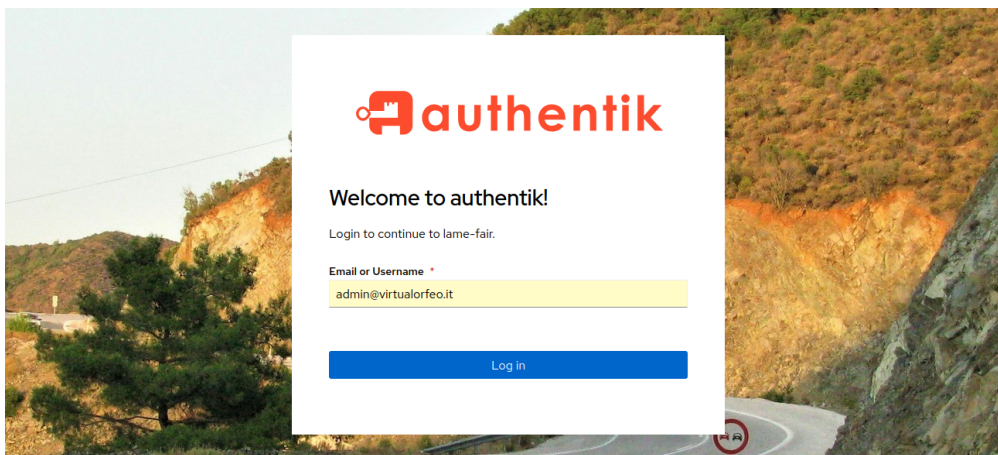


Figure 5.7: Authentication via Authentik using OpenID Connect.

After successful login, users are taken to the dashboard (Figure 5.8), which acts as an entry point to the management board and other functions. If no proposals have been created yet, the board appears empty (Figure 5.9).

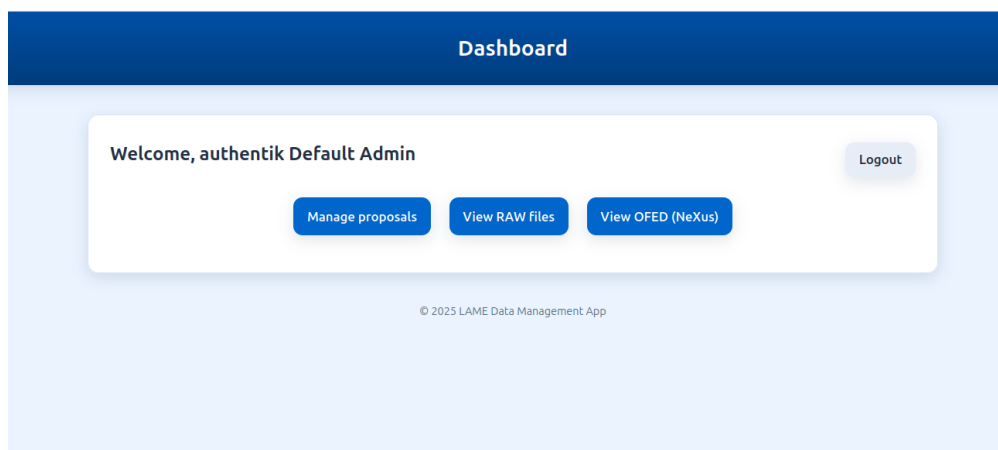


Figure 5.8: Dashboard view after successful login, showing navigation options.

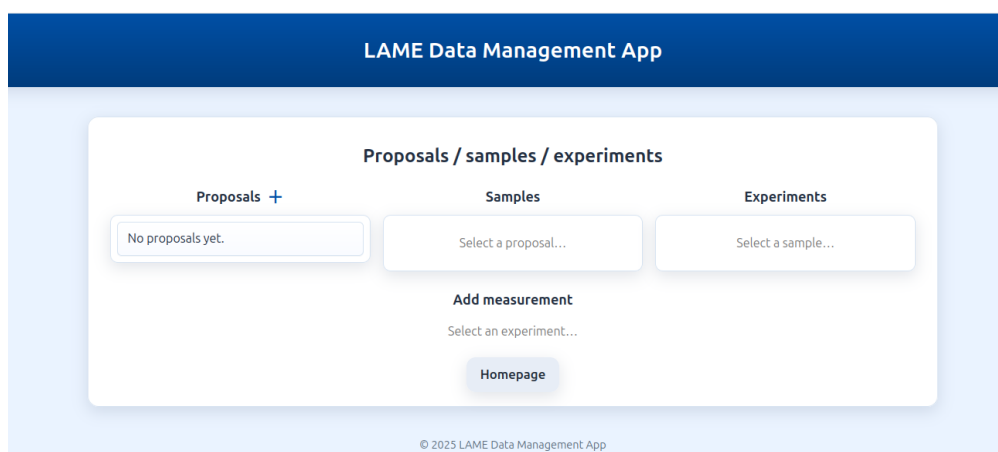
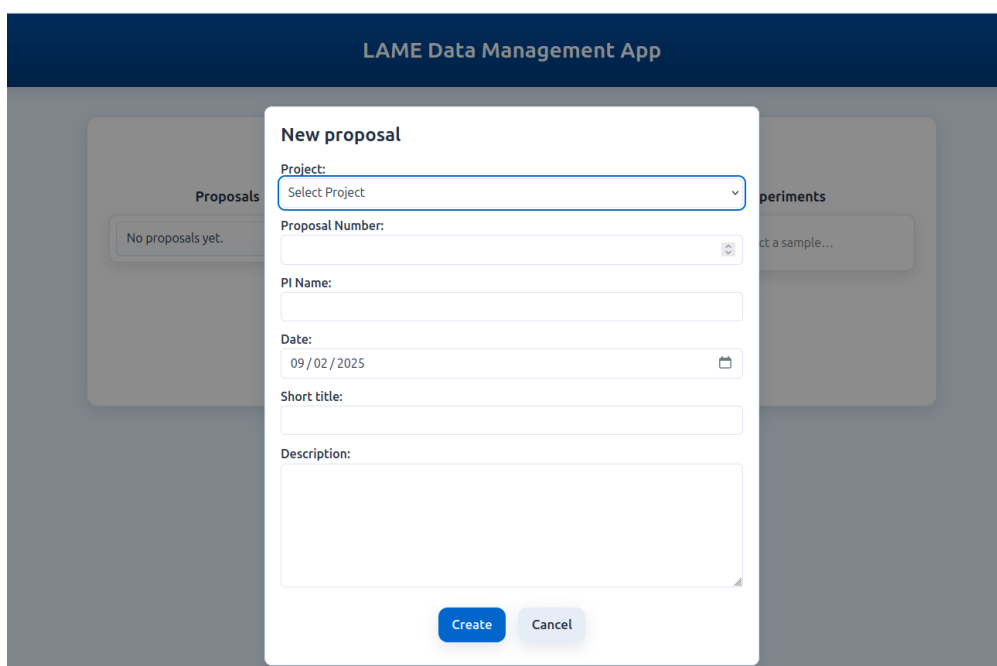


Figure 5.9: Management board before any proposals are created.

New proposals are created via modal dialogs. Figure 5.10 shows the form for entering the proposal number, principal investigator, and description. Once submitted, the new proposal appears in the left pane of the management board, and its details can be inspected as shown in Figure 5.11.

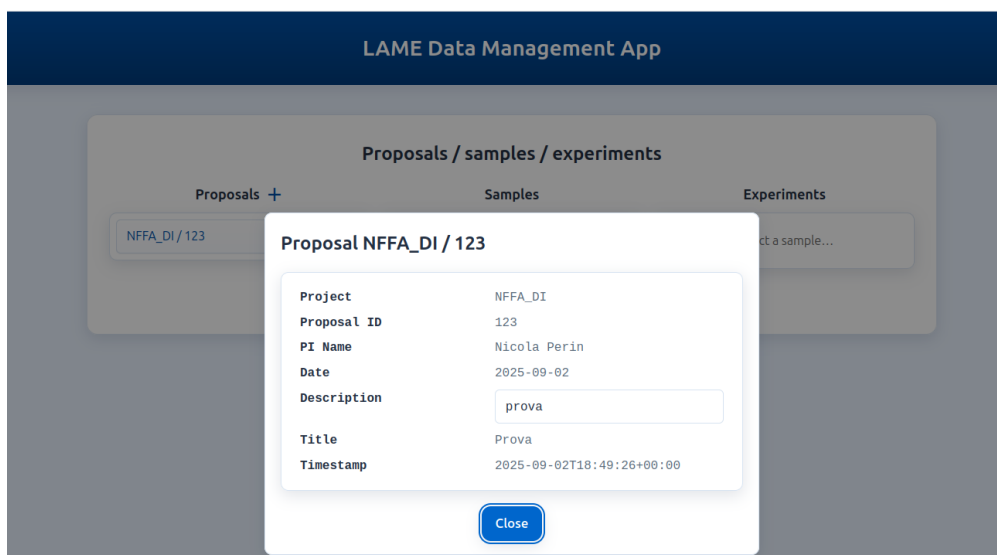


The screenshot shows the 'LAME Data Management App' interface. A modal form titled 'New proposal' is centered on the screen. The form contains the following fields:

- Project:** A dropdown menu with 'Select Project' as the placeholder.
- Proposal Number:** A text input field with a small icon on the right.
- PI Name:** A text input field.
- Date:** A date picker showing '09/02/2025'.
- Short title:** A text input field.
- Description:** A large text area.

At the bottom of the modal are two buttons: 'Create' (blue) and 'Cancel' (grey).

Figure 5.10: Modal form for creating a new proposal.



The screenshot shows the 'LAME Data Management App' interface. A modal form titled 'Proposal NFFA_DI / 123' is centered on the screen. The form displays the following details:

Project	NFFA_DI
Proposal ID	123
PI Name	Nicola Perin
Date	2025-09-02
Description	prova
Title	Prova
Timestamp	2025-09-02T18:49:26+00:00

At the bottom of the modal is a 'Close' button (blue).

Figure 5.11: Viewing details of a created proposal.

Within a proposal, users can register samples. The modal in Figure 5.12 allows adding a new sample with its metadata. Measurements can then be attached to experiments within each sample. Figure 5.13 shows the interface for creating a measurement entry, which is then linked to raw data files.

The image shows a modal window titled "New sample for NFFA_DI / 123" overlaid on a blurred background of the application interface. The modal contains the following fields and controls:

- Sample name:** A text input field containing "sample001".
- Sample identifier:** A text input field containing "smp11".
- Preparation date:** A date input field showing "09 / 02 / 2025" with a calendar icon on the right.
- Atom types (comma-separated):** A text input field containing "Fe,Ag".
- Physical form:** A dropdown menu with "powder" selected.
- Buttons:** "Create" (blue) and "Cancel" (grey) buttons at the bottom.

Figure 5.12: Modal form for creating a new sample within a proposal.

The image shows the main interface of the "LAME Data Management App" with the breadcrumb "Proposals / samples / experiments". It features three tabs: "Proposals +", "Samples for NFFA_DI / 123 +", and "Experiments for sample001 +".

- Under the "Proposals" tab, there is a card for "NFFA_DI / 123" with an "Info" button.
- Under the "Samples" tab, there is a card for "sample001" with an "Info" button.
- Under the "Experiments" tab, there is a card for "exp001 — some description" with an "Info" button.

Below the tabs, the text "Add measurement to exp001 (sample sample001)" is displayed. A "Detector" dropdown menu is set to "TEM_JEOL_F200 - TVIPS_camera". At the bottom, there are three buttons: "Choose file(s)", "Choose folder", and "Upload & register", followed by a "Homepage" button.

Figure 5.13: Adding a new measurement under a sample and experiment.

Raw data, typically TIFF images, is uploaded through a drag-and-drop interface (Figure 5.14). Files are sent directly to object storage using presigned URLs, ensuring efficient transfer without passing through the Django server.

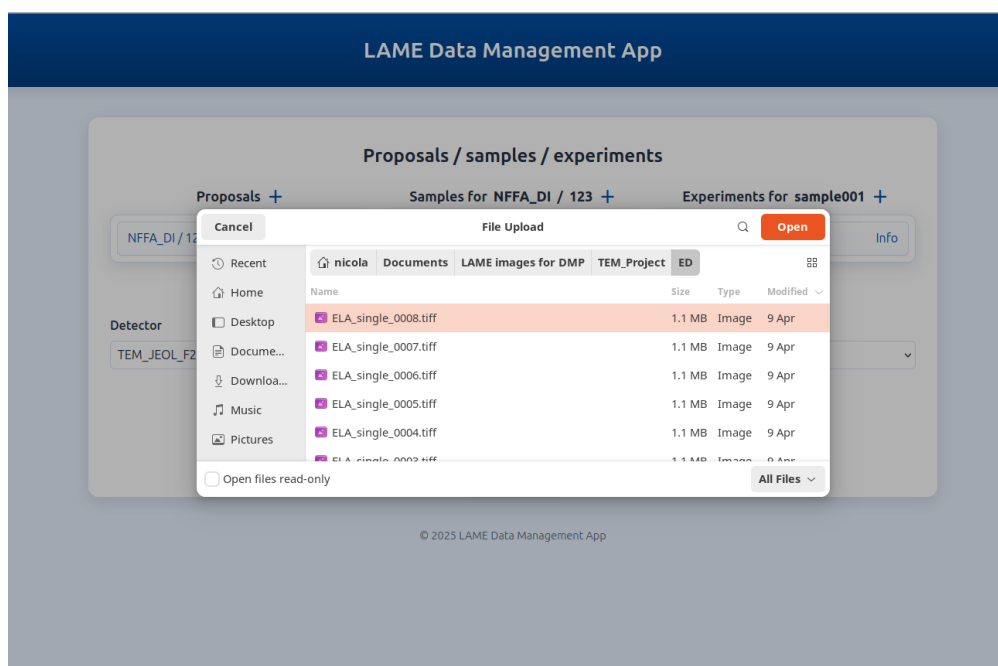


Figure 5.14: Selecting TIFF files for upload from the local filesystem.

Once data is uploaded, users can browse it in the bucket view. At the top level, proposals are listed with their associated storage buckets (Figure 5.15). Drilling down, users can see the files belonging to a specific experiment (Figure 5.16), with options to download individual files, stream a ZIP archive, or generate aria2 manifests for high-throughput transfers.

The same browsing interface is available for the “OFED” (derived data) view, which mirrors the raw data layout but contains NeXus files generated from the uploaded TIFFs. Because its appearance is identical to the raw data browser, we omit a separate screenshot here.

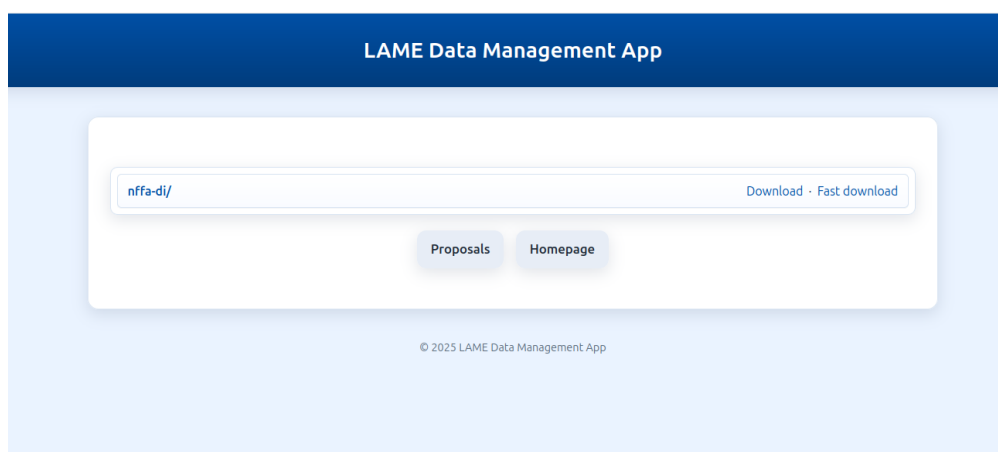


Figure 5.15: Top-level view of available proposals in the bucket browser.

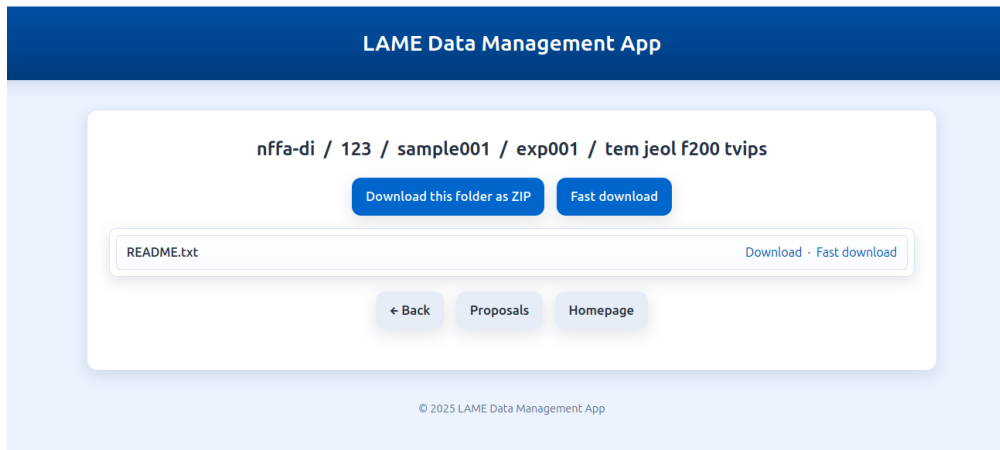


Figure 5.16: Browsing files inside a specific experiment folder, with options for ZIP or fast download.

5.8 Security model in-app

Authentication and login. The application never manages passwords itself. Instead, it delegates login to an external identity provider (Authentik) using OpenID Connect (OIDC). When a user tries to log in, they are redirected to Authentik. After successful login, Authentik returns an ID token (a signed JWT) which the app validates. From this token, a normal Django session is created. Logout also goes through OIDC: the app passes an ID token hint and a post-logout redirect so that both the identity provider and the local session are closed at the same time.

Claims and group mapping. Each token from Authentik carries *claims*¹⁴. The custom backend (LameOIDCBackend in `auth_backends.py`) reads these claims and synchronizes them into the Django user model:

- `preferred_username` → `user.username`,
- `given_name` / `family_name` → `first_name` and `last_name`,
- `groups` → Django group membership (created on demand if missing).

This way, group membership defined in Authentik or FreeIPA flows into Django and can be reused for permission checks.

Authorization rules. Access control in views builds on this group information:

- `@login_required` ensures that only authenticated users can reach sensitive endpoints.
- `@user_passes_test(is_boss)` marks administrator-only actions; `is_boss` checks whether the user belongs to the right group.

Dangerous operations such as bucket recreation, access to mirrored views, or management dashboards are strictly limited to admins. For normal data, each Proposal stores a `created_by` reference so that ownership and provenance remain visible.

¹⁴OIDC claims are fields embedded in the ID token, such as username, first/last name, or group membership.

Protecting data transfers. Uploads and downloads rely on presigned S3 URLs. Each presigned URL is valid only for:

- a single object (by key),
- a single method (GET or PUT),
- and a limited lifetime (usually one hour).

Buckets are configured with strict CORS rules so that browsers can use these URLs only from the expected frontend origin. Sensitive secrets (S3 keys, OIDC client secrets, CA bundles) are never baked into container images. They are mounted as Kubernetes Secrets and injected at runtime. TLS checks are pinned to the internal FreeIPA CA by setting `AWS_CA_BUNDLE` and letting `boto3`/requests pick it up.

Recording what happened. Every uploaded file is checksummed in the background. The resulting SHA-256 digests are stored in the `MeasurementFileChecksum` table, ensuring that each file has a permanent fingerprint. Alongside the files, plain-text README files describe who created what, when, and with which description. Logs capture all major events — such as scheduling or finishing background jobs — together with the job ID and storage location. This combination of checksums, README files, and logs makes it possible to trace back the history of data and verify its consistency over time.

5.9 Performance and scalability

Separating upload from conversion. Large files are never sent through the Django server itself. When a user uploads, the browser first asks the backend for a *presigned URL*¹⁵. The browser then sends the file straight to S3/MinIO. The only parts that go through the Django server are very small: the request for the presigned URL (`/s3_presign`) and the follow-up registration of metadata. Expensive work like computing checksums, converting TIFFs into NeXus files, and copying objects into mirror buckets runs later in background workers, not during the upload. This way, web responsiveness and heavy processing can scale independently.

Handling many things at once.

- **Uploads:** The Uppy¹⁶ uploader can open several parallel connections at once. Object storage systems like Ceph RGW or MinIO are built to handle concurrent input/output, so the bottleneck is usually the network link, not the application. Since Django never touches the file bytes, its CPU and memory stay stable even under heavy uploads.
- **Workers:** Background work is handled by Redis Queue (RQ)¹⁷. Operators can start as many worker processes as needed. Each worker can compute checksums or build NeXus files in parallel. A stable job identifier is used so that if two requests try to process the same TIFF, only one job is actually run.

¹⁵A presigned URL is a temporary web address that allows the browser to upload or download a specific object from S3 directly, without passing the data through the application server.

¹⁶<https://uppy.io/>, a JavaScript library for drag-and-drop uploads with retries and progress bars.

¹⁷RQ (<https://python-rq.org/>) is a Python library that uses Redis as a queue to manage jobs.

- **Downloads:** For reading data, browsers again receive presigned URLs and fetch files directly from S3. For folder downloads, the app streams a ZIP file on the fly. Streaming means the server never holds the whole ZIP in memory: it sends each chunk as it is produced, letting the browser apply back-pressure¹⁸.

Database performance. The database tables enforce uniqueness (for example, one proposal number per project, one sample per proposal). Django automatically creates the necessary indexes to support these constraints. Extra indexes can be added for frequent lookups, such as checksums (`MeasurementFileChecksum(bucket, key)`). To avoid unnecessary database queries, list views use `select_related`, which fetches related rows in a single query.

Where caching helps.

- **UI fragments:** Proposal and sample lists can be cached for a short time, which reduces the load when users rapidly switch panes in the HTMX interface.
- **Presigned URLs:** These should not be cached on the server side, because they expire quickly and are tied to a specific file. Clients, however, may reuse them during multi-file uploads.
- **README parsing:** When a modal needs to display a README file, it can cache the parsed lines until the file changes (detected via its ETag¹⁹).

Network and TLS security. All S3 clients are configured to use TLS certificates from a trusted CA bundle²⁰. This bundle is mounted at runtime as a Kubernetes Secret, not embedded in the image. For the application endpoints, TLS termination is handled by Ingress²¹ in combination with MetalLB²². Presigned URLs point at S3/MinIO with HTTPS, so encryption holds end-to-end.

Dealing with failures. The S3 client is configured to retry failed requests a few times before giving up. Background jobs also use retry policies with exponential backoff, and they are written to be safe to run multiple times without causing duplicates or corruption. If a bucket does not exist, the code can recreate it and reapply its CORS configuration. The `/buckets/` view shows clear error messages when something is missing, and administrators can fix it with one click.

Scaling in operations. Web processes and background workers can scale separately. The main scaling factor is storage: moving from local volumes to a CSI-backed block store²³ or to an external S3 cluster requires no code change because everything goes through the storage gateway abstraction. Redis can be replaced by a managed service if needed; RQ workers keep working the same way. The API and UI servers are stateless (except for the shared database), which means they can be horizontally scaled behind the load balancer without special coordination.

¹⁸Back-pressure is when the receiving side slows down the sending side so that buffers do not overflow.

¹⁹An ETag is an identifier provided by the storage service that changes whenever the file content changes.

²⁰A Certificate Authority (CA) bundle is a set of trusted certificates used to verify secure connections.

²¹Ingress is a Kubernetes component that manages external access to services, often with TLS termination.

²²MetalLB is a Kubernetes load-balancer implementation for bare-metal clusters.

²³CSI (Container Storage Interface) is a standard for connecting Kubernetes to different storage systems.

Conclusions

This thesis investigated how the FAIR data principles can be applied to the daily work of an electron microscopy laboratory. The problem addressed was the increasing amount and complexity of data produced by modern instruments, and the difficulty of storing, organizing, and reusing these data in a systematic way. The goal was to design and implement a data pipeline and platform that make microscopy datasets easier to manage and more useful in the long term.

The work resulted in a solid and FAIR-by-design pipeline that starts from the moment of data acquisition. Raw files are placed in an object store, enriched with metadata, and where possible converted into standardized NeXus/NXem files. The system records checksums and README files to keep track of provenance, and separates raw from processed data using dedicated buckets. All of this is integrated into a web application that allows researchers to create projects, proposals, samples, and experiments, and to upload and browse their data. Background workers handle the heavier tasks such as checksum calculation and file conversion, so that the web interface remains responsive.

To ensure that the platform could be deployed safely in production, all components were first developed and validated in the *VirtualOrfeo* environment, a digital twin of the ORFEO cluster. This made it possible to test authentication, storage, and deployment choices without risk, before moving to the real infrastructure. The layered software design — with clear separation between the domain model, storage gateway, background tasks, and user interfaces — further contributes to robustness and maintainability. Overall, the work demonstrates that a FAIR-by-design pipeline for electron microscopy can be deployed on ORFEO, the high-performance computing and data center operated by Area Science Park in Trieste, providing a reproducible and future-proof solution for managing scientific data.

There are several areas for future work. The NeXus writers can be extended to validate files more strictly, convert units into standard forms, and support multi-frame datasets. More instrument mappings should be added so that the system can be used with a wider variety of microscopes. Connections to external research infrastructures, including persistent identifiers and metadata standards, would improve the visibility and reuse of the data. Finally, scaling up the system in production will require monitoring how it behaves under higher data rates and adjusting the storage and queueing backends as needed.

Although this work focused on the electron microscopy workflows of LAME, its design is not specific to a single technique. The same architecture — combining structured uploads, object storage, NeXus-based metadata handling, and background processing — can be adapted to other scientific domains such as spectroscopy, tomography, or genomics, and reused across laboratories both inside and outside Area Science Park. By keeping the components modular and standards-based, the pipeline offers a general blueprint for FAIR data management beyond microscopy.

The software stack developed in this thesis is openly released on GitHub²⁴ under the permissive *MIT License*. This allows other researchers to freely adopt, extend, and integrate the

²⁴<https://github.com/NicolaPerin/fair-em-pipeline>

platform into their own workflows, ensuring that the results of this thesis contribute not only to local needs but also to the broader goals of open science and reproducible research. In this sense, the project applies the FAIR principles not only to scientific data but also to the tools that manage them.

In summary, this thesis shows that FAIR data management for electron microscopy is possible with a pipeline and platform that combine standardized formats, automated processing, and integration with laboratory workflows. The result is a practical approach that supports both local research needs and the broader objectives of scientific infrastructures, positioning ORFEO and Area Science Park at the forefront of open and reproducible science in Europe.

Appendix A

Tools and technologies

This appendix lists the main tools, libraries, and platform components used in the project, in alphabetical order, with brief notes on how each is used.

ACME

Protocol used by `cert-manager` to automatically obtain and renew TLS certificates from the internal FreeIPA CA.

Ansible

Automation tool used (together with Vagrant) to provision the VirtualOrfeo VMs and bootstrap services.

aria2

Command-line downloader. The app generates `.aria2.txt` manifests for high-throughput, parallel downloads from presigned S3 URLs.

Authentik

Open-source Identity Provider (IdP) providing OIDC/OAuth2. It syncs users/groups from FreeIPA and issues the tokens used by the Django app.

Boto3

Python SDK used by the storage gateway to talk to S3-compatible object stores (Ceph RGW / MinIO): create buckets, list, upload, presign URLs.

Ceph

Distributed storage platform used in VirtualOrfeo to replicate ORFEO's storage architecture.

Ceph RADOS Gateway (RGW)

The S3-compatible object storage interface of Ceph. The application stores raw and derived data here via the S3 API.

CephFS

(Where relevant) POSIX-style distributed filesystem on Ceph, used in the real ORFEO environment for shared storage tiers.

cert-manager

Kubernetes add-on that automates TLS certificates (via ACME) for Ingresses; integrated with the FreeIPA CA in VirtualOrfeo.

Django

Python web framework used for the application (MVT pattern, forms, views, templates, ORM, admin).

Django REST Framework (DRF)

Toolkit used to expose a REST API (serializers, viewsets, routers) for programmatic access to projects, proposals, samples, experiments.

django-rq

Django integration for Redis Queue, providing a simple web UI to inspect queues, running and failed jobs.

Docker

Containerization of the web app and worker processes; images are deployed to K3s via Helm.

FreeIPA

Authoritative directory for users/groups and internal Certificate Authority (CA). Authentik binds to FreeIPA over LDAP.

Git Version control for the codebase, Helm charts, and infrastructure manifests.

Gunicorn

WSGI server running the Django application; paired with an NGINX sidecar in Kubernetes.

HDF5

Binary container format underlying NeXus. The generated .nxs files store arrays and rich metadata together.

Helm

Kubernetes package manager. The app is shipped as a Helm chart (web deployment, worker, Redis, Postgres, Ingress, Secrets, Jobs).

HTMX

Small JS library enabling declarative AJAX. Powers the modal forms and dynamic pane refreshes in the management board.

JSON

Data format for API payloads and for the instrument-to-NeXus mapping files used during metadata translation.

JWT (JSON Web Token)

Signed tokens issued by Authentik (OIDC) and validated by the app for authentication and claims (e.g., groups).

K3s Lightweight, CNCF-certified Kubernetes distribution used to run the virtual cluster in VirtualOrfeo.

Kubernetes

Orchestrates containers (Deployments, Services, Ingress, Secrets, ConfigMaps, Jobs) for the application stack.

KVM/QEMU

Virtualization stack used by VirtualOrfeo to run the VMs that emulate ORFEO's services.

MetalLB

Bare-metal load balancer for Kubernetes. Assigns a routable IP to the Ingress controller inside VirtualOrfeo.

MinIO

S3-compatible object store used as a simpler alternative to Ceph RGW in tests; the app speaks to it via the same S3 API.

NeXus

Community standard (built on HDF5) for storing scientific data + metadata. The app builds .nxs outputs for interoperability.

NGINX

Reverse proxy sidecar for static files and request handling in front of Gunicorn; also used as the cluster Ingress controller.

NXem

NeXus application definition for electron microscopy. Guides the schema the writers populate from TIFF metadata.

OpenID Connect (OIDC)

Auth protocol used for SSO: the Django app is an OIDC client of Authentik.

PostgreSQL

Relational database backing the Django ORM in deployment (SQLite is used only for quick local tests).

Python

Primary programming language for the backend, workers, and tooling.

Redis

In-memory data store used as the broker/backend for Redis Queue (RQ).

Redis Queue (RQ)

Lightweight background job system. Handles checksum computation and NeXus builds without blocking web requests.

Requests

Python HTTP library used indirectly (via boto3 and integrations) for HTTPS calls honoring the internal CA bundle.

S3 API (Amazon S3-compatible)

Object storage protocol used by the app (presigned PUT/GET, buckets, keys). Implemented by Ceph RGW and MinIO.

tiff file

Python library to read TIFF headers/data during ingestion and NeXus construction.

TLS

Encryption for all web endpoints and S3 URLs; certificates issued by the internal CA via cert-manager.

Uppy

Browser file uploader (drag-and-drop, retries, progress). Uses presigned URLs to upload directly to object storage.

Vagrant

VM lifecycle manager used to spin up the VirtualOrfeo machines in a reproducible way.

YAML

Configuration format for Kubernetes manifests and Helm values.

zipstream-ng

Server-side streaming ZIP generator used to download large folder trees without buffering entire archives in memory.

Bibliography

- Poger, David, Lisa Yen and Filip Braet (2023). ‘Big data in contemporary electron microscopy: challenges and opportunities in data transfer, compute and management’. In: *Histochemistry and Cell Biology* 160.3, pp. 169–192. DOI: [10.1007/s00418-023-02191-8](https://doi.org/10.1007/s00418-023-02191-8). URL: <https://pmc.ncbi.nlm.nih.gov/articles/PMC10492738/>.
- Moore, Josh et al. (2021). ‘OME-NGFF: a next-generation file format for expanding bioimaging data-access strategies’. In: *Nature Methods* 18.12, pp. 1496–1498. DOI: [10.1038/s41592-021-01326-w](https://doi.org/10.1038/s41592-021-01326-w). URL: <https://pubmed.ncbi.nlm.nih.gov/34845388/>.
- Korir, Paul K., Gerard J. Kleywegt et al. (2024). ‘Ten recommendations for organising bioimaging data for archival’. In: *F1000Research* 13. Includes discussion on proprietary formats and file-organization practices, p. 112. DOI: [10.12688/f1000research.142422.2](https://doi.org/10.12688/f1000research.142422.2). URL: <https://pmc.ncbi.nlm.nih.gov/articles/PMC10938051/>.
- Wilkinson, Mark D. et al. (2016). ‘The FAIR Guiding Principles for scientific data management and stewardship’. In: *Scientific Data* 3, p. 160018. DOI: [10.1038/sdata.2016.18](https://doi.org/10.1038/sdata.2016.18). URL: <https://www.nature.com/articles/sdata201618>.
- GO FAIR Initiative (2016). *FAIR Principles*. <https://www.go-fair.org/fair-principles/>. Accessed 2025-08-26. URL: <https://www.go-fair.org/fair-principles/>.
- Hodson, Simon et al. (2018). *Turning FAIR into Reality: Final Report and Action Plan from the European Commission Expert Group on FAIR Data*. Publications Office of the European Union. DOI: [10.2777/1524](https://doi.org/10.2777/1524). URL: <https://op.europa.eu/en/publication-detail/-/publication/7769a148-f1f6-11e8-9982-01aa75ed71a1>.
- European Commission (2021). *Horizon Europe Programme Guide*. https://ec.europa.eu/info/funding-tenders/opportunities/docs/2021-2027/horizon/guidance/programme-guide_horizon_en.pdf. Accessed 2025-08-26. URL: https://ec.europa.eu/info/funding-tenders/opportunities/docs/2021-2027/horizon/guidance/programme-guide_horizon_en.pdf.
- Area Science Park (2025a). *LAME — Electron Microscopy Laboratory*. <https://www.areasciencepark.it/en/research-infrastructures/innovative-materials/lame-electron-microscopy-laboratory/>. Accessed 2025-08-26. URL: <https://www.areasciencepark.it/en/research-infrastructures/innovative-materials/lame-electron-microscopy-laboratory/>.
- Könnecke, Mark et al. (2015a). ‘The NeXus data format’. In: *Journal of Applied Crystallography* 48.1, pp. 301–305. DOI: [10.1107/S1600576714027575](https://doi.org/10.1107/S1600576714027575). URL: <https://journals.iucr.org/j/issues/2015/01/00/po5029/po5029.pdf>.
- NeXus International Advisory Committee (NIAC) (2024). *NXem — NeXus Contributed Application Definition for Electron Microscopy*. https://manual.nexusformat.org/classes/contributed_definitions/NXem.html. Version 2024.02; Accessed 2025-08-26. URL: https://manual.nexusformat.org/classes/contributed_definitions/NXem.html.
- FAIRmat (NFDI) (2024). *NXem — NeXus Contributed Application Definition (FAIRmat mirror)*. https://fairmat-nfdi.github.io/nexus_definitions/classes/contributed_

- definitions/NXem.html. Accessed 2025-08-26. URL: https://fairmat-nfdi.github.io/nexus_definitions/classes/contributed_definitions/NXem.html.
- NFFA-DI (Mar. 2024). *New Master in Data Management and Curation Kicks Off*. Accessed: 2025-03-31. URL: <https://nffa-di.it/en/news/new-master-in-data-management-and-curation-kicks-off/>.
- Bazzocchi, Federica (Sept. 2024a). *Data Management in NFFA-DI : OFED (Overarching Fair Ecosystem for Data)*. Version v.1.0.0. DOI: [10.5281/zenodo.13773924](https://doi.org/10.5281/zenodo.13773924). URL: <https://doi.org/10.5281/zenodo.13773924>.
- FAIRmat Consortium (2024). *FAIRmat – FAIR Data Infrastructure for Condensed-Matter Physics and the Chemical Physics of Solids*. URL: <https://www.fairmat-nfdi.eu/fairmat/> (visited on 31/03/2025).
- The HDF Group (July 2021). *FAIRmat and HDF5: Experiments – Calculations – Integrate the Experimental World*. Presentation slides. URL: https://www.hdfgroup.org/wp-content/uploads/2021/07/20210707_FAIRmat_hdf5.pdf (visited on 31/03/2025).
- Metilli, Lorenzo et al. (2020). ‘Latest advances in imaging techniques for characterizing soft, multiphasic food materials’. In: *Advances in Colloid and Interface Science* 279, p. 102154. ISSN: 0001-8686. DOI: <https://doi.org/10.1016/j.cis.2020.102154>. URL: <https://www.sciencedirect.com/science/article/pii/S0001868619304567>.
- NeXus International Advisory Committee (2024a). *NeXus File Format Specification Manual*. URL: <https://manual.nexusformat.org/fileformat.html> (visited on 31/03/2025).
- NXcanSAS Working Group (2012). *Introduction to the NeXus Data Format*. URL: https://www.cansas.org/NXcanSAS/_downloads/nexus.pdf (visited on 31/03/2025).
- FAIRmat-NFDI (2025). *Validate a Nexus File with nxvalidate*. URL: <https://fairmat-nfdi.github.io/pynxtools/how-tos/validate-nexus-file.html> (visited on 26/04/2025).
- Bazzocchi, Federica (2024b). *NeXus and Electron Microscopy*. Presentation, NFFA-DI. URL: https://nffa-di.it/media/n2nn0qo1/8_nexus-and-electron-microscopy-bazzocchi.pdf (visited on 31/03/2025).
- NeXus International Advisory Committee (2024b). *Electron Microscopy Structure*. Accessed: 31 March 2025. URL: https://manual.nexusformat.org/classes/contributed_definitions/em-structure.html (visited on 31/03/2025).
- FAIRmat-NFDI Development Team (2025). *pynxtools: Nexus Validation*. <https://fairmat-nfdi.github.io/pynxtools/learn/nexus-validation.html>. Accessed: 2025-06-20.
- ORFEO Team, Area Science Park (2023a). *Welcome to ORFEO documentation!* <https://orfeo-doc.areasciencepark.it/>. Accessed 2025-08-27.
- Area Science Park (2025b). *LADE — Laboratory of Data Engineering*. <https://www.areasciencepark.it/en/research-infrastructure/data-engineering-lade/>. Accessed 2025-08-26.
- ORFEO Team, Area Science Park (2023b). *Computational resources — ORFEO documentation*. <https://orfeo-doc.areasciencepark.it/HPC/computational-resources/>. Internal and external networking overview; Accessed 2025-08-27.
- (2023c). *Storage resources — ORFEO documentation*. <https://orfeo-doc.areasciencepark.it/HPC/storage-resources/>. Details on Ceph cluster size, fast/home/scratch areas, and LTS; Accessed 2025-08-27.
- (2024). *Changelog 2024 — ORFEO documentation*. <https://orfeo-doc.areasciencepark.it/changelog/2024/>. Entry dated 2024-12-20; Accessed 2025-08-27.
- Project, Ceph (2025a). *Ceph Storage Cluster (RADOS) — Official Documentation*. <https://docs.ceph.com/en/reef/rados/>. Cluster roles and architecture; Accessed 2025-08-27.
- (2025b). *Pools — Data protection strategies in Ceph (replicated vs erasure)*. <https://docs.ceph.com/en/reef/rados/operations/pools>. Pool types and durability/performance trade-offs; Accessed 2025-08-27.

- Project, Ceph (2025c). *Erasure code — Concepts and Operations*. <https://docs.ceph.com/en/reef/rados/operations/erasure-code/>. Explains k+m chunks and trade-offs vs replication; Accessed 2025-08-27.
- (2025d). *Ceph File System (CephFS) — Official Documentation*. <https://docs.ceph.com/en/reef/cephfs/>. POSIX-like shared filesystem on top of RADOS; Accessed 2025-08-27.
- Hat, Red (2025). *Introduction to the Ceph File System*. https://docs.redhat.com/en/documentation/red_hat_ceph_storage/4/html/file_system_guide/introduction-to-the-ceph-file-system. Background and semantics of CephFS; Accessed 2025-08-27.
- Project, Ceph (2025e). *Ceph Object Gateway (radosgw)*. <https://docs.ceph.com/en/reef/radosgw>. Overview of S3/Swift compatibility and user management; Accessed 2025-08-27.
- (2025f). *Ceph Object Gateway S3 API*. <https://docs.ceph.com/en/latest/radosgw/s3/>. Operations and authentication for S3-compatible access; Accessed 2025-08-27.
- goauthentik (2025a). *LDAP Sources (Directory Sync)*. <https://goauthentik.io/docs/core/sources/ldap/>. Accessed 2025-08-28.
- FreeIPA Project (2025a). *FreeIPA — Identity Management for Linux*. <https://www.freeipa.org/>. Accessed 2025-08-28.
- goauthentik (2025b). *OAuth2 / OpenID Connect Provider*. <https://goauthentik.io/docs/providers/oauth2/>. Accessed 2025-08-28.
- Sakimura, Nat et al. (2014). *OpenID Connect Core 1.0*. Tech. rep. Final specification (incorporating errata set 2). OpenID Foundation. URL: https://openid.net/specs/openid-connect-core-1_0.html (visited on 29/08/2025).
- goauthentik (2025c). *Scopes, Claims, and Property Mappings*. <https://goauthentik.io/docs/core/property-mappings/>. Accessed 2025-08-28.
- FreeIPA Project (2025b). *FreeIPA Documentation*. <https://freeipa.readthedocs.io/en/latest/>. Accessed 2025-08-28.
- Django Software Foundation (2025a). *Django at a glance*. <https://docs.djangoproject.com/en/stable/intro/overview/>. Accessed 2025-08-28.
- (2025b). *FAQ: General — Why does Django call the MVC pattern “Model-View-Template”?* <https://docs.djangoproject.com/en/stable/faq/general/#why-does-django-call-the-mvc-pattern-model-view-template>. Accessed 2025-08-28. URL: <https://docs.djangoproject.com/en/stable/faq/general/%5C#why-does-django-call-the-mvc-pattern-model-view-template>.
- (2025c). *Databases — Supported database backends*. <https://docs.djangoproject.com/en/stable/ref/databases/>. Accessed 2025-08-28.
- (2025d). *Views — The view layer*. <https://docs.djangoproject.com/en/stable/topics/http/views/>. Accessed 2025-08-28.
- (2025e). *Templates — The Django template language*. <https://docs.djangoproject.com/en/stable/topics/templates/>. Accessed 2025-08-28.
- Collins, Jo (2021). *How Django MVT architecture works*. Accessed: 2025-09-06. URL: <https://www.freecodecamp.org/news/how-django-mvt-architecture-works/> (visited on 06/09/2025).
- Django Software Foundation (2025f). *How Django processes a request*. <https://docs.djangoproject.com/en/stable/topics/http/shortcuts/#how-django-processes-a-request>. Accessed 2025-08-28. URL: <https://docs.djangoproject.com/en/stable/topics/http/shortcuts/%5C#how-django-processes-a-request>.
- (2025g). *URL dispatcher*. <https://docs.djangoproject.com/en/stable/topics/http/urls/>. Accessed 2025-08-28.
- (2025h). *Design philosophies — DRY (Don’t Repeat Yourself)*. <https://docs.djangoproject.com/en/stable/misc/design-philosophies/>. Accessed 2025-08-28.

- Elmasri, Ramez and Shamkant B. Navathe (2015). *Fundamentals of Database Systems*. 7th ed. Boston, MA: Pearson.
- The PostgreSQL Global Development Group (2025a). *About PostgreSQL*. <https://www.postgresql.org/about/>. Accessed 2025-08-28.
- SQLite Consortium (2025). *Appropriate Uses For SQLite*. <https://www.sqlite.org/whentouse.html>. Accessed 2025-08-28.
- Django Software Foundation (2025i). *Migrations — Schema and data migrations*. <https://docs.djangoproject.com/en/stable/topics/migrations/>. Accessed 2025-08-28.
- The PostgreSQL Global Development Group (2025b). *JSON Types and Functions*. <https://www.postgresql.org/docs/current/functions-json.html>. Accessed 2025-08-28.
- (2025c). *Full Text Search*. <https://www.postgresql.org/docs/current/textsearch-intro.html>. Accessed 2025-08-28.
- Django Software Foundation (2025j). *PostgreSQL specific model fields*. <https://docs.djangoproject.com/en/stable/ref/contrib/postgres/fields/>. Accessed 2025-08-28.
- (2025k). *Making queries — Django ORM*. <https://docs.djangoproject.com/en/stable/topics/db/queries/>. Accessed 2025-08-28.
- (2025l). *Security in Django*. <https://docs.djangoproject.com/en/stable/topics/security/>. Accessed 2025-08-28.
- (2025m). *The Django admin site*. <https://docs.djangoproject.com/en/stable/ref/contrib/admin/>. Accessed 2025-08-28.
- (2025n). *Performing raw SQL queries*. <https://docs.djangoproject.com/en/stable/topics/db/sql/>. Accessed 2025-08-28.
- Kammeyer, Alexander et al. (2023). ‘Towards an HPC cluster digital twin and scheduling framework for improved energy efficiency’. In: *Proceedings of the 18th Conference on Computer Science and Intelligence Systems (FedCSIS)*. Vol. 35. Annals of Computer Science and Information Systems. Design specification of a digital twin for the PTB HPC cluster, pp. 265–268. DOI: [10.15439/2023F3797](https://doi.org/10.15439/2023F3797). URL: <https://annals-csis.org/proceedings/2023/drpf/3797.pdf>.
- Rancher Labs and CNCF (2025). *K3s — Lightweight Kubernetes*. Project documentation; certified Kubernetes distribution packaged as a single binary. URL: <https://docs.k3s.io/> (visited on 29/08/2025).
- Weil, Sage A. et al. (2006). ‘Ceph: A Scalable, High-Performance Distributed File System’. In: *OSDI’06: 7th USENIX Symposium on Operating Systems Design and Implementation*. Foundational paper describing the Ceph architecture. Seattle, WA, USA: USENIX Association. URL: <https://ceph.io/assets/pdfs/weil-ceph-osdi06.pdf>.
- Könnecke, Mark et al. (2015b). ‘The NeXus data format’. In: *Journal of Applied Crystallography* 48.1. NeXus built on HDF5; format for neutron, X-ray and muon experiments, pp. 301–305. DOI: [10.1107/S1600576714027575](https://doi.org/10.1107/S1600576714027575). URL: <https://journals.iucr.org/j/issues/2015/01/00/po5029/po5029.pdf>.
- Wiggins, Adam (2011). *The Twelve-Factor App*. Principle III: Store config in the environment. Accessed: 2025-09-01. URL: <https://12factor.net>.

Acknowledgements:

I would like to thank my supervisor, Dr. Federica Bazzocchi, for her guidance and support throughout this work. Her input was valuable during both the planning and development phases of the project.

A sincere thank you goes to Isac Pasianotto and Ruggero Lot, who originally built the *VirtualOrfeo* environment. Their work provided the foundation for this thesis, making it possible to test and validate the application safely. In particular, I thank Isac for his assistance with Authentik integration and testing, and Ruggero for his support with system configuration and deployment.

I am also grateful to Dr. Marco Prenassi for helping me take the first steps in Django, which proved fundamental in the early development of the application.

I would like to extend my thanks to the entire LADE team for their collaboration and technical insights, with special mention to Gianfranco Gallizia for his constant availability and advice. My gratitude also goes to the LAME team for their openness and support, in particular to Paolo Ronchese, whose expertise in microscopy was essential to ground this project in real laboratory practice.

Last but not least, I warmly thank my parents and my sister Francesca, for their continuous encouragement and support, without which this work would not have been possible.

Sincerely, Nicola