SCIENCE PLATFORM FOR MACHINE LEARNING OF BIG ASTRONOMICAL DATA - CLOUD INFRASTRUCTURE

Olexandr Burakov
Bachelor's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Software Engineering
Study program: Informatics
Specialisation: Web Engineering 2021
Supervisor: RNDr. Petr Škoda, CSc.
May 16, 2025

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Science platform for machine learning of big astronomical data - cloud infrastructure
Student:	Olexandr Burakov
Supervisor:	RNDr. Petr Škoda, CSc.
Study program:	Informatics
Branch / specialization:	Web Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2025/2026

Instructions

Science Platform is an emerging cloud-based technology for processing and exploratory analysis of big data sets in astronomy and earth sciences. The goal of the thesis is the design and implementation of an extensible, easily deployable engine and a custom API , which allows to manage simple workflows for conducting machine learning experiments supported by web-based interactive visualization and labelling of millions of astronomical spectra . This thesis, focused on development of a general cloud infrastructure, is complemented by thesis of Alisher Laiyk focused on implementation of individual preprocessing, visualization and machine learning modules communicating through the above mentioned API.

The key tasks are:
1) Analyse the key Big Data technology of current science platforms as Pangeo, SciServer, ESA Datalabs or Astro Data Lab as well as the basic functionality of the already obsolete VO-CLOUD system.
2) Make survey of relevant solutions for key components of system as data format (FITS, Parquet, HDF5), data containers (e.g. Pandas, Dask DataFrame) , parallel computing engine (Spark, Dask), interactive web interface for visualization of Big data in cloud (e.g. Flask with Websockets, FastAPI, JupyterLab), and database system (e.g. PostgreSQL, MariaDB, Elasticsearch, Redis) and select the optimal one according to the user requirements.
3) Define an API allowing to run the stand-alone modules according to the simple

---

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 9 November 2024 in Prague.

workflow description in a configuration file.
4) Design the prototype using open source SW libraries
5) Implement the platform and make it easily deployable (consider Docker, Podman or even Kubernetes)
6) Integrate it with the modules developed in the thesis of Alisher Laiyk and demonstrate the working system on the small dataset.
6) Discuss the performance and user experience of your solution and suggest future improvements and extensions towards much larger and different type of astronomical data sets (e.g. light curves, images).

Recommended literature and suggested tools and libraries will be provided by supervisor.

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Burakov Olexandr. *Science platform for machine learning of big astronomical data - cloud infrastructure.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

I want to express my gratitude to everyone who has supported me on this journey.
First and foremost, I want to thank my family for their love and wise counsel, which have guided me from the earliest days of my studies through the final stages of this thesis project.
I am also grateful to my partner and friends for understanding my ambitious ideas and believing that all my efforts and decisions were not in vain.
Thanks to my thesis advisor, Professor Petr Škoda, for his initiative and advice throughout the development process.
I am grateful to each of you for being with me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

In Prague on May 16, 2025

Abstract

VO-CLOUD is a cloud-based web platform with over a decade of history, designed to work with astronomical spectral data. Its purpose is to provide researchers with access to extensive collections of spectra and specialized machine-learning tools for detecting sophisticated cosmic anomalies that may form the basis for discoveries. The platform supports manipulating data from astronomical archives, distributed computation, and detailed visualization of results in the browser.

VO-CLOUD remains a relatively unique open-source solution that has adhered to the Virtual Observatory paradigm—employing the then-current UWS protocol to decouple client-side interactions from backend computation in a distributed environment.

By mid-2024, however, the legacy infrastructure had lost the ability to execute key experimental workflows or integrate new modules. This thesis, therefore, aims to evolve VO-CLOUD by adopting modern technologies, up-to-date standards, and established development practices.

As part of this work, core components have been completely redesigned and implemented: new RESTful API and job-management mechanisms for machine learning (including active learning and spectrum preprocessing) have been developed; experiment-session persistence and management workflows have been established; seamless access to a multi-terabyte file storage has been provided; and full containerization has been implemented to allow easy addition of new components. The redesigned architecture, with a clear separation of concerns across distinct responsibility zones, creates a robust "skeleton" for integrating new analytical modules and supporting future extensions. The new system has been named ML Job Manager, underscoring its focus on orchestrating and managing computational jobs.

Keywords FastAPI, Celery, Docker, ML Job Manager, astroinformatics, RESTful API, machine learning workflows, astronomical big data, distributed computing

Abstrakt

VO-CLOUD je cloudová webová platforma s více než desetiletou historií, určená pro práci s astronomickými spektrálními daty. Jejím cílem je poskytnout vědcům přístup k rozsáhlým souborům spekter a specializované nástroje strojového učení pro detekci sofistikovaných kosmických anomálií, která mohou položit základy nových objevů. Platforma podporuje manipulaci s přidělenými daty z astronomických archivů, distribuované výpočty a podrobnou vizualizaci výsledků přímo v prohlížeči.

Významným rysem VO-CLOUD je, že jde o relativně ojedinělé open-source řešení, které vycházelo z koncepcí Virtuální observatoře – využívajících tehdy aktuální protokol UWS ke zpřehlednění a oddělení interakcí na straně klienta od výpočetních úloh na serveru v distribuovaném prostředí.

Avšak ke polovině roku 2024 původní infrastruktura ztratila schopnost spouštět hlavní experimentální scénáře a integrovat nové moduly. Tato práce si klade za cíl vyvinout VO-CLOUD s využitím moderních technologií, standardů a osvědčených vývojových postupů.

V rámci této práce byly zcela přepracovány a realizovány hlavní komponenty: navrženo a realizováno nové RESTful API a mechanismy pro řízení úloh strojového učení (včetně aktivního učení a předzpracování spekter); zavedeny postupy pro trvalé ukládání a řízení experimentálních relací; zajištěn plynulý přístup k souborovému úložišti o velikosti řádově terabajtů; a kompletní kontejnerizace umožňující snadné přidávání dalších komponent. Přepracovaná architektura s jasným rozdělením odpovědností mezi jednotlivými zónami vytváří robustní „kostru“ pro integraci nových analytických modulů a podporu budoucích rozšíření. Obnovený systém byl pojmenován ML Job Manager, čímž je zdůrazněn jeho zaměr na orchestraci a správu výpočetních úloh.

Klíčová slova FastAPI, Celery, Docker, ML Job Manager, astroinformatika, RESTful API, postupy strojového učení, astronomická velká data, distribuované výpočty

Contents

List of Figures

List of Tables

List of Code listings

List of abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CDI	Contexts and Dependency Injection
CI/CD	Continuous Integration and Continuous Delivery
CLI	Command-Line Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSV	Comma-Separated Values
DevOps	Development Operations
FIFO	First In, First Out
FITS	Flexible Image Transport System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDF	Hierarchical Data Format
HPC	High-Performance Computing
HTTP	HyperText Transfer Protocol
I/O	Input/Output
IVOA	International Virtual Observatory Alliance
JSON	JavaScript Object Notation
LAMOST	Large Sky Area Multi-Object Fibre Spectroscopic Telescope
ML	Machine Learning
MLOps	Machine Learning Operations
MVC	Model–View–Controller
NFS	Network File System
ORM	Object-Relational Mapping
OS	Operating System
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SPA	Single-Page Application
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier
UWS	Universal Worker Service
VLAN	Virtual Local Area Network
VM	Virtual Machine
VO	Virtual Observatory
VRAM	Video Random Access Memory
YAML	Ain't Markup Language

Chapter 1

Introduction

The swift advancement of machine learning techniques has transformed scientific research by automating tasks that once required long hours of manual work. As these algorithms detect subtle patterns and make accurate predictions on massive datasets, they accelerate discoveries across disciplines. Open access to terabyte-scale spectral archives in astronomy has opened new insights into stellar composition, galaxy evolution, and other cosmic phenomena.

However, leveraging these vast archives requires a flexible and scalable infrastructure. Machine learning scripts and pipelines break down when experiments must be rerun, tuned, or expanded with new models. VO-CLOUD, a cloud platform built initially by CTU students for Professor Petr Škoda under the Virtual Observatory paradigm, had, by mid-2024, become technically obsolete and unsupported, unable to orchestrate its machine learning tasks or integrate new analysis modules.

This thesis introduces ML Job Manager, a complete reengineering of the core of VO-CLOUD[1]. The container-first, microservices-style system cleanly separates the server from the client: job submissions enter a queue through a RESTful API server, and dedicated worker services pull and execute jobs. It handles terabyte-scale LAMOST spectral datasets by converting raw spectra into uniform wavelength grids and driving active-learning loops where an oracle user iteratively labels the most uncertain spectra to refine a deep convolutional network. This architecture provides a solid foundation for current research and future developments by distinguishing API routing, task orchestration, and data management.

The thesis is structured as follows. Chapter 2 reviews VO-CLOUD's legacy and shortcomings; Chapter 3 establishes the new system's requirements; Chapter 4 surveys the chosen technologies; Chapter 5 details the ML Job Manager design and implementation; Chapter 6 discusses future enhancements; and Chapter 7 concludes with a summary of contributions.

[1] `https://vocloud-dev.asu.cas.cz`

Chapter 2

Analysis of the legacy solution

Before defining the requirements for the new system, it is essential to examine the current state and limitations of the existing VO-CLOUD platform. This chapter comprehensively analyzes its **architecture**, **core modules**, and **operational workflows**.

The analysis is grounded in 3 seminal theses that document the evolution of VO-CLOUD: Jakub Koza's Bachelor's thesis, his Master's thesis, and Tomáš Mazel's Bachelor's thesis. Furthermore, the official VO-CLOUD GitHub repository[1], which contains the source code and components of the platform, was reviewed. [1, 2, 3]

The architectural choices and technical limitations uncovered in this review will serve as the foundation for defining the functional and non-functional requirements of the new solution in Chapter 3.

2.1 Key architectural concepts

This section reviews three foundational concepts: **Virtual Observatory** standards, the **Universal Worker Service**, and the **Master-Worker** pattern that underpin the design and operation of VO-CLOUD.

2.1.1 Virtual Observatory

The **Virtual Observatory** (VO) is a federated distributed concept defined by the International Virtual Observatory Alliance[2] (IVOA) that integrates heterogeneous astronomical datasets, computational services, and analytical tools. Its components adhere to common protocols and metadata standards, allowing researchers to access multiple archives and services through a single interface without a concern for the underlying data locations or formats. [4, 5]

[1] `https://github.com/vodev`
[2] `https://ivoa.net`

Layered, distributed architecture. VO is based on a three-layer model (see Figure 2.1):

- *Resource layer*: distributed archives of spectra, images, and compute nodes.
- *Service layer*: standardized RESTful interfaces for discovery, retrieval, and server-side processing, including the Simple Spectral Access Protocol (SSAP) and the DataLink extension.
- *Client layer*: user-facing applications that integrate VO services and hide underlying infrastructure details.

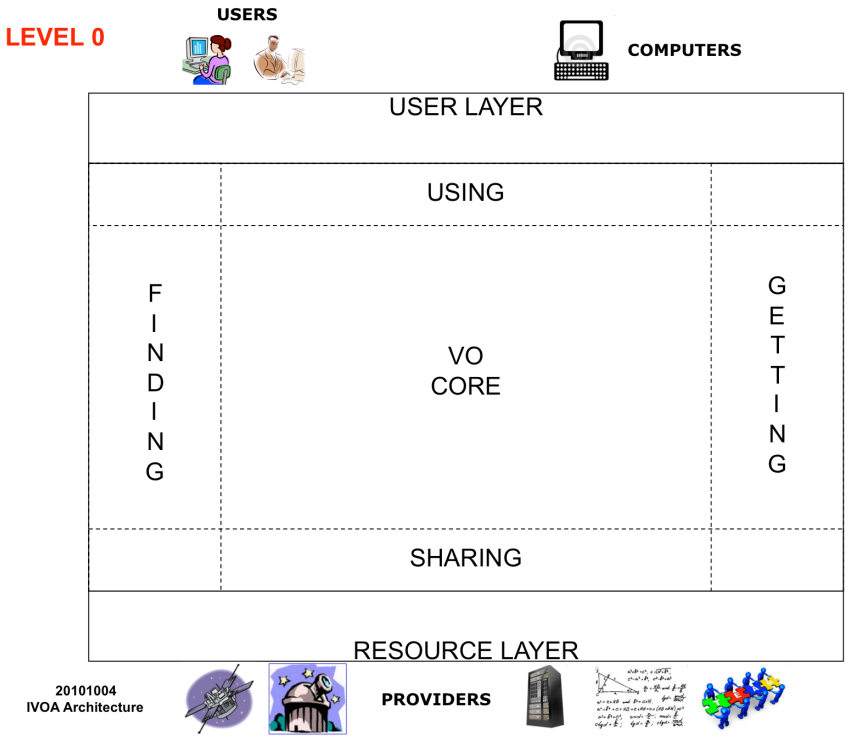

Figure 2.1 VO layered architecture. [6]

Spectral access protocols. VO defines standardized protocols for querying and retrieving spectral data (see Figure 2.2):

- *Simple Spectral Access Protocol (SSAP):* a RESTful interface that lets clients specify parameters such as wavelength range, resolution, and object coordinates to discover matching spectra. [7]
- *DataLink:* a companion service that provides links for on-the-fly operations, such as subsetting, rebinning, combining spectra, and chaining requests across multiple archives, while preserving a consistent response format. [8]

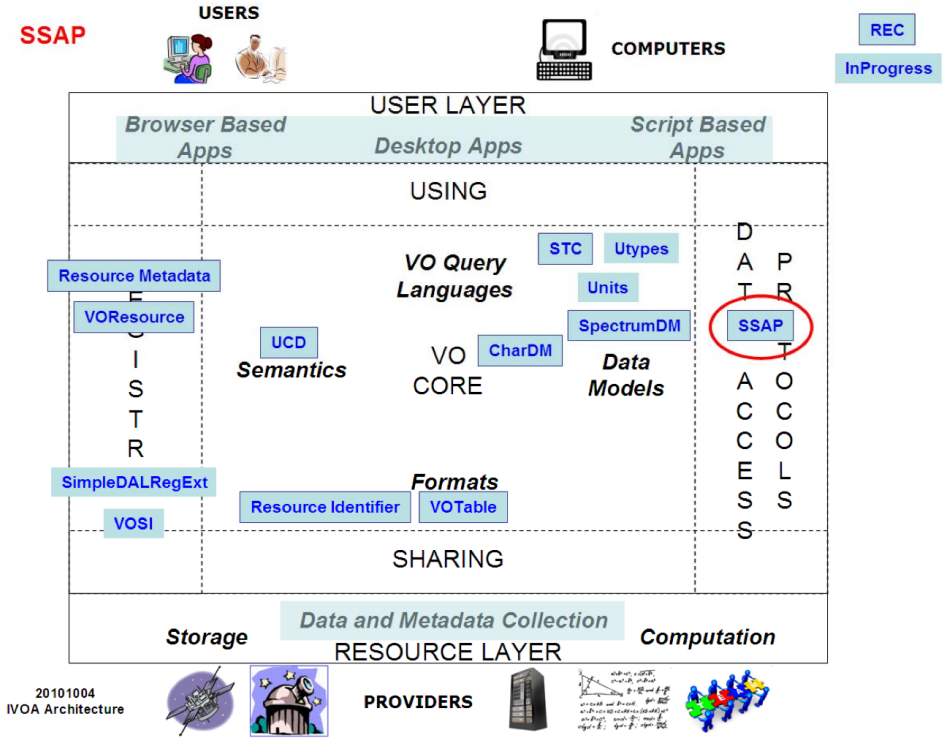

Figure 2.2 SSAP placement in the VO architecture. [9]

Data exchange formats. VO relies on two complementary formats for packaging and transmitting spectral data (see Figure 2.3):

- *Flexible Image Transport System (FITS):* a versatile binary format that encapsulates spectra in one or more Header/Data Units. Each header contains standardized metadata, such as wavelength calibration, instrument settings, and observation details, while the data units store the actual spectral values for efficient download and local processing. [10]
- *VOTable:* an XML-based table format following the IVOA Spectrum Data Model. It encodes both the spectral values and their metadata in a self-describing structure, allowing clients to parse query results directly and integrate them into visualization or analysis tools. [11]

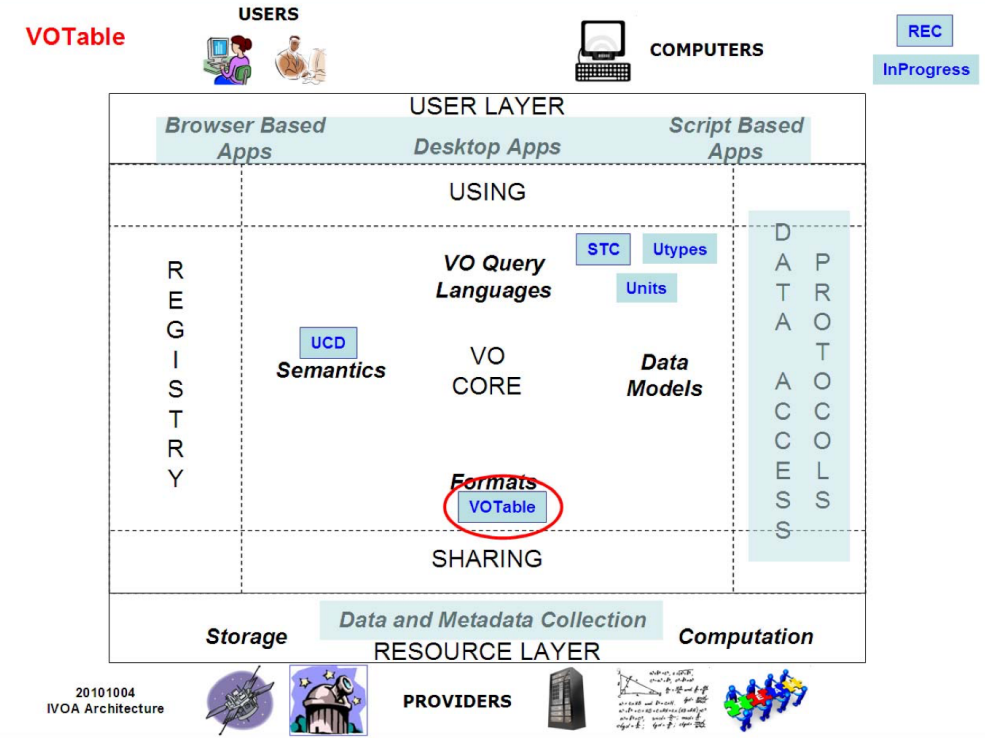

Figure 2.3 VOTable placement in the VO architecture. [12]

VO-CLOUD adheres closely to VO principles in its layered architecture and its use of SSAP, DataLink, FITS, and VOTable while selectively extending metadata and interfaces to support custom workflow and ML-driven analysis.

2.1.2 Universal Worker Service

The **Universal Worker Service** (UWS) is an IVOA recommendation for asynchronous job management within the VO framework (see Figure 2.4). By decoupling job submission from execution, UWS enables clients to submit long-running tasks, monitor their progress, and retrieve results without maintaining a continuous connection. [13]

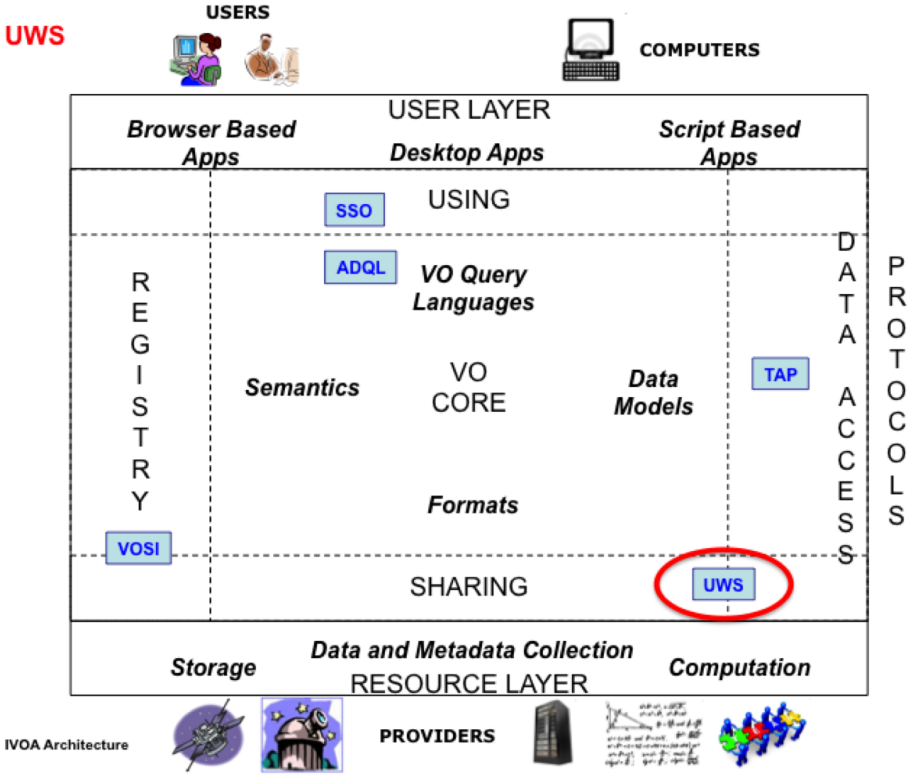

Figure 2.4 UWS placement in the VO architecture. [14]

Standardized job representation. A Job resource encapsulates parameters (inputs), ownership metadata, and a Result List pointing to outputs (files, tables, or data streams), all described in XML (see Figure 2.5).

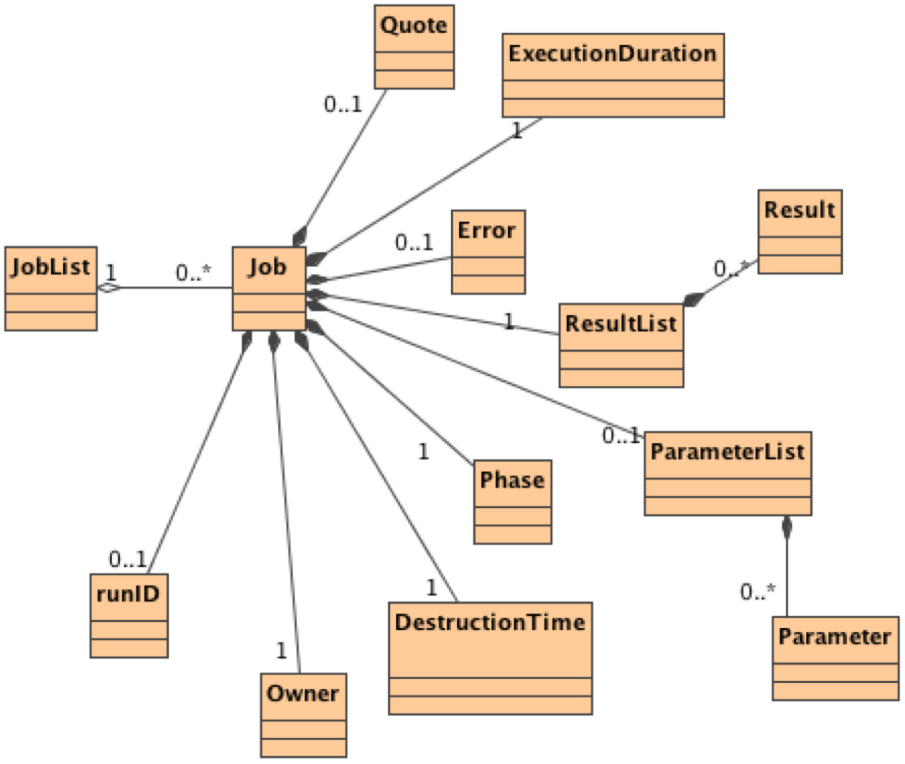

Figure 2.5 UWS Job domain. [15]

Asynchronous job handling. Clients create jobs using a RESTful endpoint and receive an identifier immediately, allowing the server to queue and execute tasks independently of the client's session.

Job lifecycle states. Each job transitions through a well-defined finite-state machine: `PENDING` (created), `QUEUED` (awaiting execution), `EXECUTING` (in progress) and one of {`COMPLETED`, `ERROR`, `ABORTED`} (finished). These primary phases allow clients to monitor the progress of the job (see Figure 2.6).

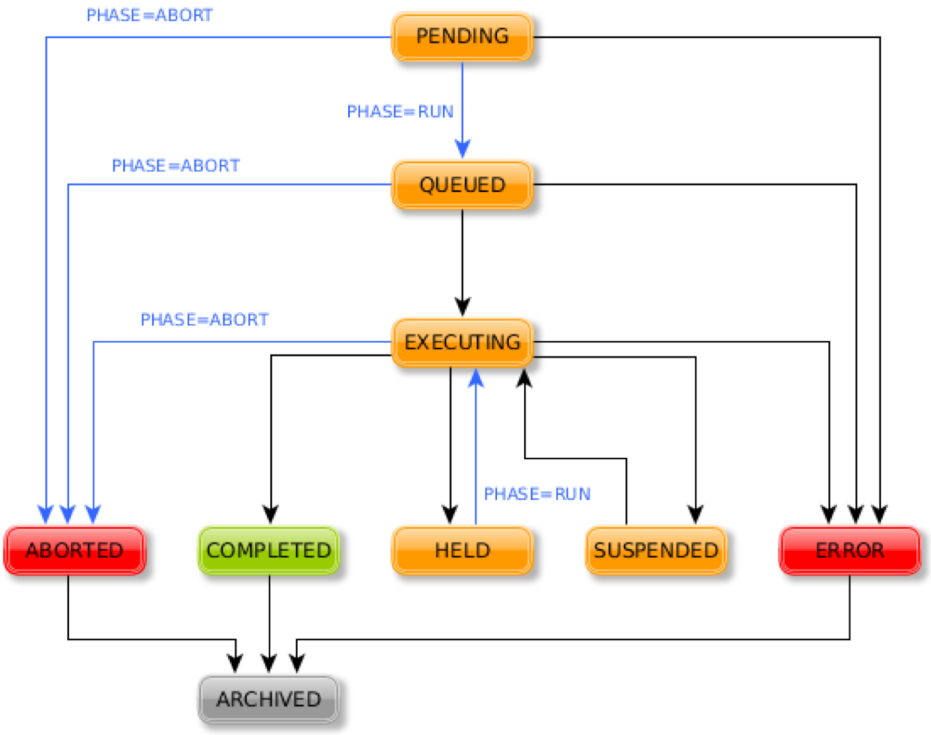

Figure 2.6 UWS Job transitions. [16]

Job cancellation. Clients can abort a running or queued job by issuing the `ABORT` action. This immediately transitions the job to the `ABORTED` state and triggers any configured cleanup actions.

RESTful bindings. Key RESTful API endpoints for UWS operations:

Method	URI	Action
POST	`/jobs`	Create a new job
POST	`/jobs/{job-id}/phase?PHASE=RUN`	Start job execution
GET	`/jobs/{job-id}/phase`	Retrieve job phase
DELETE	`/jobs/{job-id}`	Delete job

Table 2.1 Key UWS operations mapping to HTTP methods and URIs.

VO-CLOUD follows the UWS domain model, maintaining job phases and REST endpoints, but adapts job metadata and extensions for experiment tracking, ML parameters, and integration with its custom orchestration API.

2.1.3 Master–Worker

The **Master–Worker** pattern is a standard distributed computing model that separates coordination from execution (see Figure 2.7). A central Master accepts and dispatches jobs, while Workers poll for them, perform computations, and return results. [17]

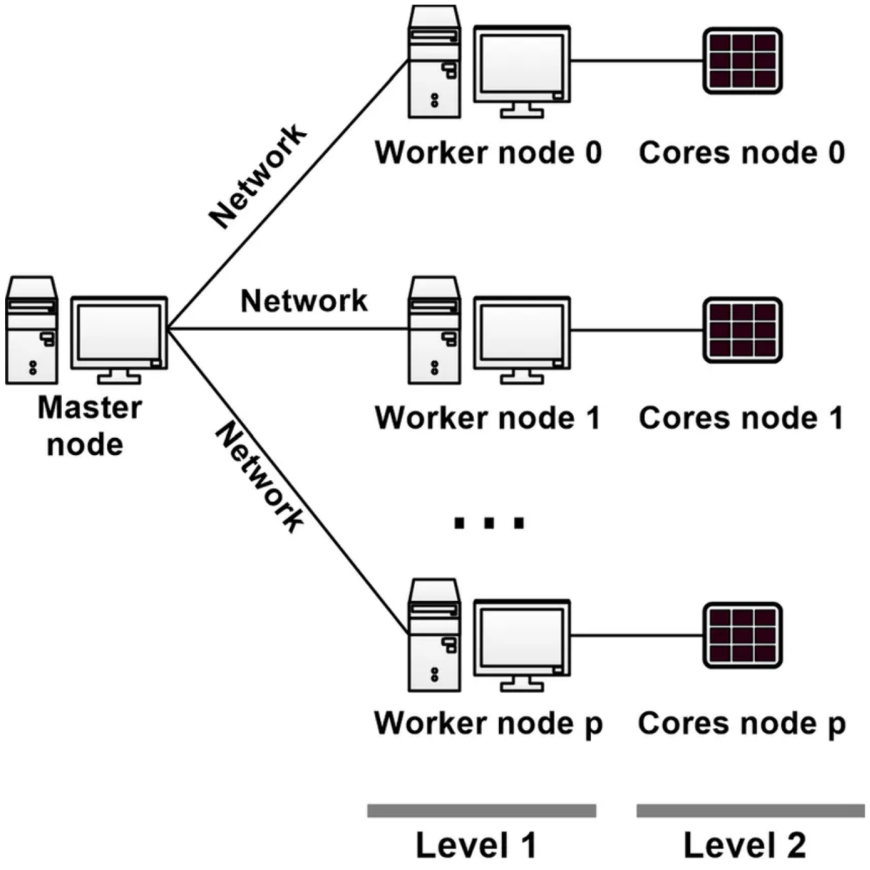

Figure 2.7 Master–Worker distributed pattern. [18]

Master server. Implements the UWS REST API as the entry point for job submission. Upon receiving a request, it creates a new Job resource with `PENDING` state, persists its parameters, and then transitions it to `QUEUED` for assignment to an available Worker.

Workers. Independently running processes or applications that periodically query the Master for `QUEUED` jobs. Each Worker fetches job parameters, loads the necessary data from storage, executes the job, and posts results back via the UWS endpoints.

Asynchronous communication. Interaction between Master and Workers is fully decoupled: the Master does not block waiting for job completion, and Workers operate in parallel. Job states are updated independently through standard UWS calls, ensuring reliability and transparency.

Scalability and fault tolerance. The Master can distribute workload across any number of Workers, which may run on different nodes. If a Worker fails or a job times out, the Master automatically re-queues the job for another Worker without user intervention.

Resource isolation. Each Worker executes its job in a dedicated directory to avoid data conflicts. Concurrency limits per node prevent resource exhaustion and enable controlled parallelism.

VO-CLOUD uses this pattern to decouple request handling from computation, allowing query coordination and data processing to run on separate nodes. By simply adding new Workers that adhere to the UWS protocol, the system can scale horizontally to meet growing workloads without altering the core submission and monitoring logic.

2.2 Core system components

This section presents VO-CLOUD's three primary infrastructure elements: the **Master Server**, **Worker Services**, and **Data Storage**, detailing their roles, technologies, and interactions in the overall platform (see Figure 2.8).

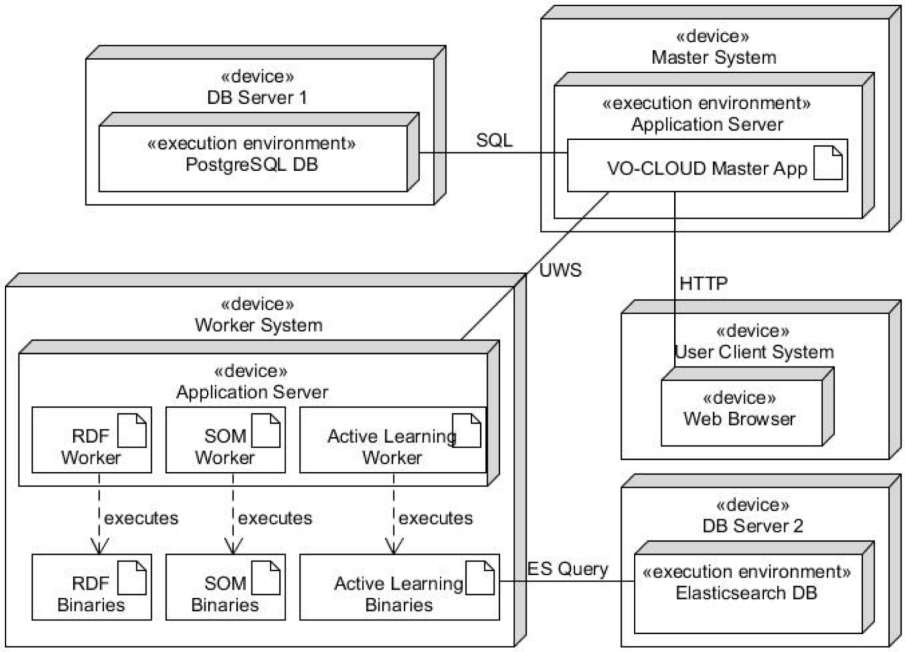

Figure 2.8 VO-CLOUD internal infrastructure. [19]

2.2.1 Master server

The **Master Server** is a single Java EE[3] application that combines user interface rendering, protocol handling, and job orchestration into one cohesive service.

MVC layers. The Master Server follows the Model–View–Controller[4] (MVC) pattern to separate concerns and improve maintainability.

- *Presentation:* JavaServer Faces[5] (JSF) provides component-based page construction, while Facelets and XHTML templates define view layouts. The JSF lifecycle manages the handling and navigation of events. Client-side JavaScript augments these views with animations, pagination, and responsive form validation.
- *Business logic:* stateless Enterprise JavaBeans[6] (EJB) encapsulate core

[3] https://www.oracle.com/java/technologies/java-ee-glance.html
[4] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-model-view-control-mvc
[5] https://www.oracle.com/java/technologies/javaserverfaces.html
[6] https://www.oracle.com/java/technologies/enterprise-javabeans-technology.html

domain behavior, such as job parameter validation, and user authentication/authorization, while Contexts and Dependency Injection[7] (CDI) wires together services and manages conversational contexts.

- *Persistence:* the Java Persistence API[8] (JPA) with Hibernate[9] implements object-relational mapping. Entity classes represent spectra metadata, job definitions, and user records; Hibernate automatically generates SQL for CRUD operations against the underlying relational database.

Unified REST API back end and UI front end. The Master Server delivers its service endpoints and user interface from a single deployment.

- *REST endpoints:* developed with the Java API for RESTful Web Services[10] (JAX-RS), these Java classes map HTTP verbs to methods and serialize data as JSON or XML. All endpoints live on the `/api` path and implement IVOA standards (SSAP, DataLink) plus a custom UWS-style job API on the `/jobs` path.
- *Static content:* packaged in the same Web Application Archive (WAR), XHTML templates define the HTML structure, CSS files apply styling, and JavaScript files provide client-side behavior. Serving both API and UI from one archive ensures that page rendering and data services share the exact same codebase, paths, and versioning.

Job orchestration and workflow management. The Master Server handles user access, data retrieval, and end-to-end job processing.

- *User authentication and authorization:* integrates container-managed Java Authentication and Authorization Service[11] (JAAS) managed by containers to register users, assign roles, and enforce access control on API endpoints and UI pages.
- *Spectrum storage integration:* connects to the centralized file repository to locate and stream FITS and VOTable files, resolve URIs, and manage access permissions for large spectral datasets.
- *Job creation and control:* exposes UI forms and REST endpoints for users to define job parameters, select input data, and submit new jobs; supports pausing, resuming, or aborting jobs via the UWS-style protocol.

[7] `https://docs.oracle.com/javaee/6/tutorial/doc/giwhl.html`
[8] `https://www.oracle.com/java/technologies/persistence-jsp.html`
[9] `https://hibernate.org/orm/`
[10] `https://www.oracle.com/technical-resources/articles/java/jax-rs.html`
[11] `https://docs.oracle.com/javase/7/docs/technotes/guides/security/jaas/JAASRefGuide.html`

- *Worker dispatch:* evaluates job requirements, selects an appropriate Worker Service instance, and issues a REST call to initiate execution.
- *Result aggregation and reporting:* receives callbacks or polls Workers for completion status, consolidates output references and logs into the database, and surfaces results through the REST API and generated UI pages.

This Master Server component forms the backbone of VO-CLOUD, unifying user interaction, data services, and job orchestration in a single application that coordinates all downstream processing.

2.2.2 Worker Services

Worker Services execute VO-CLOUD tasks within Java EE containers, providing modules for data preprocessing, active learning workflows, and real-time visualization, while offloading intensive numerical computation and machine learning routines to embedded Python scripts.

Java EE deployment and API contract. Two main Worker types run:

- *Spark Worker:* integrates with the Spark[12] ecosystem to preprocess spectra at scale.
- *Universal Worker:* handles active learning loops, model training, and result packaging.

Each exposes a RESTful API that mirrors the UWS-style protocol, accepts a JSON configuration payload, and returns the output upon completion.

Embedded Python processing. Within each Worker, Python scripts carry out specialized data workflows:

- *Preprocessing:* reads raw spectra, constructs a uniformly spaced wavelength grid based on configuration parameters, interpolates spectral fluxes onto that grid, and applies normalization and standardization routines. These operations leverage NumPy[13] for numerical operations and Scikit-Learn[14] for additional feature scaling or transformation.
- *Machine learning:* drives an active learning loop using a convolutional neural network implemented in TensorFlow[15]. The CNN is first trained on spectra manually labeled by an oracle user, and then applies its learned weights to classify the emission types of new, unseen spectra.

[12] `https://spark.apache.org/docs/latest/`
[13] `https://numpy.org/doc/2.2/numpy-user.pdf`
[14] `https://scikit-learn.org/stable/user_guide.html`
[15] `https://www.tensorflow.org/tutorials`

The Java EE layer invokes these Python scripts as external processes and inspects exit codes to transition job state back to the Master Server.

Interactive visualization and labeling. A dedicated Python service using WebSockets[16] hosts the spectrum viewer and labeling UI. Workers start this service on demand and generate real-time spectral plots with Matplotlib[17] for the browser client.

Worker Services provide the scalable execution layer of VO-CLOUD, performing preprocessing, learning, and visualization tasks, and returning results via the Master API.

2.2.3 Data storage

VO-CLOUD relies on a **multi-tiered storage** architecture to manage metadata and raw data files, ensuring performance and scalability.

Relational metadata store. A PostgreSQL[18] database holds structured records for spectra metadata (object identifiers, observation timestamps, wavelength ranges), job definitions, and user accounts. Access is provided via JPA/Hibernate on the Master Server. Complex joins and transactional integrity guarantee reliable updates to job state and metadata consistency.

Elasticsearch index. An Elasticsearch[19] cluster maintains an index of spectral metadata for fast, faceted search and analytics. The Master Server uses the Elasticsearch REST client to synchronize spectrum ingest and job completion updates, enabling subsecond query response times for the web UI filters.

Network file storage. All raw and processed data, configuration files, and logs are kept on a shared network filesystem (NFS) accessible to both Master and Worker nodes:

- *Raw spectral archives:* original FITS files from multiple LAMOST data releases[20] (DR1, DR2, DR3, …) and other surveys, stored in a standardized directory hierarchy by survey and observation date.
- *Job artifacts:* per-job JSON configuration files, execution logs, and intermediate output, such as normalized spectra, used to audit any workflow run.

[16] https://websockets.readthedocs.io/en/stable/
[17] https://matplotlib.org/stable/users/index
[18] https://www.postgresql.org/about/
[19] https://www.elastic.co/docs/solutions/search
[20] https://www.lamost.org/lmusers/

File paths and URIs are persisted in PostgreSQL, enabling clients to stream or download any file via the Master Server's REST API.

This layered storage strategy ensures that VO-CLOUD can efficiently handle terabyte-scale spectral archives, support interactive exploration, and maintain full traceability of each processing workflow.

2.3 Integration and deployment

This section explains how VO-CLOUD is set up and deployed on the two physical servers at the Astronomical Institute in Ondřejov. First, the **purpose and hardware of each server** are described. Next, the **servers** where Java and Python applications run are described. Finally, the use of Docker **containers** for specific components is covered.

2.3.1 Deployment hosts

The VO-CLOUD system is deployed across **two dedicated machines**, each providing compute and large-scale spectral storage.

betelgeuse

- *Hardware:* equipped with a 12-core (24-thread) Intel CPU, NVIDIA GTX 980 GPU (4 GiB VRAM), 128 GiB RAM and a multiterabyte local disk array.
- *Role:* the primary host for the Master Server, Worker Services, and Python visualization; the central filesystem for user uploads and job output is mounted here.
- *Network:* located on a private VLAN with no direct Internet exposure; client access is mediated via a reverse proxy.

antares

- *Hardware:* provides an 8-core Intel CPU, 24 GiB RAM, and a multiterabyte disk for spectral archives.
- *Role:* functions as a secondary Hadoop DataNode, storing and serving HDFS[21] data blocks for fault-tolerant distributed file storage, and as a YARN worker (NodeManager), launching and monitoring Spark and other containerized jobs.
- *Network:* granted direct Internet connectivity; participates in the HDFS cluster alongside betelgeuse.

[21] `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

In summary, betelgeuse is the compute and storage nexus for core services, while antares extends distributed processing capacity and provides external cluster access.

2.3.2 Application servers

This subsection maps the Java EE and Python applications to their respective **runtime environments**.

Java EE services. The VO-CLOUD Master Server and the Universal Worker REST endpoints are deployed on WildFly[22], an enterprise-grade Java EE application server, for seamless lifecycle, transaction, and security management on betelgeuse.

Python visualization. The spectrum viewer and the Active Learning UI for the labeling purposes, both implemented with Tornado[23], a lightweight asynchronous Python web framework, are also executed on betelgeuse, accessing the same shared filesystem for data.

Thus, Java applications run on bare metal WildFly for integrated lifecycle management, while Python services are isolated in containers and access the same file storage.

2.3.3 Containerization

VO-CLOUD employs a **hybrid deployment model**, combining native host-system installations with Docker[24]-based isolation for selected services.

Native services.

- *Java EE components:* the Master Server and Universal Worker REST endpoints run directly on WildFly, deployed on betelgeuse without containerization.
- *Hadoop daemons:* Hadoop components operate natively on betelgeuse and antares.
- *Elasticsearch & HDFS proxy:* the Elasticsearch indexer and the NFS proxy for HDFS mounting are installed and managed as standard Linux services on the host.

[22] `https://docs.wildfly.org`
[23] `https://www.tornadoweb.org/en/stable/`
[24] `https://docs.docker.com/build-cloud/`

Containerized services.

- *PostgreSQL database:* the Master Server relational database is containerized to simplify versioning, upgrades, and backups.
- *Spectrum viewer:* the Tornado-based interactive plotter is packaged in a dedicated Docker image, mounting the read-only VO-CLOUD storage directory.
- *JupyterHub stack:* JupyterHub[25], its proxy, and per-user notebook servers run under Docker Compose. Each user's notebook launches in its own container, with shared read-only access to VO-CLOUD storage and a private writable workspace.

Only some components, PostgreSQL, the spectrum viewer, and JupyterHub, are containerized. Most services still run directly on the hosts. Complete isolation and uniform deployment remain for future work.

2.4 Basic usage scenarios

This section analyzes three core user workflows in VO-CLOUD: **importing new spectra**, **configuring and submitting computational jobs**, **viewing outputs**, and inspecting error logs.

2.4.1 Spectra import

The user may **add new spectra** to the workspace via three distinct mechanisms:

1. **Direct upload:** drag and drop or file-chooser upload through the web GUI.
2. **VO protocols:** SSAP and DataLink clients built into the Master Server query VO services for spectral data.
3. **Administrator import:** administrators may place files directly on the server, under the project home tree.

All imported files become visible in the VO-CLOUD UI. Users can then move, rename, delete, or download them according to their permissions.

[25] `https://jupyter.org/hub`

2.4.2 Job submission

To **launch a job computation**, the user selects *Create Job* from the main menu and chooses one of the registered types of jobs (see Figures 2.9, 2.10). The user either uploads or selects an existing JSON configuration, or chooses from predefined templates (see Figure 2.11). Upon submission, the Master Server enqueues the job via the UWS REST API, assigns it to an appropriate Universal Worker or Spark Worker, and returns a job identifier. The UI then displays the job phase in real time by polling the UWS endpoint.

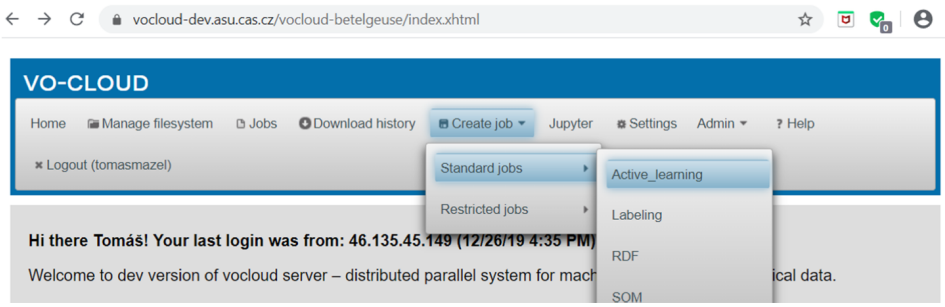

Figure 2.9 VO-CLOUD job creation UI section. [20]

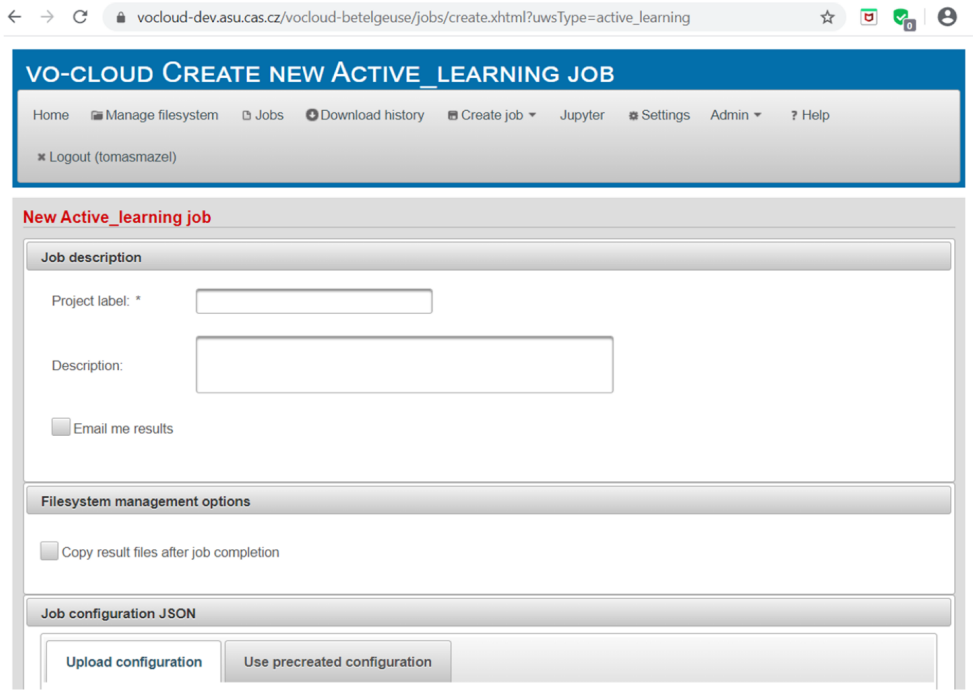

Figure 2.10 VO-CLOUD job specification UI section. [21]

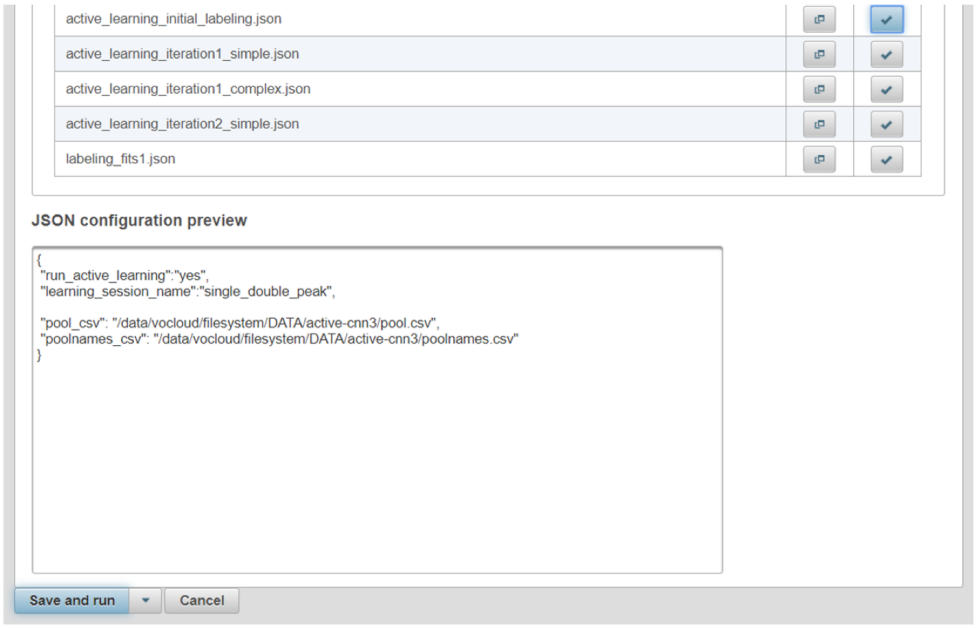

Figure 2.11 VO-CLOUD job configuration UI section. [22]

2.4.3 Result visualization

When a job enters a terminal state (`COMPLETED`, `ERROR`, or `ABORTED`), the user can **view all the output** and any logs (see Figure 2.12).

For successful data processing jobs, the Master Server displays the job directory in the browser. The user can see preprocessed spectra (CSV, FITS) and download links for each file.

For active learning jobs, the results page embeds the labeling interface. The spectra with the highest model uncertainty appear as zoomable plots of raw and normalized data. The user assigns labels (by click), adds comments, and then saves their changes (see Figure 2.13).

If a job fails or is aborted, the worker's stdout stream appears in a log pane. The user can review these logs online or download them for offline debugging.

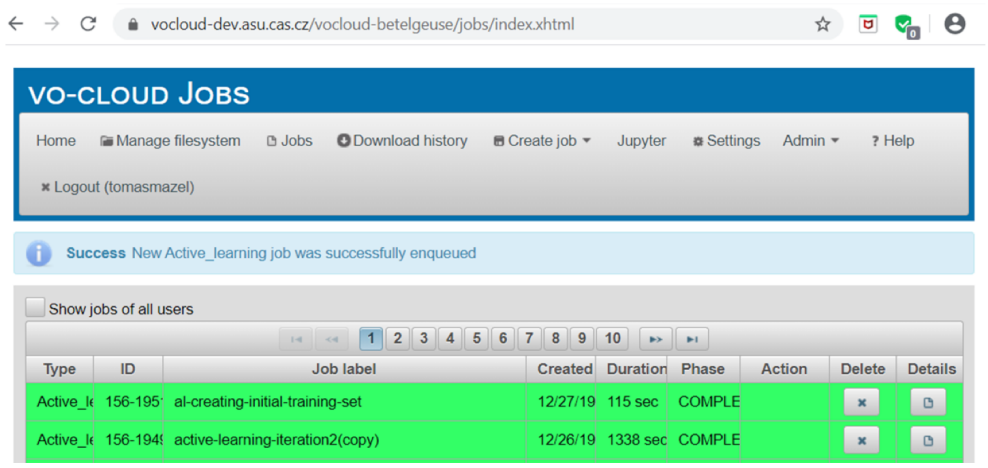

Figure 2.12 VO-CLOUD jobs UI page. [23]

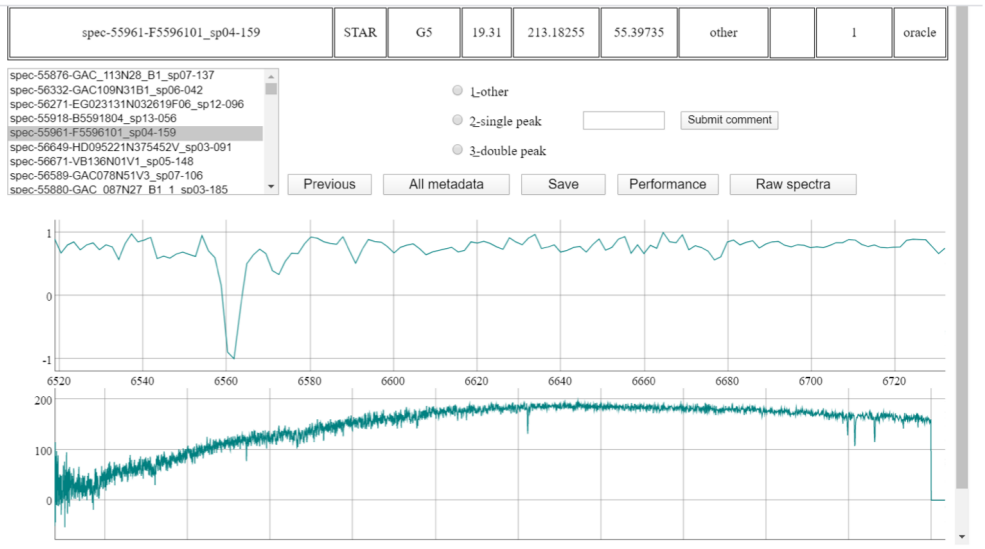

Figure 2.13 VO-CLOUD job result and visualization UI page. [24]

2.5 Analysis summary

The VO-CLOUD platform integrates a rich set of capabilities, such as data ingestion, job orchestration, computation, and result presentation in a single environment. Despite its comprehensive workflow support, the following **limitations** were identified:

- **Monolithic architecture.** Core services, including the user interface, job management, and data handling, are packaged as one extensive Java

EE application. This tight coupling prevents individual components from being updated or replaced independently, which slows down development.

- **Server-side UI rendering.** Views are generated using JavaServer Faces and XHTML templates, resulting in page reloads for most interactions. Modern single-page application frameworks with JSON-based APIs would deliver a more responsive user experience and simplify client-side logic.
- **Inconsistent containerization.** Only a few components: PostgreSQL, the spectrum viewer, and JupyterHub are deployed in Docker. Java EE services and the Hadoop ecosystem run directly on host machines, leading to heterogeneous deployment scripts, environment drift, and increased operational complexity.
- **Underutilized VO standards.** Although SSAP and DataLink are supported, they are bypassed in practice because bulk downloads of extensive spectral archives over these protocols can take weeks. The user prefers to transfer data outside VO-CLOUD, rendering these protocol implementations redundant, yet still demanding maintenance.
- **Custom UWS integration.** Job states and interactions are defined in XML/XHTML and handled by bespoke REST mappings. A shift to JSON-first libraries could eliminate boilerplate code, reduce parsing overhead, and align with current web API best practices.
- **Sparse developer documentation.** The codebase offers limited inline comments. As a result, understanding, debugging, or extending the system requires extensive code exploration.
- **Redundant Hadoop cluster.** Although a Hadoop/Spark cluster was provisioned, it is no longer used for production data processing. All spectral transformations and machine learning tasks run on the Universal Worker, making the cluster an unnecessary operational burden.

Taken together, these factors argue strongly in favor of a complete rewrite, replacing the monolithic stack with clearly separated components, modern APIs, full containerization, and concise and up-to-date documentation.

Chapter 3

Requirements for the new system

Building on the analysis of the legacy platform, this chapter specifies the **functional** and **non-functional requirements** for a modern, modular VO-CLOUD replacement. These requirements cover the back-end API, job management, job queue, Worker behavior, and overall infrastructure.

3.1 General Functional Requirements

FR-G1 – RESTful API for job management. The system shall expose HTTP endpoints that implement Create, Read, Update, and Delete (CRUD) operations for jobs. All request and response bodies must be encoded in JSON to ensure easy integration with external clients and services.

FR-G2 – Flexible, modular job types with asynchronous execution. Job types shall be defined as independent, self-contained modules. Adding a new type of job must require only minimal changes to the core codebase. Initially, the system must support at least two job types:

- *Data preprocessing.*
- *Active machine learning* with interactive spectra labeling.

All jobs must be performed asynchronously so that the API remains responsive at all times.

FR-G3 – UWS-inspired phase management. Each job shall progress through a well-defined set of states: `PENDING`, `PROCESSING`, `COMPLETED`, `ERROR`, and `ABORTED`. Users and Workers may initiate valid state transitions; any invalid transition must produce a clear error message. Jobs may be deleted in

any terminal state. In the event of failure, a human-readable log file must be preserved for diagnosis.

FR-G4 – Configuration-driven execution. Every job shall reference a JSON configuration file stored on the server. This file must specify input and output paths, algorithm parameters, and required environment settings. The system shall persist these configuration files and supply them to Workers at execution time.

FR-G5 – Ordered job scheduling. Jobs shall be recorded in the database in the exact order of their creation, along with a timestamp. By default, the processing must follow a first-come, first-served policy. Workers will consume tasks from a shared queue as soon as they become available.

3.2 Job management functional requirements

FR-M1 – Job data model. Each job record shall include the following fields:

- `job_id`: a globally unique identifier.
- `type`: the job category (e.g., preprocessing, active-ML).
- `label`: a brief, human-readable name.
- `created_at`: the timestamp of job creation.
- `dir_path`: the file system path to the job's output directory.

FR-M2 – Job creation via JSON. An API endpoint must accept a JSON payload to instantiate new jobs. All required fields (`type`, `label`, etc.) must be validated before acceptance. Upon successful creation, the job shall enter the `PENDING` state.

FR-M3 – Job storage and access. All job records (including their metadata and output paths) must be persisted in a database. The API shall allow clients to retrieve, modify, or delete any job. Records of completed or failed jobs must remain queryable for audit and analysis purposes.

FR-M4 – Phase transitions. When a job begins execution, its state must move from `PENDING` to `PROCESSING`. A job may transition from `PROCESSING` to `COMPLETED`, `ERROR`, or `ABORTED`.

FR-M5 – Job abortion. The system shall allow users or Workers to abort any job that has not yet been completed. Aborted jobs must immediately move to the `ABORTED` state, and any allocated resources must be released.

FR-M6 – Job listing order. When listing jobs through the API or UI, the newest jobs appear first and the oldest last. Pagination controls must respect this ordering.

FR-M7 – Queue integration. Upon creation or start, every job must be queued in the shared queue. Aborted or deleted jobs must be removed from this queue. Both the job manager and Worker processes shall be able to add, remove, and poll jobs in the same queue.

3.3 Job queue functional requirements

FR-Q1 – FIFO ordering. The job queue shall ensure that jobs are queued in the same order in which they were queued.

FR-Q2 – Job removal. The queue must support removing any job that has been aborted or is no longer needed.

FR-Q3 – Shared access. The central job manager and all Worker instances must share access to the same queue for enqueuing, dequeuing, and status updates.

3.4 Worker functional requirements

FR-W1 – Queue consumption. Workers must poll or subscribe to the queue and retrieve jobs in FIFO order.

FR-W2 – Job type execution. Each Worker shall declare which types of jobs it supports. Upon fetching a job, the Worker must verify its compatibility before execution.

FR-W3 – Error handling and logging. If a job fails due to invalid parameters or runtime exceptions, the Worker must capture the complete traceback in a log file and mark the job as `ERROR`.

FR-W4 – Metrics reporting. Upon completion (success or failure), Workers must report key execution metrics back to the Master Server via a designated API endpoint.

3.5 Non-functional requirements

3.5.1 Infrastructure and maintainability

NFR-I1 – Containerization. Every service component must be buildable and executable within Docker containers (or Podman). Orchestration is provided through Docker Compose or an equivalent tool.

NFR-I2 – Cross-platform execution. The platform must run reliably on Linux servers. Containerized deployment must also allow operation on other OS hosts without major adjustments.

NFR-I3 – Modern technology stack. Development must employ up-to-date frameworks, libraries, and language versions to ensure maintainability and security.

3.5.2 Extensibility, modularity, and performance

NFR-E1 – Modular architecture. The platform must separate the API, queue management, Worker execution, and storage layers into distinct modules with well-defined interfaces.

NFR-E2 – Horizontal scalability. It must be possible to scale out by dynamically adding or removing Worker instances to meet workload demands.

NFR-E3 – Non-blocking operations. Long-running tasks shall not block the main API or degrade responsiveness. Resources allocated for individual jobs must be released promptly when no longer needed.

3.5.3 Compatibility, documentation, and openness

NFR-C1 – VO-independence. The new system should focus on core job management and ML workflows, rather than mandating legacy VO protocols.

NFR-C2 – Clear documentation and self-describing API. A specification must be published for all HTTP endpoints. The comprehensive deployment, usage, and developer guides shall be maintained in a public repository.

Chapter 4

Researching suitable technologies

In this chapter, **technologies are reviewed and selected** to achieve a clear separation of responsibilities among the server infrastructure of the new system. Leading scientific platforms and proven architectural patterns are examined to adopt best practices without reinventing the wheel. The goal is to assemble a modern, maintainable technology stack that meets the modularity, scalability, and asynchronous execution requirements in Chapter 3.

4.1 Overview of large scientific platforms

This section summarizes the **architecture** and **key technologies** of three large scientific platforms: Pangeo, SciServer, and ESA Datalabs, each designed to support large-scale data analysis through modular, scalable infrastructures.

4.1.1 Pangeo

> *"The Pangeo community is actively engaged in developing geoscientific software and platforms. Many of these tools are useful far beyond the geoscientific domain."* [25]

Pangeo is an open-source ecosystem designed for reproducible analysis of large multidimensional datasets. Although its origins lie in geoscience, the cloud-native architecture and scalable tools of the platform support any domain that requires parallel processing of terabyte-scale data (see Figure 4.1). [26]

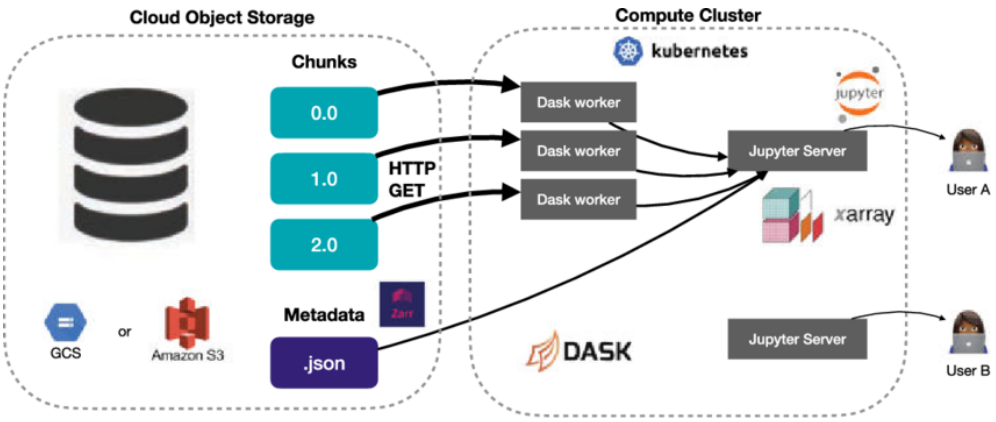

Figure 4.1 Pangeo internal infrastructure. [27]

Key architectural concepts.

- *Modular ML extensions:* the Pangeo-ML project extends core Pangeo libraries with high-level interfaces to integrate extensive datasets into ML workflows seamlessly.
- *Cloud-native storage:* chunked, compressed formats are hosted on distributed object stores or High Performance Computing (HPC) filesystems to enable efficient parallel I/O.
- *Distributed compute engine:* computational frameworks orchestrate task graphs across clusters, supporting both in-memory and out-of-core computation for workloads exceeding available RAM.
- *Labeled data model:* ML data-handling frameworks enrich N-dimensional arrays with coordinate metadata, simplifying indexing, alignment, and broadcasting of complex data structures.
- *Interactive analysis:* code UI notebooks provide a web-based interface for consoles and dashboards, while the Holoviz technology stack provides responsive large-data visualizations.

Core technologies and formats.

- *Zarr*[1], *HDF5*[2] for scalable chunk-based storage optimized for parallel reads and writes.
- *Dask*[3] as the scheduler and distributed executor, enabling seamless scaling from single machines to large clusters.

[1] `https://zarr.dev`

[2]

[3] `https://www.dask.org`

- *Xarray*[4] for metadata-aware, multidimensional array management and analysis.
- *JupyterLab*[5] for interactive computing and exploration within the browser.
- *hvPlot*[6], *SpatialPandas*, *GeoViews* for high-level interactive visualizations of big data.

4.1.2 SciServer

> *"SciServer is a revolutionary new approach to achieving productive science research by bringing the analysis to the data."* [28]

SciServer is a cloud-based science platform developed by the Institute for Data Intensive Engineering and Science (IDIES) at Johns Hopkins University. It evolved from SkyServer, the Sloan Digital Sky Survey (SDSS) web portal, where SDSS is an international astronomy project that maps the universe in three dimensions through optical imaging and spectroscopy. SciServer extends SkyServer's server-side analytics to multiple domains, hosting large relational databases and file stores, while offering interactive and batch analyses through Docker/VM-based environments (see Figure 4.2). [29, 30]

[4] `https://docs.xarray.dev/en/stable/`
[5] `https://jupyterlab.readthedocs.io/en/latest/`
[6] `https://hvplot.holoviz.org`

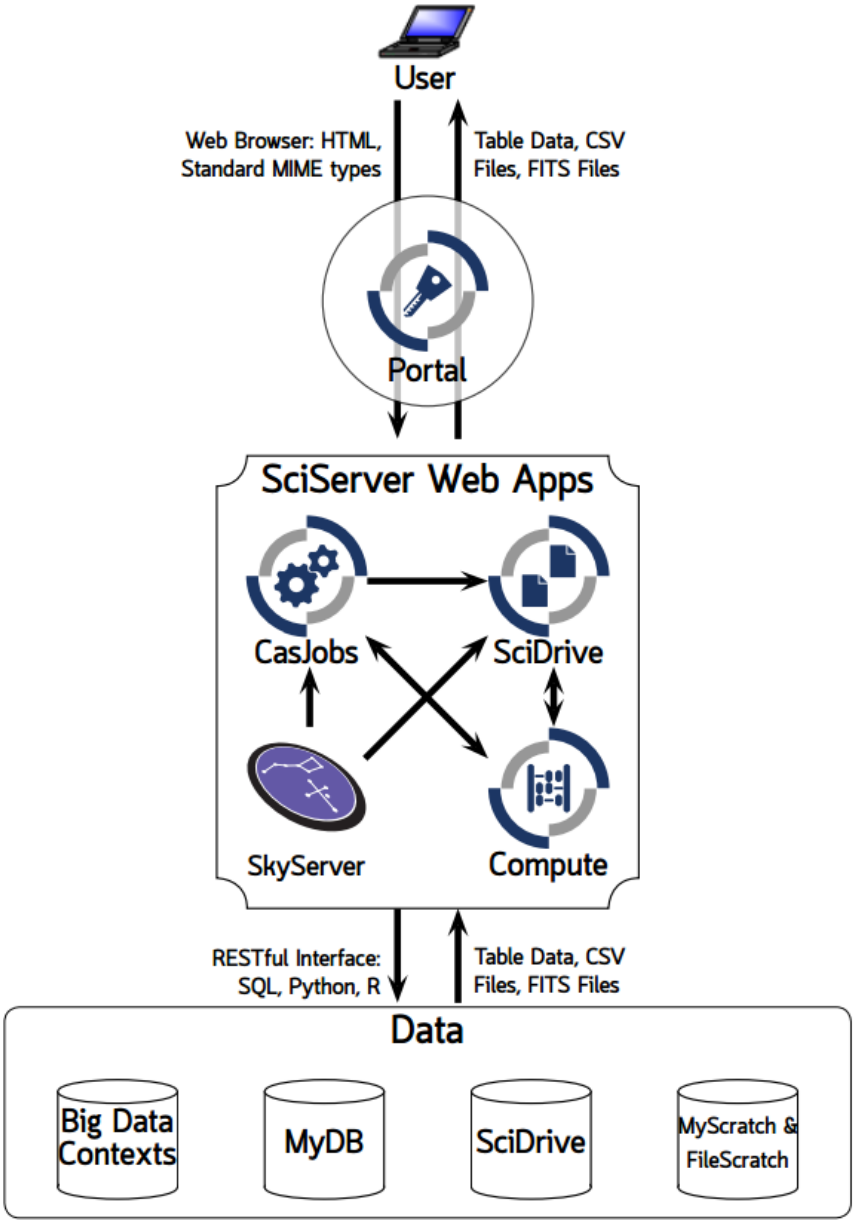

Figure 4.2 SciServer internal infrastructure. [31]

Key architectural concepts.

- *Server-side data access:* core datasets (SDSS catalog) reside in relational databases and are accessed via the SkyServer Web interface or the SQL programming endpoints.

- *Containerized analysis environments:* SciServer Compute provides code UI notebooks and command-line sessions running in containers, co-located with the data to eliminate large transfers.
- *Asynchronous and interactive querying:* CasJobs enables long-running queued SQL queries with personal database spaces and shared scratch spaces, while SkyServer supports immediate interactive queries via a web portal.
- *Integrated file storage:* SciDrive offers RESTful file storage for private and shared data, and temporary FileScratch volumes are mounted in the compute sessions to minimize data movement.
- *Unified REST API layer:* all core services, data queries, compute session management, and file operations are exposed through consistent REST APIs for automation and external integration.

Core technologies.

- *Docker/VM containers* to encapsulate interactive and batch computing environments (SciServer Compute) adjacent to the data.
- *JupyterLab, RStudio*[7] for in-browser interactive analysis sessions using Python, R, or MATLAB.

4.1.3 ESA Datalabs

> *"ESA datalabs offers the possibility of searching ESA's data holdings across domains using the ESA data discovery portal we are developing. To start, you will be able to search for data collections in astronomy, earth observation, navigation, but we expect to offer representative data collections from all of ESA's directorates."* [32]

ESA Datalabs is a collaborative e-science platform provided by the European Space Agency (ESA) to support data-intensive research in space and Earth sciences. Instead of transferring large datasets to users, ESA Datalabs allows code and workflows to run in situ on ESA's infrastructure, with seamless access to mission archives and federated data collections (see Figure 4.3). [33, 34, 35]

[7] `https://posit.co/download/rstudio-desktop/`

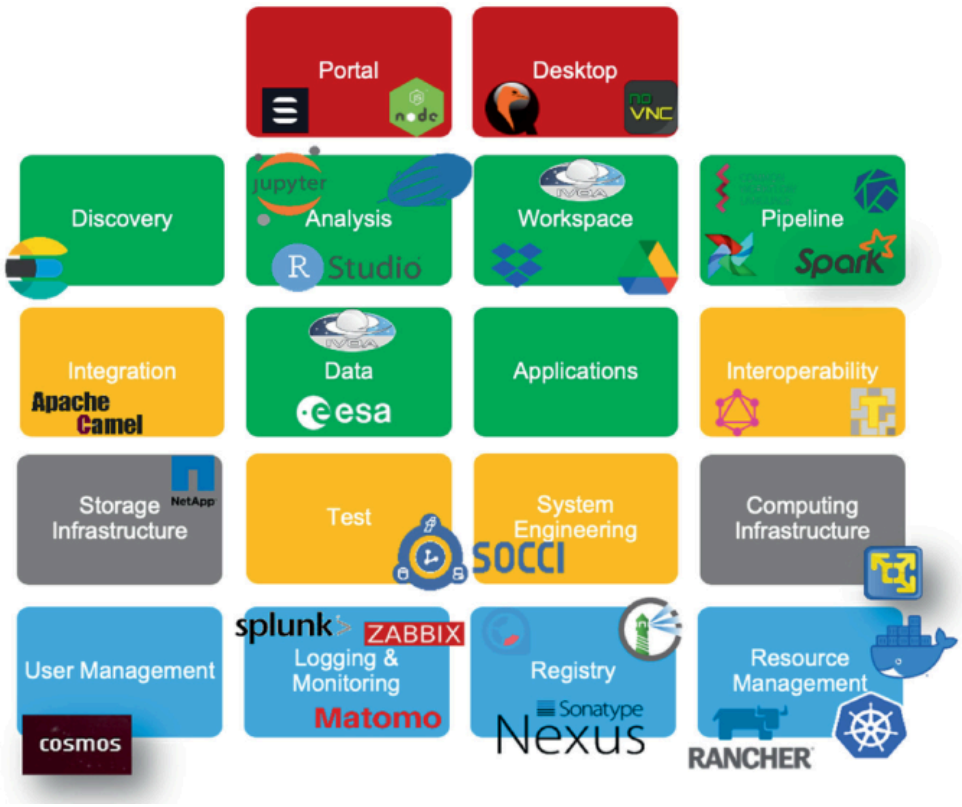

Figure 4.3 ESA Datalabs internal infrastructure. [36]

Key architectural concepts.

- *Bring-code-to-data paradigm:* Users select data collections through a web portal, then mount them into their workspace without copying, enabling immediate access and execution.
- *N-tier, modular design:*
  - **User Layer:** web platform (customized JupyterLab, Octave, Zeppelin) decouples presentation from processing.
  - **Exploitation/Preservation Layer:** domain-specific and generic analysis modules operate on native or federated datasets.
  - **Support Layer:** shared libraries and glue-code components provide standard services between modules.
  - **Infrastructure Layer:** storage and orchestration solutions are configured to meet specific requirements.
- *Federated data discovery:* a unified portal allows searching ESA's holdings across astronomy, Earth observation, navigation, and other domains, then mounting chosen collections into user workspaces via RESTful APIs.

- *Collaborative workspaces:* private and team storage areas support fine-grained access control, persistent backups, and project sharing without data duplication.
- *Reproducible computational narratives:* notebooks and workflows capture code, dependencies, and data references to ensure long-term reproducibility of the analyses.

Core technologies.

- *Kubernetes*[8], *Rancher*[9] for container orchestration, service management, scalability, and high availability.
- *Docker* to package analysis environments, including JupyterLab, Octave, and Zeppelin, with domain-customized extensions.
- *JupyterLab, Octave*[10], *Zeppelin*[11] as in-browser IDEs and notebook interfaces supporting Python, R, MATLAB, and domain-specific languages.

4.2 Selection of architectural patterns and concepts

To decompose the legacy monolith into independent, scalable components, three architectural approaches are considered: **microservices**, **job queues**, and **containerized orchestration**.

4.2.1 Microservices

> *"Microservices architecture is a new trend embraced by many organizations as a way to modernize their legacy applications."* [37]

In contrast to a monolithic application, where all modules are packaged and deployed as a single executable, **microservice architecture** organizes an application as a collection of small autonomous services that can be developed, deployed and scaled independently (see Figure 4.4).

[8] `https://kubernetes.io`
[9] `https://www.rancher.com`
[10] `https://gnu-octave.github.io/packages/`
[11] `https://zeppelin.apache.org`

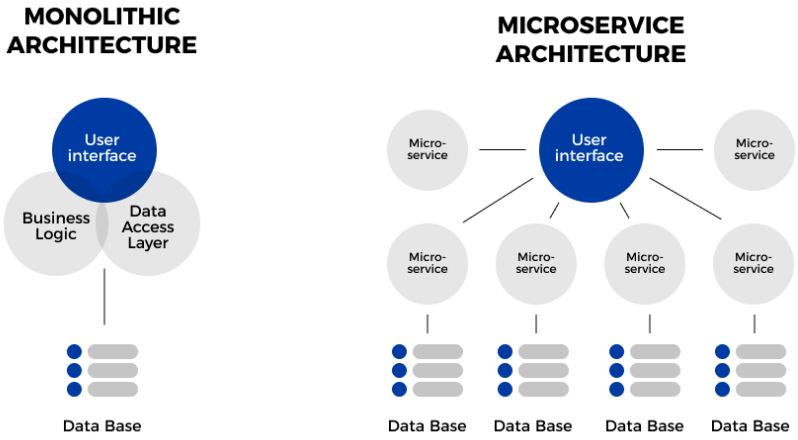

Figure 4.4 Microservices architecture. [38]

Principles and motivations. Key principles guiding the decomposition into microservices include:

- *Modularization:* services encapsulate functionality and expose only the interfaces required by others, reducing coupling and improving comprehensibility.
- *Independent deployability:* services can be released on separate schedules, minimizing the risk of system-wide downtime.
- *Polyglot governance:* each service may adopt the most suitable language, framework, and persistence mechanism for its domain.
- *Fault isolation:* service failures are contained, preventing cascading errors that would destroy an entire monolith.

Advantages and drawbacks. Microservices offer:

- Enhanced scalability through fine-grained resource allocation.
- Accelerated development process.
- Improved resilience by isolating faults within individual services.

However, they introduce operational complexity in service communication, orchestration, and data consistency management.

Application to the new solution. For the new system, the monolithic VO-CLOUD platform will be restructured into three collaborating services:

1. *UI client service:* a single-page application (SPA) handling user interaction, developed in parallel by Alisher Layik as part of his thesis ("Science Platform for Machine Learning of Big Astronomical Data – Data Analysis Modules").
2. *API service:* implements REST endpoints, validates requests, and queues tasks.
3. *Job queue service:* provides the messaging backbone for decoupling the API from workers and ensuring reliable delivery.
4. *Worker Services:* consume jobs from the queue, performs computations, and reports status.

This split clarifies the responsibilities, enables independent scaling of the API and compute layers, and aligns with modern DevOps practices.

4.2.2 Job queue

> *"Distributed task processing plays a crucial role in modern cloud computing architectures, enabling efficient execution of large-scale computations by dividing tasks across multiple nodes."* [39]

A dedicated **job queue** decouples request submission from execution, allowing tasks to be enqueued rapidly and consumed asynchronously by Worker processes (see Figure 4.5).

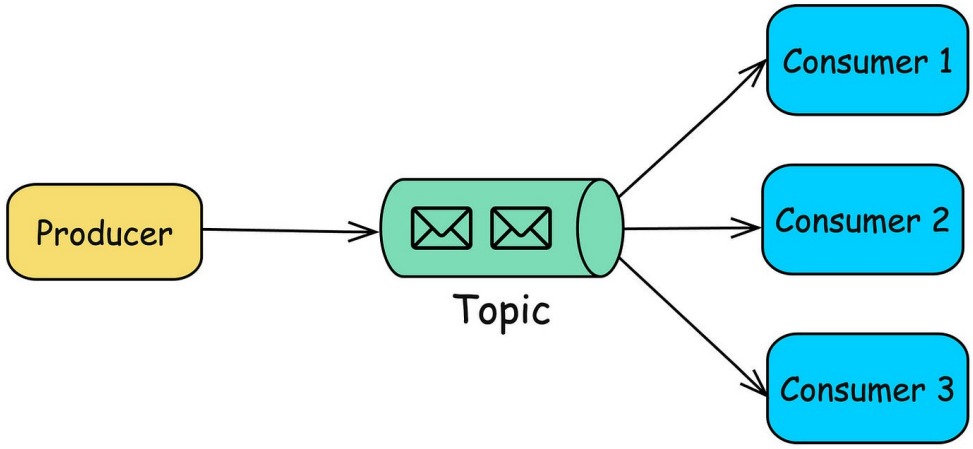

Figure 4.5 Job queue pattern. [40]

Conceptual considerations. Key requirements for an effective job queue include:

- *Decoupling and asynchrony:* producers submit jobs without blocking execution, while consumers pull job when resources allow, smoothing load and improving responsiveness.
- *Ordering guarantees:* queues should preserve insertion order (FIFO) or support priority schemes to ensure predictable processing sequences.
- *Reliability and durability:* jobs must be stored persistently until acknowledged, preventing loss in the face of failures.
- *Scalability:* the queue must handle high job volumes and allow multiple consumers to process jobs in parallel.
- *Fault tolerance:* failed or timed-out jobs should be re-tried or re-queued automatically, isolating errors and preventing system-wide disruption.

Application to the new solution. In the redesigned platform, a standalone queue service will:

- Accept job messages from the API service immediately upon request validation.
- Persist jobs durably until they are fetched and acknowledged by worker instances.
- Support multiple Worker consumers pulling jobs concurrently, enabling horizontal scaling of compute capacity.

This dedicated queue layer ensures that job submission remains fast and that compute resources are utilized efficiently and resiliently.

4.2.3 Containerization & orchestration

> *"Containerization provides many promising features like super lightweight, faster spin-up/down, efficient energy and resource utilization, impressive workload distribution capabilities, achieving server consolidation, and many more, but at the same time it has few major problems such as weaker isolation, higher chance of container sprawl, lack of capable tools for container orchestration and cross-platform supports and container portability limitations."* [41]

By packaging each service and its dependencies into **isolated containers**, the system achieves consistent and reproducible deployments across environments. The **orchestration tool** then automates the scheduling, scaling, and management of these containers (see Figure 4.6).

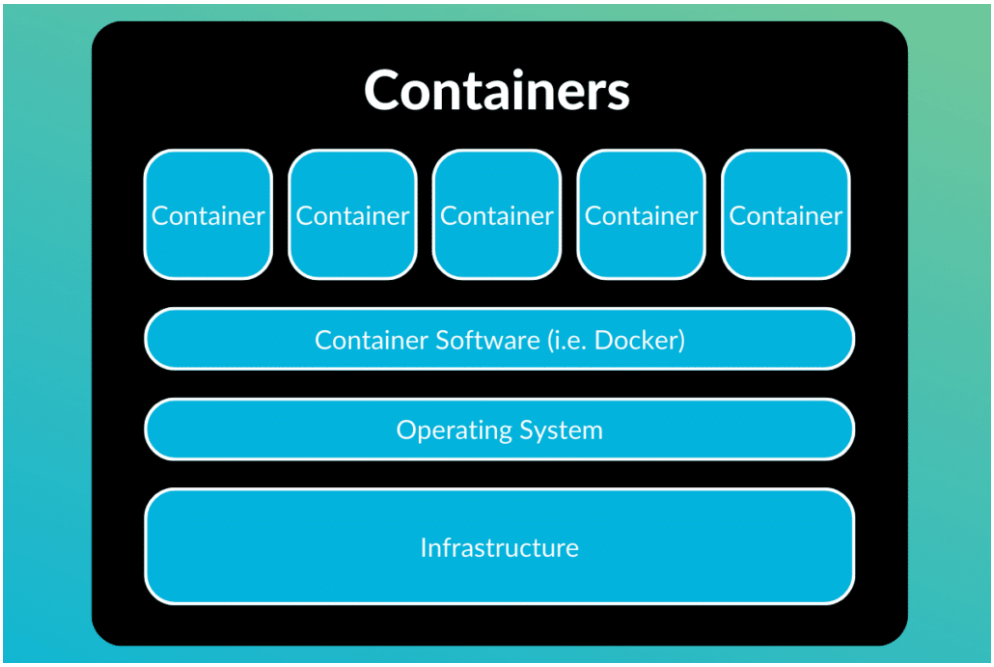

Figure 4.6 Containerization approach. [42]

Conceptual considerations. A containerized architecture must address:

- *Lightweight virtualization:* containers share a common host kernel, minimizing CPU, memory, and storage overhead compared to VMs.
- *Portability:* identical container images can run on any compliant host OS, ensuring "write once, run anywhere" deployments.
- *Resource efficiency:* fast start and shutdown facilitate dynamic scaling and server consolidation for energy and cost savings.
- *Orchestration needs:* automated scheduling, rolling updates, and self-healing require a robust control plane.
- *Persistent storage and networking:* stateful workloads demand volume management and software-defined networking across datacenters and clouds.

Application to the new solution. The entire platform will be delivered as container images and managed by a single orchestration layer:

- Each microservice (UI client, API, queue, workers) is encapsulated in its own Docker image with explicit dependency manifests.
- Docker Compose will schedule containers across the cluster, handle service discovery, and perform rolling upgrades with zero downtime.
- Persistent volumes will be provisioned to support job metadata, logs, and any stateful components.

This unified containerized approach guarantees portability, simplifies configuration, and provides the foundation for automated CI/CD and multi-cloud deployments.

4.3 Choice of core infrastructure components

This section justifies the selection of the fundamental technologies for the new system, focusing on the **implementation language**, the **relational database**, and the **message broker**.

4.3.1 Implementation language

> *"Python is colloquially known as the* lingua franca of data*. It is a non-compiled, not strongly typed, and multi-paradigm programming language that has a clear and simple syntax. Its tooling ecosystem is also extensive, especially in the analytics and ML space."* [43]

Python was selected for both the API and Worker services based on several compelling factors:

- **Ecosystem maturity:** Python 3.13 (API) and 3.11 (Worker) support a wide array of scientific ML libraries: NumPy, SciPy, Scikit-Learn, Pandas, TensorFlow, most of which pioneered their primary APIs in Python and continue to prioritize it over other languages.
- **Lightweight infrastructure:** compared to Java EE, Python services incur a lower memory and startup overhead, simplifying containerization and horizontal scaling.
- **Data-centric tooling:** "Almost every major new software library for use in the data world has a Python API." Its preeminence in data science and machine learning ensures seamless access to the latest frameworks and community-driven innovations.
- **Popularity and community support:** as of August 2023, Python ranked as the most popular programming language worldwide (TIOBE index[12]).

[12] `https://www.tiobe.com/tiobe-index/`

This widespread adoption results in abundant third-party packages, tutorials, and commercial tools.

- **Interoperability:** Python easily interfaces with database drivers, message brokers, and cloud development kits. This makes it straightforward to wrap high-performance native code or cloud-native services without friction.
- **DevOps and MLOps alignment:** modern CI/CD and deployment pipelines (Docker, Podman, Kubernetes) have first-class support for Python.
- **Operational efficiency:** compared to JVM-based stacks, Python services exhibit lower memory footprints and faster startup times, facilitating denser container packing, rapid autoscaling and cost-effective resource utilization.

Together, these factors establish Python as the de facto choice for building scalable, maintainable, and data-intensive microservices in this project.

4.3.2 Relational database management system

> *"PostgreSQL is one of the most popular open source database management systems. It is highly versatile and used across different industries and areas as diverse as particle physics and geospatial databases. One of the defining characteristics of PostgreSQL is its extensibility, which enables developers to add new database functionality without forking from the original project. Many companies have leveraged the rich functionality and ecosystem of PostgreSQL to build advanced, successful applications."* [44]

PostgreSQL was selected as the project RDBMS for the following reasons:

- **Demonstrated scalability:** achieves consistent throughput improvements on gigabyte scales without performance degradation.
- **Operational maturity:** enterprise-grade backup and tools have already underpinned VO-CLOUD's production workloads.
- **Structured-domain modeling:** relational schemas, ACID transactions, and declarative constraints (foreign keys, unique indexes, check constraints) enable precise encoding of domain entities and relationships.
- **Full open-source stack:** PostgreSQL and its ecosystem of extensions (PostGIS, pgML, MADlib, etc.) are released under permissive open source licenses.

The latest stable PostgreSQL 17 release will be used to take advantage of its long-term support, performance enhancements, and wide ecosystem compatibility.

4.3.3 Message broker

> *"RabbitMQ acts as an intermediary between the various services. It reduces the load and delivery time on server web applications by delegating tasks that would typically take a lot of time and resources. Message queuing allows web servers to respond quickly to requests rather than being forced to perform complex procedures that can take more time and resources."* [45]

A standalone **RabbitMQ** broker will be deployed to manage the system job queue. Publishers (the API service) will queue jobs via the Advanced Message Queuing Protocol (AMQP) exchanges, and consumers (Worker services) will pull and process these messages asynchronously.

Key characteristics.

- *Standardized protocol:* Advanced Message Queuing Protocol (AMQP) ensures interoperability, defining exchanges, queues, bindings, and routing keys for reliable message delivery.
- *Decoupling of components:* API and Worker services communicate only through the broker, eliminating direct dependencies and allowing each service to scale and evolve independently.
- *Asynchronous processing:* tasks persist in durable queues until acknowledged, allowing the API to respond immediately and Workers to consume at their own pace.
- *Load leveling and scalability:* multiple Worker instances can subscribe to the same queue, evenly distributing workload, and supporting horizontal scaling with high throughput.
- *Fault tolerance:* unacknowledged or failed messages can be re-queued or routed to dead-letter queues, preventing message loss and isolating errors.
- *Lightweight operations:* RabbitMQ was chosen for its lower infrastructure overhead and simpler configuration, avoiding the operational complexity required by high-throughput brokers such as Apache Kafka[13].

Application to the new solution. In the redesigned platform, RabbitMQ will:

- Host an exchange to which the API service publishes validated task messages (jobs).

[13] https://kafka.apache.org

- Maintain durable queue that buffer jobs until they are consumed by Worker services.
- Support bindings that route messages based on job type.
- Provide monitoring and management endpoints for visibility into queue depth, message rates, and consumer health.

This dedicated message broker ensures that job submission remains fast, reliable, and resilient, while compute capacity can be scaled independently of request traffic.

4.4 Selection of frameworks and libraries

To support the **development**, **validation**, and **deployment** of microservices and data pipelines, a curated set of tools and libraries was selected for their robustness, interoperability, and community support.

4.4.1 Package management

> *"Poetry is a tool for dependency management and packaging in Python. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you. Poetry offers a lockfile to ensure repeatable installs, and can build your project for distribution."* [46]

Poetry is adopted as the dependency and environment manager for the project.

Usage of Poetry. [47]

- *Isolated environments:* automatically creates a dedicated virtual environment per project, so that API, Worker, and CLI components run on the same interpreter and library versions.
- *Single manifest:* declares all dependencies in `pyproject.toml`, unifying the project metadata and requirements.
- *Reproducible installs:* generates a `poetry.lock` file that pins exact package versions, ensuring identical installations on developers' machines and CI pipelines.
- *Conflict prevention:* employs an advanced dependency resolver to detect and prevent version conflicts before installation, avoiding "dependency hell".
- *Integrated packaging:* provides built-in commands to build and publish internal Python packages, simplifying long-term maintenance and distribution.

4.4.2 Web framework

"FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints." [48]

FastAPI was chosen as the project's web framework for the following reasons:

Key features. [48]

- *High performance:* one of the fastest Python frameworks available.
- *Rapid development:* type-hint driven code generation and automatic interactive documentation (OpenAPI/JSON Schema).
- *Asynchronous support:* first-class `async`/`await` endpoints handle I/O-bound tasks efficiently, improving responsiveness to data-intensive workloads.
- *Data validation:* built-in integration for request/response models ensures strong typing, automatic serialization, and runtime validation.
- *Dependency injection:* intuitive and declarative dependency management keeps the code modular, testable, and maintainable.
- *Standards-based:* full compatibility with OpenAPI interactive documentation and JSON Schema simplifies tooling and integration.
- *Robust ecosystem:* widely adopted by major organizations (Microsoft, Uber, Netflix, Cisco) for ML services and microservice architectures.

Comparison to other frameworks. [49]

- *Django*[14]*:* batteries-included, heavy-weight MVC stack; great for full-stack apps, but slower and more monolithic.
- *Flask*[15]*:* minimal micro-framework; flexible but requires manual assembly of extensions for validation, docs, and async support.
- *FastAPI:* strikes a balance: light weight, async-first, and comes with built-in validation and documentation, making it ideal for ML-driven, API-first backends.

[14] `https://www.djangoproject.com`
[15] `https://flask.palletsprojects.com/en/stable/`

4.4.3 Data validation & serialization

> *"Powered by type hints—with Pydantic, schema validation and serialization are controlled by type annotations; less to learn, less code to write, and seamless integration with your IDE and static analysis tools."* [50]

Pydantic provides fast, declarative schemas and JSON Schema export, with a Rust-based core for performance. Its deep integration with FastAPI ensures that all request and response data are automatically validated and serialized.

4.4.4 Database Object Related Mapping

> *"SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL."* [51]

SQLAlchemy's Core and Object Related Mapping (ORM) layers enable both schema-centric SQL construction and domain-centric object persistence. It supports asynchronous engines (via `asyncio`), fully integrates with tools for migrations, and is the de facto standard for Python database access.

4.4.5 Database schema migrations

> *"Alembic provides for the creation, management, and invocation of change management scripts for a relational database, using SQLAlchemy as the underlying engine."* [52]

Alembic tracks incremental schema changes in versioned migration scripts. On service start-up, pending migrations are applied automatically, ensuring database schemas remain synchronized across environments.

4.4.6 Job orchestration

> *"Celery is a simple, flexible, and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system."* [53]

Celery is the project's task orchestration framework, selected for its production-proven reliability and rich feature set:

Core capabilities. [54]

- *Producer/consumer API:* defines jobs in the API service (producer) and executes them in Worker processes (consumers) via AMQP protocol, fully decoupling request handling from heavy computation.
- *Retry and timeout policies:* built-in decorators allow per-job configuration of retry counts, backoff strategies, and execution time limits, improving resilience to transient failures.
- *Job routing:* supports multiple named queues, exchange bindings, enabling separation of pre-processing, inference, and post-processing workloads.
- *Horizontal scalability:* additional Worker instances can be added or removed at runtime, allowing independent scaling of ML inference capacity without affecting the API layer.

Application to the new solution.

- The API service enqueues validated inference and pre-processing jobs via `@celery.task` decorators.
- Worker services run Celery workers that consume from dedicated queue, execute ML model, and update job status in API app.
- Celery's configurable timeouts and retry logic ensure that failed jobs are requeued.

4.4.7 Asynchronous I/O

> *"aiofiles is a library for handling local disk files in asyncio applications by delegating operations to a thread pool."* [55]

aiofiles enables non-blocking host NFS file reads and writes in async endpoints, while the standard **os** module offers portable path and environment utilities [56]. Together, they prevent file I/O from blocking the event loop.

4.4.8 Scientific data handling

> *"The astropy package contains key functionality and common tools needed for performing astronomy and astrophysics with Python."* [57]

> *"The h5py package is a Pythonic interface to the HDF5 binary data format. It lets you store huge amounts of numerical data, and easily manipulate that data from NumPy."* [58]

Astrospectral data are read from FITS files using **astropy**, which provides:

- Comprehensive support for FITS I/O, including header parsing metadata and main spectral data (fluxes, waves) parsing.
- Utilities for unit conversions, coordinate transformations, and spectrum manipulation.
- Integration with affiliated astronomy packages for advanced analysis workflows.

The preprocessed spectra and the derived matrices are stored in HDF5 using **h5py**, chosen for:

- **High-performance, array-style access:** datasets are exposed as NumPy-like arrays with on-disk slicing and lazy loading.
- **Self-describing, portable files:** metadata and raw data coexist in a single binary container compatible with project tools.
- **Seamless ML integration:** native interoperability with NumPy, allowing zero-copy data pipelines in worker processes.

Numerical computation and machine learning are performed with:

- **NumPy**[16] for vectorized array operations and linear algebra.
- **scikit-learn**[17] for classical ML preprocessing (scaling, feature extraction, clustering).
- **TensorFlow**[18] for deep neural network training and inference in worker pipelines.

Each preprocessing job, implemented in Alisher Layik's analysis modules, loads a single LAMOST sub-catalog (400–500 MB of FITS spectra), interpolates each spectrum to a common uniform grid, standardizes, and normalizes it in a sequential pipeline. Since each job's working set is under a gigabyte and must be processed in order, the overhead of a Dask cluster or Xarray arrays isn't justified. Instead, the cleaned data are written to HDF5 via h5py, enabling fast, zero-copy slicing. Model training then streams these HDF5 datasets sequentially into the convolutional network, yielding a simple, efficient workflow without distributed-data complexity.

[16] `https://numpy.org`
[17] `https://scikit-learn.org/stable/`
[18] `https://www.tensorflow.org`

4.4.9 Containerization

> *"Docker is an open platform for developing, shipping, and running applications… Containers are lightweight and contain everything needed to run the application."* [59]

> *"Docker Compose is a tool for defining and running multi-container applications. It simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single YAML configuration file."* [60]

Docker is used to encapsulate each microservice: API, queue, workers, and auxiliary tools, into immutable self-contained images. These containers include the exact runtime, system libraries, and application dependencies required, ensuring reliability across different environments.

Docker Compose orchestrates the complete project stack by declaring each service (API, RabbitMQ, PostgreSQL, Workers) in `docker-compose.yml`.

By standardizing on Docker and Docker Compose (both of which are fully compatible with Podman[19]) the system gains a portable, declarative, and scriptable infrastructure layer.

4.5 Summary of technology selection

This chapter surveyed leading scientific platforms and modern architectural patterns to select the best technologies for VO-CLOUD reengineering. The case studies of Pangeo, SciServer, and ESA Datalabs highlighted the advantages of modular, container-based systems co-located with data. Combined with DevOps best practices, this led to three core concepts: responsibility separation through microservices, asynchronous work via job queues, and reliable and portable deployments through containers.

Consequently, the following **technology stack** was adopted:

- **Implementation language:** Python 3.13 for API Service and Python 3.11 for Workers, chosen for its lightweight runtime, rich ML/data ecosystem, and first-class support in CI/CD pipelines.
- **Relational database:** PostgreSQL 17, selected for its proven scalability, strong relational guarantees, and mature open-source ecosystem.
- **Message broker:** RabbitMQ, providing a lightweight standards-based queueing layer with durable persistence, load leveling, and fault tolerance.
- **Package management:** Poetry to enforce isolated environments, deterministic dependency resolution, and integrated packaging workflows.

[19] `https://podman.io`

- **Web framework:** FastAPI, delivering high-performance, async-first request handling, automatic interactive API documentation, and type-safe validation.
- **Data validation & serialization:** Pydantic is for declarative, Rust-powered schema validation and serialization to JSON.
- **ORM and migrations:** SQLAlchemy (async-capable) for flexible SQL expression and object mapping; Alembic for automated, versioned database schema evolution.
- **Job orchestration:** Celery, enabling producer/consumer patterns, retry/timeout policies, and horizontal scaling of ML inference and preprocessing workloads.
- **Async I/O:** aiofiles and Python's built-in os module to prevent file operations from blocking the event loop.
- **Scientific data handling:** Astropy for FITS I/O and spectral metadata; h5py (HDF5) for high-performance, NumPy-compatible storage; NumPy, scikit-learn, and TensorFlow for numerical and ML pipelines.
- **Containerization:** Docker images and Docker Compose (Podman compatible) to encapsulate each service and declare the multicontainer stack.

This curated stack balances the demands of data-intensive astronomy workflows with modern software engineering best practices (microservices, async APIs, CI/CD). The result will be a flexible, maintainable, and scalable foundation upon which the new modular platform can be implemented and operated in various environments.

Chapter 5

Implementation of the new system

The design and implementation of the new ML Job Manager system are presented in this chapter. Building upon the limitations revealed in the legacy VO-CLOUD platform, the requirements defined in Chapter 3 and the technology stack chosen in Chapter 4 are applied to construct four interconnected microservices: the **ML Job API**, the **ML Job DB**, the **ML Job Queue**, and the **ML Job Worker**. Each component is implemented using best practices in domain-driven design and container-based deployment.

5.1 API microservice

The **ML Job API** microservice has been designed according to a layered "onion" architecture[1], in which each abstraction layer communicates only with its immediate neighbors. At the core lie the repository interfaces (implemented with SQLAlchemy), surrounded by a domain service layer that encapsulates business logic, and finally by the API routing layer (built on FastAPI) that exposes HTTP endpoints. This service cleanly separates four distinct domains: **file management**, **spectral data reading**, **job orchestration**, and **spectrum labeling**, so that each concern is implemented and evolved in isolation. It no longer contains user-interface code or direct queue-management logic. The ML Job API's sole responsibility is to validate and process client requests, coordinate across its internal domains, persist and retrieve state from the ML Job DB, and enqueue job items for downstream Workers.

[1] `https://dev.to/yasmine_ddec94f4d4/onion-architecture-in-domain-driven-design-ddd-35gn`

5.1.1 File management domain

The **File management** domain encapsulates all operations on the server's file storage.

Capabilities. The following operations are provided to fully interact with the system's dedicated file storage:

- *Upload* arbitrary files (configurations, auxiliary data) into specific directories.
- *Download* result or log files produced by completed jobs.
- *Create* and *delete* directories to organize job outputs.
- *List* the contents of any directory (files and subdirectories).

Layered architecture. The File management domain is decomposed into four layers (from bottom to top):

1. *Repository interface* (`src/files/repository.py`): defines the CRUD contract over files and directories.
2. *Repository implementation* (`src/files/repositories/lfs.py`): an LFS repository using os, aiofiles, and aioshutil[2] libraries.
3. *Service layer* (`src/files/service.py`): orchestrates files' repository calls and returns Pydantic serializers.
4. *API router* (`src/files/api/rest.py`): FastAPI Class-Based View[3] (CBV) files' router exposing HTTP methods, paths, parameters, response models, and error mappings.

Domain entities. As in the legacy VO-CLOUD implementation, each file and directory is represented by a self-describing metadata typed model. Each file and directory is represented by a Pydantic "entity" model that carries both its identity and the metadata. The `FileEntity` includes the exact file name `filename` (including extension), the normalized relative path of its parent directory `parent_dir_path`, the file size in bytes `size`, and a UTC timestamp of last modification `modified_at`. The `DirectoryEntity` similarly encapsulates the directory's name `dirname` and the relative path of its containing folder `parent_dir_path`, allowing clients to reconstruct the full storage hierarchy and navigate the server filesystem with precise context.

[2] `https://pypi.org/project/aioshutil/`
[3] `https://fastapi-utils.davidmontague.xyz/user-guide/class-based-views/`

```
// ml-job-api/src/files/entities.py
class FileEntity(BaseModel):
    filename: str
    parent_dir_path: str
    size: int
    modified_at: datetime

class DirectoryEntity(BaseModel):
    dirname: str
    parent_dir_path: str
```

Code listing 5.1 Domain entity models for file and directory metadata.

Local filesystem repository implementation. The `FileRepository` abstraction defines the full set of supported file and directory operations for the ML Job API system. In the `FileLFSRepository` implementation, these operations are mapped directly onto the server's local filesystem: creating and deleting directories, listing contents, uploading files, and streaming downloads. To support large files without blocking the FastAPI event loop, a helper function `download_file` (in `src/files/utils.py`) reads from disk in fixed-size chunks and yields each fragment as an asynchronous iterator (see Code 5.2). Parent directories are created on demand, and path normalization via `get_norm_path` prevents traversal attacks. Missing-resource checks (using `os.path.exists` and `os.path.isfile`/`isdir`) raise domain-specific errors (`FileNotExistError`, `DirectoryAlreadyExistError`, etc.), to be translated into precise HTTP responses at the API layer (see Code 5.3).

```
// ml-job-api/src/files/utils.py
async def download_file(file_path: str, chunk_size: int) ->
↪  AsyncIterator[bytes]:
    async with aiofiles.open(file_path, "rb") as file_reader:
        while chunk := await file_reader.read(chunk_size):
            yield chunk
```

Code listing 5.2 Utility function for LFS file download.

```
// ml-job-api/src/files/repositories/lfs.py
class FileLFSRepository(FileRepository):
    ...
    async def download_by_filename_and_parent_dir_path(self,
    ↪  filename: str, parent_dir_path: str) ->
    ↪  AsyncIterator[bytes]:
        # Build relative and absolute paths
        rel_file_path = get_norm_path(parent_dir_path,
        ↪  child_name=filename)
        abs_file_path = get_norm_path(parent_dir_path,
        ↪  prefix=self.shared_dir_path, child_name=filename)

        # Ensure file exists
        if not os.path.exists(abs_file_path) or not
        ↪  os.path.isfile(abs_file_path):
            raise FileNotExistError(f"Cannot download
            ↪  file='{rel_file_path}'.")

        return download_file(abs_file_path, self.chunk_size)
    ...
```

Code listing 5.3 Repository implementation for LFS file and directory manipulation.

Service usecases. The `FileService` encapsulates business logic by delegating all storage interactions to the abstract file repository. It translates client parameters into repository calls and converts returned entities into Pydantic serializers, remaining completely agnostic to the underlying storage medium. For example, when handling a download request, the `FileService` method `download_file_by_filename` simply forwards the given parameters to the repository, then returns the async byte stream (see Code 5.4).

```
// ml-job-api/src/files/service.py
class FileService:
    ...
    async def download_file_by_filename(self, filename: str,
    ↪  params: EntryLocateParams) -> AsyncIterator[bytes]:
        return await self.repository
            .download_by_filename_and_parent_dir_path(filename,
            ↪  params.parent_dir_path)
    ...
```

Code listing 5.4 Service usecases for file and directory operations.

API router. The REST API router, implemented as a FastAPI REST router in `src/files/api/rest.py`, glues together HTTP transport and the file-management service. Each endpoint is fully annotated with path, HTTP method, operation tags, response model, status codes, and human-readable summaries and descriptions. Input validation is performed automatically by FastAPI and Pydantic: malformed paths, missing query parameters, or invalid payloads result in `HTTP 422`. Within each handler, domain exceptions such as `FileNotExistError` or `DirectoryNotExistError` are caught and translated into `HTTP 404`, while `DirectoryAlreadyExistError` yields `HTTP 409`. Any unanticipated failures bubble up as `HTTP 500`. By mapping every possible outcome to a precise status code and structured JSON response, the router guarantees a clear, consistent contract for clients and automatically generates interactive OpenAPI documentation with example values.

Method	URI	Success	Errors
POST	`/{filename}/upload`	201	422, 500
GET	`/{filename}/download`	200	404, 422, 500
DELETE	`/{filename}`	204	404, 422, 500
POST	`/directories/{dirname}`	201	409, 422, 500
DELETE	`/directories/{dirname}`	204	404, 422, 500
GET	`/`	200	404, 422, 500

Table 5.1 API REST mapping for file and directory endpoints.

5.1.2 Spectral data domain

The **Spectral data** domain provides on-demand access to raw LAMOST FITS spectra.

Capabilities. This domain supports a single operation:

- *Retrieve* both header metadata and the full wavelength and flux arrays for a given FITS spectrum.

Layered architecture. The Spectral data domain mirrors the same four-layer onion structure:

1. *Repository interface* (`src/spectra/repository.py`): declares the contract `get_by_filename` returning a `SpectrumEntity`.
2. *Repository implementation* (`src/spectra/repositories/lfs.py`): computes the LAMOST-encoded path, verifies file existence, and offloads FITS parsing to a background thread via `asyncio.to_thread`[4].

[4] `https://docs.python.org/3/library/asyncio.html`

3. *Service layer* (`src/spectra/service.py`): invokes the repository, wraps the result in a Pydantic serializer, and remains storage-agnostic.

4. *API router* (`src/spectra/api/rest.py`): FastAPI REST router with one `GET` endpoint, parameter validation, response model, and HTTP error mappings.

Domain entity. Each spectral measurement records the distribution of electromagnetic energy as a function of wavelength (or, equivalently, frequency). A photon's energy is given by

$$E = h\,\nu = h\,\frac{C_0}{\lambda},$$

where $h = 6.626 \times 10^{-34}\,\mathrm{J\,s}$ (Planck's constant), ν is the photon frequency in hertz (Hz, cycles s^{-1}), λ its wavelength in meters (m), and $C_0 = 2.998 \times 10^{8}\,\mathrm{m\,s}^{-1}$ the constant speed of light in vacuum. In spectroscopic practice, it is often expressed λ in ångströms (1 Å $= 10^{-10}$ m) and converted E to more convenient units only when needed.

The spectral intensity, or energy flux, across a unit area is defined as

$$I \;=\; Q \;=\; \frac{\partial E/\partial t}{\text{area}} \quad \left[\mathrm{W\,m}^{-2}\right],$$

which at the photon level corresponds to the rate of photon arrival per time and area. In astronomical spectroscopy, the calibrated flux density per unit wavelength is recorded, $F(\lambda)$, typically in units of [61]

$$10^{-17}\,\mathrm{erg\,s}^{-1}\,\mathrm{cm}^{-2}\,\text{Å}.$$

In the ML Job API system, each spectrum therefore provides two principal arrays:

- λ_i: the wavelength vector in ångströms,
- F_i: the corresponding flux density measurements in $10^{-17}\,\mathrm{erg\,s}^{-1}\,\mathrm{cm}^{-2}$ Å.

Together, these define the spectral "wave" (the sampled wavelengths) and its "flux" (the photon-derived intensity) across the observed band, providing the foundational data for all downstream visualization and analysis.

Beyond these raw data arrays, every spectrum in the system carries metadata describing the source and acquisition: the telescope pointing (RA/DEC), the observing date/time, the instrument configuration (spectrograph and fiber IDs), and classification results (e.g., redshift, spectral type). This combination of physical measurements and contextual metadata allows to relate each spectral measurement back to its celestial origin and observing conditions.

Each spectrum in the ML Job API is represented by a Pydantic domain entity `SpectrumEntity` that captures exactly the set of metadata and data

arrays needed for downstream visualization and analysis. In LAMOST Data Release 2 (DR2), the FITS filename `filename` follows the special defined pattern `spec-<LMJD>-<PLANID>_sp<SPID>-<FIBERID>.fits`, encoding observation time, plan ID, spectrograph and fiber numbers. The `targetname` parameter comes from the `DESIG` header, while `observed_at` parameter is parsed from the `DATE-OBS` header. Celestial coordinates (`ra`, `dec`), spectral classification (`type`, `subtype`), photometric magnitudes (`mag_1`…`mag_7`), and signal-to-noise ratios (`sn_u`…`sn_z`) reflect key header keywords. Finally, the analysis results redshift (`z`) and its uncertainty (`z_err`) are taken from the `Z` and `Z_ERR` headers. The two principal data arrays: `wave` and `flux`, are extracted from the primary data unit's third and first rows, respectively. This compact entity model thus provides all information required to reconstruct each spectrum's provenance, calibration, and science-ready data in a single object.

```
// ml-job-api/src/spectra/entity.py
class SpectrumEntity(BaseModel):
    filename: str
    targetname: str
    observed_at: datetime
    type: SpectrumType
    subtype: str
    ra: float
    dec: float
    magtype: str
    mag_1, mag_2, ..., mag_7: float
    sn_u, sn_g, ..., sn_z: float
    z: float
    z_err: float
    wave: list[float]
    flux: list[float]
```

Code listing 5.5 Domain entity model for spectral metadata, wave, and its flux data.

Local filesystem repository implementation. The `SpectrumRepository` abstraction declares the single supported operation, retrieving a parsed spectrum entity by its FITS filename, without prescribing where or how the data are stored. In the `SpectrumLFSRepository` implementation, this operation is mapped directly onto the server's local filesystem: the relative directory path is computed from the LAMOST-encoded filename. Existence and file-type checks using `os.path.exists` and `os.path.isfile` guard against missing or invalid files, raising `SpectrumNotExistError` when appropriate.

Because each FITS file is only ∼200 KB and is accessed on demand for

UI-driven visualization, the overhead of maintaining a separate database table (and associated indexing) for spectral metadata, as done in the legacy VO-CLOUD system, was judged superfluous. Instead, blocking FITS parsing is offloaded to a background thread via `asyncio.to_thread`, invoking a synchronous helper function `read_file` (which employs `astropy.io.fits`) to extract header metadata and data arrays directly from disk. The extracted dictionary is subsequently validated into a `SpectrumEntity` before being returned (see Code 5.6 and Code 5.7), with end-to-end latency remaining under one second per spectrum.

```
// ml-job-api/src/spectra/utils.py
def read_file(file_path: str) -> dict[str, Any]:
    with fits.open(file_path) as hdul_reader:
        header = hdul_reader[0].header
        data = hdul_reader[0].data
        raw_spectrum = dict(
            filename=header["FILENAME"],
            targetname=header["DESIG"],
            observed_at=header["DATE-OBS"],
            type=header["CLASS"],
            subtype=header["SUBCLASS"],
            ra=header["RA"],
            dec=header["DEC"],
            magtype=header["MAGTYPE"],
            mag_1, mag_2, ..., mag_7=header["MAG7"],
            sn_u, sn_g, ..., sn_z=header["SN_Z"],
            z=header["Z"],
            z_err=header["Z_ERR"],
            wave=data[2].tolist(),
            flux=data[0].tolist(),
        )

    return raw_spectrum
```

Code listing 5.6 Utility function for LFS spectrum read.

```
// ml-job-api/src/spectra/repositories/lfs.py
class SpectrumLFSRepository(SpectrumRepository):
    ...
    async def get_by_filename(self, filename: str) ->
    ↪  SpectrumEntity:
        # Build relative and absolute paths
        rel_parent_dir_path = self._get_dir_path(filename)
        rel_file_path = get_norm_path(rel_parent_dir_path,
        ↪  child_name=filename)
        abs_file_path = get_norm_path(rel_parent_dir_path,
        ↪  prefix=self.shared_dir_path, child_name=filename)

        # Ensure file exists
        if not os.path.exists(abs_file_path) or not
        ↪  os.path.isfile(abs_file_path):
            raise SpectrumNotExistError(f"Cannot get spectrum
            ↪  from file='{rel_file_path}'.")

        spectrum_file_data = await asyncio.to_thread(read_file,
        ↪  abs_file_path)

        return SpectrumEntity.model_validate(spectrum_file_data)
    ...
```

Code listing 5.7 Repository implementation for LFS spectra reading.

Service usecases. The `SpectrumService` encapsulates all business logic for spectral retrieval by delegating data access to the abstract repository and remaining agnostic to the storage backend. Upon receiving a request, it invokes method `get_by_filename`, handles any `SpectrumNotExistError` propagation, and transforms the resulting `SpectrumEntity` into a Pydantic serializer for API output (see Code 5.8).

```
// ml-job-api/src/spectra/service.py
class SpectrumService:
    ...
    async def retrieve_spectrum_by_filename(self, filename: str)
    ↪  -> SpectrumReadSerializer:
        entity = await self.repository.get_by_filename(filename)

        return SpectrumReadSerializer(**entity.model_dump())
    ...
```

Code listing 5.8 Service usecases for spectra operations.

API router. The REST API router for spectral data, implemented as a FastAPI REST router in `src/spectra/api/rest.py`, binds HTTP transport to the `SpectrumService`. The single `GET` endpoint is fully annotated with path parameter validation (against the LAMOST filename pattern), operation tags, response model, status codes, and human-readable summaries and descriptions. Invalid inputs (e.g., invalid filenames) are rejected with `HTTP 422`, a missing spectrum triggers a caught `SpectrumNotExistError` translated into `HTTP 404`, and any unexpected failures result in `HTTP 500`.

Method	URI	Success	Errors
GET	`/{filename}`	200	404, 422, 500

Table 5.2 API REST mapping for spectra endpoints.

5.1.3 Job management domain

The **Job management** domain encapsulates the full lifecycle and orchestration of background ML jobs.

Capabilities. This domain supports a bundle of operations:

- *Initialize* a new job with type, label and optional description.
- *Retrieve* full details of an existing job by its UUID.
- *Edit* mutable job attributes (e.g., `label`, `description`).
- *Remove* a job (if in `PENDING`, `COMPLETED`, `ERROR`, or `ABORTED` phase).
- *List* jobs with pagination (`offset`, `limit`).
- *Process* a job: `RUN` (enqueue for execution) or `ABORT`.
- *End* a job: mark as `COMPLETE` or `ERROR`, recording execution metrics.

Layered architecture. The Job management domain follows the same pattern:

1. *Repository interface* (`src/jobs/repository.py`): declares CRUD and lifecycle operations, e.g., (`create`, `get_by_job_id`, `update_by_job_id`, `total`, `list_by_offset_and_limit`).
2. *Repository implementation* (`src/jobs/repositories/postgres.py`): the developed PostgreSQL-backed repository using SQLAlchemy ORM.
3. *Service layer* (`src/jobs/service.py`): orchestrates job creation, editing, deletion, queueing (via Celery framework), phase transitions, and pagination.
4. *API router* (`src/jobs/api/rest.py`): FastAPI router exposing CRUD, action and list endpoints, mapping domain errors to HTTP statuses.

Domain entity. Each ML job is represented by a Pydantic `JobEntity` (see Code 5.9) carrying both its unique identity and all relevant metadata required to track and orchestrate its lifecycle. Upon initialization, a new `job_id` (a UUID) is generated and a corresponding `dir_path` is constructed by normalizing the user-provided `label` (e.g. "LAMOST 2025 V883 Orionis Spectra Learning") into a filesystem-safe string (lowercased, spaces replaced with "_") and appending the UUID, yielding paths such as

`/JOBS/job_lamost_2025_v883_orionis_spectra_learning_3fa85f64....`

The `type` field (one of `DATA_PREPROCESSING` or `ACTIVE_ML`) categorizes the job's functional purpose. Its current `phase` (an enumerated `PhaseType`: `PENDING`, `PROCESSING`, `COMPLETED`, `ERROR` or `ABORTED`) reflects where it stands in the lifecycle. A human-readable `label` and optional `description` provide contextual information. Timestamps, `created_at`, `started_at`, and `ended_at`, record key events in UTC, and `execution_duration` captures the total run time once available.

```
// src/jobs/entity.py
class JobEntity(BaseModel):
    job_id: UUID
    dir_path: str
    type: JobType
    phase: PhaseType = PhaseType.PENDING
    label: str
    description: str | None = None
    created_at: datetime | None = None
    started_at: datetime | None = None
    ended_at: datetime | None = None
    execution_duration: float | None = None
```

Code listing 5.9 Domain entity model for job metadata.

PostgreSQL repository implementation. All job records are persisted in a PostgreSQL table via SQLAlchemy ORM. CRUD operations and pagination are implemented in `JobPostgresRepository`. Each method (see Code 5.10) loads or manipulates ORM instances (the `JobPostgresModel`), commits the transaction, refreshes state, and maps to `JobEntity`, raising the defined error `JobNotExistError` when appropriate:

```
// src/jobs/repositories/postgres.py
class JobPostgresRepository(JobRepository):
    ...
    async def create(self, dto: JobCreateDTO) -> JobEntity:
        orm = self.model(**dto.model_dump(exclude_none=True))

        self.session.add(orm)
        await self.session.commit()
        await self.session.refresh(orm)

        return JobEntity.model_validate(orm)
    ...
```

Code listing 5.10 Repository implementation for PostgreSQL jobs manipulation.

Service usecases. The `JobService` encapsulates all business logic for managing ML job lifecycles, remaining agnostic to both the persistence layer and the job queue implementations. Its responsibilities include:

- *Initialize*: a new UUID is generated and combined with a normalized,

label-based folder name to form `dir_path`; a `PENDING`-phase record is persisted via the repository.

- *Retrieve/Edit/Delete*: existing jobs are fetched or updated through the repository; deletion is allowed only in terminal or pending phases (`PENDING`, `COMPLETED`, `ERROR`, `ABORTED`), otherwise a `JobPhaseConflictError` is raised.
- *List*: jobs are listed in descending creation order with total count, offset, and limit parameters to support pagination.
- *Process (`RUN`/`ABORT`) 5.12*:
  - **RUN** (allowed only from `PENDING`) enqueues a Celery job via the method `run_by_job_id_and_job_type`, then transitions the phase to the phase `PROCESSING`.
  - **ABORT** (allowed only from `PROCESSING`) sends a revoke to Celery via the method `abort_by_job_id`, then transitions the phase to `ABORTED`.
- *End (`COMPLETE`/`ERROR`)*:
  - Both `COMPLETE` and `ERROR` are allowed only from `PROCESSING`. The service computes `execution_duration` from provided timestamps, updates `started_at`, `ended_at` and `phase` to `COMPLETE` or `ERROR`, and persists the changes.

Phase transitions follow the strict sequence

$$\text{PENDING} \xrightarrow{\text{RUN}} \text{PROCESSING} \xrightarrow{\text{ACTION}} \{\text{COMPLETED},\ \text{ERROR}\}$$

with an additional ABORTED state reachable from `PROCESSING`.

A key part of this logic is the integration with a Celery-backed `JobQueue`, configured as follows (see Code 5.11):

```
// src/jobs/clients/celery.py
class JobCeleryQueue(JobQueue):
    ...
    def run_by_job_id_and_job_type(self, job_id: UUID, job_type:
    ↪  JobType, dto: JobStartDTO) -> None:
        self.queue.send_task(task_id=str(job_id), name=job_type,
        ↪  kwargs={"dto": dto.model_dump()})

    def abort_by_job_id(self, job_id: UUID) -> None:
        self.queue.control.revoke(task_id=str(job_id),
        ↪  terminate=True)
    ...
```

Code listing 5.11 Queue implementation for Celery jobs manipulation.

```
// src/jobs/service.py
class JobService:
    ...
    async def manage_job_by_job_id_and_process_action(
        self, job_id: UUID, process_action: ProcessActionType
    ) -> JobReadSerializer:
        entity = await self.repository.get_by_job_id(job_id)
        allowed_operation_phases = {
            ProcessActionType.RUN: {
                PhaseType.PENDING,
            },
            ProcessActionType.ABORT: {
                PhaseType.PROCESSING,
            },
        }

        ...

        if process_action == ProcessActionType.RUN:
            self.queue.run_by_job_id_and_job_type(entity.job_id,
            ↪  entity.type,
            ↪  JobStartDTO(dir_path=entity.dir_path))
            entity = await
            ↪  self.repository.update_by_job_id(entity.job_id,
            ↪  JobUpdateDTO(phase=PhaseType.PROCESSING))

        elif process_action == ProcessActionType.ABORT:
            self.queue.abort_by_job_id(entity.job_id)
            entity = await
            ↪  self.repository.update_by_job_id(entity.job_id,
            ↪  JobUpdateDTO(phase=PhaseType.ABORTED))

        return JobReadSerializer(**entity.model_dump())
    ...
```

Code listing 5.12 Service usecases for jobs operations.

API router. The FastAPI REST router (`src/jobs/api/rest.py`) binds HTTP transport to the `JobService`. Each endpoint is annotated with path, method, tags, response model, status codes, and human-readable summaries and descriptions. Request validation errors are returned as `HTTP 422`, missing-job errors as `HTTP 404`, invalid-phase conflicts as `HTTP 409`, and any other unexpected failures as `HTTP 500`. Interactive OpenAPI documentation is gen-

erated automatically with example values.

Method	URI	Success	Errors
POST	`/`	201	422, 500
GET	`/{job_id}`	200	404, 422, 500
PATCH	`/{job_id}`	200	404, 422, 500
DELETE	`/{job_id}`	204	404, 409, 422, 500
POST	`/{job_id}/process/{p...}`	202	404, 409, 422, 500
POST	`/{job_id}/end/{end_a...}`	200	404, 409, 422, 500
GET	`/`	200	422, 500

Table 5.3 API REST mapping for job management endpoints.

5.1.4 Spectral labelling domain

This is a new, standalone domain, previously absent in VO-CLOUD, dedicated exclusively to storing and managing annotations (both model-generated and human-provided) for LAMOST spectra within Active ML jobs. By decoupling labelling records from the spectra themselves, the system allows the same spectrum to be revisited in multiple jobs and labelled differently according to each job's configuration.

Capabilities. This domain manages the creation, retrieval, editing and bulk operations of spectrum labellings within Active ML jobs. In particular:

- *Initialize* (`POST /`): a single labelling (model prediction, user label/comment) for a spectrum within a job.
- *Retrieve* (`GET /{labelling_id}`): used whenever an existing labelling must be fetched for display or further editing.
- *Edit* (`PATCH /{labelling_id}`): called when a user updates the label or adds a comment for a specific spectrum.
- *Initialize batch* (`POST /batch/`): algorithm providing its model prediction (with optional user override) .
- *Edit batch* (`PATCH /batch/`): used by the client-side UI after manual review, once the user has annotated many spectra, the client submits all edited labellings in a single batch update.
- *List* (`GET /?job_id=...`): employed when fetching all labelling records for a given job, for result visualization or audit.

Layered architecture. The labelling domain is structured in four layers:

1. *Repository interface* (`src/labellings/repository.py`): declares contract methods to be implemented for concrete labelling storage.
2. *PostgreSQL repository* (`src/labellings/repositories/postgres.py`): a SQLAlchemy ORM is used to bulk-insert, bulk-update, and query database table, raising domain errors (e.g.,`LabellingAbsentJobError`).
3. *Service layer* (`src/labellings/service.py`): handles UUID generation, single/batch creation, retrieval, single/batch edits, and listing, delegating to the repository and enforcing batch-length validation.
4. *API router* (`src/labellings/api/rest.py`): FastAPI REST API exposing CRUD, batch, and list endpoints, mapping validation to `HTTP 422`, missing-job or missing-labelling to `HTTP 404`, batch mismatches to code `HTTP 400`, and others to `HTTP 500`.

Domain entity. A spectrum labelling in the system is captured by the `LabellingEntity`, which embeds both its unique identity and all relevant annotation metadata. Upon creation, a new `labelling_id` (UUID) is assigned and paired with the `job_id` of the Active ML job under which this spectrum was evaluated. The specific spectrum is referenced by its FITS filename (`spectrum_filename`), and each labelling records which partition of the workflow it belongs to, whether it was a low-confidence candidate (`CANDIDATE`), a performance-estimation sample (`PERFORMANCE_ESTIMATION`), or one selected for human review (`ORACLE`). The integer `sequence_iteration` denotes the zero-based iteration of the active-learning cycle when this spectrum was presented to the model. After the model produces its own label (`model_prediction`), a human oracle user may supply or correct a `user_label` (optionally) and leave an optional free-form `user_comment` for later investigation. Each field is rigorously validated against its centralized definition, ensuring that every labelling record is complete, consistently typed, and traceable back to both the originating job and the specific spectrum instance (see Code 5.13).

```
// src/labellings/entity.py
class LabellingEntity(BaseModel):
    labelling_id: UUID
    job_id: UUID
    spectrum_filename: str
    spectrum_set: SpectrumSetType
    sequence_iteration: int
    model_prediction: str | None = None
    user_label: str | None = None
    user_comment: str | None = None
```

Code listing 5.13 Domain entity for spectrum labelling metadata.

PostgreSQL repository implementation. The developed repository class `LabellingPostgresRepository` provides a concrete, PostgreSQL-backed implementation of the abstract `LabellingRepository` interface, enabling all CRUD and bulk operations against the labellings table via SQLAlchemy ORM. Individual creation and retrieval simply map DTOs and ORM instances to and from the database, with missing-record lookups raising appropriate error `LabellingNotExistError` and foreign-key violations (e.g., a non-existent `job_id`) yielding `LabellingAbsentJobError`. Bulk initialization and batch updates are optimized into single `INSERT` and `UPDATE` statements over the entire input list, ensuring high throughput even when persisting or modifying thousands of labellings at once; integrity checks in these operations similarly roll back and raise domain errors if any referenced job or labelling is invalid (see Code 5.14). Finally, listing by `job_id` orders results according to the configured `spectrum_set` priority (`ORACLE` → `PERFORMANCE_ESTIMATION` → `CANDIDATE`), returning fully validated `LabellingEntity` instances for downstream consumption.

```
// src/labellings/repositories/postgres.py
class LabellingPostgresRepository(LabellingRepository):
    ...
    async def create_batch(self, batch:
    ↪  list[LabellingCreateDTO]) -> None:
        # Prepare and execute bulk creation
        batch_creation_data = [
            dto.model_dump(exclude_none=True) for dto in batch
        ]
        query = insert(self.model)

        try:
            await self.session.execute(query,
            ↪  batch_creation_data)
            await self.session.commit()

        except IntegrityError as e:
            await self.session.rollback()
            raise LabellingAbsentJobError("Cannot create some
            ↪  labelling of the batch.") from e
    ...
```

Code listing 5.14 Repository implementation for PostgreSQL labellings manipulation.

Service usecases. The `LabellingService` encapsulates all business logic for creating, updating and retrieving spectrum labellings independently of how they are persisted. Upon initialization it generates a new UUID for each record and wraps the client payload in a `LabellingCreateDTO` before delegating to the repository's `create` or `create_batch` methods; in the batch case it ensures that each input payload is paired with a fresh identifier and then issues a single bulk-insert (see Code 5.15). Retrieval by `labelling_id` simply invokes the repository's `get_by_labelling_id` and transforms the returned `LabellingEntity` into a serializer model. Batch edits first verify equal lengths of ID and DTO arrays, raising `LabellingInvalidBatchError` on mismatch, then invoke `update_batch_by_labelling_ids` for an efficient multi-row update. Finally, listing all labellings for a given job delegates to `list_by_job_id` and packages the resulting entities into the appropriate list serializer. At every step the service remains agnostic to the underlying database implementation, translating domain models to DTOs and serializers and surfacing only domain-level exceptions for invalid operations or inputs.

```
// src/labellings/service.py
class LabellingService:
    ...
    async def initialize_labellings_batch(self, batch:
    ↪  list[LabellingInitializeDTO]) -> None:
        labelling_ids = [self._generate_labelling_id() for _ in
        ↪  batch]
        batch = [
            LabellingCreateDTO(labelling_id=labelling_id,
            ↪  **dto.model_dump())
            for labelling_id, dto in zip(labelling_ids, batch)
        ]

        await self.repository.create_batch(batch)
    ...
```

Code listing 5.15 Service usecases for labellings operations.

API router. The FastAPI REST router in `src/labellings/api/rest.py` exposes six endpoints for managing spectrum labelling records. Each endpoint is fully documented with summaries and descriptions. Input validation errors yield `HTTP 422`, any unexpected failures `HTTP 500`. When creating a single labelling, a missing associated job raises `HTTP 404`. Retrieving or editing a non-existent labelling also returns `HTTP 404`. Bulk initialization is intended for Worker processes at the end of an Active ML run and likewise returns `HTTP 404` if any job is absent. Bulk edits, used by the client after manual labelling during result visualization, will return `HTTP 400` on ID/payload length mismatch or `HTTP 404` if any labelling is missing.

Method	URI	Success	Errors
POST	`/`	201	404, 422, 500
GET	`/{labelling_id}`	200	404, 422, 500
PATCH	`/{labelling_id}`	200	404, 422, 500
POST	`/batch/`	204	404, 422, 500
PATCH	`/batch/`	204	400, 404, 422, 500
GET	`/`	200	422, 500

Table 5.4 API REST mapping for spectral labelling endpoints.

5.2 DB microservice

The **ML Job DB** microservice provides persistent storage for the Job management and Spectral labelling domains. It exposes no HTTP API itself, but

underlies the ML Job API microservice by housing two relational tables, `jobs` and `labellings`, linked by a foreign-key constraint. All ORM mappings use SQLAlchemy against PostgreSQL.

Configuration. The PostgreSQL connection settings are driven by the definded Pydantic `PostgresSettings` class, loading its values from environment variables or a configuration file (see Code 5.16). A custom validator transforms the base Data Source Name (DSN) into an `asyncpg` URI suitable for SQLAlchemy's async engine.

```
// ml-job-db/src/settings/storages/postgres.py
class PostgresSettings(BaseSettings):
    db_url: str
    debug: bool
    engine_connection_timeout: int
    session_autocommit: bool = False
    session_autoflush: bool = False
    session_expire_on_commit: bool = False

    @field_validator("db_url")
    def build_db_async_url(cls, db_url: str) -> str:
        # Convert standard Postgres DSN to asyncpg scheme
        db_dsn = PostgresDsn(db_url)
        return str(PostgresDsn.build(
            scheme="postgresql+asyncpg",
            hosts=db_dsn.hosts(),
            path=db_dsn.path.replace("/", ""),
            query=db_dsn.query,
            fragment=db_dsn.fragment,
        ))
```

Code listing 5.16 SQLAlchemy DB settings via PostgreSQL microservice.

Initialization. Upon startup, the service creates an async SQLAlchemy engine and session maker using the above settings, and exposes a dependency that yields scoped `AsyncSession` instances for repository use (see Code 5.17). Connection timeouts, echo logging, and session behavior are all controlled by `PostgresSettings`.

```
// ml-job-db/src/infrastructure/storages.py
# Create the async engine
postgres_async_engine = create_async_engine(
    url=postgres_settings.db_url,
    echo=postgres_settings.debug,
    connect_args={"timeout":
    ↪  postgres_settings.engine_connection_timeout},
    future=True,
)

# Configure the async session maker
postgres_async_session_maker = async_sessionmaker(
    bind=postgres_async_engine,
    autocommit=postgres_settings.session_autocommit,
    autoflush=postgres_settings.session_autoflush,
    expire_on_commit=postgres_settings.session_expire_on_commit,
    future=True,
)

async def get_postgres_async_session() ->
↪  AsyncGenerator[AsyncSession, None]:
    async with postgres_async_session_maker() as session:
        yield session
```

Code listing 5.17 SQLAlchemy DB engine and session setup via PostgreSQL microservice.

This setup ensures that every repository method receives a properly scoped `AsyncSession`, with connection pooling and transaction semantics governed by the centralized settings, and without duplicating configuration logic across the codebase.

5.2.1 Jobs table

The `jobs` table holds one row per ML job, recording its full lifecycle metadata. A B-tree PostgreSQL index on `created_at` supports fast pagination by creation time (newest first).

Column	Type
job_id	UUID pk
dir_path	VARCHAR(255)
type	ENUM
phase	ENUM
label	VARCHAR(50)
description	VARCHAR(255)
created_at	UTC TIMESTAMP WITH TIME ZONE
started_at	UTC TIMESTAMP WITH TIME ZONE
ended_at	UTC TIMESTAMP WITH TIME ZONE
execution_duration	FLOAT

Table 5.5 Schema for the `jobs` table.

Indexes and constraints.

- `PRIMARY KEY (job_id)` ensures unique job identifiers.
- `INDEX(created_at)` allows efficient ordering and pagination by creation time.

5.2.2 Labellings table

The `labellings` table stores one row per spectrum annotation. A foreign key `job_id` → `jobs.job_id` (ON DELETE CASCADE) ties each labelling to its job, and an index on `job_id` supports lightning-fast retrieval of all labellings for a given job, critical when displaying thousands of annotations.

Column	Type
labelling_id	UUID PRIMARY KEY
job_id	UUID REF jobs(job_id) ON DELETE CASCADE
spectrum_filename	VARCHAR(70)
spectrum_set	ENUM
sequence_iteration	INTEGER
model_prediction	VARCHAR(30)
user_label	VARCHAR(30)
user_comment	VARCHAR(50)

Table 5.6 Schema for the `labellings` table.

Indexes and constraints. The defined table foreign-key constraint between `labellings.job_id → jobs.job_id` is a cornerstone, binding the Job management and Spectral labelling domains at the database level. By declaring `ON DELETE CASCADE`, PostgreSQL automatically removes all annotations

whenever the parent job is deleted, ensuring that no orphaned labelling records remain and that data consistency is maintained without extra application logic.

Moreover, the B-tree index on `labellings.job_id` guarantees that querying all annotations belonging to a given job remains efficient even as the table grows to millions of rows. In practice, this means that a single query like the following: `SELECT … WHERE job_id = :id` can scan only the small subset of matching rows, returning thousands of annotations in milliseconds, a critical requirement when the Active ML front end must display or paginate large annotation sets in real time.

- `PRIMARY KEY (labelling_id)`: ensures unique annotation IDs.
- `FOREIGN KEY(job_id) REFERENCES jobs(job_id) ON DELETE CASCADE`: maintains referential integrity and automatically removes annotations when a job is deleted.
- `INDEX(job_id)`: enables rapid look-up of all labellings for a particular job, even at large scale.

Relationship. Each labelling row points to exactly one job via `job_id`. Conversely, a job may have zero or many associated labellings. This one-to-many relationship underpins the Active ML workflow: spectra are evaluated and annotated in the context of a job, and those annotations can be fetched or purged together with their parent job record.

5.3 Queue microservice

The **ML Job Queue** microservice provides the messaging backbone for asynchronous job dispatch and control. It leverages the Celery framework to configure, publish, and revoke tasks over an AMQP RabbitMQ broker, decoupling the ML Job API (producer) from downstream Worker processes (consumers).

Configuration. Celery's connection and behavior are driven entirely by a Pydantic settings class, loading its values from environment variables or a configuration file (see Code 5.15).

```
// ml-job-api/src/settings/clients/celery.py
class CelerySettings(BaseSettings):
    broker_url: str
    task_default_queue: str
    task_publish_retry: bool = True

    @field_validator("broker_url")
    def build_broker_url(cls, broker_url: str) -> str:
        # Normalize and validate the AMQP DSN
        return str(AmqpDsn(broker_url))
```

Code listing 5.18 Celery queue configuration via RabbitMQ microservice.

Initialization. At application startup, a global Celery client is created and configured from these settings. No manual connection code is needed: Celery handles the AMQP handshake, queue declaration, and retry logic automatically (see Code 5.19).

```
// ml-job-api/src/infrastructure/clients.py
# Create the global Celery client
celery_client = Celery()

# Load broker URL, default queue, and retry settings from
↪  CelerySettings
celery_client.config_from_object(
    obj=celery_settings,
    silent=False,
    force=True,
)
```

Code listing 5.19 Celery queue configuration at startup.

This single client producer instance knows the broker URL, default queue name, and retry policy, and is responsible for all job publication and revocation across the ML Job Manager ecosystem.

5.4 Worker microservice

The **ML Job Worker** microservice runs the actual data-preprocessing and active learning pipelines as Celery consumers. It subscribes to the RabbitMQ broker, executes domain jobs, writes logs and outputs to the shared filesystem, and reports status and labellings back to the ML Job API.

Configuration. All worker behavior: broker connection, queue name, acknowledgment, and retry policies, is driven by a Pydantic `AppSettings` class, sourcing values from environment variables or a config file (see Code 5.20). A validator ensures the AMQP DSN is normalized.

```
// ml-job-worker/src/settings/app.py
class AppSettings(BaseSettings):
    broker_url: str
    broker_connection_timeout: int
    task_default_queue: str
    task_acks_late: bool
    task_track_started: bool = True
    task_reject_on_worker_lost: bool = True
    broker_connection_retry_on_startup: bool = True
    broker_connection_max_retries: int | None = None
    worker_prefetch_multiplier: int = 1
    worker_send_task_events: bool = True

    @field_validator("broker_url")
    def build_broker_url(cls, v: str) -> str:
        return str(AmqpDsn(v))
```

Code listing 5.20 Celery settings in Worker microservice.

Initialization. On startup, a Celery consumer is created, configured from `AppSettings`, and pointed at the default queue. Job modules for both data preprocessing and active learning are registered so Celery can discover them (see Code 5.21). These job modules, and their domain logic, including Alisher Layik's `run` functions, reside alongside the Celery entrypoint in the Worker's domain directories `src/data_preprocessing` and `src/active_ml`.

```
// ml-job-worker/src/main.py
# Create the main Celery worker application instance
app = Celery()

# Include task modules so Celery can discover and register tasks
app.conf.include = [
    "src.active_ml",
    "src.data_preprocessing",
]

# Load Celery configuration directly from the Pydantic-based
↪  settings object
app.config_from_object(
    obj=app_settings,
    silent=False,  # Raise errors if config keys are missing
    force=True,    # Override any existing config values
)
```

Code listing 5.21 Celery initialization in Worker microservice.

This single Celery instance handles broker connections (with timeouts and retries), enqueues and acknowledges jobs according to the configured policies, and emits jobs events for monitoring.

Job modules. Two domain-specific Celery jobs are provided as native Celery `@shared_task` functions:

- **Data preprocessing** (`name=DATA_PREPROCESSING`): reads a JSON config, runs the spectral interpolation/scaling pipeline, writes HDF5 results and logs, and notifies the ML Job API of completion or error via `JobHttpxAPI`.
- **Active ML** (`name=ACTIVE_ML`): loads an active-learning configuration, executes either the zero or regular iteration, collects model predictions, calls `LabellingHttpxAPI.initialize_labellings_batch` to persist annotations, and signals job end to the ML Job API.

Because these modules live directly in the Worker's domain folders, adding new pipeline stages or replacing the `run` logic is as straightforward as dropping in a new job module. Each job binds its `task_id` to the ML job's UUID, measures `started_at` and `ended_at`, and handles manual aborts and unexpected exceptions identically.

HTTP clients. Two lightweight HTTP clients encapsulate API interactions:

- `JobHttpxAPI`: posts job end actions (`COMPLETE` or `ERROR`) with execution timestamps.
- `LabellingHttpxAPI`: submits bulk labelling records after an Active ML iteration.

They wrap an httpx.Client[5] configured with the ML Job API base URL, serializing response into JSON and handling HTTP response codes internally.

File utilities. Helpers for reading JSON configs and writing task logs reside in a shared module:

- `read_config_file(path)`: loads a JSON file into a Python dict (see Code 5.22).
- `write_log_file(path, log)`: writes a text log, overwriting existing content Worker (see Code 5.23).

```
// ml-job-worker/src/common/utils.py
def read_config_file(file_path: str) -> dict[str, Any]:
    with open(file_path, "r") as file_reader:
        raw_config = json.load(file_reader)

    return raw_config
```

Code listing 5.22 Utility config reading function in Worker microservice.

```
// ml-job-worker/src/common/utils.py
def get_error_log() -> str:
    return traceback.format_exc()

def write_log_file(file_path: str, log: str) -> None:
    with open(file_path, "w") as file_writer:
        file_writer.write(log)
```

Code listing 5.23 Utility logging functions in Worker microservice.

These functions are used uniformly by both job modules to load pipeline parameters and persist job-level logs back into the shared server filesystem.

[5] `https://www.python-httpx.org`

This Worker microservice thus cleanly separates job orchestration (Celery), domain logic (Alisher Layik's job pipelines), API callbacks (HTTP clients), and I/O utilities, making it straightforward to extend with new job types or pipeline stages in the future.

5.5 Microservice orchestration

The ML Job Manager suite, comprising the API, Worker, RabbitMQ broker, and PostgreSQL database, is orchestrated via **Docker Compose**. All four services are connected over a dedicated bridge network, share named volumes for persistence, and are configured by a single `.env` file. This arrangement guarantees isolation from the host OS, reproducible deployments, and consistent behavior across environments. An approximate orchestration topology looks like this:

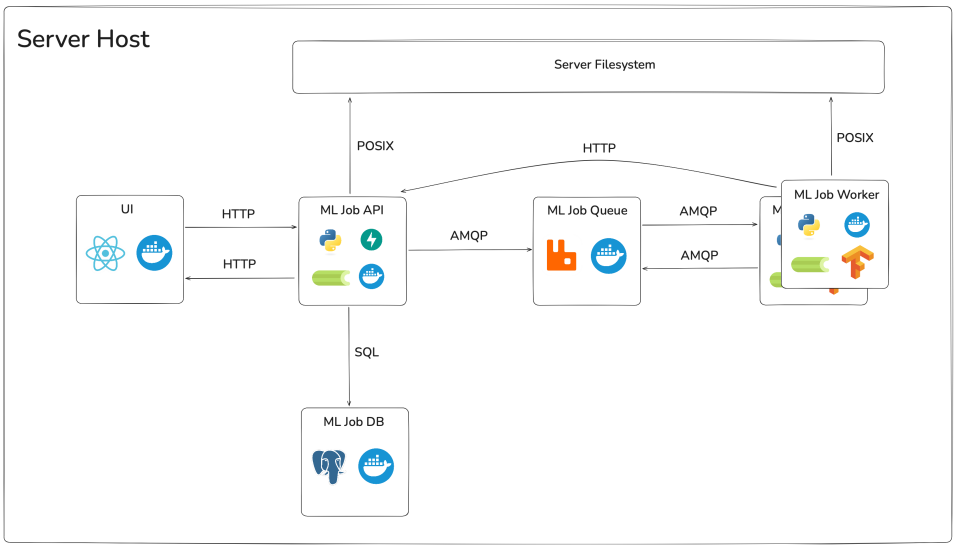

Figure 5.1 ML Job Manager orchestration.

Compose topology.

- **Network:** A custom bridge network `ml-job-network` enables internal containers' communication.
- **Volumes:**
  - `ml-job-db-data`: persists the PostgreSQL data directory.
  - `ml-job-queue-data`: persists RabbitMQ queue state.
- **Services:**

- `ml-job-api`: the FastAPI application, built from `./ml-job-api`, listening on `$API_PORT`, mounting host directories for file uploads (`/FILES`) and read-only spectra (`/SPECTRA`).
- `ml-job-worker`: the Celery consumer, built from `./ml-job-worker`, also mounting `/FILES` and read-only `/SPECTRA`, with GPU support via `runtime: nvidia`.
- `ml-job-queue`: RabbitMQ with management UI service hosted on the configured `$RABBITMQ_MANAGEMENT_PORT`.
- `ml-job-db`: PostgreSQL 17 on `5432`, using the `ml-job-db-data` volume.

Environment variables, such as database DSN, broker URL, file paths, and timeouts, are injected from `.env`, keeping configuration DRY and centralized.

API service entrypoint. When `ml-job-api` starts, its entrypoint script (see Code 5.24):

- Runs all pending Alembic database migrations (`alembic upgrade head`).
- Launches Gunicorn with Uvicorn server worker on inner Docker container port 8000.

```
#!/bin/sh

# Exit immediately if any command fails
set -e

# Run all pending database migrations
alembic upgrade head

# Launch the FastAPI application via Gunicorn with Uvicorn
↪  workers
exec gunicorn "src.main:app" \
    -k uvicorn.workers.UvicornWorker \
    --bind 0.0.0.0:8000 \
    --workers 1
```

Code listing 5.24 Entrypoint for ML Job API container.

This ensures the database schema is up-to-date before serving requests.

Worker service entrypoint. The `ml-job-worker` container's entrypoint script (see Code 5.25):

- Exits on any error (`set -e`).
- Starts a Celery worker consuming from the configured queue.
- Spawns two worker processes (`--concurrency=2`) and logs at INFO level.

```
#!/bin/sh

# Exit immediately if any command fails
set -e

# Launch the Celery worker:
#   -A: module:attribute pointing to the Celery app instance
#   --concurrency=2: number of worker processes
#   --loglevel=info: set logging verbosity
exec celery -A "src.main:app" worker \
    --concurrency=2 \
    --loglevel=info
```

Code listing 5.25 Entrypoint for ML Job Worker container.

Setting concurrency to 2 reflects production experience from VO-CLOUD, balancing throughput and resource usage for heavy preprocessing and training jobs.

Isolation and portability. By building dedicated Docker images and orchestrating them on a single network, the entire ML Job system, from the ML Job API to ML Job Worker, to database and broker, runs in self-contained containers. This guarantees:

- *Environment parity:* identical behavior in development, staging, and production.
- *Dependency isolation:* no conflicts with host system libraries or tools.
- *Ease of scaling:* services can be replicated, updated, or replaced independently.

In sum, Docker Compose serves as the orchestration backbone, wiring together all microservices with minimal operational overhead.

Chapter 6

Future improvements & new workflow example

As it stands, the ML Job Manager is a fully modular, containerized system built on modern technologies (FastAPI, SQLAlchemy, Celery, Docker). Its four microservices communicate over well-defined HTTP or AMQP boundaries, and share no inline UI or host-specific code. This architecture makes the system trivial to deploy on new infrastructure (betelgeuse and antares clusters) simply by adjusting environment variables and pointing Docker Compose at the target hosts.

6.1 Planned enhancements

The following key features are slated for future release:

- **Authentication & user management:** a fifth domain, "Users & Auth," will be introduced to secure all endpoints, issue JWTs, and associate user context with job and labelling records, thereby enabling Internet-facing deployments.
- **Pluggable worker domains:** additional Celery task modules may be simply dropped into the Worker's `src/` directories alongside the existing `data_preprocessing` and `active_ml` packages. Celery's auto-discovery mechanism will register any new `@shared_task` definitions without modification to the core application.

6.2 New data-preprocessing workflow

To illustrate the ease of spinning up and running a new job in the current system, consider a simple test of the data-preprocessing pipeline.

1. Creating a new job via the OpenAPI UI. Interactive OpenAPI documentation (available at `http://localhost:10000/docs`) is used to issue a `POST /jobs/` request, selecting the `DATA_PREPROCESSING` type and providing a short label. This call returns a newly generated `job_id` and the corresponding directory path under `/FILES`, confirming that the job record has been initialized (see Figure 6.1).

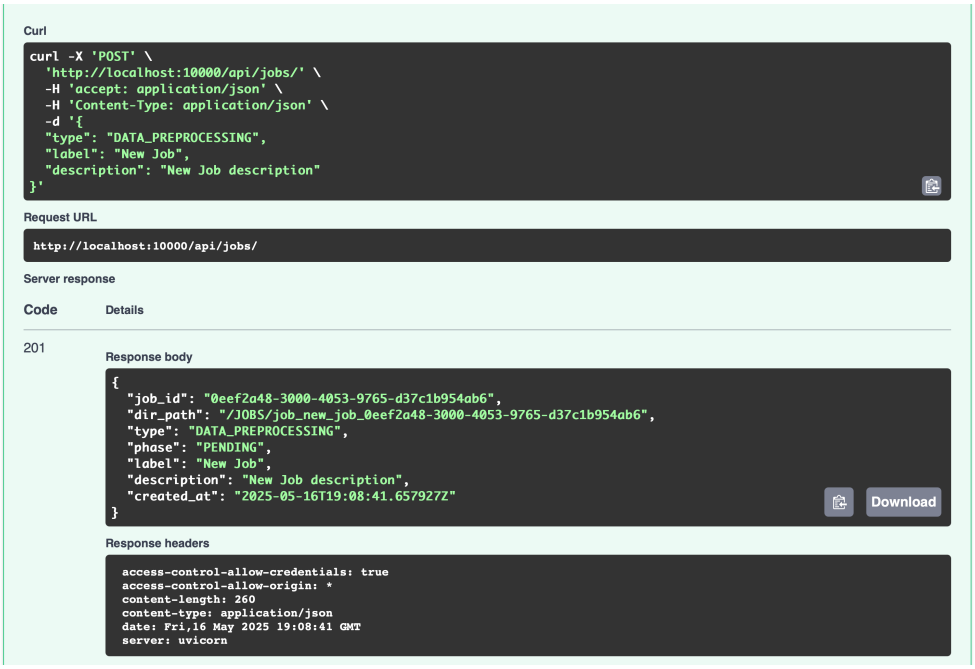

Figure 6.1 Data preprocessing job worklfow - creating new job.

2. Preparing the job configuration. In the filesystem (mounted at `/FILES`), a `config.json` is created inside the job's folder. Its contents adhere to the schema described in the previous Chapter, specifying input data directory, output file name, and any preprocessing parameters. This ensures that when the Worker runs, it will have a valid pipeline configuration (see Figure 6.2).

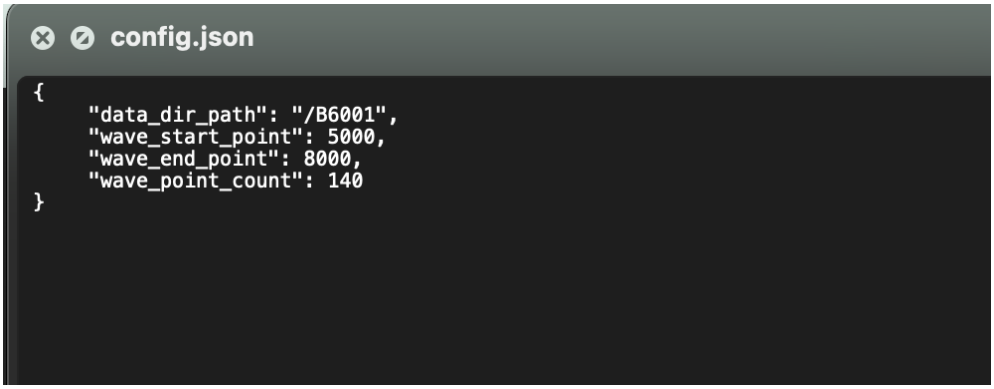

Figure 6.2 Data preprocessing job worklfow - creating new config.

3. Verifying the job directory. A quick check of the job's directory confirms the presence of `config.json` and an empty `log.txt` / `result.h5` placeholders. This step validates that the ML Job API handled the workspace correctly and that the configuration file is in place (see Figure 6.3).

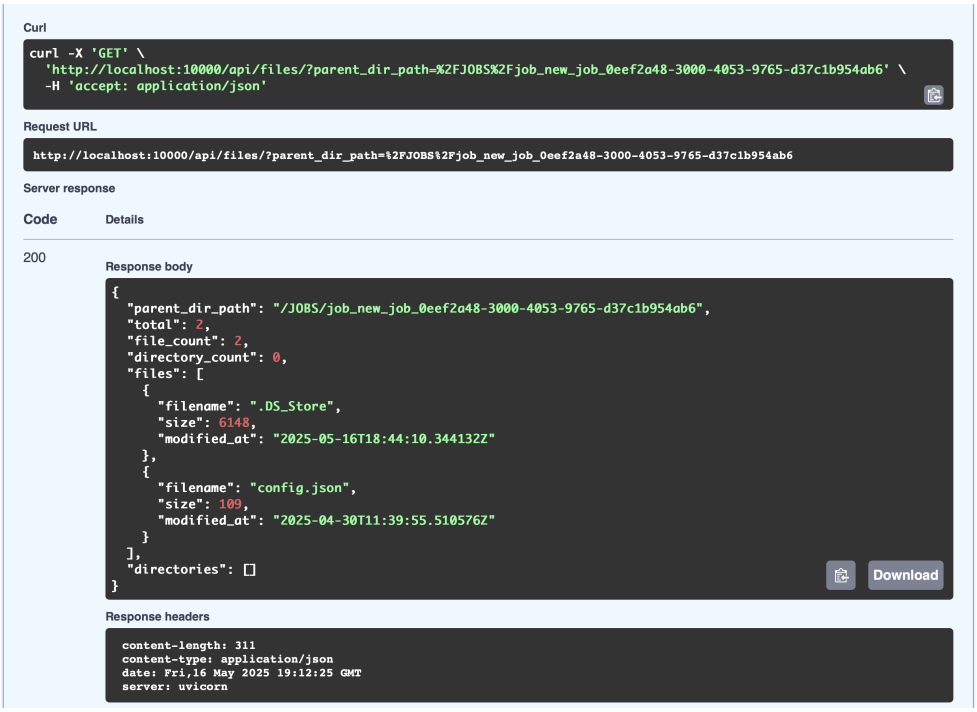

Figure 6.3 Data preprocessing job worklfow - checking new job.

4. Enqueuing the preprocessing task. Using the OpenAPI UI again, a `POST /jobs/{job_id}/process/RUN` request is issued. The API enqueues a Celery job with the `job_id` as its `task_id` and transitions the job's phase to `PROCESSING`. A `202` Accepted response confirms that the Worker has received

the job (see Figure 6.4).

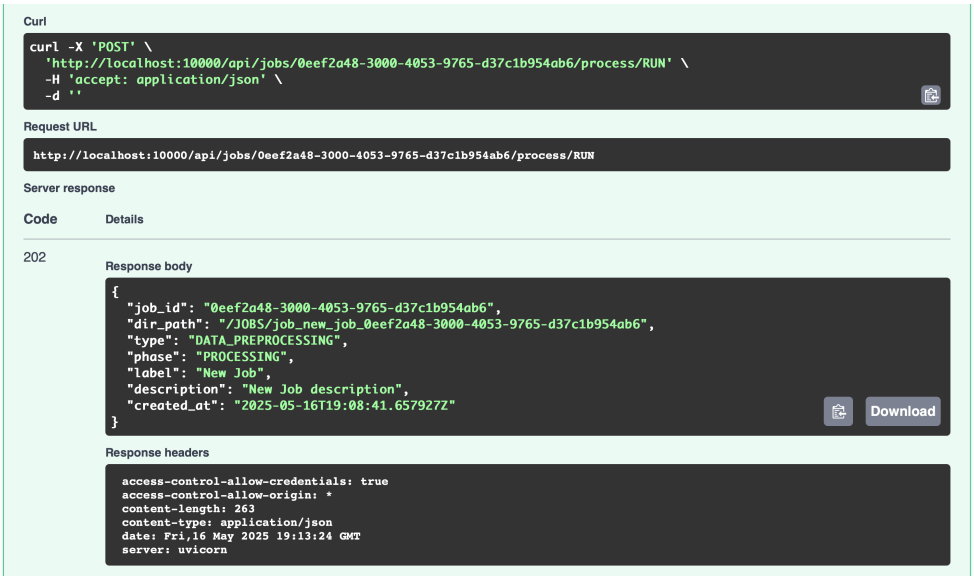

Figure 6.4 Data preprocessing job worklfow - running new job.

5. Monitoring job completion. Polling the job status via API path `GET /jobs/{job_id}` shows the phase change to `COMPLETED` and populated `started_at`, `ended_at`, and `execution_duration` fields. This indicates that the Worker finished the pipeline without errors (see Figure 6.5).

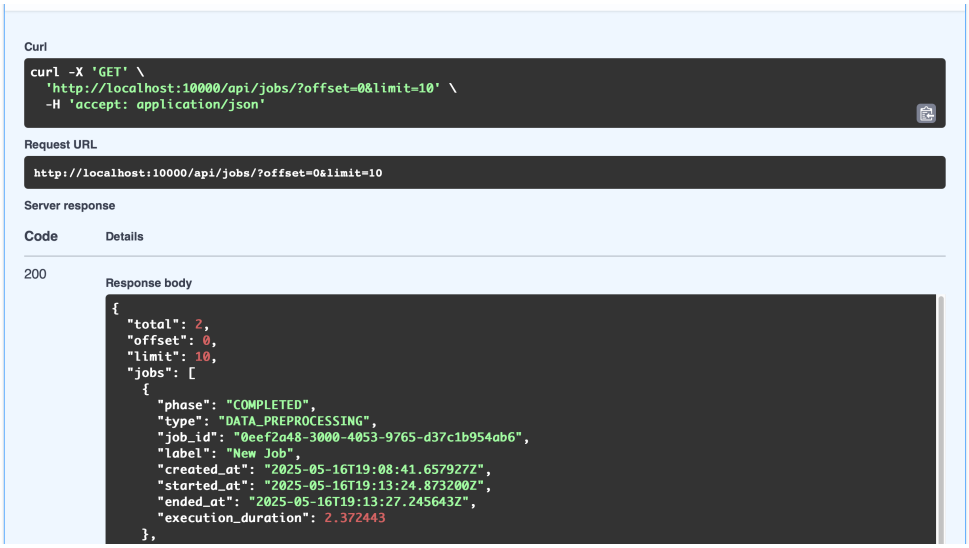

Figure 6.5 Data preprocessing job worklfow - checking new job completion.

6. Inspecting the job outputs. Finally, the contents of the job directory

are listed (or browsed via the Files API). The `result.h5` file and the updated `log.txt` confirm that the preprocessing ran successfully and produced the expected artifacts, completing the end-to-end test workflow (see Figure 6.6).

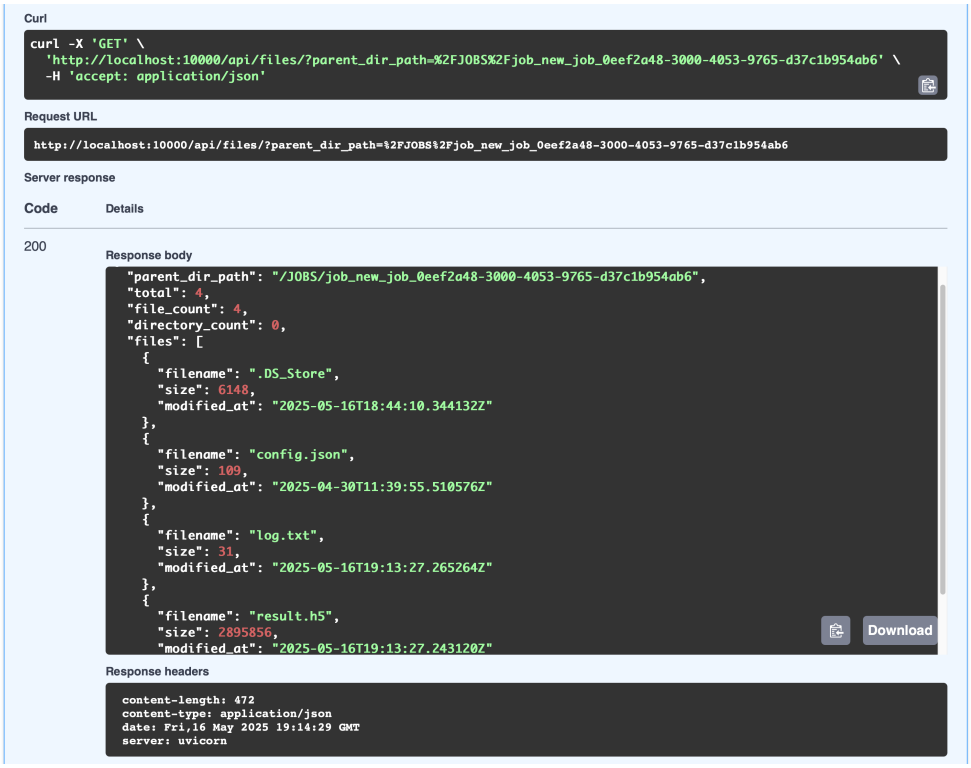

Figure 6.6 Data preprocessing job worklfow - checking new job output.

Chapter 7

Conclusion

A comprehensive analysis of the legacy VO-CLOUD platform was performed, revealing a tightly coupled Java EE monolith, server-side UI rendering, partial containerization, and outdated protocol implementations. These findings specified a complete rewrite with precise functional and non-functional requirements: a JSON-first REST API, UWS-inspired job phases, modular job types, FIFO task queuing, and fully containerized deployment.

To meet these requirements, modern technologies were selected according to best practices. FastAPI application now powers the back-end API, Celery with RabbitMQ manages asynchronous job dispatch, and PostgreSQL provides reliable metadata storage. Docker containers and Docker Compose orchestrate all services. This combination establishes a lightweight microservices architecture with clear separation of concerns between API, queue, Workers, and data storage.

The machine learning and data preprocessing modules from Alisher Laiyk's work have been integrated and adapted into this new infrastructure. His front-end application was connected via a shared Docker Compose configuration, aligning with the designed zones of responsibility.

An initial implementation, ML Job Manager, realizes this architecture. It converts raw LAMOST FITS files into uniform wavelength grids, drives active-learning loops with user-in-the-loop labeling, and stores results in a PostgreSQL database. Job definitions, logs, and metrics flow through FastAPI endpoints and Celery Workers, forming the skeleton of a scalable, maintainable system.

Looking ahead, the result is a unique platform tailor-made for large-scale astronomical spectra analysis and interactive ML workflows.

The original VO-CLOUD core has been thoroughly reengineered into a container-first, microservices-based system that meets today's portability, scalability, and developer productivity standards. All project goals have been met, and the system is now ready for future enhancements.

Bibliography

1. KOZA, Jakub. *Design and implementation of a distributed platform for data mining of big astronomical spectra archives* [online]. 2015. [visited on 2025-04-18]. Available from: `https://dspace.cvut.cz/bitstream/handle/10467/63145/F8-BP-2015-Koza-Jakub-thesis.pdf`. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology.
2. KOZA, Jakub. *Interactive Cloud-Based Platform for Parallelized Machine Learning of Astronomical Big Data* [online]. 2017. [visited on 2025-04-18]. Available from: `https://dspace.cvut.cz/bitstream/handle/10467/69141/F8-DP-2017-Koza-Jakub-thesis.pdf`. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology.
3. MAZEL, Tomáš. *Cloud-Based Platform for Active Learning of Astronomical Spectra* [online]. 2020. [visited on 2025-04-18]. Available from: `https://dspace.cvut.cz/bitstream/handle/10467/88254/F8-BP-2020-Mazel-Tomas-thesis.pdf`. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology.
4. ARVISET, Christophe; GAUDET, Severin. *IVOA Architecture Version 1.0* [online]. 2010. [visited on 2025-04-18]. Available from: `https://ivoa.net/documents/Notes/IVOAArchitecture/20101123/IVOAArchitecture-1.0-20101123.pdf`.
5. HANISCH, Robert; QUINN, Peter. *The International Virtual Observatory* [online]. 2003. [visited on 2025-04-18]. Available from: `http://www.ivoa.net/about/TheIVOA.pdf`.
6. GROUP, IVOA Technical Coordination. *IVOA Architecture Level 0* [online]. 2010. [visited on 2025-04-18]. Available from: `https://www.ivoa.net/documents/Notes/IVOAArchitecture/20101123/IVOAArchitecture-1.0-20101123.pdf`.

7. TODY, Doug; DOLENSKY, Markus; MCDOWELL, Jonathan; BONNAREL, Francois; BUDAVARI, Tamas; BUSKO, Ivo; MICOL, Alberto; OSUNA, Pedro; SALGADO, Jesus; SKODA, Petr, et al. Simple spectral access protocol version 1.1. *IVOA Recommendation* [online]. 2012, vol. 10 [visited on 2025-04-18]. Available from: https://ivoa.net/documents/SSA/20120210/REC-SSA-1.1-20120210.pdf.

8. DOWLER, Patrick; BONNAREL, François; MICHEL, Laurent; DEMLEITNER, Markus. IVOA DataLink Version 1.0. *IVOA Recommendation* [online]. 2015, vol. 17 [visited on 2025-04-18]. Available from: https://ivoa.net/documents/DataLink/20231215/REC-DataLink-1.1.pdf.

9. GROUP, IVOA Technical Coordination. *Role of SSA within the overall IVOA architecture* [online]. 2012. [visited on 2025-04-18]. Available from: https://www.ivoa.net/documents/SSA/20120210/REC-SSA-1.1-20120210.pdf.

10. GROSBØL, Preben. *Databases & On-line Data in Astronomy*. The FITS data format. Springer, 1991. ISBN 978-94-010-5433-1. Available also from: https://link.springer.com/chapter/10.1007/978-94-011-3250-3_25.

11. OCHSENBEIN, François; WILLIAMS, Roy; DAVENHALL, Clive; DEMLEITNER, Markus; DONALDSON, Tom; DURAND, Daniel; FERNIQUE, Pierre; GIARETTA, David; HANISCH, Robert, et al. IVOA VOTable Format Definition Version 1.4. *IVOA Recommendation* [online]. 2019 [visited on 2025-04-18]. Available from: https://ivoa.net/documents/VOTable/20191021/REC-VOTable-1.4-20191021.pdf.

12. GROUP, IVOA Technical Coordination. *VOTable in the IVOA Architecture* [online]. 2019. [visited on 2025-04-18]. Available from: https://www.ivoa.net/documents/VOTable/20191021/REC-VOTable-1.4-20191021.pdf.

13. HARRISON, Paul; RIXON, Guy. IVOA Universal Worker Service Pattern Version 1.1. *IVOA Recommendation* [online]. 2016 [visited on 2025-04-18]. Available from: https://ivoa.net/documents/UWS/20161024/REC-UWS-1.1-20161024.pdf.

14. GROUP, IVOA Technical Coordination. *UWS in the IVOA Architecture* [online]. 2016. [visited on 2025-04-18]. Available from: https://www.ivoa.net/documents/UWS/20161024/REC-UWS-1.1-20161024.pdf.

15. GROUP, IVOA Technical Coordination. *Objects within a UWS* [online]. 2016. [visited on 2025-04-18]. Available from: https://www.ivoa.net/documents/UWS/20161024/REC-UWS-1.1-20161024.pdf.

16. GROUP, IVOA Technical Coordination. *UWS execution phases* [online]. 2016. [visited on 2025-04-18]. Available from: https://www.ivoa.net/documents/UWS/20161024/REC-UWS-1.1-20161024.pdf.

17. RUSIA, Vikalp. *Master Worker Architecture: Unleashing the Power of Parallel Processing* [online]. Medium, 2023. [visited on 2025-04-18]. Available from: https://medium.com/@vikalprusia/master-worker-architecture-unleashing-the-power-of-parallel-processing-eed241da30a.

18. RUSIA, Vikalp. *Master node assigning tasks to worker nodes* [online]. 2023. [visited on 2025-04-18]. Available from: https://miro.medium.com/v2/resize:fit:1400/format:webp/1*fx1xvD2LF7bwc5qSaRn4yw.png.

19. MAZEL, Tomáš. *VO-CLOUD deployment diagram modified* [online]. 2020. [visited on 2025-04-18]. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88254/F8-BP-2020-Mazel-Tomas-thesis.pdf.

20. MAZEL, Tomáš. *Starting active learning job in VO-CLOUD environment* [online]. 2020. [visited on 2025-04-18]. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88254/F8-BP-2020-Mazel-Tomas-thesis.pdf.

21. MAZEL, Tomáš. *Specifying new active learning job* [online]. 2020. [visited on 2025-04-18]. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88254/F8-BP-2020-Mazel-Tomas-thesis.pdf.

22. MAZEL, Tomáš. *Simple JSON configuration to create initial training set* [online]. 2020. [visited on 2025-04-18]. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88254/F8-BP-2020-Mazel-Tomas-thesis.pdf.

23. MAZEL, Tomáš. *List of jobs - going to details* [online]. 2020. [visited on 2025-04-18]. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88254/F8-BP-2020-Mazel-Tomas-thesis.pdf.

24. MAZEL, Tomáš. *Viewing both preprocessed and original spectra* [online]. 2020. [visited on 2025-04-18]. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88254/F8-BP-2020-Mazel-Tomas-thesis.pdf.

25. DEVELOPERS, Xarray core. *Pangeo A community for open, reproducible, scalable geoscience.* [online]. Pangeo, 2025. [visited on 2025-04-18]. Available from: https://pangeo.io.

26. JONES, Max. *Pangeo ML: Open Source Tools and Pipelines for Scalable Machine Learning Using NASA Earth Observation Data* [online]. NASA, 2025. [visited on 2025-04-18]. Available from: https://www.earthdata.nasa.gov/about/competitive-programs/access/pangeo-ml.

27. MUNTEANU, Alexandru. *The cloud native architecture of Pangeo 3* [online]. 2024. [visited on 2025-04-18]. Available from: https://www.researchgate.net/publication/384487656/figure/fig2/AS:11431281281399761@1727855091674/The-cloud-native-architecture-of-Pangeo-3.png.

28. IDIES.JHU.EDU. *SciServer* [online]. The Institute for Data-Intensive Engineering and Science, 2024. [visited on 2025-04-18]. Available from: https://www.idies.jhu.edu/what-we-offer/sciserver/.

29. IDIES.JHU.EDU. *SciServer Compute Bringing Analysis Close to the Data* [online]. The Institute for Data-Intensive Engineering and Science, 2016. [visited on 2025-04-18]. Available from: https://www.idies.jhu.edu/sciserver-compute-bringing-analysis-close-to-the-data/.

30. TAGHIZADEH-POPP, Manuchehr; KIM, Jai Won; LEMSON, Gerard; MEDVEDEV, Dmitry; RADDICK, M Jordan; SZALAY, Alexander S; THAKAR, Aniruddha R; BOOKER, Joseph; CHHETRI, Camy; DOBOS, Laszlo, et al. SciServer: a science platform for astronomy and beyond. *Astronomy and Computing* [online]. 2020, vol. 33, p. 100412 [visited on 2025-04-18]. Available from: https://www.sciencedirect.com/science/article/abs/pii/S2213133720300664.

31. SCISERVER, IDIES Data Center. *SciServerSystem* [online]. 2016. [visited on 2025-04-18]. Available from: https://dev-idies-jhu.pantheonsite.io/wp-content/uploads/2016/12/SciServerSystem.png.

32. DATALABS.ESA.INT. *ESA Datalabs* [online]. European Space Agency, 2025. [visited on 2025-04-18]. Available from: https://datalabs.esa.int.

33. COSMOS.ESA.INT. *CREATING A COLLABORATIVE SCIENCE PLATFORM – ESA DATALABS:* [online]. European Space Agency, 2024. [visited on 2025-04-18]. Available from: https://www.cosmos.esa.int/web/data-science/edl.

34. NAVARRO, Vicente; RIO, Sara; DIEGO, Miguel; BASO, Daniel; MARQUES, André; ZLOBIN, Veniamin; RAMOS, Nuno; PEREIRA, Alfredo; REITSAK, Aapo; MARSEILLE, Matthieu; MESSAGER, Fabien; LÓPEZ, Marcos; ALVAREZ, Roly; ARVISET, Christophe. ESA DATALABS: TOWARDS A COLLABORATIVE E-SCIENCE PLATFORM FOR ESA. *Astronomy and Computing* [online]. 2021 [visited on 2025-04-18]. Available from DOI: 10.13140/RG.2.2.36173.56807.

35. NAVARRO, Vicente; RIO, Sara del; DIEGO, Miguel Angel; LOPEZ-CANIEGO, Marcos; MARINIC, Filip; KRUK, Sandor; REERINK, Jan; ARVISET, Christophe. ESA Datalabs: Digital Innovation in Space Science. In: *Space Data Management*. Springer, 2024, pp. 1–13.

36. NAVARRO, Vicente; RIO, Sara del; ANGEL DIEGO, Miguel, et al. *ESA Datalabs: Digital Innovation in Space Science | SpringerLink* [online]. 2024. [visited on 2025-04-18]. Available from: https://media.springernature.com/lw685/springer-static/image/chp%3A10.1007%2F978-981-97-0041-7_1/MediaObjects/605372_1_En_1_Fig2_HTML.png.

37. VELEPUCHA, Victor; FLORES, Pamela. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. *IEEE Access*. 2023, vol. 11, pp. 88339–88358. Available from DOI: 10.1109/ACCESS.2023.3305687.

38. , taufiq hidayat taufiq. *Microservice Versus Monolithic Architecture. What Are They?* [online]. 2020. [visited on 2025-04-18]. Available from: https://www.cloudflight.io/app/uploads/2024/10/frame-1-4.png.

39. ROMERO, Isaac. A Multi-Cloud Architecture for Distributed Task Processing Using Celery, Docker, and Cloud Services. *International Journal of AI, BigData, Computational and Management Studies*. 2025, vol. 1, no. 2, pp. 1–10. Available from DOI: 10.63282/3050-9416.IJAIBDCMS-V1I2P101.

40. ASTHA. *Message Queue* [online]. 2024. [visited on 2025-04-18]. Available from: https://miro.medium.com/v2/resize:fit:1400/0*L3fPUOtYddvNE3BY.png.

41. WATADA, Junzo; ROY, Arunava; KADIKAR, Ruturaj; PHAM, Hoang; XU, Bing. Emerging Trends, Techniques and Open Issues of Containerization: A Review. *IEEE Access*. 2019, vol. 7, pp. 152443–152472. Available from DOI: 10.1109/ACCESS.2019.2945930.

42. TANNER, Matt. *How does containerization software work?* [online]. 2024. [visited on 2025-04-18]. Available from: https://vfunction.com/wp-content/uploads/2024/04/containerization-software-how-does-it-work-1024x683.png.

43. SORVISTO, Dayne. Infrastructure for MLOps. In: *MLOps Lifecycle Toolkit: A Software Engineering Roadmap for Designing, Deploying, and Scaling Stochastic Systems*. Springer, 2023, pp. 103–138.

44. CUBUKCU, Umur; ERDOGAN, Ozgun; PATHAK, Sumedh; SANNAKKAYALA, Sudhakar; SLOT, Marco. Citus: Distributed PostgreSQL for Data-Intensive Applications. In: *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, 2021, pp. 2490–2502. ISBN 9781450383431. Available from DOI: 10.1145/3448016.3457551.

45. ĆATOVIĆ, Amar; BUZAĐIJA, Nevzudin; LEMES, Samir. Microservice development using RabbitMQ message broker. *Science, engineering and technology*. 2022, vol. 2, no. 1, pp. 30–37.

46. PYTHON-POETRY.ORG. *Introduction | Documentation | Poetry - Python dependency management and packaging made easy* [online]. Poetry, 2025. [visited on 2025-04-18]. Available from: https://python-poetry.org/docs/.

47. TUYCHIEV, Bex. *Python Poetry: Modern And Efficient Python Environment And Dependency Management* [online]. 2024. [visited on 2025-04-18]. Available from: https://www.datacamp.com/tutorial/python-poetry.

48. FASTAPI.TIANGOLO.COM. *FastAPI* [online]. 2025. [visited on 2025-04-18]. Available from: https://fastapi.tiangolo.com/.

49. VERBINA, Evgenia. *Which Is the Best Python Web Framework: Django, Flask, or FastAPI?* [online]. 2025. [visited on 2025-04-18]. Available from: https://www.geeksforgeeks.org/fastapi-introduction/.

50. DOCS.PYDANTIC.DEV. *Welcome to Pydantic - Pydantic* [online]. Pydantic, 2025. [visited on 2025-04-18]. Available from: https://docs.pydantic.dev/latest/.

51. DOCS.SQLALCHEMY.ORG. *Overview - SQLAlchemy 2.0 Documentation* [online]. SQLAlchemy, 2025. [visited on 2025-04-18]. Available from: https://docs.sqlalchemy.org/en/20/intro.html.

52. ALEMBIC.SQLALCHEMY.ORG. *Tutorial - Alembic 1.15.2 documentation* [online]. 2025. [visited on 2025-04-18]. Available from: https://alembic.sqlalchemy.org/en/latest/tutorial.html.

53. DOCS.CELERYQ.DEV. *Introduction to Celery - Celery 5.5.2 documentation* [online]. Celery, 2025. [visited on 2025-04-18]. Available from: https://docs.celeryq.dev/en/latest/getting-started/introduction.html.

54. THANH VO, Minh. *Deploy ML models as A Task Queue Distributed Service with Python and Celery* [online]. 2022. [visited on 2025-04-18]. Available from: https://medium.com/@vmthanhit/deploy-ml-models-as-a-task-queue-distributed-service-with-python-and-celery-5a5bae82240f.

55. PYPI.ORG. *aiofiles: file support for asyncio* [online]. Python Software Foundation, 2024. [visited on 2025-04-18]. Available from: https://pypi.org/project/aiofiles/.

56. DOCS.PYTHON.ORG. *os — Miscellaneous operating system interfaces* [online]. Python Software Foundation, 2025. [visited on 2025-04-18]. Available from: https://docs.python.org/3/library/os.html.

57. DOCS.ATROPY.ORG. *astropy: A Community Python Library for Astronomy* [online]. Astropy Developers, 2025. [visited on 2025-04-18]. Available from: https://docs.astropy.org/en/stable/index.html.

58. CONTRIBUTERS, Andrew Collette. *HDF5 for Python* [online]. Andrew Collette and contributors, 2025. [visited on 2025-04-18]. Available from: https://www.h5py.org.

59. DOCS.DOCKER.COM. *What is Docker?* [online]. Docker Inc., 2025. [visited on 2025-04-18]. Available from: https://docs.docker.com/get-started/docker-overview/.

60. DOCS.DOCKER.COM. *Docker Compose* [online]. Docker Inc., 2025. [visited on 2025-04-18]. Available from: https://docs.docker.com/compose/.

61. BLACUS, Victor. The electromagnetic wave spectrum. *PHYSICS ACROSS OCEANOGRAPHY: FLUID MECHANICS AND WAVES* [online]. 2020 [visited on 2025-04-18]. Available from: https://uw.pressbooks.pub/ocean285/chapter/electromagnetic-spectrum/.