



**MESHCHERYAKOV**  
**LABORATORY of**  
**INFORMATION**  
**TECHNOLOGIES**



# **Workload Management System for SPD Online filter**

VII SPD Collaboration Meeting. 23.05.2024  
Nikita Greben, MLIT

## SPD Online Filter as a middleware software

«**SPD OnLine filter**» – hardware and software complex providing multi-stage high-throughput processing and filtering of data for SPD detector.

## ❖ Data management system

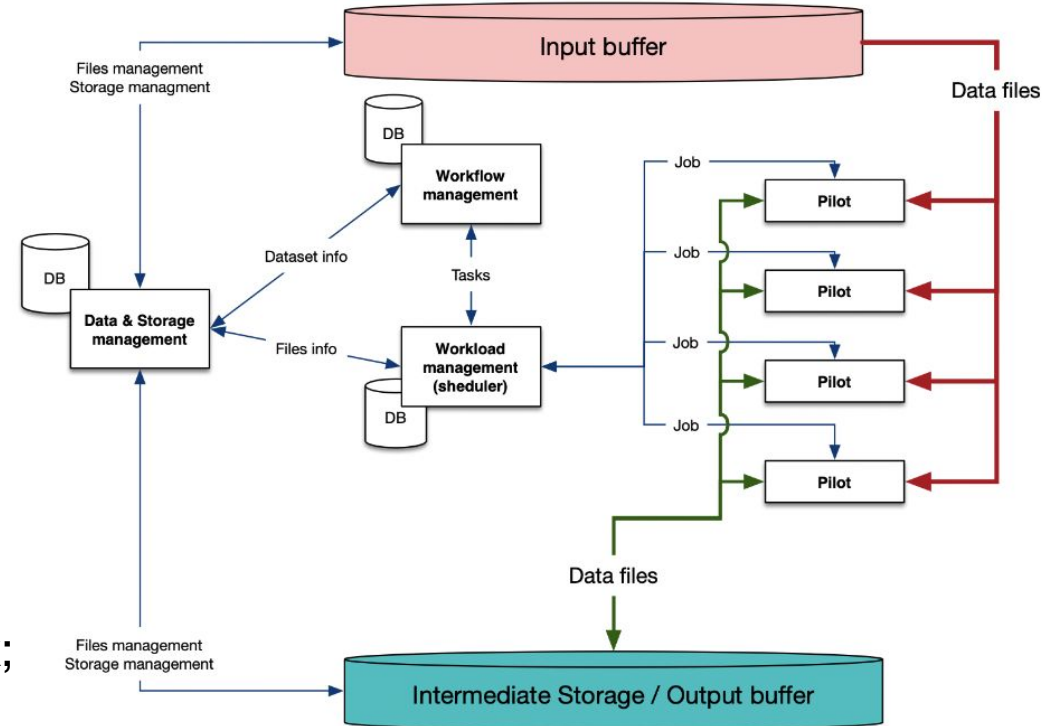
- Data lifecycle support (data catalog, consistency check, cleanup, storage);

## ❖ Workflow Management System:

- Define and execute processing chains by generating the required number of computational tasks;

❖ **Workload management system:**

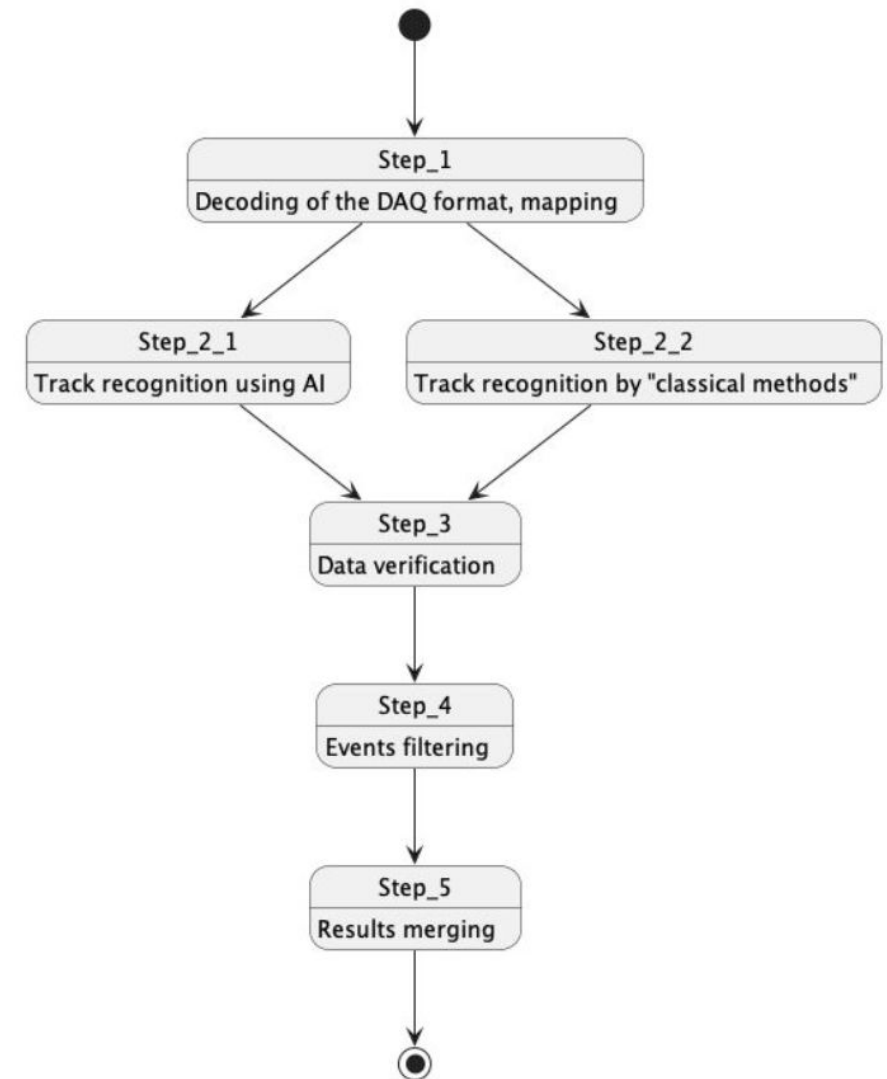
- Create the required number of jobs to perform the task;
- Dispatch jobs to working nodes via pilots;
- Control job execution;
- Pilot control (identification of "dead" pilots);
- Efficient resource management;



## Architecture of SPD Online Filter

# High-throughput computing

- **HTC** is defined as a type of computing that simultaneously executes numerous simple and computationally independent jobs to perform a data processing task.
- Since each data element can be processed simultaneously, this can be applied to data aggregated by a data acquisition system (DAQ).
- To ensure efficient utilization of computational resources, data processing should be multi-stage:
  - One stage of processing → **task**
  - Processing a block of data (file) → **job**



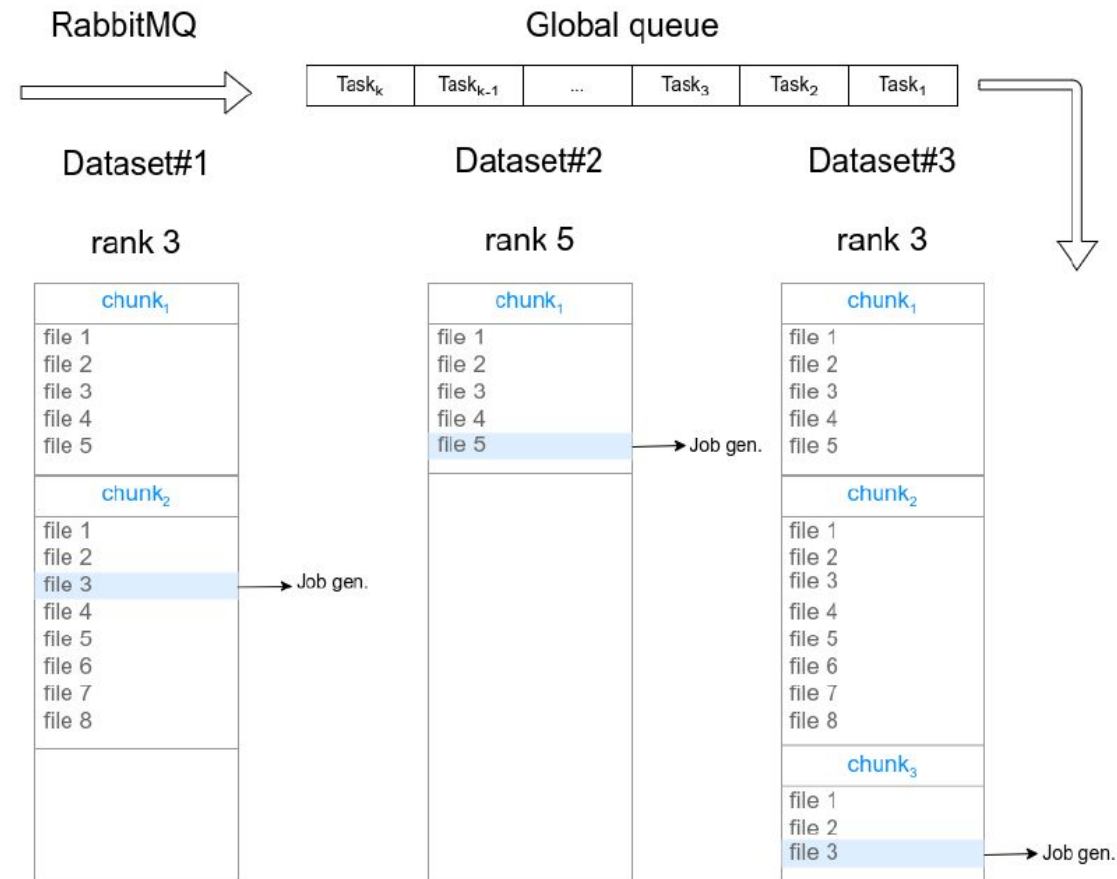
# Task and job definition

- A **task** is a workload unit responsible for processing a block of homogeneous data - **dataset**.
- A processing request is a set of input data, which may consist of multiple files, and a handler.
- The completion criterion of the **task** is the processing of the data block.
- The **Workflow Management System** is responsible for defining and executing workflows, as well as defining a processing request, which is a **task**.
  
- A job (payload) is a unit of work that processes a unit of data (file).
- The unit responsible for processing a single file in terms of workload is called a job.
- The **Workload Management System** (WMS) is responsible for generating jobs, sending them to compute nodes, and executing them.

# Workload management system requirements

The key requirement - systems must meet the high-throughput paradigm.

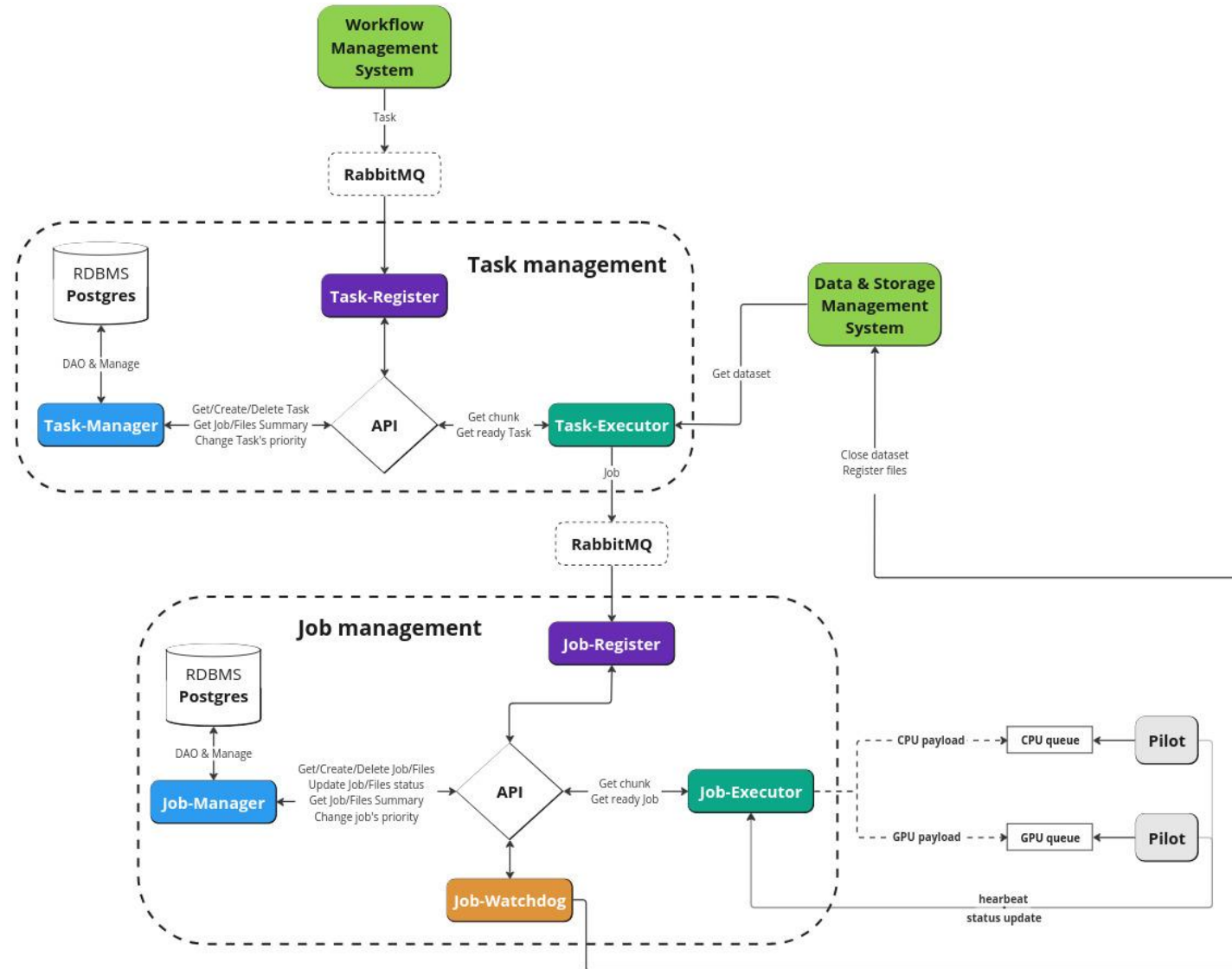
- ❖ **Task registration:** formalized task description, including job options and required metadata registration.
- ❖ **Jobs definition:** generation of required number of jobs to perform task by controlled loading of available computing resources.
- ❖ **Jobs execution management:** continuous job state monitoring by communication with pilot, job retries in case of failures, job execution termination.



Forming jobs based on dataset contents, one file per one job

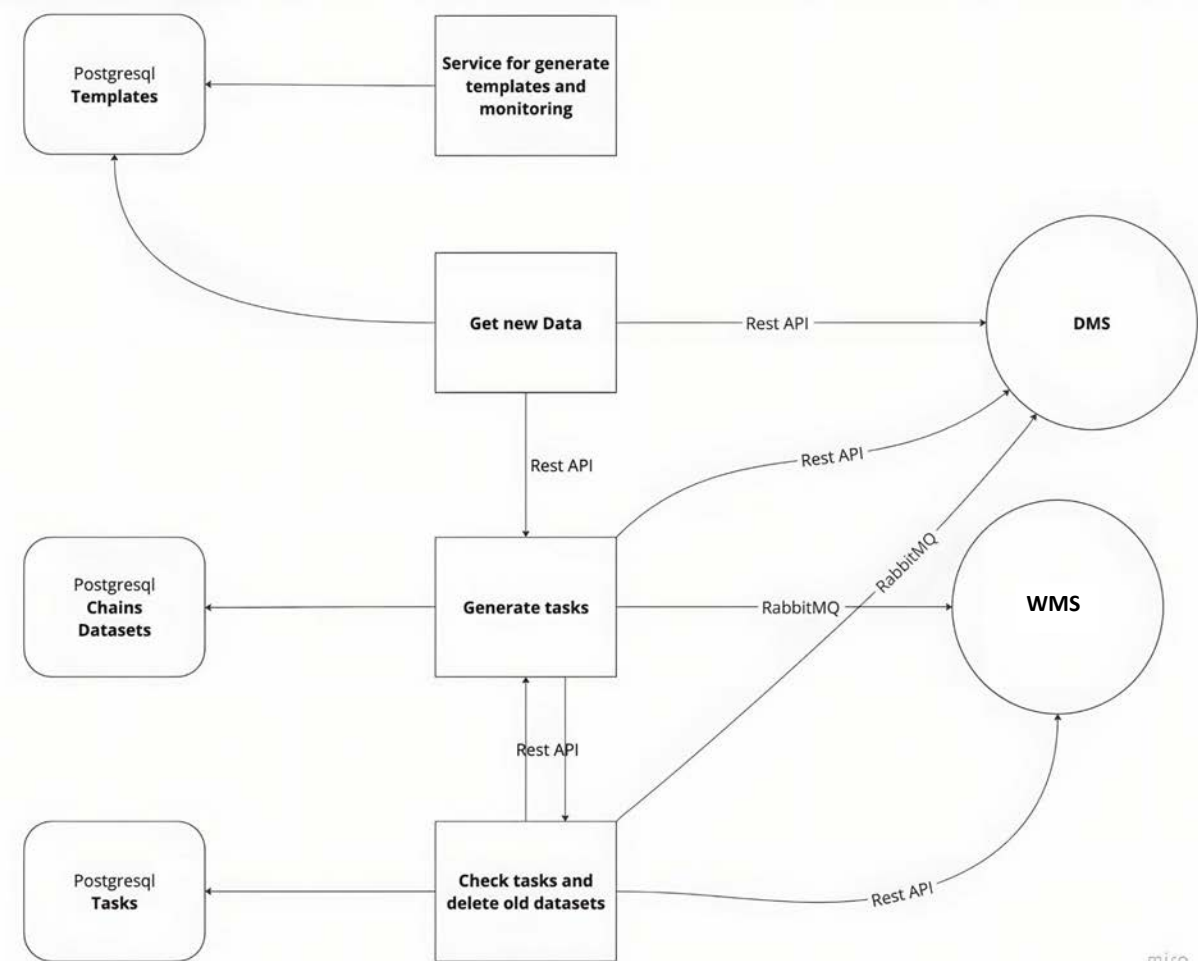
# Architecture and functionality of Workload Management System

- ❖ **task-manager** – implements both external and internal REST APIs. Responsible for registering tasks for processing, cancelling tasks, reporting on current output files and tasks in the system.
- ❖ **task-executor** – responsible for forming jobs in the system by dataset contents.
- ❖ **job-manager** – accountable for storing jobs and files metadata, as well as providing a REST API for the executed jobs.
- ❖ **job-executor** – responsible for distribution of jobs to pilot applications, updating the status of jobs, registering output files and closing the dataset.
- ❖ **pilot** – responsible for running jobs on compute nodes, organizing their execution, and communicating various information about their progress and status.



# Interaction with the Workflow Management System

- Registration of a task for processing
  - **WfMS** passes the task description into message queue
- Summary of current intermediate properties of jobs/files in the system
  - Aggregated information about the status of each job/file for further decision making
- Task cancellation
  - Based on the decision made on the **WfMS** (*too many errors occurring*) or operator side
- Change priority of a task
  - Control management





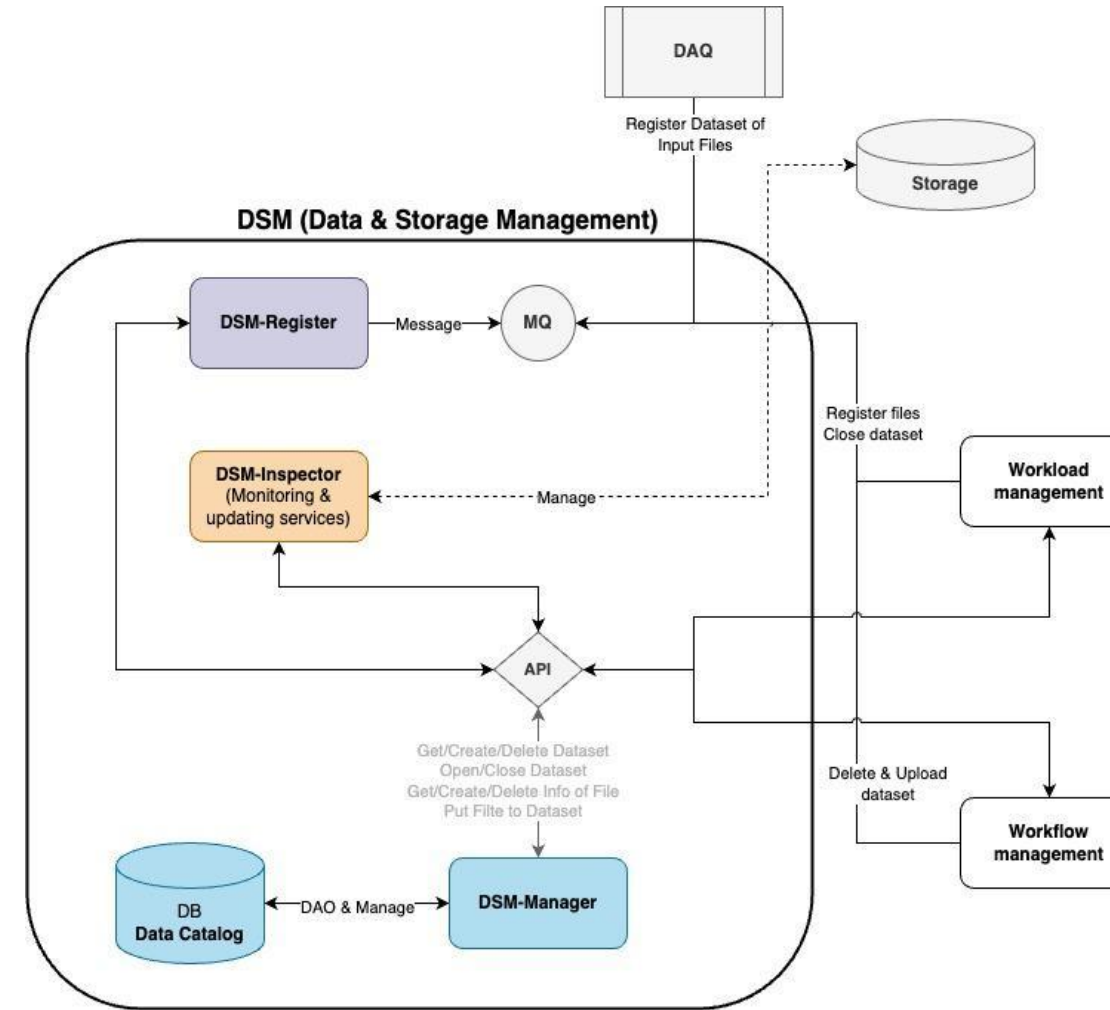
# Interaction with the Data Management System

Routing Key	Msg	Algo
<b>dataset.close</b>	Dataset info <ul style="list-style-type: none"> <li>Dataset UID</li> <li>File check list (file names)</li> </ul>	Request the registered files in the dataset. If they match the checklist, set the status to <b>CLOSED</b> . Otherwise, return the messages back to the queue for deferred execution.
dataset.upload	Dataset UID	Marking dataset for uploading ( <b>TO_UPLOAD</b> )
dataset.delete	Dataset UID	Marking dataset for deletion ( <b>TO_DELETE</b> )

Signature and algorithm of message receiving gateways for the **dsm-register** service

Within a **Workload Management System**, there are several scenarios for interacting with the data management system:

- Obtain information about dataset contents for forming jobs from **DSM-Manager (Data Catalog REST API)**
- Register files in datasets after executing payload on compute node – **DSM-Register (Data Registration)**
- Close dataset after cancellation or sufficient number of successfully processed files – **DSM-Register**



Architecture of Data Management



# Database design

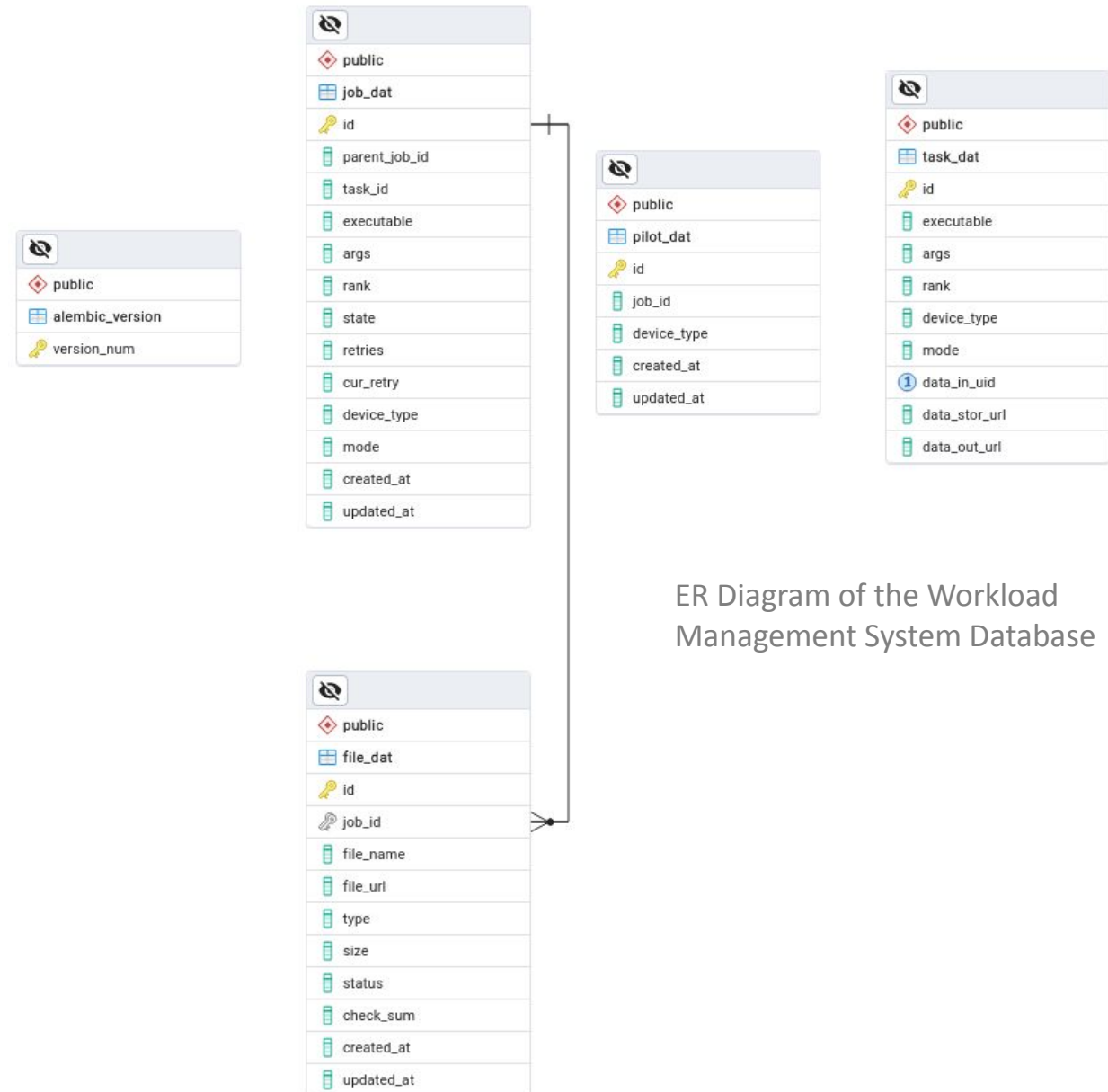
## RDBMS - PostgreSQL 16

### Tables:

- ❖ **alembic\_version** – managing and tracking database schema changes
- ❖ **file\_dat** – a directory specifying the output files and logs generated on the pilot
- ❖ **job\_dat** – jobs currently being processed in the system
- ❖ **task\_dat** – current tasks in the system

### Extra mechanisms:

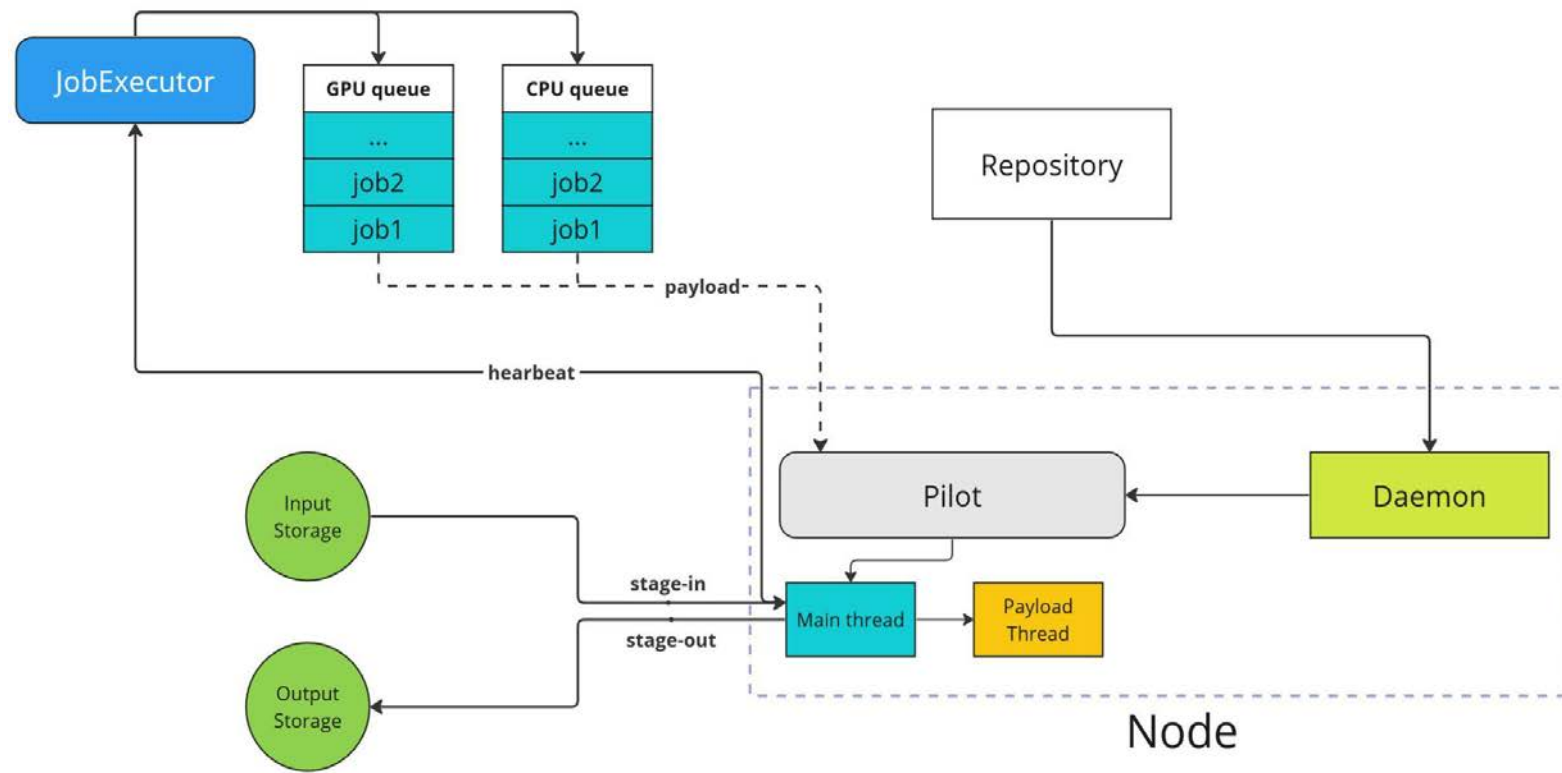
- ❖ **Indexes** – on filter fields for optimization of operations
- ❖ **Procedures** – task and job generation for test purposes
- ❖ **Triggers** – rank update logic
- ❖ **Decomposition** – single database per microservice



ER Diagram of the Workload Management System Database

# Internal design of Pilot Agent

- The agent application is deployed on a compute node and consists of the following two components: a UNIX daemon and the pilot itself.
- The UNIX daemon's objective is to run the next pilot by downloading an up-to-date version from the repository.
- Pilot itself is a multi-threaded Python application responsible for
  - Receiving and validating jobs from the message broker.
  - Downloading input files for the payload stage and uploading the result files to the output storage.
  - Launching a subprocess to execute a payload (decoding DAQ format, track recognition algorithm, etc.)
  - Keeping the upstream system informed of the current status of the payload and the pilot itself via heartbeat/status updates during each phase of pilot execution.



- Compute nodes differ only in the availability of specialized co-processors (GPUs) and are assigned to the appropriate message broker based on the computational needs of the job.
- Regardless of the presence of an error, when the pilot finishes, the UNIX daemon launches a new instance of the pilot.

# Interaction with the Pilot Agent

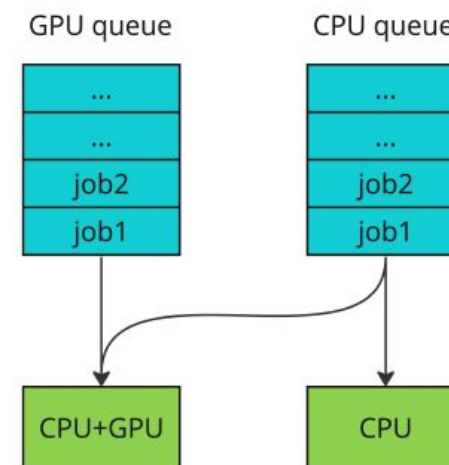
- ❖ Pilot has a series of preprocessing stages before running a job itself:
  - a. start logging
  - b. read configuration
  - c. getting a job from message queue
  - d. validation
- ❖ After those steps the Pilot launches another thread where it does
  - a. environment setup script
  - b. copying files locally from the input storage
  - c. starts execution of a job itself in a separate sub-process
  - d. analysis of the result of a job
  - e. copying output data and logs to storage
  - f. sends regular messages to **WMS**
  - g. cleaning up the local environment
- ❖ Pilot sends status-update message at any point of internal changes
- ❖ **WMS** may terminate the job if the corresponding task is cancelled or if an error occurs.
- A detailed job status model has been described
- Error codes introduced
- Pilot ran through all major stages of the job execution (**DAG**)
- Pilot at this stage runs a script that does a basic hash compute
- Further debugging needed

Two communication channels:

- HTTP (aiohttp)
- AMQP (message broker - RabbitMQ)

Two types of nodes:

- Multi-CPU
- Multi-CPU + GPU



<b>Common</b> <ul style="list-style-type: none"><li>➤ Python 3.12</li><li>➤ docker compose - running multi-container applications</li></ul>	<b>Frameworks</b> <ul style="list-style-type: none"><li>➤ aio-pika (RabbitMQ + asyncio) - asynchronous API with RabbitMQ</li><li>➤ FastAPI + uvicorn</li></ul>
<b>DB</b> <ul style="list-style-type: none"><li>➤ PostgreSQL - RDBMS</li><li>➤ Alembic (Migration)</li><li>➤ SQLAlchemy 2.0</li><li>➤ asyncpg - Postgres DBAPI</li></ul>	<b>Extra</b> <ul style="list-style-type: none"><li>➤ aiohttp - asynchronous HTTP client/server framework</li><li>➤ Pydantic - validate and serialize data schemes</li><li>➤ pytest-asyncio - test purposes</li></ul>

# Key results

## Design of services:

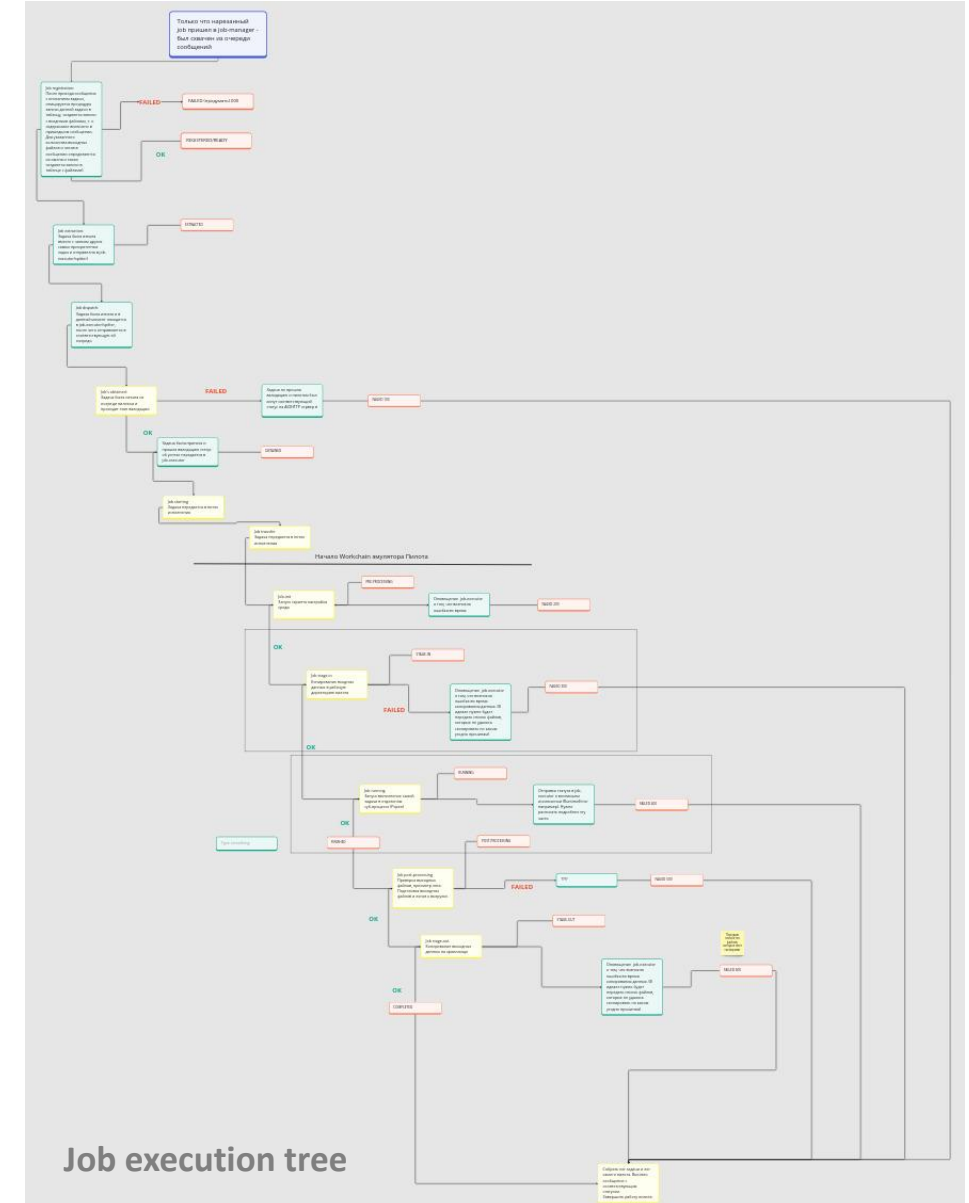
- Implemented a mechanism for declaring the data model in the database based on ORM and migration scripts;
- Designed and implemented a list of required REST API methods and their signatures;
- Configured CD tools (build and deployment) on the JINR LIT infrastructure;
- Designed inter-service interaction scenarios;
- Redesigned Pilot internal architecture;

## Prototype of services:

- Run through all job execution state model, debugging interactions with the pilot;
- Job management subsystem is the most advanced: most interactions implemented and being tested, job-watchdog microservice is being developed;
- Pilot is in active stage of development (*Leonid Romanychev SPbU*).

# Next major steps

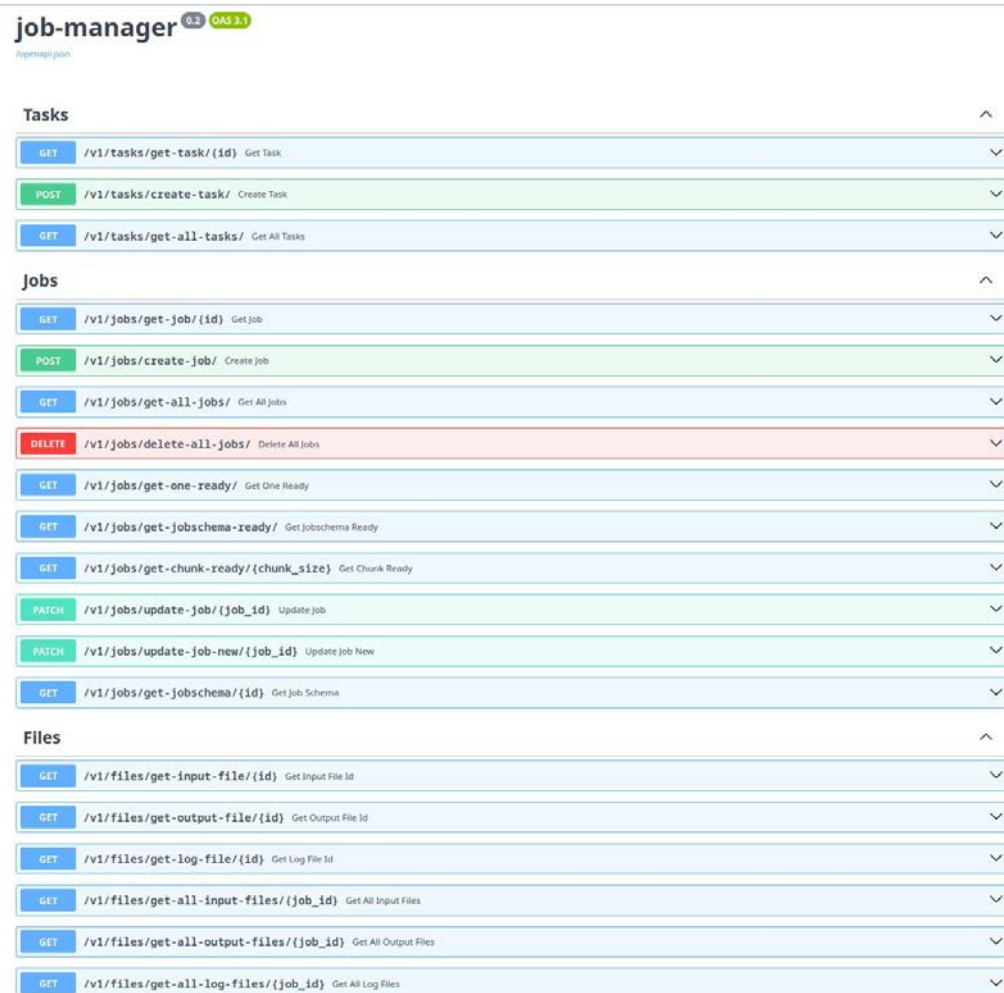
- **Task processing**
  - Implementing task-partitioning algorithm.
  - Closing datasets for DSM.
  - Execute the entire workchain set up on the level of **WfMS**.
- **Logging**
  - Currently, each microservice's logs are mapped to the host via a shared file system between Docker and the host.
  - Ideally – **ELK** (Elastic-Logstash-Kibana) stack to build a log analysis platform.
- **Configuration**
  - Consider to centralize some of the shared configurations across multiple services.
- **Documentation**
  - Given the increasing complexity of the internal logic of the software, it is necessary to document each step of the development, for example: job state model could grow in complexity.
- **Metrics and monitoring**
  - For example, service query-per-second, API responsiveness, service latency etc.
  - InfluxDB, Prometheus, Graphana.



Job execution tree

# Prototyping Job-Manager (API)

- The chosen framework for building the service is FastAPI + Uvicorn asynchronous framework
- A basic set of CRUD operations on data in the form of REST API is developed.
- API description autogeneration according to OpenAPI 3.0 specification is implemented (available in Swagger UI at <server address>/docs)



**job-manager** 0.2 OAS 3.1

**Tasks**

- GET /v1/tasks/get-task/{id} Get Task
- POST /v1/tasks/create-task/ Create Task
- GET /v1/tasks/get-all-tasks/ Get All Tasks

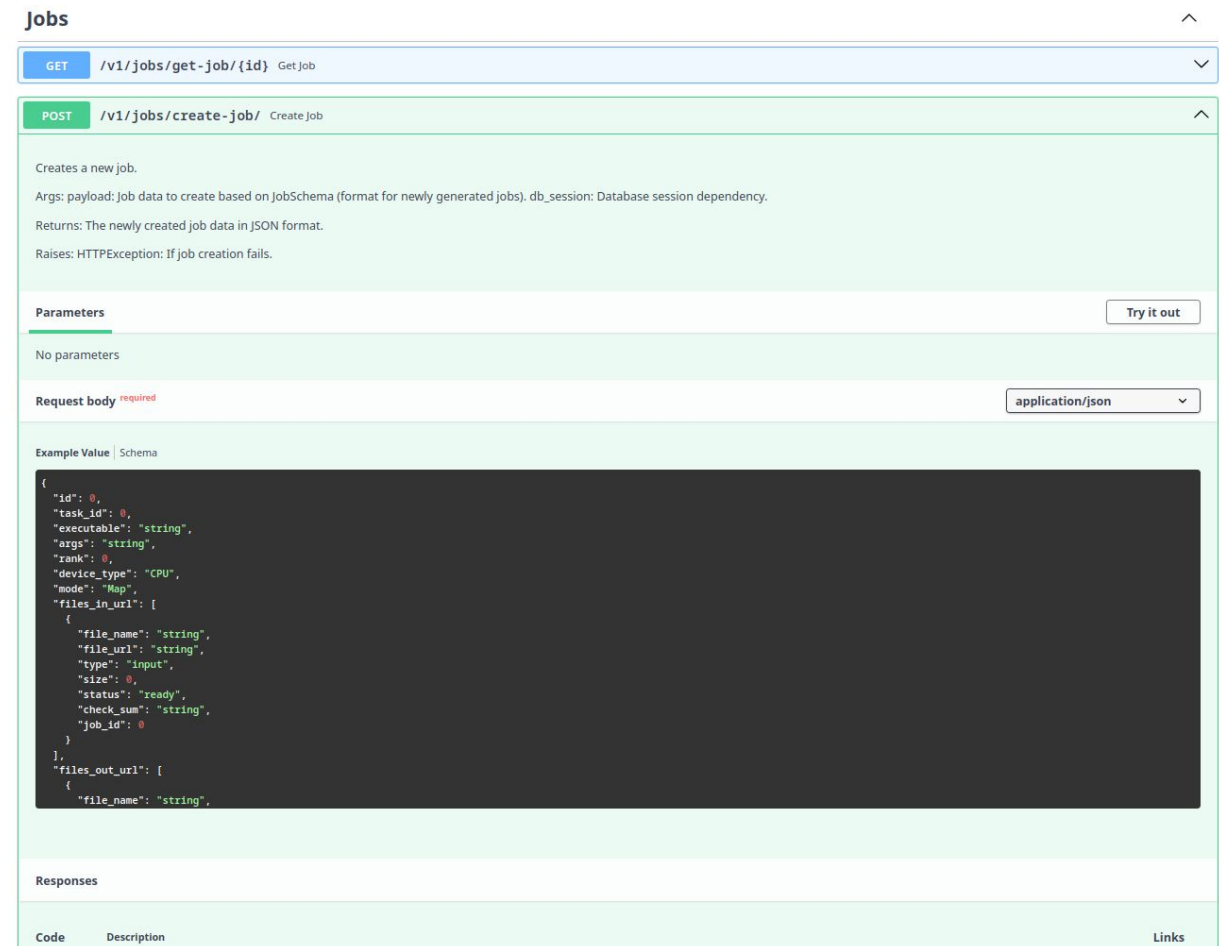
**Jobs**

- GET /v1/jobs/get-job/{id} Get Job
- POST /v1/jobs/create-job/ Create Job
- GET /v1/jobs/get-all-jobs/ Get All Jobs
- DELETE /v1/jobs/delete-all-jobs/ Delete All Jobs
- GET /v1/jobs/get-one-ready/ Get One Ready
- GET /v1/jobs/get-jobschema-ready/ Get Jobschema Ready
- GET /v1/jobs/get-chunk-ready/{chunk\_size} Get Chunk Ready
- PATCH /v1/jobs/update-job/{job\_id} Update Job
- PATCH /v1/jobs/update-job-new/{job\_id} Update Job New
- GET /v1/jobs/get-jobschema/{id} Get Job Schema

**Files**

- GET /v1/files/get-input-file/{id} Get Input File Id
- GET /v1/files/get-output-file/{id} Get Output File Id
- GET /v1/files/get-log-file/{id} Get Log File Id
- GET /v1/files/get-all-input-files/{job\_id} Get All Input Files
- GET /v1/files/get-all-output-files/{job\_id} Get All Output Files
- GET /v1/files/get-all-log-files/{job\_id} Get All Log Files

Swagger UI with job-manager service API description



**Jobs**

GET /v1/jobs/get-job/{id} Get Job

POST /v1/jobs/create-job/ Create Job

Creates a new job.

Args: payload: Job data to create based on JobSchema (format for newly generated jobs). db\_session: Database session dependency.

Returns: The newly created job data in JSON format.

Raises: HTTPException: If job creation fails.

**Parameters**

No parameters

**Request body** required application/json

**Example Value** | Schema

```
{
  "id": 0,
  "task_id": 0,
  "executable": "string",
  "args": "string",
  "rank": 0,
  "device_type": "CPU",
  "mode": "Map",
  "files_in_url": [
    {
      "file_name": "string",
      "file_url": "string",
      "type": "input",
      "size": 0,
      "status": "ready",
      "check_sum": "string",
      "job_id": 0
    }
  ],
  "files_out_url": [
    {
      "file_name": "string",

```

**Responses**

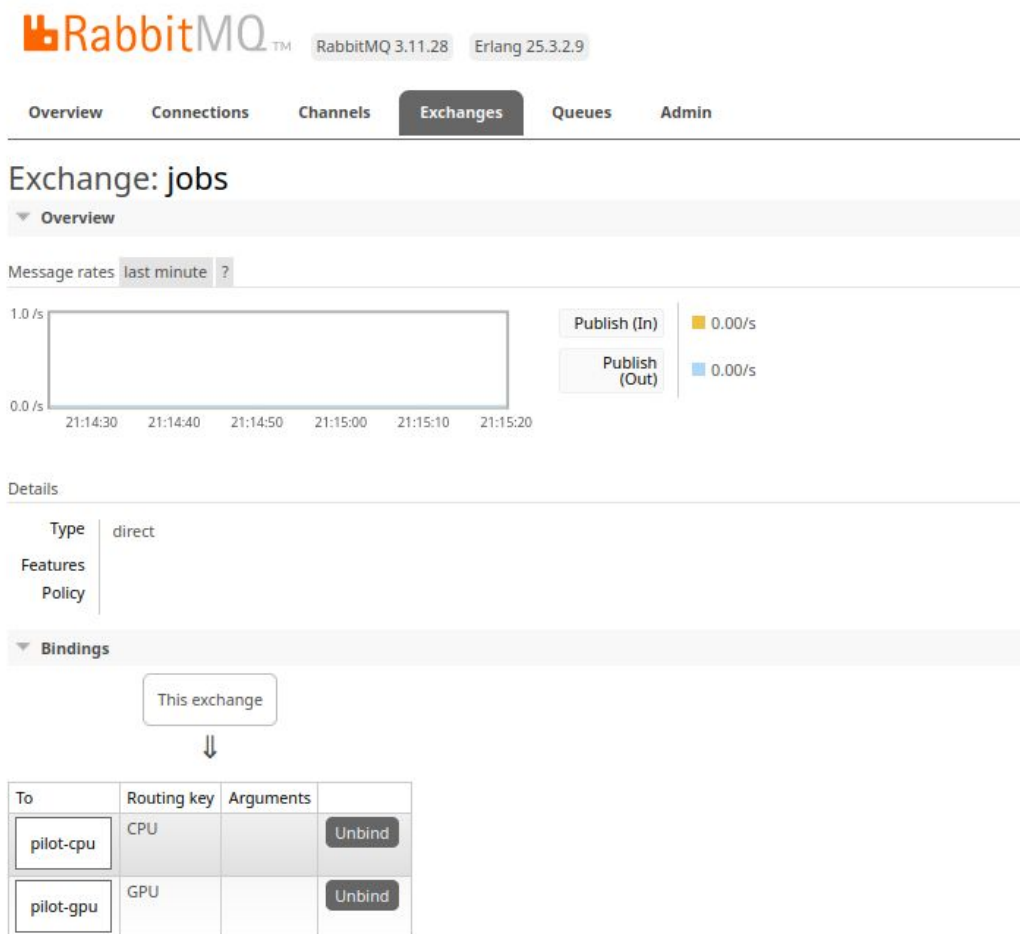
Code Description Links

Example of a service call to post a new job

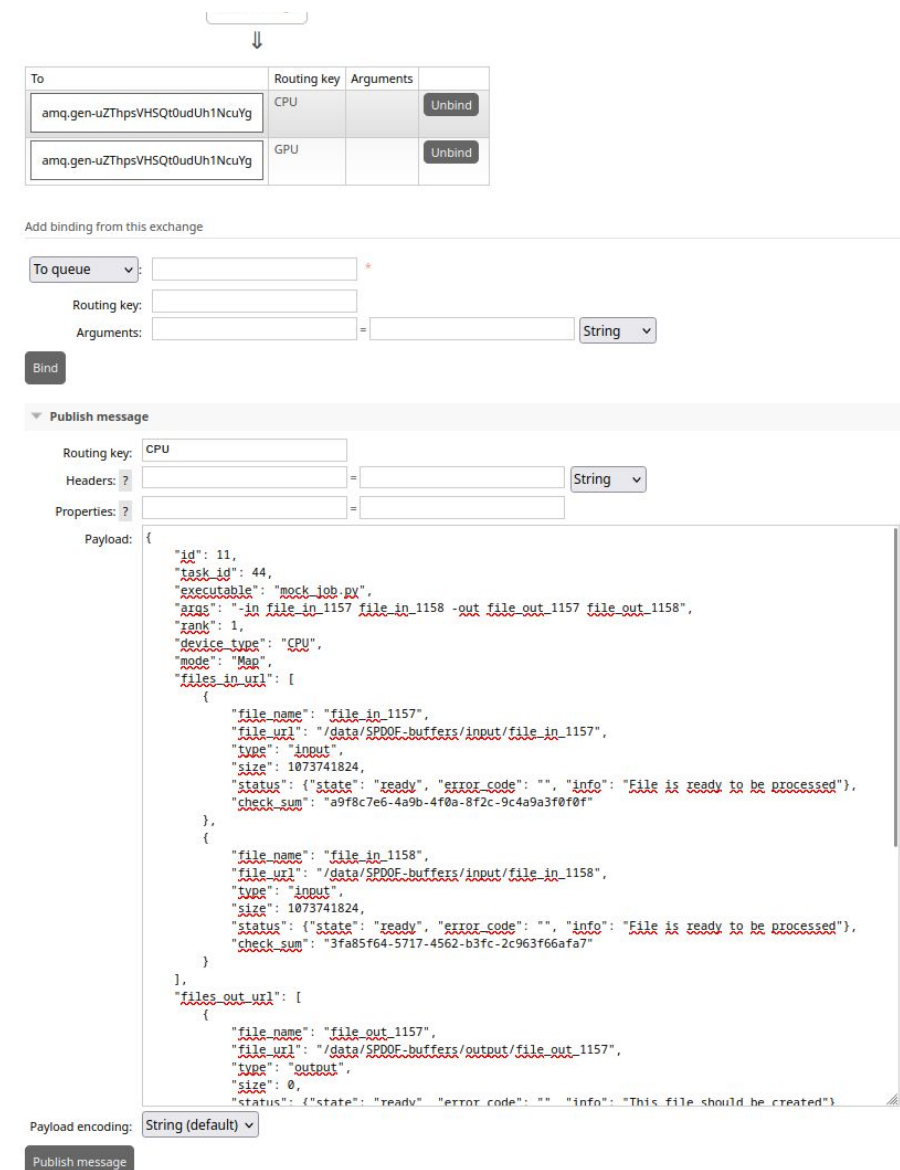


# Prototyping Job-Executor - Pilot (RabbitMQ queues)

- RabbitMQ is selected as the message broker
- Queues are defined using the declarative notation of the aio-pika tool
- At the start of the application their unfolding is performed



Configured RabbitMQ queues



The screenshot shows the 'Add binding from this exchange' and 'Publish message' sections of the RabbitMQ Admin interface. The 'Add binding' section has a 'To queue' dropdown, a 'Routing key' field, and an 'Arguments' field. The 'Publish message' section has a 'Routing key' dropdown, a 'Headers' field, a 'Properties' field, and a 'Payload' text area. The 'Payload' field contains a JSON message with fields for 'id', 'task\_id', 'executable', 'args', 'rank', 'device\_type', 'mode', 'files\_in\_url', and 'files\_out\_url'. The 'Payload encoding' dropdown is set to 'String (default)'.

Routing key: CPU

Headers: ?

Properties: ?

Payload:

```
{
  "id": 11,
  "task_id": 44,
  "executable": "mock_job.py",
  "args": "-in file_in_1157 file_in_1158 -out file_out_1157 file_out_1158",
  "rank": 1,
  "device_type": "CPU",
  "mode": "Mock",
  "files_in_url": [
    {
      "file_name": "file_in_1157",
      "file_url": "/data/SPDOE/buffers/input/file_in_1157",
      "type": "input",
      "size": 1073741824,
      "status": {
        "state": "ready",
        "error_code": "",
        "info": "File is ready to be processed"
      },
      "check_sum": "a9f8c7e6-4a9b-4f0a-8f2c-9c4a9a3f0f0f"
    }
  ],
  "files_out_url": [
    {
      "file_name": "file_in_1158",
      "file_url": "/data/SPDOE/buffers/input/file_in_1158",
      "type": "input",
      "size": 1073741824,
      "status": {
        "state": "ready",
        "error_code": "",
        "info": "File is ready to be processed"
      },
      "check_sum": "3fa85f64-5717-4562-b3fc-2c963f66afa7"
    }
  ],
  "status": {
    "state": "ready",
    "error_code": "",
    "info": "This file should be created"
  }
}
```

Payload encoding: String (default)

Publish message

Jobs could be delivered manually

# Summary

We have designed the components of the **Workload Management System**, taking into account the characteristics and internal requirements of both the **WFMS** and **DSM** systems.

Our goal is to complete the prototyping phase and fully integrate with the application layer components of the «SPD On-Line Filter» platform.

## Current plans:

- Run a simple data flow: create dataset, define a task, propagate through **WMS**, register files, close dataset
- Deploy pilot on multiple machines

## Plans for the year:

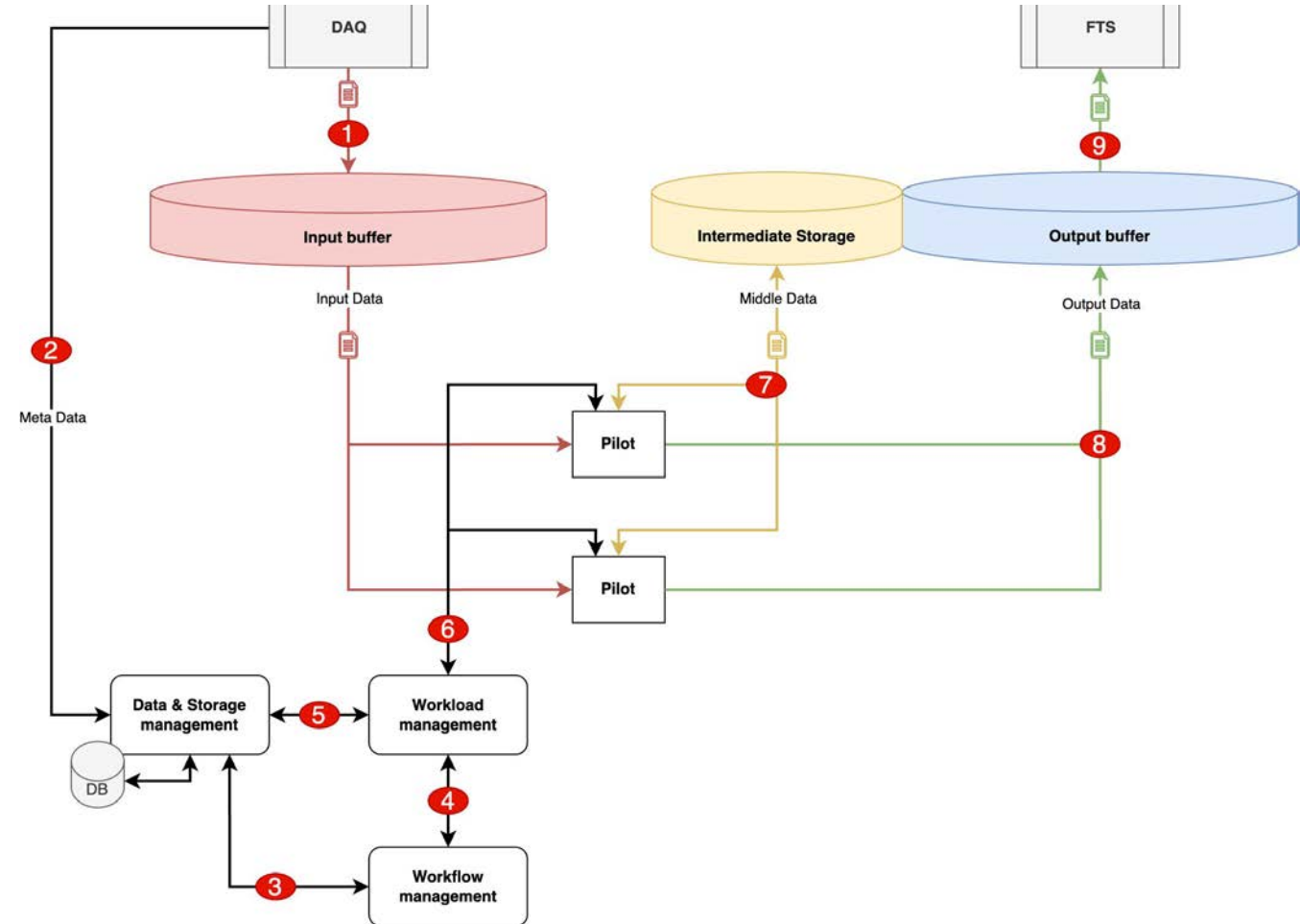
- Defining and implementing obvious data processing pipelines;
- Debugging basic algorithms and external interfaces;
- Work out integration with application software and test on SPD-DAQ modelled data.

**Thank you for your attention!**

# Dataflow and data processing concept

Main data streams:

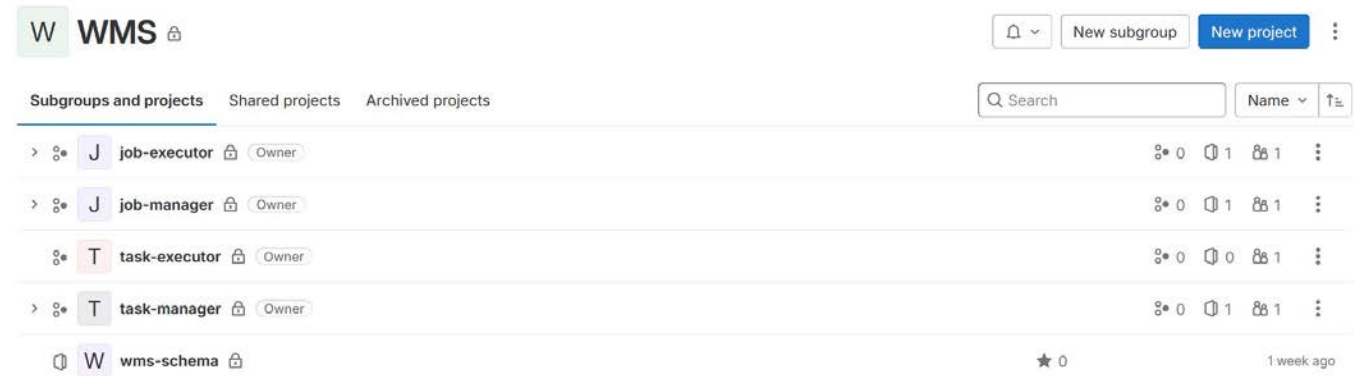
- ❖ SPD DAQs, after dividing sensor signals into time blocks, send data to the SPD Online Filter input buffer as files of a consistent size.
- ❖ The workflow management system creates and deletes intermediate and final data sets
- ❖ The **workload management system** “populates” the data sets with information about the resulting files
- ❖ At each stage of data processing, pilots will read and write files to storage and create secondary data



# Modularization: deploying and using own packages

Following tools are used

- ❖ Poetry
  - Particularly good at handling complex dependency trees and ensuring that the different modules can integrate with each other without version conflicts
- ❖ Python packages
  - separate GitLab repositories for each package
  - Poetry for packaging and dependency management
- ❖ Gitlab
  - *Access Tokens* used as kind of credentials for scripts and other tools
  - CI/CD for automate testing and building



**wms-schema is a package that contains a scheme for task and job data that is used in almost every other service**

- Jobs scheduling (algo)
- Partitioning of a task
  - Imagine a multitasking operating system.
  - Each dataset represents a process, and each record within a dataset is like a thread within that process.
  - The algorithm acts as the operating system's scheduler, allocating processing time to threads based on their priority.
- Chunk size and rank/priority of a job as a basic control unit:

$$rank_{i+1} = \alpha \times x_i + \beta \times y_i + \gamma \times rank_i$$

$x_i$  – aging,  $y_i$  – retries

---

**Algorithm 1** Task Scheduling Algorithm
 

---

**Variables:**

global\_queue – global queue with tasks

dataset – array of datasets

$N$  – number of datasets

rank\_max – maximum task priority

heap – binary heap storing maximum task priorities

rank – array with task priorities

**Algorithm:**

```

1: initialize_datasets(dataset)
2: build_heap(rank)
3: while true do
4:   rank_max = heap.top()
5:   for  $r = 1$  to rank_max do
6:     for  $i = 1$  to  $N$  do
7:       if not dataset[ $i$ ].chunk.empty() and rank[ $i$ ]  $\geq r$  then
8:         await dataset[ $i$ ].chunk.cur_item
9:         update(dataset[ $i$ ].chunk –  $i$ , cur_item)
10:      else if dataset[ $i$ ].chunk.empty() then
11:        if dataset[ $i$ ].chunk.cur_item then
12:          dataset[ $i$ ] = global_queue.head()
13:        end if
14:        update(rank[ $i$ ])
15:        update(heap)
16:      end if
17:    end for
18:  end for
19: end while
  
```

---

Proposed task-partitioning algorithm