D3.2 Production Release

EOSC Data Commons – First Software Release

30/06/2026

Abstract

This deliverable presents the first production release of the EOSC Data Commons platform. EOSC Data Commons contributes to establishing EOSC as the European Research Commons, a global trusted ecosystem that provides seamless access to high-quality interoperable research outputs and services, enabling European researchers to collaborate more easily, be more productive and achieve higher levels of excellence. A documentation of the architecture, deployed software components, infrastructure operation, observability and environments is provided in this deliverable.

Funded by the European Union

EOSC Data Commons receives funding from the European Union's Horizon Europe research and innovation programme under grant agreement No. 101188179.
This Project has received support from the EOSC EU Node.

Document Description

EOSC Data Commons D3.2 First Software Release v1.0			
Work Package number: 3			
Due date	30/06/2026	**Actual delivery date:**	30/06/2026
Nature of document	Report	**Dissemination level**	Public
Document Status	Under EC Review	**Version**	1.0
Lead Partner	UST/AGH		
Authors	Draženko Celjak (SRCE), Reggie Cushing (DANS), Vincent Emonet (SIB), Mateo Hitl (SRCE), Michał Kołomański (UST/AGH), Aleš Křenek (CESNET), Chuan Ping Lee (SWITCH), Lukasz Opiola (UST/AGH), Marco Rorro (EGI), Tobias Schweizer (SWITCH), Ritwik Shanker (FZJ), Jusong Yu (ETH), Wojtek Ziajka (UST/AGH), Sebastian Sigloch (SWITCH)		
Reviewers	Sebastián Luna-Valero (EGI), Ilaria Fava (EGI)		
Approved by	Enol Fernandez (EGI)		
Document link	https://documents.egi.eu/document/4238		
DOI	https://zenodo.org/records/21068518		
Keyword	EOSC Data Commons, Metadata Warehouse, Matchmaker, Data Player Virtual Research Environments, Metadata Harvesting, Data Discovery, Data Interoperability		

Revision History

Issue	Date	Description	Author/Reviewer
V 0.1	27/05/2026	ToC	Michał Kołomański (UST/AGH)
V 0.2	22/06/2026	Ready for review version	Draženko Celjak (SRCE), Reggie Cushing (DANS), Vincent Emone (SIB), Mateo Hitl (SRCE), Michał Kołomański (UST/AGH), Aleš Křenek (CESNET), Chuan Ping Lee (SWITCH), Lukasz Opiola (UST/AGH), Marco Rorro (EGI), Tobias Schweizer (SWITCH), Ritwik Shanker (FZJ), Jusong Yu (ETH), Wojtek Ziajka (UST/AGH), Sebastian Sigloch (SWITCH)
V 0.3	25/06/2026	Revised version	Sebastián Luna-Valero (EGI), Ilaria Fava (EGI)
V 1.0	**30/06/2026**	**Final**	Michał Kołomański (UST/AGH)

Table of contents

Figures

Tables

Executive Summary

This deliverable documents the first production release of the EOSC Data Commons platform, a key outcome of the EOSC Data Commons project. The platform provides an integrated environment that enables researchers to discover, access, combine, analyse, and reuse research data across heterogeneous repositories and computing infrastructures. Release 1 establishes the technical foundation of the platform and delivers the first operational deployment of its core services.

The EOSC Data Commons platform is organised around two services: the EOSC Matchmaker and the EOSC Data Player. The Matchmaker supports metadata aggregation, dataset discovery, and tool matchmaking, while the Data Player enables the execution of analytical tools and workflows in distributed Virtual Research Environments (VREs). Together, these services support an end-to-end research workflow, allowing users to move seamlessly from data discovery to computational analysis.

Release 1 delivers a production-ready implementation of the EOSC Data Commons platform architecture. The release integrates the software required to support metadata aggregation of 10 repositories across 26 harvesting endpoints, dataset discovery, matchmaking, orchestration, and operational monitoring across the EOSC Data Commons ecosystem. The production deployment includes frontend and backend services, metadata ingestion pipeline, OpenSearch and large language model-based (LLM-based) datasets discovery, files and tools matching, and operational infrastructure supporting development and production environments. All the software components and multiple supporting databases have been deployed and integrated into a common operational environment.

The deliverable also documents the production deployment, including the underlying infrastructure, operational environments, monitoring and observability services, and security mechanisms. The platform is deployed using containerised services and Infrastructure-as-Code practices, with monitoring provided through Prometheus, Grafana, Loki, and Alertmanager, and user authentication enabled through EGI Check-in.

Overall, Release 1 demonstrates the successful integration of the EOSC Data Commons architecture into a functioning production platform. It establishes the foundation for future service expansion, additional repository integrations, enhanced AI-assisted discovery capabilities, broader tool interoperability, and increased automation of research workflows. Future releases will focus on increasing the maturity of individual components, onboarding additional repositories and VREs,

improving metadata quality and search relevance, and expanding the operational capabilities of the platform to support a growing EOSC user community.

1. Introduction

The purpose of this deliverable is to formally describe Release 1 of the EOSC Data Commons platform, including the deployed software components, architecture and infrastructure setup. The EOSC Data Commons platform integrates metadata harvesting, search, tool discovery and tool execution into a coherent set of services supporting the objectives of the project. This first release represents a major milestone in the project, providing the first production-ready implementation of the core services. Deployment characteristics, infrastructure environments and observability are also provided. The document is structured as follows:

Chapter 2 presents an overview of the EOSC Data Commons architecture and describes the relationship between the major platform services and components.

Chapter 3 describes the EOSC Matchmaker, including the Metadata Warehouse, search services, user interface, tool registry, metadata harvesting services, coordinator components, and supporting databases.

Chapter 4 presents the EOSC Data Player and its Dispatcher component, which enables the execution of tools and workflows across supported VREs and infrastructures.

Chapter 5 describes the deployment and operational aspects of the platform, including development and production environments, observability, monitoring, security, and reliability measures.

Finally, chapter 6 summarises the current results of the first production release and outlines future development plans and platform evolution.

2. Architecture Overview

EOSC Data Commons delivers a federated framework enabling AI-enhanced discovery, access, analysis, and reuse of research data across distributed repositories and compute environments. It's composed of two core services: the EOSC Matchmaker, which integrates heterogeneous metadata, supports discovery and pairs datasets with analytical tools; and the EOSC Data Player, which orchestrates data access, provisioning, and workflow execution across the compute continuum. A detailed description of the services and their components is available in D3.1 (EOSC Data Commons architecture, functional specifications no.1). In this section we provide a short overview of the components and how they relate together.

15 software components and five databases are developed for EOSC Data Commons Release 1 as shown in the overview diagram in Figure 1.

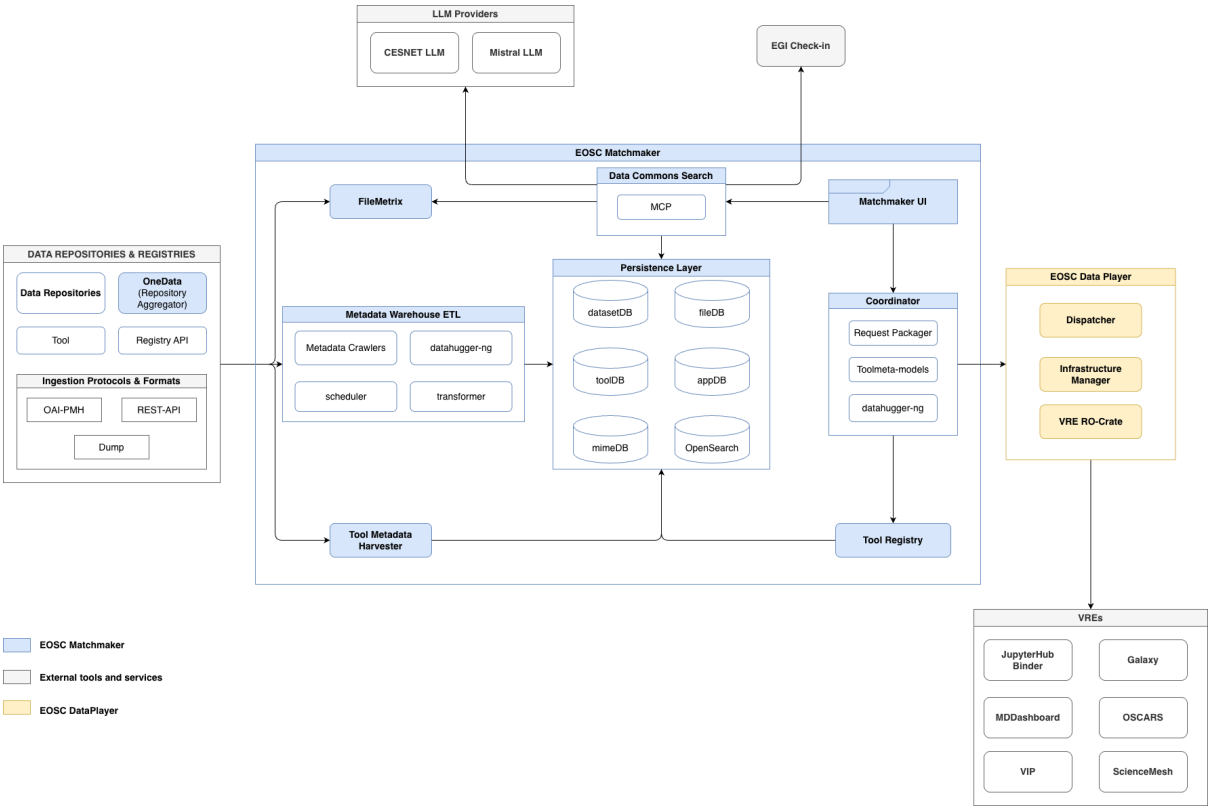

Figure 1 - EOSC Data Commons Release 1 component overview.

First, the **Metadata Crawler** collects dataset metadata from external data sources such as OneData, OAI-PMH repositories, and REST APIs, and stores the harvested information in the dataset database (datasetDB) component of the **Metadata Warehouse**. The stored dataset metadata is transformed batch-wise by a

transformer for semantic embedding and OpenSearch indexing, and interpreted by **datahugger-ng** for file-level metadata stored in the fileDB. Tools and workflows metadata are: (1) harvested actively by the **Tool Metadata Harvester** from dedicated data sources or (2) registered by tool users using the **Tool Registry API**. The **scheduler** automates the execution of metadata harvesting, transformation, and indexing workflows across all Metadata Warehouse components before being optimised for an AI-performant DB, namely the appDB for **Data Commons Search.** Users can search datasets and match the datasets with relevant tools in **Matchmaker UI**. The natural text response will be fine-tuned by **Cesnet LLM** with a fallback option to **Mistral LLM**. Once the user decides to run analysis on the selected dataset, the **Coordinator** will retrieve the required file, tool and workflow metadata before being packaged by the **VRE Research Object Crate (RO-Crate)** to enable data analysis workflow storage and provenance. If the file metadata cannot be found in fileDB before dispatching, **FileMetrix** will be triggered at runtime for file search of the selected dataset. Lastly, the RO-Crate created by users will be launched in selected tool-specific Virtual Research Environments (VREs) for data analysis, saving the time of users in searching for the right dataset and tools for their research. Please note that a user can use AI search without login authentication, but they must be authenticated by EGI Check-in during data and tool orchestration for enabling their data analysis workflow traceability and visibility (user memory).

Table 1 - List of all internal software components in the first production release.

Component	Version	TRL	Reason
Matchmaker User Interface (UI)	0.8.7	TRL7	Being used in real application
Data Commons Search	0.7.11	TRL7	Being used in real application
Metadata Crawlers	1.0.0	TRL7	Scheduled harvesting
Metadata Warehouse	1.5.0	TRL7	Being used in real application
Transformer	1.5.0	TRL7	In production with controlled rolled out
Scheduler	1.5.0	TRL7	In production with controlled rolled out
Tool Registry	1.0.0	TRL7	Being used in real application
Tool Metadata Harvester	1.0.0	TRL6	Still testing other edge cases
Filemetrix	1.0.0	TRL7	Being used in real application
Datahugger-ng	0.6.4	TRL7	Scheduled harvesting
Coordinator	0.1.0	TRL7	Being used in real application
RO-Crate Creator	1.0.0	TRL6	Simulating real use case
Dispatcher	1.0.0	TRL6	Simulating real use case

Table 2 - List of all external software components in the first production release.

Component	Version	TRL	Reason
Cesnet LLM	external	TRL9	Mature and stable
Mistral LLM	external	TRL9	Mature and stable
EGI Check-in	external	TRL9	Mature and stable
Data Repositories (see Section 3.1.4)	external	TRL8	Mature and stable where some repositories are undergoing revision

3. EOSC Matchmaker

3.1. Metadata Warehouse

Table 3 - Metadata Warehouse repository details.

Attribute	Details
Name	Metadata Warehouse
Release number	1.5.0
Repository	https://github.com/EOSC-Data-Commons/metadata-warehouse
URLs	Data models are shown in the appendix. Releases: https://github.com/EOSC-Data-Commons/metadata-warehouse/releases Documentation: https://github.com/EOSC-Data-Commons/metadata-warehouse/blob/development/README.md
Licence	Apache-2.0
TRL	7
Owner	WP4

The Metadata Warehouse serves as the central hub for aggregating and managing metadata across the Matchmaker. It provides a scalable infrastructure for harvesting, transforming, and indexing metadata from diverse repositories. It relies on an extract-transform-load (ETL) pipeline (see Figure 2) that preserves raw metadata in its original format while generating normalized and enriched records for advanced search capabilities.

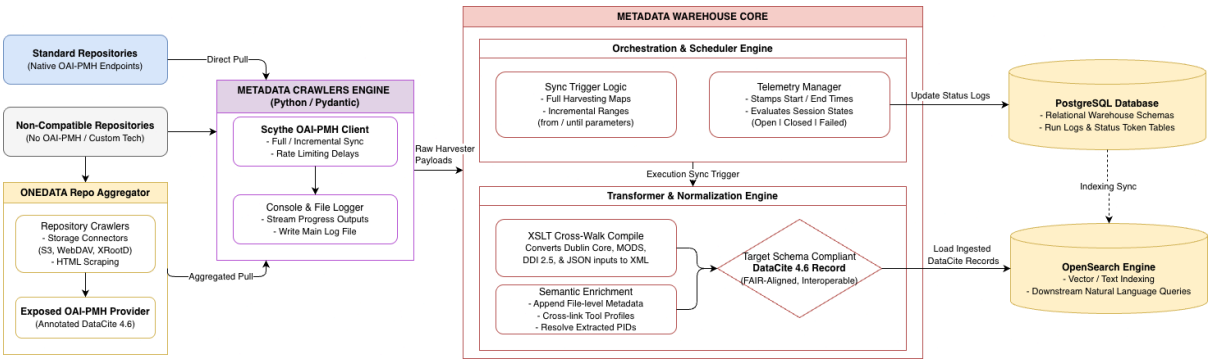

Figure 2 - Metadata Warehouse Pipeline

3.1.1. Metadata Crawlers

Table 4 - Metadata Crawlers repository details.

Attribute	Details
Name	Metadata Crawlers
Release number	1.0.0
Repository	https://github.com/EOSC-Data-Commons/metadata-crawlers
URLs	Images: https://github.com/EOSC-Data-Commons/metadata-crawlers/pkgs/container/metadata-crawlers Documentation: https://github.com/EOSC-Data-Commons/metadata-crawlers/blob/master/README.md
Licence	Apache-2.0
TRL	7
Owner	WP4

Role in the ETL pipeline

The Metadata Crawler is the first component in the ETL pipeline. Its main task is to collect metadata from external repositories and pass the collected records to the next stage of the pipeline. The component acts as a bridge between external metadata sources in EOSC Data Commons and the internal processing system. After metadata is collected, it is stored in the Metadata Warehouse and passed to the Transformer component. Metadata crawlers are responsible for obtaining the data, while the Transformer processes, transforms, and enriches the harvested records. This separation of responsibilities keeps the pipeline modular and easier to maintain.

Implementation and configuration

The Metadata Crawlers are implemented in Python. It uses Python modules to manage harvesting workflows, communication with external APIs, logging, and configuration handling. Extensible Stylesheet Language Transformations (XSLT) are used to convert other metadata formats into a DataCite 4.6 format. DataCite Metadata Schema 4.6 is used in EOSC Data Commons Project because it provides a FAIR-compliant, PID-enabled, interoperable metadata standard that supports discovery, citation, linking, and reuse of research outputs. Version 4.6 introduces enhanced support for projects, awards, identifier interoperability, and richer relationships between research objects, aligning well with EOSC interoperability and Open Science requirements. The crawler uses a configuration system written in Python using Pydantic Settings. This allows configuration values to be loaded from environment variables, *.env* files, or default values depending on the environment. The system supports multiple environments including development, staging, production, and local. Each environment can define different values for services like the Warehouse API URL, logging configuration, and access tokens.

Supported Metadata Sources

The component Metadata Crawlers is designed to support multiple metadata sources through individual crawler implementations. It mainly supports harvesting metadata from OAI-PMH endpoints. If a repository does not provide an OAI-PMH interface, a custom harvester is implemented for that specific source. For OAI-PMH sources, the preferred metadata format is DataCite 4.6. If a repository does not support this format, the crawler uses XSLT transformations to convert available formats, such as Dublin Core, MODS, or DDI 2.5, into the DataCite 4.6 format. In cases where metadata is not available in XML through OAI-PMH, the crawler can also map JSON responses into XML so that the final output is always a standard DataCite record. This ensures that all records stored in the warehouse follow a single consistent format. Some repositories also provide additional APIs or *richer* sets of metadata that contain extra information about datasets, especially file-level metadata. When this is available, the crawler sends additional requests to these APIs to collect and attach additional information to the harvested record.

In this architecture, Onedata serves as a meta-repository that aggregates datasets from sources lacking interfaces directly compatible with the Matchmaker — primarily repositories without OAI-PMH support or a file metadata discovery API compatible with datahugger-ng.

To accommodate diverse technological setups, the Onedata team has developed a generic tool called repository crawlers. It provides a framework for implementing repository-specific crawlers capable of collecting datasets and files from a given source using:

- custom protocols supported by the repository,
- HTML scraping techniques,
- storage driver-level connectors (e.g. S3, WebDAV, XRootD).

In essence, a crawler can be implemented as arbitrary Python code. Once datasets and file metadata have been crawled, they are registered in Onedata in aggregator mode — meaning no data is copied. Instead, Onedata stores point to resources residing on external systems (e.g. HTTP, S3, WebDAV) and constructs the desired logical structure at the metadata level, fully controllable through the crawler implementation.

The aggregated datasets are exposed via OAI-PMH, with persistent identifiers assigned on demand or reused where the source repository already provides them. Each dataset is annotated with DataCite metadata compatible with the Metadata Warehouse. DataCite records are likewise built within the crawler, giving the implementer full flexibility to collect and map relevant metadata from the source repository into the required format.

All data and metadata aggregated through Onedata is accessible via standardised interfaces compatible with the Metadata Warehouse and Dispatcher. As a result, every repository onboarded through Onedata is immediately aligned with the project's technical requirements.

Harvesting Process

The Metadata Crawlers component can perform both full repository harvesting (collecting all available records) and incremental harvesting using the OAI-PMH *from* and *until* parameters to retrieve only records within a specific time range. It supports the harvesting of multiple sets from a single OAI-PMH endpoint when needed. For communication with OAI-PMH endpoints, the component uses the *oaipmh-scythe* client, which handles requests, retries, and basic connection reliability. The component can also add delays between requests to avoid overloading external services. During execution, it prints progress messages and errors to the terminal and also writes them to a main log file. It records both the start and end time of each harvest run and sends this information to the database. Each harvest run is marked with a status: *open* when it is running, *closed* when it finishes successfully, and *failed* if any record fails to be fetched or if the server does not respond properly. The crawler also includes basic tests that check whether transformations work correctly (for example, converting from Dublin Core to DataCite) and whether the overall harvesting workflow behaves as expected.

3.1.2. Transformer

Table 5 - Transformer repository details.

Attribute	Details
Name	Transformer
Release number	1.5.0
Repository	https://github.com/EOSC-Data-Commons/metadata-warehouse/blob/a370e8a75a8c7baeb3775aab48b0048c2ea42abf/src/transform.py
URLs	Images: https://github.com/EOSC-Data-Commons/metadata-warehouse/pkgs/container/transformer Releases: https://github.com/EOSC-Data-Commons/metadata-warehouse/releases Documentation: https://github.com/EOSC-Data-Commons/metadata-crawlers/blob/master/README.md
Licence	Apache-2.0
TRL	7
Owner	WP4

The Transformer is a service that processes harvested metadata records and makes them available for search and discovery. It sits between the Metadata Crawlers (harvester which collects raw OAI-PMH XML) and the downstream stores (PostgreSQL and OpenSearch).

Input

Records enter the system as *harvest events* — raw XML metadata in DataCite format, connected to a *harvest run*. A harvest run represents either an initial full harvest or an incremental update for a given endpoint. Records are read from the *harvest_events* table in PostgreSQL, grouped into configurable-size batches, and queued for asynchronous processing via Celery.

Job Types

Two types of Celery jobs run in parallel per batch:

- *transform_batch* handles the core transformation pipeline. For each record, it parses the raw XML, converts it to normalised DataCite JSON, validates it against a schema, and calculates vector embeddings for semantic search. Results are bulk-indexed into OpenSearch and upserted into the `records` table in PostgreSQL. Deleted records are removed from both stores. Errors at the individual record level are logged to the *harvest_events* table without

failing the whole batch; errors at the batch level (e.g. embedding failure, bulk index failure) roll back the PostgreSQL transaction for that batch. For failed individual records, a report can be generated to fix validation problems at the source (e.g., invalid dates, missing required information in the source data).

- *add_file_metadata* extracts file-level metadata (names, sizes, checksums, download URLs) from provider-specific APIs (Dataverse, Zenodo, HAL, DABAR etc.) and writes it to the *record_files* table.

API

A FastAPI application exposes the control interface. The */index* endpoint triggers batch creation for a given harvest run and OpenSearch index. Supporting endpoints manage harvest run lifecycle (create, close, status check) and endpoint configuration. A */scheduler* endpoint group allows an external orchestrator to poll for completion before triggering downstream steps.

3.1.3. Scheduler

Table 6 - Scheduler repository details.

Attribute	Details
Name	Scheduler
Release number	1.5.0
Repository	https://github.com/EOSC-Data-Commons/metadata-warehouse/tree/a370e8a75a8c7baeb3775aab48b0048c2ea42abf/scheduler
URLs	Images: https://github.com/EOSC-Data-Commons/metadata-warehouse/pkgs/container/scheduler Releases: https://github.com/EOSC-Data-Commons/metadata-warehouse/releases Documentation: https://github.com/EOSC-Data-Commons/metadata-crawlers/blob/master/README.md
Licence	Apache-2.0
TRL	7
Owner	WP4

The Scheduler is responsible for orchestrating the metadata harvesting and indexing workflow. It is implemented as a scheduled CRON job and, in the current production deployment, is executed once per week.

The scheduler coordinates the end-to-end harvesting process and ensures that newly harvested metadata becomes available for search and discovery. The workflow consists of the following steps:

1. The scheduler queries the Transformer API to determine which repository endpoints are due for harvesting.
2. For each eligible endpoint, the scheduler triggers the Metadata Crawler. The crawler is responsible for creating and managing the corresponding harvest run and collecting metadata records from the source repository.
3. The scheduler monitors the status of all harvest runs and waits until harvesting is complete for every triggered endpoint.
4. Once all harvest runs have successfully finished, the scheduler invokes the Transformer API to start metadata transformation, embedding calculation, and OpenSearch indexing. This process is executed for all harvest events associated with the newly completed harvest runs.

By centralising the orchestration logic, the Scheduler provides an automated and reliable mechanism for coordinating harvesting activities and ensuring that newly collected metadata is processed, enriched, and made available through the platform's search infrastructure.

3.1.4. Integrated Repositories

The Metadata Crawlers support onboarding multiple external repositories with different endpoint configuration. Each repository may expose one or more OAI-PMH endpoints, or in some cases a repository in-house API that requires a customized harvesting implementation.

Table 7 lists all repositories which are successfully integrated into the Metadata Warehouse, and Table 8 lists ongoing issues for those repositories with partial harvesting status in release 1 of the Matchmaker. Tables 7 and 8 distinguish between internal repositories (in the consortium) and external repositories.

Table 7 - All integrated repositories in Metadata Warehouse.

Repository	Description	Base URL	Endpoints	Status
Internal repositories				
DaSCH	DaSCH metadata repository	https://dasch.swiss	Single OAI-PMH endpoint	Partial
DABAR - Digital Academic Archives and	Croatian national repository system	https://dabar.srce.hr/en	Single OAI-PMH endpoint	Complete

Repository	Description	Base URL	Endpoints	Status
Repositories				
FinBIF	Finnish Biodiversity Information Facility	https://laji.fi	Custom harvester (no OAI-PMH)	Complete
Data Archiving and Networked Services	Dutch national archive for research data	https://dans.knaw.nl	5 OAI-PMH endpoints: ● Archaeology Data Station ● Social Sciences Data Station ● Life Sciences ● Physical and Technical Sciences ● Generalist	Complete
HAL Science	French open archive	https://hal.science	Single OAI-PMH endpoint	Complete
Molecular Dynamics Data Bank	Unified database of the data generated by molecular dynamics simulations	https://mddbr.eu/	Custom harvester (no OAI-PMH)	Partial
Onedata	Onedata demo	https://demo.onedata.org	4 OAI-PMH endpoints: ● Bgee ● EODC ● GWAS ● VIP	Complete
SwissUbase	Swiss data repository	https://www.swissubase.ch	Single OAI-PMH endpoint	Partial
External repositories				
Dataverse Latvia EOSC Node	Latvian research data repository	https://dataverse.lv	3 OAI-PMH endpoints: ● DataverseLV ● Riga Stradins University ● CLARIN-IV	Complete
PaNOSC	Scientific data infrastructure repository	https://www.panosc.eu	11 OAI-PMH endpoints: ● DESY ● Elettra ● ESRF ● ESS ● EuXFEL ● HZB ● HZDR	Partial

Repository	Description	Base URL	Endpoints	Status
			• ILL • ISIS • MAX IV • PSI	

Table 8 - Repositories with unresolved issues.

Repository	Issues
Internal repositories	
Molecular Dynamics Data Bank	Unable to fetch additional file metadata because of connection and too many requests errors
SwissUbase	They have unique records for each different metadata prefix in their OAI-PMH endpoint
DaSCH	4 records have duplicated IDs
External repositories	
PaNOSC	• Repository ALBA throws Error 500 at the end of an initial harvest • Incremental harvests for repositories DESY, ESS and MAX IV doesn't work

3.2. User Interface

Attribute	Details
Name	Matchmaker User Interface (UI)
Release number	0.8.7
Repository	https://github.com/EOSC-Data-Commons/matchmaker
URLs	UI: https://matchmaker.eosc-data-commons.eu/ Images: https://github.com/EOSC-Data-Commons/matchmaker/pkgs/container/matchmaker-frontend Documentation: https://github.com/EOSC-Data-Commons/matchmaker/blob/main/README.md
Licence	MIT
TRL	7

Attribute	Details
Owner	WP4

Table 9 - Matchmaker repository details.

The User Interface is the main entry point to the EOSC Data Commons platform, taking the user from a natural-language question to a ranked list of relevant datasets before launching an analysis tool on a chosen dataset.

3.2.1. Views

The interface is organised around three main views detailed below.

Dataset search (/search)

Provides users with the results from their searches. The question typed on the landing page is sent to the Backend, and the results are rendered in two phases: initial OpenSearch matches are displayed immediately while the LLM-based processing continues, after which the reranked datasets are shown with relevance-score badges and a natural-language summary. Results can be refined further without additional server requests through client-side facets (publication year, author, subject) computed locally from the result set; the filter state is synchronised to the URL so that a filtered view can be bookmarked and shared. Each result card shows the origin repository, links to the dataset via its DOI or URL, and offers citation export: citations are retrieved from the official DOI metadata (content negotiation against doi.org) with a locally generated BibTeX fallback.

Conversational search (/chat)

Authenticated users can switch the search input to AI mode and continue dataset discovery as a conversation. The chat view streams the Backend's responses as they are produced, renders the retrieved datasets inline as result cards, and keeps the multi-turn context within a conversation thread. A sidebar lists the user's past conversations (served by the Backend's */conversations* endpoints), which allow them to be resumed or deleted with a confirmation prompt.

Profile and API keys (/profile)

Authenticated users have a profile page for managing personal API keys that allow the platform to act on their behalf when launching tools. Two keys are supported: a VIP API key, which lets the Data Sandbox interoperate with the VIP Virtual Research Environment on the user's behalf, and a GitHub personal access token, used to raise the rate limit on GitHub API requests. Keys are never stored in the browser or in the application database; they are held in the EGI Secret Store (HashiCorp Vault) and accessed only through the cookie-authenticated /auth/keys API, where each key can be set, replaced, revealed, or removed. The page is reached through the "API Keys" entry in the user menu.

Tool launching (*/dataplayer*)

Provides users with an interface to interact with the dataset, namely, processing data from the dataset. This includes functionalities such as discovering and matching tools (see the definition of “tool” in section 3.5) and using these tools to process the data identified through the matchmaker data search interface described above.

It consists of two main components: a) **Dataset File Browser**: as shown in Figure 3. After the dataset is selected, the system will try to automatically search for tools that are potentially able to process the data from the dataset based on the MIME-type. In this release, to let users precisely target the tool they know and need, we provide a search text bar to pick out the tool they want. This component lists all files within a dataset, along with detailed metadata such as file size and MIME type. It also provides a direct download option, allowing users to retrieve files from their original data source, b) **Tool Matching and Execution Panel**: This component enables users to search for and match appropriate tools. Users can interact with the selected tool and launch it at the location where the VRE is hosted, using data obtained from the Matchmaker search results. The task is then submitted through the Coordinator. The Coordinator execution state is streamed live to the browser as Server-Sent Events; once the task is ready, the result is opened through the callback URL returned by the Coordinator.

In addition, the UI provides an “Additional Datasets” feature that allows users to attach datasets by providing links to datasets that have not been harvested into the dataset database. This allows users to process these external datasets alongside the files retrieved from the Matchmaker search results. The supported datasets are listed in the table in the documentation of datahugger-ng.

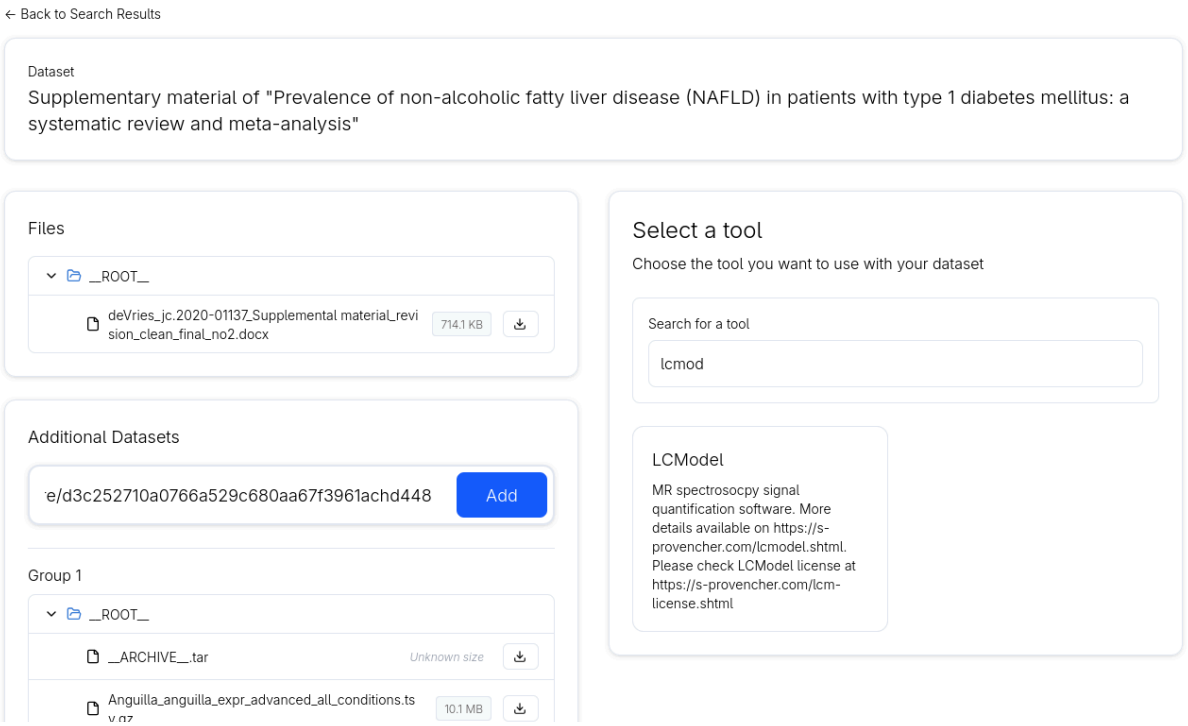

Figure 3 - The screenshot of the dataset file list panel (on the left) and the tool search and selection panel widget (on the right).

When a tool is selected, users are guided to an input configuration panel, where they must provide the required parameters to launch the tool. The structure of this panel depends on the tool type (see Figure 4 for one example of the input widget from a LCModel tool from VIP Virtual Research Environment).

Three categories of tools are supported.

- First, the dataset-level tools operate on the entire dataset of files. In this case, the dataset URL is passed as the sole input parameter when launching the tool.
- Second, the slot-based tools require users to map inputs to a predefined set of slots before execution. Each slot must be filled with the appropriate input types, such as a file, number, flag, or string
- Thirdly, there are file storage tools which accept a collection (array) of files as input. Unlike slot-based tools, no explicit mapping is required and users simply provide the relevant files, which are then passed directly to the tool. One example of this tool in the release is the CernBox.

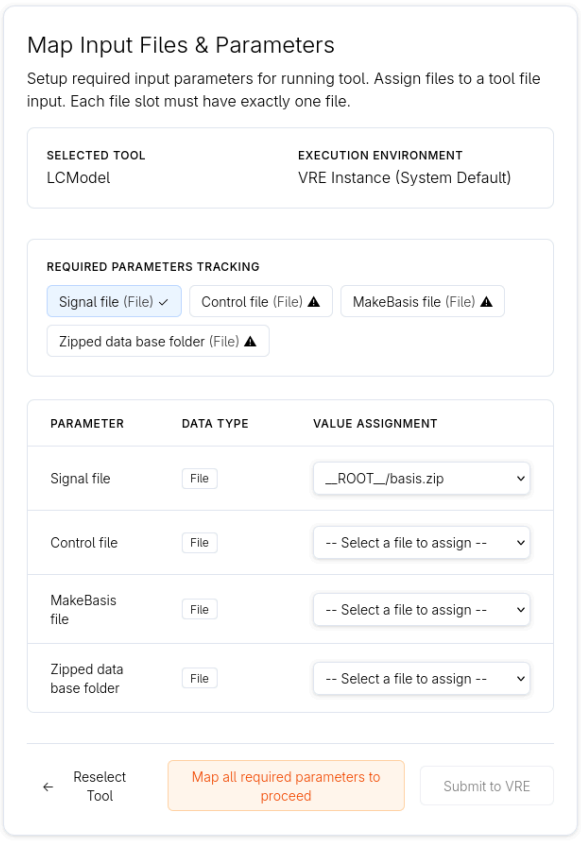

Figure 4 - Screenshot from the inputs config panel of the LCModel tool that runs at VIP VRE. Users need to select files for every input slot.

3.2.2. Architecture and integration

The UI is a server-side rendered web application written in TypeScript with React 19 and React Router 7, styled with Tailwind CSS and served by a Node.js (Express) server. The Express server has a dual role. It performs the server-side rendering and serves the client bundle, and it acts as the API gateway for the browser: search, chat, and authentication calls are proxied to the Backend, while tool- and task-related requests are translated into gRPC calls to the Coordinator. The gRPC client code is generated from the protobuf contract shared with the Coordinator code base (included as a git submodule), keeping both sides of the contract in sync. Login uses the Backend's EGI Check-in integration; the session lives in HttpOnly cookies, and the UI server attaches the user's EGI access token to the Coordinator calls that launch tools and fetch files, so these actions are performed under the user's identity. Search remains available to anonymous users.

Pull requests are gated by build and dependency-audit checks, and an npm minimum-release-age policy reduces exposure to freshly published, potentially malicious package versions. In production, the compiled server runs in cluster mode under the pm2 process manager.

Privacy and acceptable-use considerations are surfaced directly in the interface: the application links to a dedicated Privacy Policy page (/privacy-policy) and an Acceptable Use Policy page (/acceptable-use-policy), which set out the personal data processed, its legal basis under the GDPR, and the conditions under which the service may be used

3.3. Data Commons Search

Table 10 - Data Commons Search repository details.

Attribute	Details
Name	Data Commons Search
Release number	0.7.11
Repository	https://github.com/EOSC-Data-Commons/data-commons-search
URLs	Images: https://github.com/EOSC-Data-Commons/data-commons-search/pkgs/container/data-commons-search Releases: https://github.com/EOSC-Data-Commons/data-commons-search/releases Documentation: https://github.com/EOSC-Data-Commons/data-commons-search/blob/main/README.md
Licence	MIT
TRL	7
Owner	WP4

An HTTP API written in Python (FastAPI), described by an auto-generated OpenAPI specification, served at */docs*. It communicates and coordinates among the UI, PostgreSQL, OpenSearch, and EGI Check-in, turning a natural-language question into a ranked list of relevant datasets or tools.

The service exposes four groups of endpoints:

Search Tools

A Model Context Protocol (MCP) server, mounted at */mcp* over streamable HTTP transport (stateless, JSON responses). It exposes the search capabilities as tools that any MCP-compatible LLM client (Claude Code, GitHub Copilot, etc.) can call.

This lets users reach the search functions directly, without being limited to the chat agent provided by the EOSC Matchmaker. The main tool, *search_data*, runs a hybrid query against OpenSearch (see Section 3.7.2). It accepts optional structured filters (date range, creator name) that the LLM extracts from the user question. Two companion tools, *get_dataset_files* (file-level metadata from the FileMetrix API) and *search_tools* (relevant tools and services), complete the discovery flow.

Conversational interface (*/chat*)

A direct conversational endpoint for search, used by the UI conversational search view, that drives the tools exposed on */mcp* through an LLM. A request opens an agentic tool-calling loop: the model selects and calls tools, the results are fed back, and a final reranking step asks the LLM to score the retrieved datasets against the user question and produce a natural-language summary. The response is streamed as Server-Sent Events following the AG-UI protocol, emitting message, tool-call and tool-result events as they happen, with the conversation *thread_id* returned in a response header. The LLM provider is configurable at deployment (e-INFRA CZ, OpenRouter, or Mistral); because inference goes through the LangChain chat-model wrapper rather than a provider-specific SDK, the service is not locked to any single vendor and can be adapted to most providers with minimal changes. The endpoint is protected against abuse by a per-user (or per-IP for anonymous traffic) rate limit and, optionally, a shared API key that gates access against bots.

Authentication (*/auth*)

Login through EGI Check-in via OpenID Connect (OIDC). The */auth/login*, */auth/callback*, */auth/logout* and */auth/user* endpoints implement the Authorisation Code flow with PKCE and a state parameter for CSRF protection. The provider configuration is discovered from the EGI Check-in well-known endpoint. After a successful exchange, the access and refresh tokens are stored in HttpOnly cookies; the access token is silently refreshed when it expires, and refreshed cookies are reattached even on streaming responses. Authentication is optional on search and chat (anonymous users are allowed), but required for conversation history.

Conversation history (*/conversations*)

Endpoints to list, retrieve and delete a user's past conversations, each scoped to the authenticated owner.

Persistence and rate limiting

All persistent states live in the shared PostgreSQL *appDB* database. The backend defines and owns four tables: *users* (provisioned from OIDC userinfo on first login), *conversations* and *messages* (chat history, messages stored as JSONB), and *rate_limits*. The rate limiting is enforced directly in Postgres through an atomic upsert

on the *rate_limits* table, keyed per authenticated user or, for anonymous traffic, per client IP, with a shorter interval for logged-in users than for anonymous ones.

3.4. FileMetrix

Table 11 - FileMetrix repository details.

Attribute	Details
Name	FileMetrix
Release number	1.0.0
Repository	https://github.com/Dans-labs/filemetrix
URLs	Releases: https://github.com/Dans-labs/filemetrix/releases Documentation: https://github.com/Dans-labs/filemetrix/blob/main/README.md
Licence	MIT
TRL	7
Owner	WP5

FileMetrix is a FastAPI-based web service that provides a REST API for retrieving and exposing file-level metadata associated with research datasets. It offers a few endpoints that enable clients to discover, retrieve, and process detailed information about the files contained within a dataset, regardless of the underlying repository. The service is designed to support other software components by providing a consistent interface for accessing file-level metadata.

List dataset file-level metadata (/{PID})

The main API entry point accepts a persistent identifier (PID) for a dataset and returns a complete listing of the files that belong to that dataset. For each file, the service provides detailed metadata including the file name, download URL, size, checksum (hash), and other available descriptive attributes retrieved from the underlying repository.

File extensions (/extensions/{PID})

This endpoint returns the unique file extensions present within a dataset. It accepts a dataset PID as input, retrieves the corresponding file-level metadata, and processes the results to extract a deduplicated list of file extensions. By providing a concise

overview of the file types contained in a dataset, the endpoint enables clients to quickly assess the dataset's composition without retrieving the complete file listing.

3.5. Coordinator

Table 12 - Coordinator repository details.

Attribute	Details
Name	Coordinator
Release number	0.1.0
Repository	https://github.com/EOSC-Data-Commons/req-packager
URLs	Images: https://github.com/EOSC-Data-Commons/req-packager/pkgs/container/coordinator Documentation: https://github.com/EOSC-Data-Commons/req-packager/blob/main/README.md
Licence	MIT
TRL	7
Owner	WP5

To enable modularity and performance, a **coordinator** server is implemented to communicate backend services with the UI. Rather than interacting directly with individual services, the UI communicates with the coordinator, which exposes a unified interface and abstracts away service-specific details.

Through the **Coordinator**, the **Matchmaker** interacts with several backend components. These include the tool registry (see Section 3.6) for retrieving tool metadata, as well as datahugger-ng and the file metadata database for obtaining dataset and file-level information. To initiate tool execution, the coordinator utilises the request packager (see Section 3.5.1), which combines tool metadata, dataset information, and user-provided inputs into a structured request. This request is then forwarded to the dispatcher (see Section 4.1), which launches the selected tool within the Virtual Research Environment (VRE) and returns a response used to redirect the user to the running tool instance.

The use of the coordinator enables a clean separation between the UI and backend services, improving modularity and maintainability. This architectural decision is primarily motivated by the requirement for the Data Player to function as a

standalone, headless service, capable of operating independently from the Matchmaker platform.

Under the hood, the coordinator operates as a gRPC server, while the UI acts as a gRPC client. Communication between the two is bidirectional, enabling the exchange of requests and responses required to coordinate interactions with backend services. The coordinator consists of three services:

1. DatasetService, which wraps around the datahugger and fileDB
2. ToolService, which wraps around the tool-registry
3. DataplayerService wraps around the dispatcher and tool-registry.

The dataflow of three services is described in Figure 5 below. From left to right, it shows chronologically how the data flow through each service. The box with lambda is the functions of the services. The dark boxes represent the standalone components to communicate with the databases.

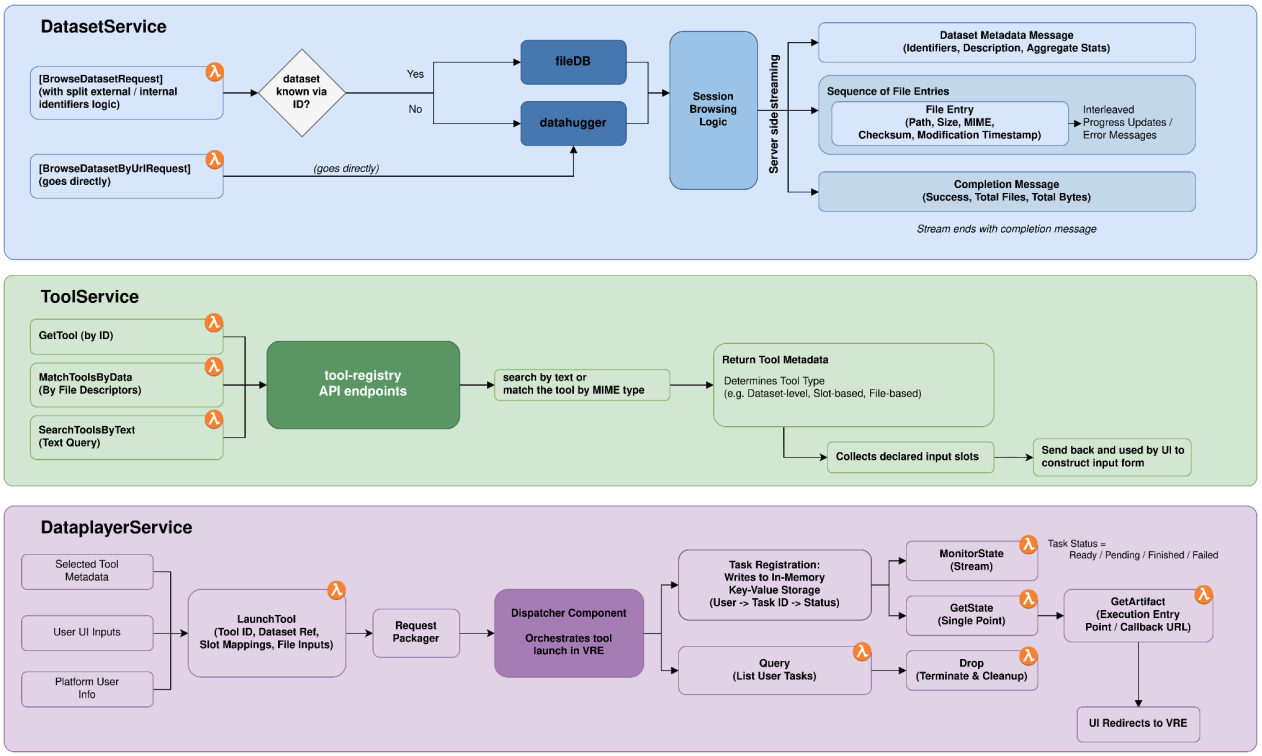

*Figure 5 - Three services of the **Coordinator** and their functions.*

3.5.1. Request packager

The request packager is a component sitting inside the coordinator to prepare the payload from three sources of input for the dispatcher. The three sources are:

1. Inputs from the matchmaker UI after the user interacts with the UI.
2. Inputs from tool-registry after the user selects a tool.
3. Inputs of user from platform to identify the user.

Those inputs are packed and serialised into the payload format (i.e. the RO-Crate) and sent to the dispatcher to be deserialised so it can be used for interacting with different VREs to launch the tool.
In this release, we are targeting five major Virtual Research Environments required from the project use cases, which are VIP, RRP, Galaxy, CernBox, MDDash. To cover those use cases, an abstraction of launch input is made. The data structure of user UI inputs is the **LaunchInput**, which is a union type that can be

1. A DatasetOnly input contains only a url as input. The typical example is Jupyter binder-like VRE (e.g. mybinder, EGI Replay), which accepts a dataset URL as input and launches a Jupyter notebook.
2. A SlotsOnly input contains a mapping of slots to the slot values that VRE needs to set specific inputs for a tool. The typical examples are VIP and Galaxy.
3. A SlotsAndFiles input contains not only a mapping to get the values for the slots but also a list of files from the dataset. These selected files do not need to be mapped specifically to any slots. The CernBox is a typical VRE in release one that accepts this kind of input.

On the UI side, after the user selects the tool, the input layout is determined; thus, in the request packager, we do not need to further validate if the input layout matches with the tool. The inputs from tool-registry are modelled in the ToolMeta data structure. The ToolMeta is an inner representation of the response from tool-registry. It extracts the necessary information (necessary for launching a tool in the corresponding VRE) from the response of a tool.

Virtual Research Environments (VREs) are typically provisioned upon user request (on behalf of Dispatcher) and are accessible within the scope of a specific project. Therefore, when launching a tool within a VRE, the command must include additional credentials. These credentials allow the VRE to verify that the request originates from an authorised user of EOSC Data Commons and to proceed with launching the tool on their behalf. In the request packager, this information is not directly attached within the inputs; instead it is represented by a user token from the **auth/** endpoint of the UI server and the coordinator and Data Player will rely on that token to acquire more information on demand.

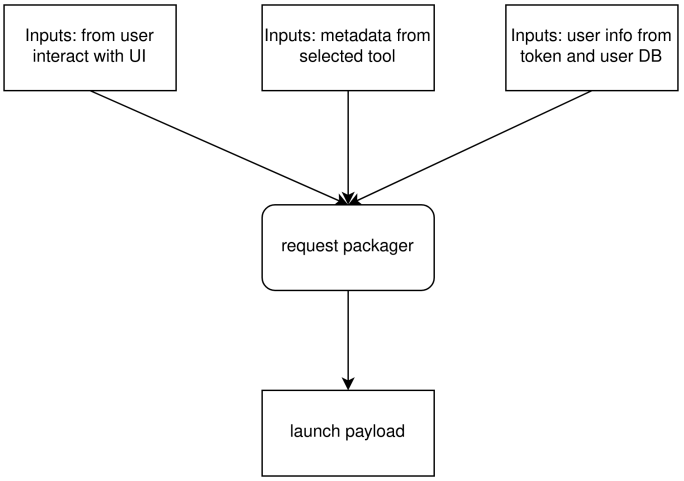

Figure 6 - Flow of how the request packager gets information from and what payload it produces for the downstream dispatcher for launching the tool with data.

3.6. Tools Registry

Table 13 - Tool Registry repository details.

Attribute	Details
Name	Tools Registry
Release number	1.0.0
Repository	https://github.com/EOSC-Data-Commons/tool-registry
URLs	Images: https://github.com/EOSC-Data-Commons/tool-registry/pkgs/container/tool-registry Documentation: https://github.com/EOSC-Data-Commons/tool-registry/blob/main/README.md
Licence	Apache-2.0
TRL	7
Owner	WP5

This FastAPI web service provides the main API for registering, discovering, retrieving, updating, deleting, and matching scientific tools in the EOSC Data Commons Tool Registry using MIME-based matching. It acts as the HTTP interface between clients and the underlying database model ToolGeneric, exposing tool metadata such as name, URI, version, licence, description, keywords, tags, supported

tool types, and supported input/output file formats. The *toolDB* database schema is explained further in the Appendix.

List tools (/tools)

Lists the tools in the database. The endpoint supports pagination through *limit* and *offset*, returns the total count in *X-Total-Count*, and can optionally return all matching tools.

The endpoint also supports filtering of tools. Clients can filter by name, description, input file format, output file format, tool type, tag, keyword, and creator.

Retrieve tool metadata (/tools/{identifier})

Retrieves the complete metadata record for a single tool by internal numeric ID. It returns the full database record for the tool. The endpoint */tools/{identifier}/raw_definition* returns the original tool definition, e.g. the Galaxy workflow json description or the Boutique definition for the VIP use case.

CRUD operations (/tools)

The */tools* endpoint implements CRUD operations through post, patch and delete HTTP methods that allow to create, update or delete a tool entry. The request body contains the tool metadata. Creation, updating and deletion require an EGI token to be validated. Creation stores the authenticated user as created_by, and rejects duplicate tool URIs. Updating and deleting is only allowed by the same user.

Matchmaking (/match)

The tool registry performs MIME type-based matchmaking. This is done by matching tools against input criteria, currently file-based matching. The service derives file extensions from MIME types and finds tools whose declared input formats match either any input format (*or*) or all input formats (*and*). MIME type matching is assisted through a separately developed component, ***mimeDB.*** mimeDB harvests MIME types from IANA, Galaxy, Pronom and Apache to build a local database with MIME-type to file-extension conversions.

Upon receiving a request, the service consults a local MIME type database to resolve MIME types into their corresponding file extensions. For example, the MIME type *text/csv* is mapped to the *csv* extension. When a MIME type is not provided, or when no corresponding extension can be determined, the service falls back to the extension extracted from the supplied filename. This approach allows matching to remain effective even when MIME information is incomplete.

The matching behaviour is controlled through an operator option. When the *or* operator is used, tools are considered compatible if they support at least one of the specified input formats. When the *and* operator is selected, only tools that support all specified input formats are returned.

3.7. Tool Metadata Harvester

Table 14 - Tool Metadata Harvester repository details.

Attribute	Details
Name	Tool Metadata Harvester
Release number	1.0.0
Repository	https://github.com/EOSC-Data-Commons/toolmeta-harvester
URLs	Releases: https://github.com/EOSC-Data-Commons/toolmeta-harvester/releases Documentation: https://github.com/EOSC-Data-Commons/toolmeta-harvester/blob/main/README.md
Licence	Apache-2.0
TRL	6
Owner	WP5

The tool harvesting pipelines collect tool and workflow metadata from external sources and transform it into the common ToolGeneric model used by the Tool Registry. They act as ingestion workflows that bridge external catalogues, Galaxy instances, ToolShed repositories, workflow hubs, and VIP application metadata with the central registry database or REST API. The current architecture of the harvester is composed of tasks and flows. Tasks are reusable functions that facilitate the extraction, transformation and loading of external tools, while flows combine tasks into pipelines to execute the full ETL pipeline. The harvester is a collection of flows.

Galaxy ToolShed Git Harvesting

The *harvest_a_shed_tool.py* pipeline harvests a specific Galaxy ToolShed tool from a Git repository. It crawls the repository, extracts tool metadata such as name, version, URI, description, categories, inputs, outputs, and file formats, converts this information into ToolGeneric, and stores it in the database.

Workflow Hub Harvesting

The *harvest_galaxy_hub_workflows.py* pipeline harvests workflows from the Galaxy Workflow Hub. It extracts workflow metadata, including UUID, name, version, tags, URL, licence, input/output slots, input/output formats, raw Galaxy definitions, raw metadata, and ToolShed tools used by the workflow. The harvested workflows are then stored in the generic tool table.

UseGalaxy Instance Harvesting

The *harvest_usegalaxy_base.py* pipeline provides a reusable workflow harvester for public Galaxy instances. It connects to a configured Galaxy instance, iterates over available workflows, extracts metadata and input/output format information, and stores each workflow in the *toolDB* database.

The *harvest_usegalaxy_eu.py, harvest_usegalaxy_fr.py, harvest_usegalaxy_ch.py* and *harvest_usegalaxy_org.py* scripts specialise the generic UseGalaxy harvesting pipeline for specific public Galaxy servers.

VIP Application Harvesting

The *harvest_vip_apps.py* pipeline harvests application metadata from the VIP Git source repository. VIP tools are workflows composed of containers written in the *Boutique* format. The harvester crawls a GitHub repository of tools to extract the metadata and ingest each workflow into the *toolDB*. The tool descriptions do not specifically define the file input types for each workflow, but often mention the type as part of a description. The pipeline uses basic NLP on the description to extract file input information.

HAL Python Notebook Harvesting

The *harvest_hal_apps.py* pipeline harvests metadata records from the HAL repository. The pipeline looks for Zenodo publications within each HAL record and crawls the Zenodo record to look for Python notebooks. These are extracted as tools and registered in the Tool Registry. Python notebooks are often published within a *ZIP* file in Zenodo, which means the pipeline needs to extract file information from the archive file to look for Python notebooks.

3.8. Persistency Layer

3.8.1. PostgreSQL

The Matchmaker utilizes PostgreSQL as its primary database for long-term persistence. It was selected for its robust support of both structured relational data and flexible semi-structured formats, which is critical for handling diverse metadata from different repositories. PostgreSQL serves as the definitive source of truth for

the ETL pipeline, storing raw XML metadata as harvested from OAI-PMH endpoints alongside processed JSONB representations.

There are four databases in the Matchmaker PostgreSQL (for detailed information and schema, check Appendix I)

- appDB: supports the search application itself, including rate limiting, user management, and chat history.
- datasetDB: supports the metadata harvesting and transformation pipeline. Stores information about the providers, endpoints, and the harvesting jobs.
- fileDB: Stores metadata for files associated with harvested datasets, it's populated by the add_file_metadata job.
- toolDB: stores metadata for tools, serving as a central registry of software tools and services.

3.8.2. OpenSearch

OpenSearch serves as the search backend, holding a transformed and enriched copy of each record optimised for both keyword and semantic search.

Documents stored in the index conform to a normalised subset of the DataCite metadata standard, validated before indexing. Every document requires an id, at least one creator with a creatorName, at least one title, and either a DOI or URL. Optional fields cover subjects, descriptions, dates, types, formats, and rights. Language fields follow ISO 639-1 two-letter codes, and the schema is strict — additionalProperties: false at all levels means unexpected fields from upstream transformation cause validation to fail rather than passing through silently.

In the index, titles, descriptions, creators, and subjects are modelled as nested objects to preserve one-to-many relationships within a record. Their text content is copied into shared convenience fields (_all_fields, _title, _description, _creator, _subject) to allow simple cross-field full-text queries without targeting nested paths directly. DOI and URL are stored as keywords for exact matching. Two internal fields, _repo and _harvest_url, carry provenance metadata identifying the origin repository and endpoint. Dynamic mapping is disabled, so any undeclared fields in incoming documents are silently ignored.

The index has kNN search enabled, with the embedding (emb) field storing dense vector embeddings using cosine similarity, the HNSW algorithm with Lucene engine, and scalar quantisation for memory efficiency. Query embeddings are computed locally with a multi-language dense embedding model, *BAAI/bge-small-en-v1.5*) and BM25 keyword matching over titles, descriptions and subjects, fused through an

OpenSearch RRF (reciprocal rank fusion) search pipeline. This supports three search modes: pure full-text, pure semantic, and hybrid — where keyword and vector scores are combined using OpenSearch's score normalisation and combination pipeline, allowing relevance from both signals to be weighted and merged into a single ranking.

4. EOSC Data Player

4.1. Dispatcher

Table 15 - Dispatcher repository details.

Attribute	Details
Name	Dispatcher
Release number	1.0.0
Repository	https://github.com/EOSC-Data-Commons/Dispatcher
URLs	API endpoint: https://player.eosc-data-commons.eu Documentation: https://github.com/EOSC-Data-Commons/Dispatcher/blob/master/README.md
Licence	MIT
TRL	6
Owner	WP6

The **Dispatcher** is a relatively thin software layer located between Matchmaker/Coordinator and the target **virtual research environments (VREs)** in the overall project stack architecture. It accepts the packages in a unified format, unwraps and translates them into specific VRE protocols, and manages execution of the payload.

4.1.1. Interface and input format

Dispatcher exposes a REST API interface with essential operations for **submitting a request** and checking the **request status**.

The requests are accepted in the RO-Crate format, which is generic enough to cover any feasible use case. All the requests refer to a workflow to be executed and its required inputs.

Options for submitting a full ZIP archive containing multiple files (auxiliary inputs), or a standalone *ro-crate-metadata.json* (using references to external entities for both the workflow and its inputs) are provided.

Handling the specifically supported RO-Crate profiles is extracted into a dedicated library. Besides parsing and validating the supported RO-Crates, the library also provides code to create RO-Crate skeletons for testing and demonstration purposes.

4.1.2. Supported VREs

At the time of writing, Dispatcher supports the following generic VREs:

- **Binder** and **Jupyterhub**. Workflow is a Jupyter notebook (.ipynb), auxiliary input files are uploaded to the target environment, and main input files must be downloaded by the notebook itself. In the Binder case, Dispatcher prepares a temporary git repository with the payload and the user is provided with a Binder-ready URL to trigger the notebook setup. With Jupyterhub, Dispatcher talks to the target service on behalf of the user, and it prepares the whole environment.

- **Galaxy**. Workflow is in Galaxy format (.ga), Dispatcher uses unauthenticated *workflow landing* Galaxy interface to prepare the environment for the user, its execution is triggered by the user directly.

Moreover, the following specialised VREs linked to the project use cases (WP7) are supported as well:

- **Oscar.** https://github.com/grycap/oscar/
- **ScienceMesh.** https://sciencemesh.io/
- **VIP**. https://www.creatis.insa-lyon.fr/vip/
- **MDDashboard**. https://github.com/CERIT-SC/mddash (experimental)
- **Scipion**. https://scipion.i2pc.es/ (experimental)

4.1.3. Supported infrastructures

The dispatcher can either talk to an externally managed, existing instance of the target VRE (like usegalaxy.eu, notebooks.egi.eu), or spawn a dedicated instance of the VRE. The latter is done via **Infrastructure Manager (IM)**. In this case, the Dispatcher input RO-Crate must be enriched with a link to IM's deployment recipe for the specific VRE, and specification of the resource requirements (CPUs, memory, etc.).

Experimentally, deployment in **Kubernetes** is supported as well; similarly to IM, the RO-Crate contains inputs to Helm chart (a standard way to specify Kubernetes deployments). In particular, Kubernetes deployments are expected to make the connection to HPC resources via interLink.

4.1.4. Internal architecture and implementation

Dispatcher is built as a FastAPI Python application. On receiving a request from the input endpoint, a Celery task is created and its unique identifier is returned to the user immediately. Processing of the request, which involves interaction with the target VRE is done asynchronously. The user can check the status/progress of the request any time.

Various VREs are supported in a semi-dynamic way through a factory mechanism which loads specific VRE classes (inherited from a common abstract ancestor) paired together with a VRE identifier string used in RO-Crate (e.g. "https://jupyter.org" maps to class *VREJupyter*).

Configuration is managed with *pydantic-settings*, coding style is enforced via GitHub actions.

Deployment is managed with *Ansible*, it consists of several on-the-fly build docker containers via Docker compose (nginx front-end, Dispatcher itself, certificate renewal, auxiliary services).

4.1.5. Authentication

With the current design, Dispatcher does not run with any service identity, which would be granted access to use other services. It operates either **anonymously** where possible (e.g. using Galaxy workflow landing interface) or it acts **on behalf of the user**, by relying on HTTP headers with user credentials.

The primary operation mode of Dispatcher is authenticated. The requests are expected to carry a user's OAuth2 bearer token in the *Authentication* HTTP header. The token is used to access the downstream VREs (and IM and Kubernetes), either unchanged or after OAuth2 token exchange. Besides authenticated mode, anonymous calls to Dispatcher are possible too; these yield anonymous calls to the VREs, which may fail eventually, returning the error back to the user.

The approach of Dispatcher acting on the user's behalf only brings the advantage of no need for setup of access of the Dispatcher identity with the downstream services, and the need for detailed mapping (accounting) of these requests to the end users. On the other hand, appropriate policies that Dispatcher can request token exchange for a given list of services still have to be configured with OIDC identity providers.

The authentication layer is implemented with *fastapi-oauth2* middleware and it is configured with Ansible on deployment.

5. Deployment and Operations

The EOSC Data Commons platform is deployed across two environments: a development environment for active development and integration testing, and a production environment serving end users.

Components are packaged as Docker images and automatically published to the GitHub Container Registry by GitHub Actions workflows on every version tag.

5.1. Environments

5.1.1. Development Environment

The development environment runs on two virtual machines hosted at ACC Cyfronet AGH on reduced resources. All platform services are deployed on this pair of VMs, allowing the full stack to be exercised without the footprint of the production setup. A dedicated VPN provides developers with direct access to internal interfaces and services.

Deployments in the development environment are automated: Watchtower monitors the container registry of each component and automatically updates running containers whenever a new image is published, so the environment always reflects the latest build without manual intervention.

5.1.2. Production Environment

The production deployment runs on five dedicated virtual machines provisioned through OpenStack. Every VM is connected through an isolated private network; only the application gateway host (`dc-apps`) exposes public endpoints over HTTP and HTTPS. Administrative interfaces, including the database management panel, OpenSearch Dashboards, and the observability stack, are accessible exclusively from within the project VPN protected by OpenStack security group rules. Each service is packaged as a Docker Compose stack deployed to a dedicated VM. The underlying OpenStack cloud runs on high-performance Ceph storage, providing hardware-level redundancy and data durability for all attached volumes.

All virtual machines run Ubuntu. Each VM has a local root disk included in the flavor. Persistent production data is stored separately on dedicated Cinder block volumes, decoupling data lifecycle from VM lifecycle. The project has 1 TB of Cinder storage allocated in total, with 350 GB currently provisioned, leaving substantial headroom for data growth.

Production deployments are intentionally manual: updates are applied by running Ansible playbooks with an explicitly specified image tag, ensuring that only reviewed and approved versions reach the live environment.

Infrastructure as Code. The entire production environment is defined and managed as code. OpenStack resources — virtual machines, networks, security groups, and block volumes — are provisioned with Terraform, ensuring that the infrastructure topology is version-controlled and reproducible. Service configuration and application deployment are managed with Ansible: idempotent playbooks handle software installation, configuration templating, and Docker Compose stack rollouts across all hosts. This approach eliminates manual configuration drift and allows the environment to be fully recreated from the codebase if needed.

As usage grows, the platform is designed to scale vertically to larger VM flavours or horizontally by introducing clustering for the database and search layers.

Table 16 - Infrastructure for the production environment.

VM	Role	vCPU	RAM	Disk	Cinder Volume
dc-apps	nginx, frontend, backend, MCP	4	16 GB	50 GB	—
dc-db	PostgreSQL	8	32 GB	50 GB	200 GB
dc-search	OpenSearch	8	32 GB	50 GB	100 GB
dc-transformer	Celery, RabbitMQ, ETL	8	32 GB	50 GB	—
dc-obs	Grafana, Prometheus, Loki	2	4 GB	25 GB	50 GB

5.2. Observability and Monitoring

The platform uses a centralised observability stack deployed on the dedicated `dc-obs` host, consisting of Prometheus for metrics collection, Grafana for visualisation and dashboards, Loki for log aggregation, and Alertmanager for alert routing and notification.

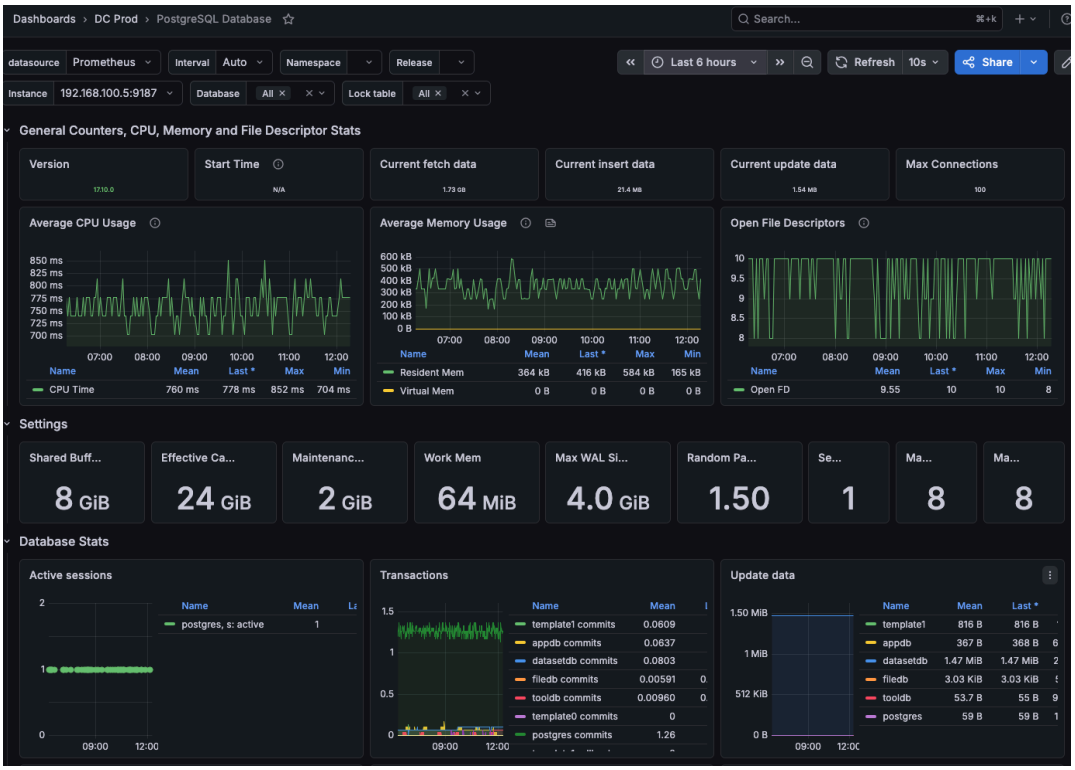

Figure 7- Monitoring dashboard of PostgreSQL database.

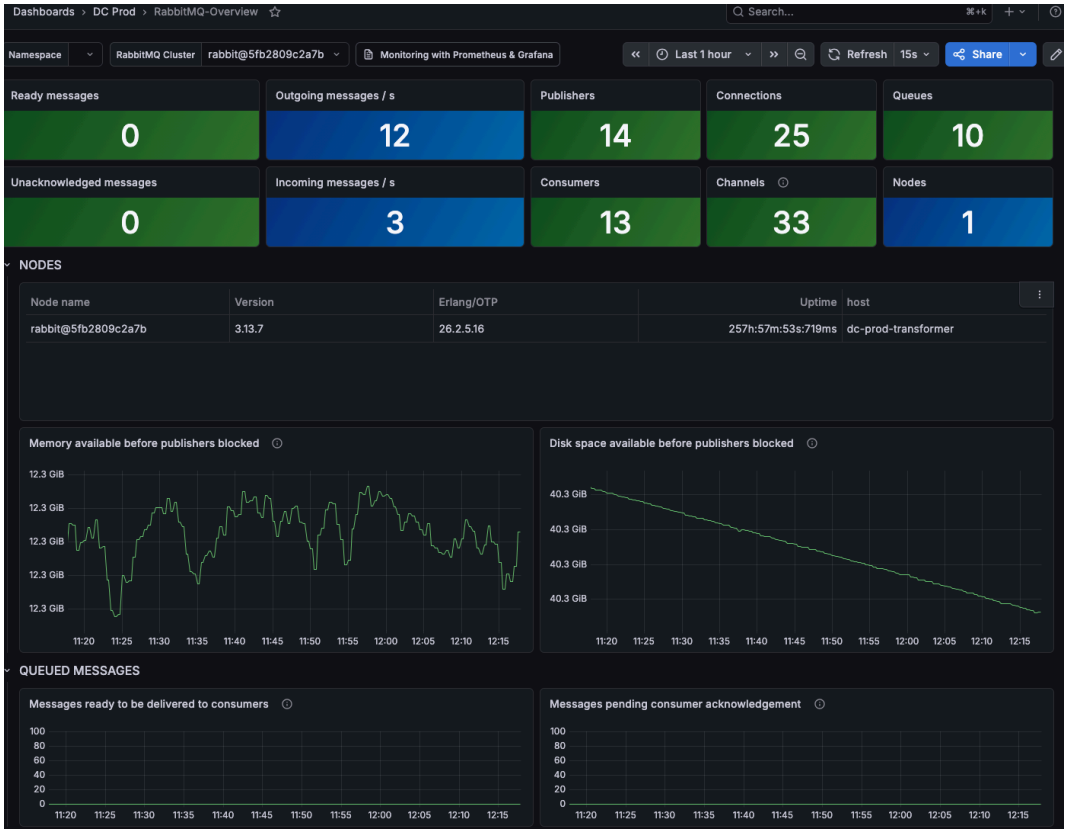

Figure 8 - RabbitMQ Overview.

Every VM in the production environment runs a set of monitoring agents deployed as a Docker Compose stack: node-exporter for host-level metrics (CPU, memory, disk, network), cAdvisor for per-container resource usage, and Promtail for log collection. Promtail discovers running Docker containers automatically via the Docker socket and ships their logs to Loki, tagging each log stream with the service name, Compose project, container name, and host. Additional exporters are deployed where relevant: an nginx metrics exporter on `dc-apps`, a PostgreSQL exporter on `dc-db`, and RabbitMQ's built-in Prometheus endpoint on `dc-transformer`.

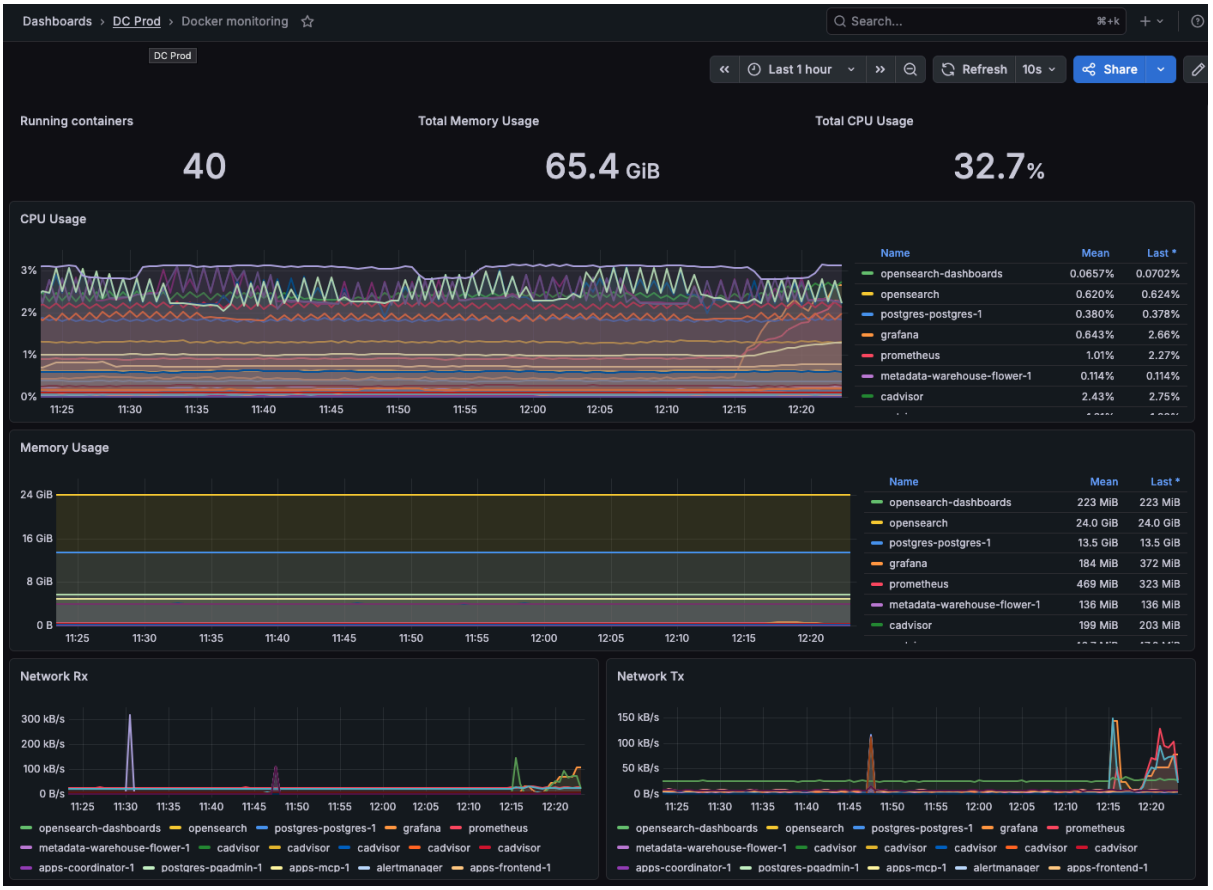

Figure 9 - Docker monitoring. Dashboard tracking container resource usage across all hosts: CPU, memory, network I/O, and container state.

Prometheus scrapes all targets every 30 seconds and retains metrics for 30 days. A set of alerting rules covers the key failure modes: target unavailability, sustained high CPU or memory utilisation, low disk space on any filesystem, and PostgreSQL availability. Alerts are routed through Alertmanager to a dedicated Slack channel, with colour-coded severity levels (warning / critical) and automatic resolution notifications when the condition clears. Critical alerts are re-notified every hour until resolved; inhibition rules suppress lower-severity warnings when a critical alert for the same host is already firing. Grafana dashboards provide a unified view across infrastructure metrics and application logs.

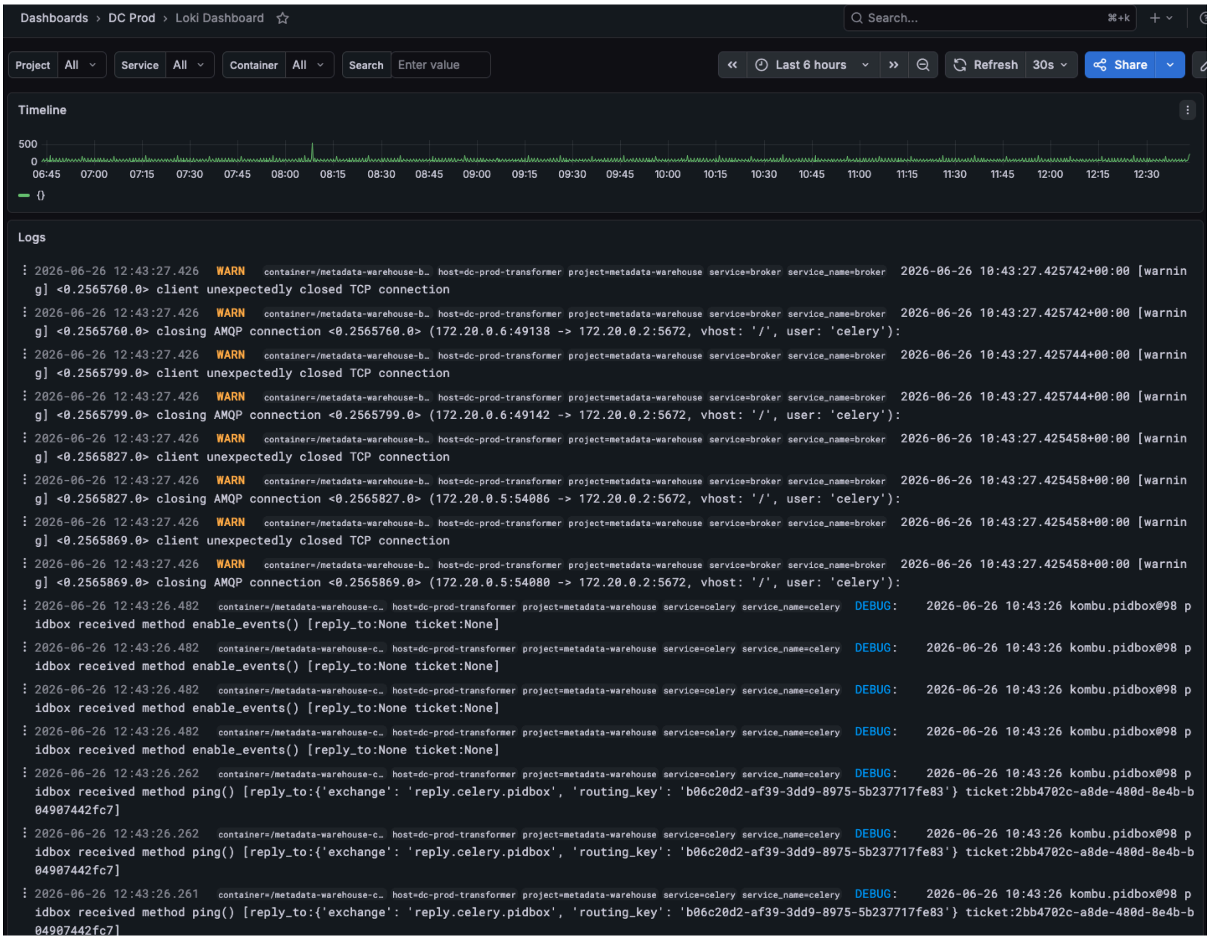

Figure 10 - Loki dashboard for browsing and querying application logs aggregated from all services across the infrastructure.

The User Interface relies on Matomo (hosted on the EGI Matomo instance) for usage analytics. In addition to page views, the UI records custom events for the main user actions - search submissions and AI-mode toggles, dataset play and source-link clicks, citation open/copy/download actions, filter apply and remove actions, sign-in gate triggers, and "Were these results helpful?" thumbs-up/down feedback on the results page - so the team can see which features are used and how users move through the platform.

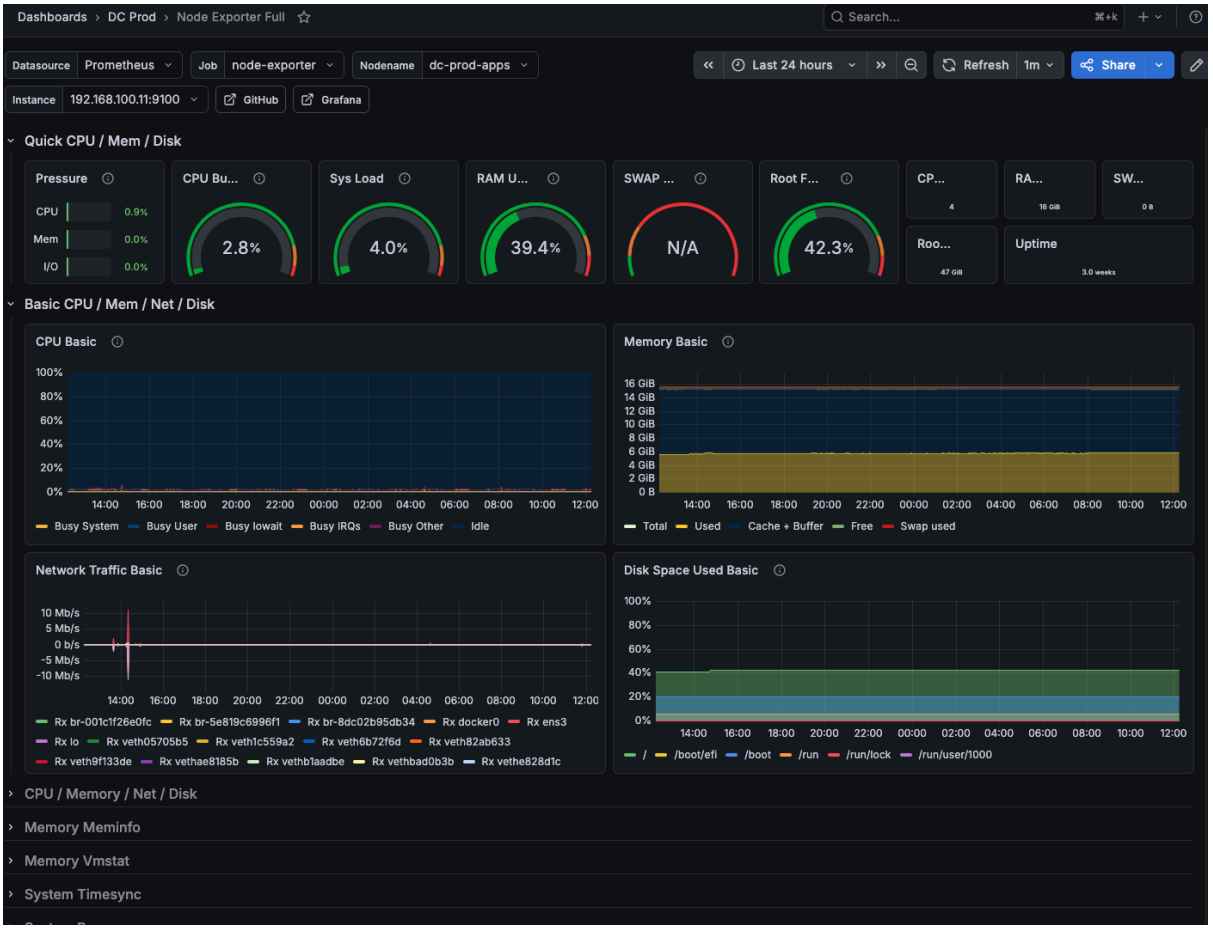

Figure 11 - Node Exporter Full. Dashboard monitoring host-level system metrics: CPU usage, memory, disk I/O, network throughput, and system load across all VMs.

5.3. Security and Reliability

Access control and network isolation. The production network topology limits the attack surface: only the `dc-apps` host is reachable from the public internet, on ports 80 and 443. All other VMs are on a private network with no direct external exposure. SSH access and all administrative interfaces are restricted to the project VPN by OpenStack security group rules. TLS termination is handled by nginx on `dc-apps` using certificates issued by Let's Encrypt and renewed automatically.

Authentication. User-facing services are protected by federated authentication through EGI Check-in using the OpenID Connect Authorisation Code flow with PKCE, as described in Section 2.1.3. Session tokens are stored in HttpOnly cookies and silently refreshed on expiry. The search endpoint is available to anonymous users; conversation history and tool execution require authentication.

Data persistence and backups. Persistent data is stored on Cinder block volumes backed by Ceph, independent of VM lifecycle. Daily snapshots of all virtual machines are taken at the OpenStack level within the primary hosting site, providing rapid

recovery from VM failure or misconfiguration. PostgreSQL and OpenSearch data are additionally backed up daily using restic and stored in S3-compatible object storage operated by Cyfronet AGH at a geographically separate facility, ensuring that a copy of all critical data is available independently of the primary site. Should the infrastructure itself need to be rebuilt, the full environment — from networking and VMs to service configuration — can be reproduced from the Terraform and Ansible codebase without manual intervention.

Operational reliability. All containers are configured to restart automatically on failure. Platform resource usage is continuously monitored (Section 5), and the infrastructure is designed to scale vertically or be clustered horizontally as demand grows.

Maintenance and support. The platform is maintained on a best-effort basis during business hours. The operations team monitors the observability stack continuously (Section 5) and aims to respond promptly to critical alerts and service disruptions.

6. Conclusions and Future Plans

This deliverable documents the first production release of the EOSC Data Commons platform. The release provides the core infrastructure required for metadata aggregation, transformation, indexing, dataset discovery, semantic search, tool matchmaking, and workflow orchestration.

The platform successfully integrates metadata from multiple repositories, establishes a scalable Metadata Warehouse based on PostgreSQL and OpenSearch, and delivers the key software components required to support EOSC Data Commons services in a production environment.

This release represents an important milestone for the project and provides a solid foundation for future platform expansion, repository onboarding, and service enhancements.

The first production release establishes the foundation of the EOSC Data Commons platform. Future work will focus on expanding the number of integrated repositories and their harvesting endpoints, improving metadata quality and enrichment processes, and enhancing search and discovery capabilities.

A dedicated staging environment will be established to enable comprehensive testing of services and their interdependencies before deploying changes to the production environment. This will further improve release quality, reliability, and operational stability.

The project team will continuously monitor platform usage, user engagement, and community feedback in order to identify areas for improvement and guide future development priorities. Ongoing efforts will focus on extending the platform's functionality while further reducing latency, improving performance, and maintaining a high level of service reliability, and continuously improving the documentation of the system components. These improvements will increase the TRLs of the system components. Ethical-by-design principles will be applied throughout future development and deployment activities with the support of the Ethics assessment deliverables of the project expected in September 2026.

.

Acronyms

Acroynm	Description
AAI	Authentication and Authorisation Infrastructure
API	Application Programming Interface
BM25	Best Matching 25 (ranking algorithm)
CRON	Command Run On (scheduled task mechanism)
CSRF	Cross-Site Request Forgery
DOI	Digital Object Identifier
EOSC	European Open Science Cloud
ETL	Extract, Transform, Load
EGI	European Grid Infrastructure
gRPC	gRPC Remote Procedure Calls
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSONB	Binary JSON (PostgreSQL JSON data type)
kNN	k-Nearest Neighbours
LLM	Large Language Model
MIME	Multipurpose Internet Mail Extensions
MCP	Model Context Protocol
OAI-PMH	Open Archives Initiative Protocol for Metadata Harvesting
OIDC	OpenID Connect
PKCE	Proof Key for Code Exchange
RRF	Reciprocal Rank Fusion
TRL	Technology Readiness Level
UI	User Interface
URI	Uniform Resource Identifier

Acroynm	Description
URL	Uniform Resource Locator
VRE	Virtual Research Environment
WP	Work Package
XSLT	Extensible Stylesheet Language Transformations

Appendix 1. Metadata Warehouse Databases

appDB

The schema backs the search application itself (rather than the harvesting pipeline), with four tables covering rate limiting, user identity, and chat history.

rate_limits is a standalone counter table used to throttle requests. Each row tracks a rate-limit *key* (e.g. a user, IP, or API token), the current request *count*, and the *window_end* timestamp at which the counter resets. The *key* is the primary key, so each limited entity has exactly one active window row that is incremented and reset as time passes.

users table holds the top-level identity entities, one row per authenticated user. The primary key *sub* is the OIDC subject identifier from the identity provider, alongside optional *email*, *name*, and *username* descriptive fields and a *created_at* timestamp recording first sign-in.

conversations table sits below *users* (one or many per user) and represents a single chat thread. Each conversation is identified by the composite key (*user_id*, *thread_id*), carries a human-readable *label*, and tracks *created_at* / *updated_at* timestamps.

messages is the leaf table storing the individual messages exchanged within a conversation. Each row has an auto-incrementing id, the type of message (e.g. human, ai, tool), and the message content stored as JSONB to accommodate arbitrary structured payloads. The composite foreign key to conversations with ON DELETE CASCADE ties messages to their thread, so removing a conversation (or the parent user) cleans up all associated messages.

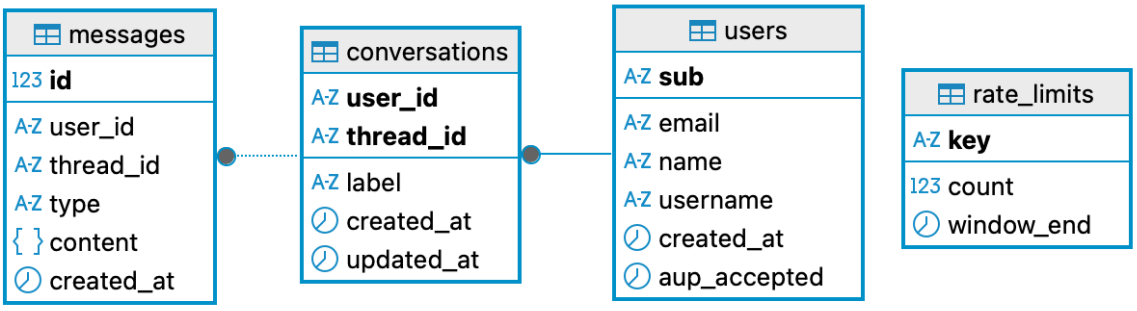

Figure 12 - appDB tables.

datasetDB

The schema models a metadata harvesting pipeline with five core tables, moving data from raw harvest_events through transformation to indexed records (job type *transform_batch*).

repositories table contain the top-level entities, representing a data provider such as DANS, Zenodo, HAL etc. Each repository has a unique short code, an activity flag, and basic descriptive metadata.

endpoints table sits below repositories (one or many per repository) and describes a specific harvesting target — its URL, protocol (OAI-PMH), scientific discipline, configurable harvest parameters stored as JSONB, and a harvest schedule interval controlling how frequently the endpoint should be polled.

harvest_runs table tracks individual harvest run executions against an endpoint. A run tracks its time window (from_date / until_date), status (open, closed, failed), and counters for records harvested, created, updated, and deleted. A partial unique index enforces that only one run per endpoint can be open at a time.

harvest_events is the staging table — each row represents a single metadata record received during a harvest run, carrying the raw XML, datestamp, a deletion flag, and an optional error message written back by the Transformer when processing fails. The unique constraint on (endpoint_id, harvest_run_id, record_identifier) prevents duplicate events within a run. Also if a transformation fails, it can always be retried without harvesting the data again.

records table is the final destination for successfully transformed records. It stores the normalised DataCite JSON, the original raw XML, DOI or URL (at least one required by constraint), vector embeddings with their model name, and OpenSearch sync state. The unique constraint on (endpoint_id, record_identifier) means a record is identified by its origin endpoint and its identifier, with upsert semantics on re-harvest of the same record (updates).

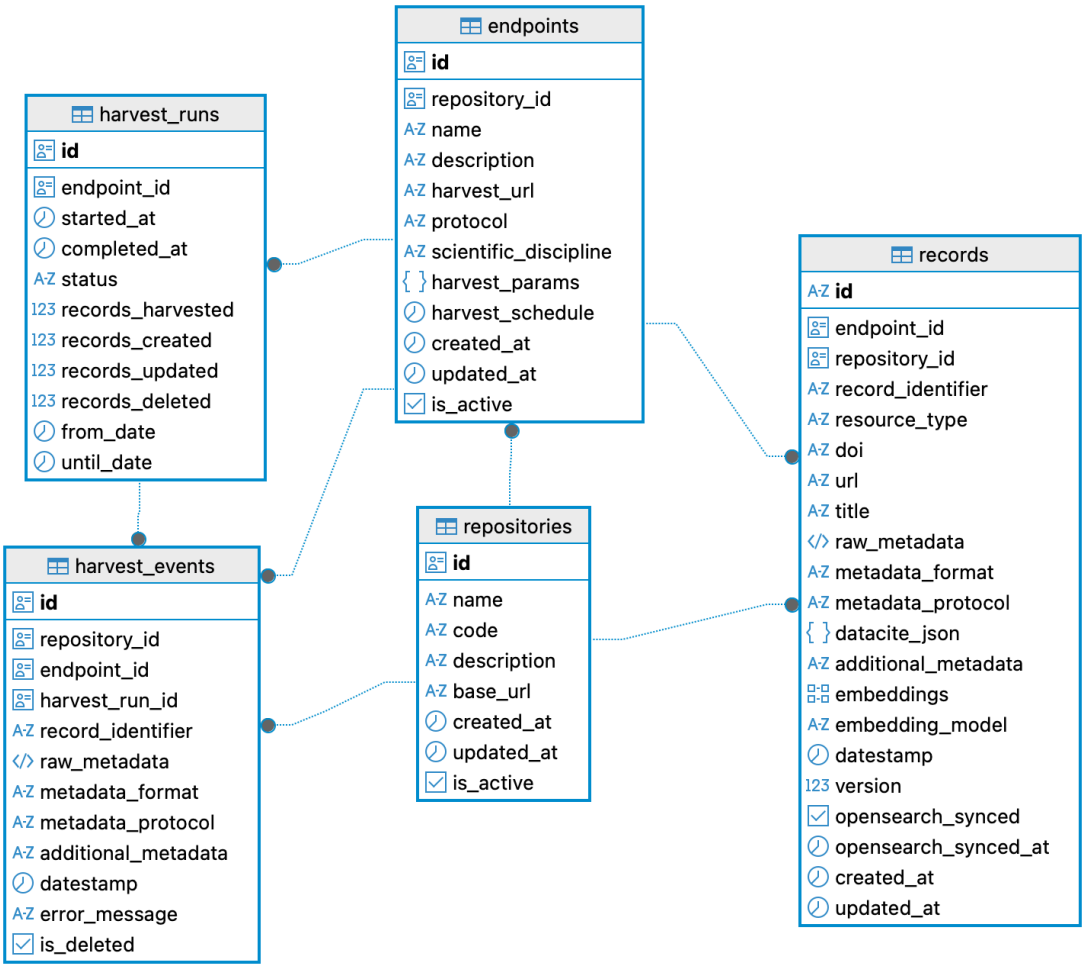

Figure 13 - datasetDB tables.

fileDB

The schema contains a single table, record_files, populated by the *add_file_metadata* job type.

record_files stores file-level metadata for individual files belonging to a harvested dataset. Each row represents one file linked to its parent record identified by harvest_url and record_identifier. The unique constraint on (harvest_url, record_identifier, file_identifier) prevents duplicate entries per file per record. A check constraint enforces that checksum_type and checksum_value are either both present or both absent.

Each row captures the file name, MIME type, size in bytes, download URL, optional checksum, version, and creation and modification timestamps as reported by the source. identifier_type and identifier_granularity describe how the file is identified at the source (DOI, URL, URN, etc.) and at what level of granularity.

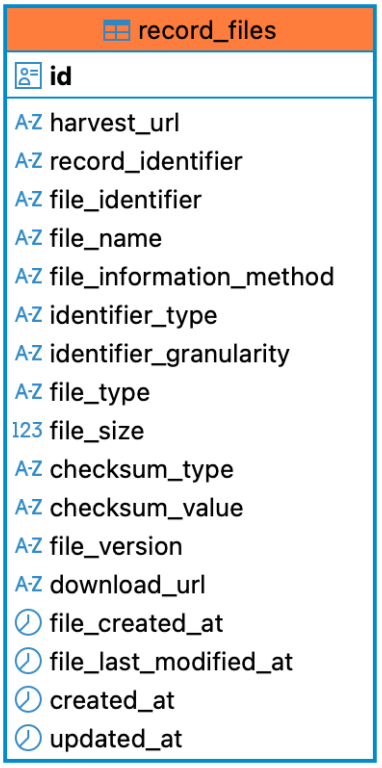

Figure 14 - fileDB record_files table.

toolDB

Tooldb contains one table, tool_generic, that serves as a central registry for software tools and services. It stores the core description technical metadata, location URL, input and output file descriptions, tool types, tags and keywords. Of particular importance is the input file information, which is needed to enable the file-based matchmaking for Release 1. Each record represents a tool e.g. a Galaxy Workflow, Python Notebook or a Container. A tool is identified globally by the URI and is used to prevent duplicate tools.

Designing a single schema that can accurately represent all tool types is challenging. To accommodate the diversity of workflows, notebooks, containers, and other computational artefacts, the schema includes several flexible fields intended for use by downstream services such as the request packager. In particular, the *raw_definition* field stores the original tool definition, for example, a Galaxy workflow definition or a Python notebook. The *raw_metadata* field preserves the metadata exactly as retrieved from the source repository, such as WorkflowHub, ensuring that no information is lost during the harvesting and normalisation process.

Since users are also able to add their own tools, a *created_by* field is added to record the EGI user id. This allows the user to update and delete his tool record.

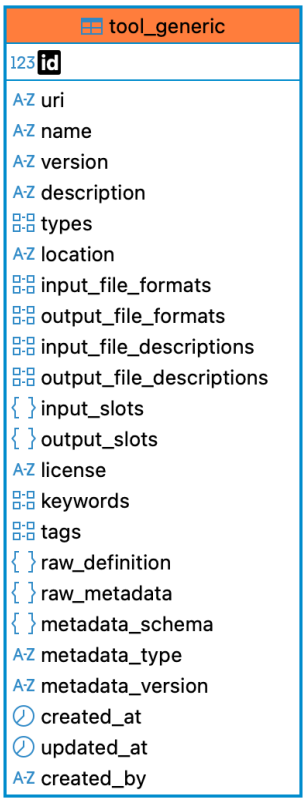

Figure 15 - toolDB tool_generic table.