MESHCHERYAKOV LABORATORY of INFORMATION TECHNOLOGIES

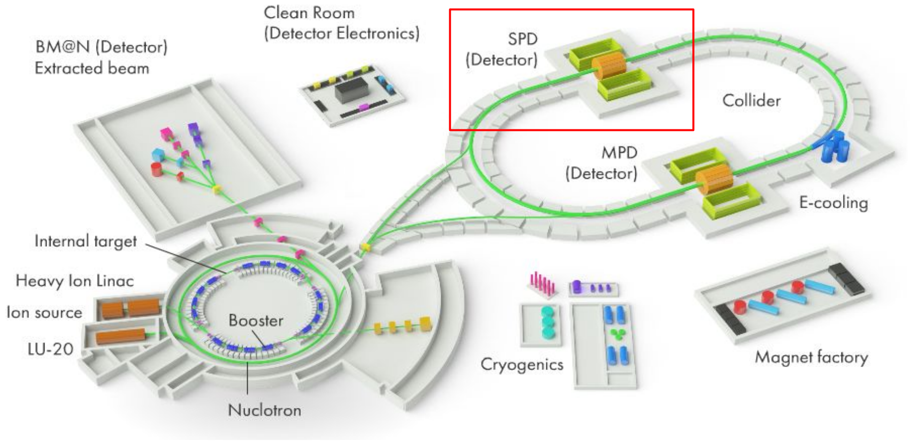
Workload Management System Development for SPD Online Filter

Nikita Greben, MLIT

AYSS-2024, Dubna

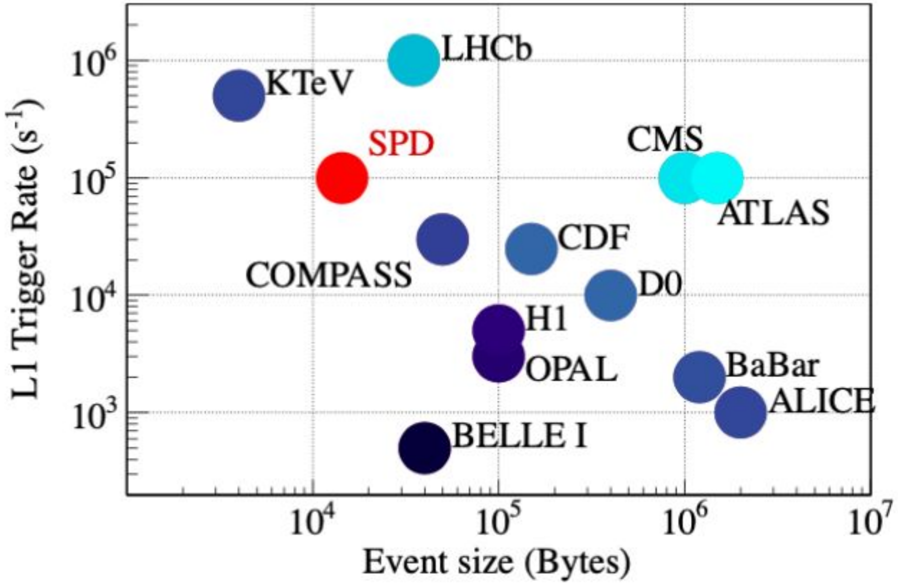
SPD experiment at NICA collider

The SPD detector (Spin Physics Detector) is one of the NICA infrastructure projects designed to study the spin and momentum of gluons and their distribution.



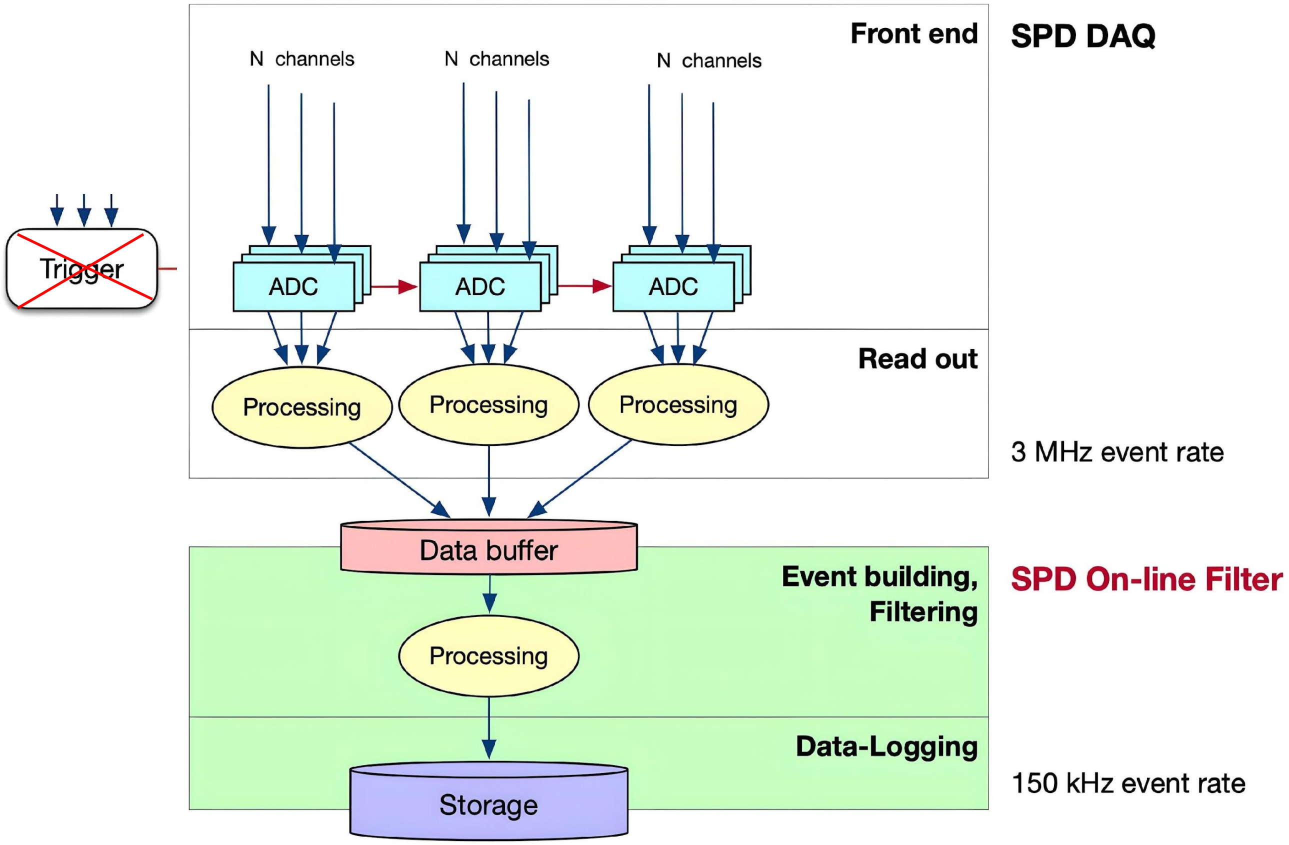
- Polarized proton and deuteron beams
- Collision energy up to 27 GeV
- luminosity up to 10^{32} cm^{-2} s^{-1}
- Bunch crossing every 80 ns = crossing rate 12.5 MHz

- Number of registration channels in SPD ~ 500000
- ~ 3 MHz event rate (at max luminosity) = pileups
  - ~ 20 GB/s (or 200PB/year) “raw” data
- Physics signal selection requires momentum and vertex reconstruction
  - => no simple trigger is possible

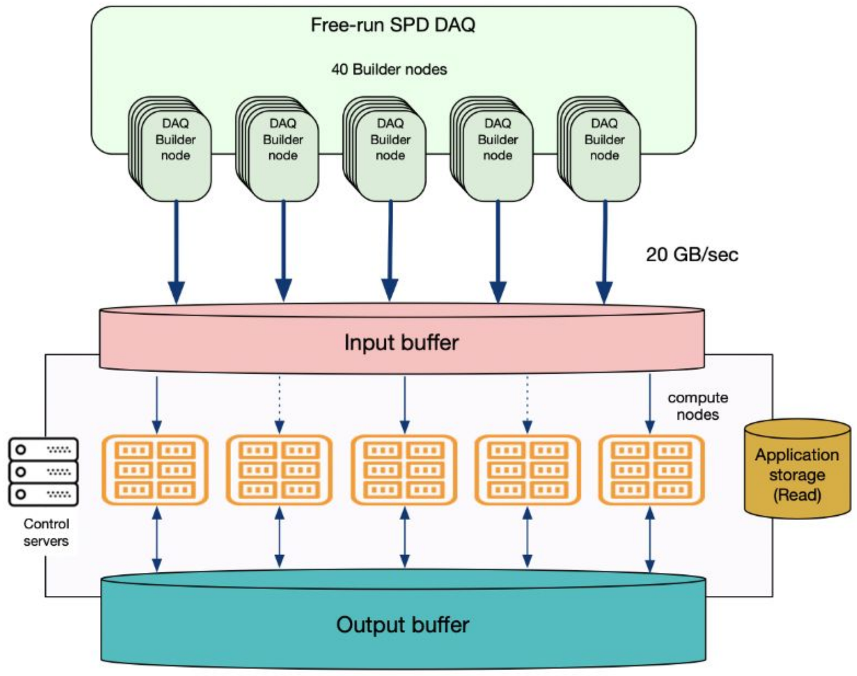


Triggerless DAQ

Triggerless DAQ means that the output of the system is not a set of raw events, but a set of signals from sub-detectors organized into time slices.



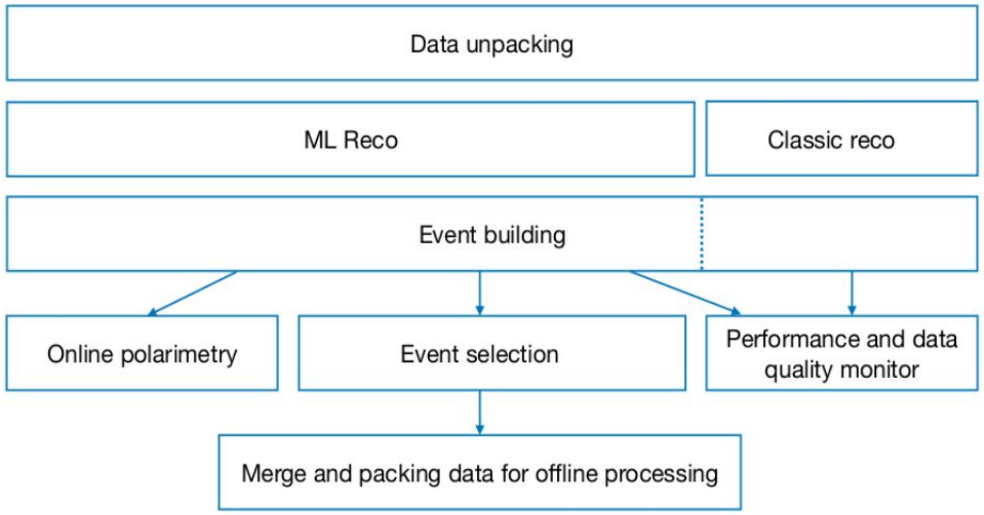
«triggerless» DAQ



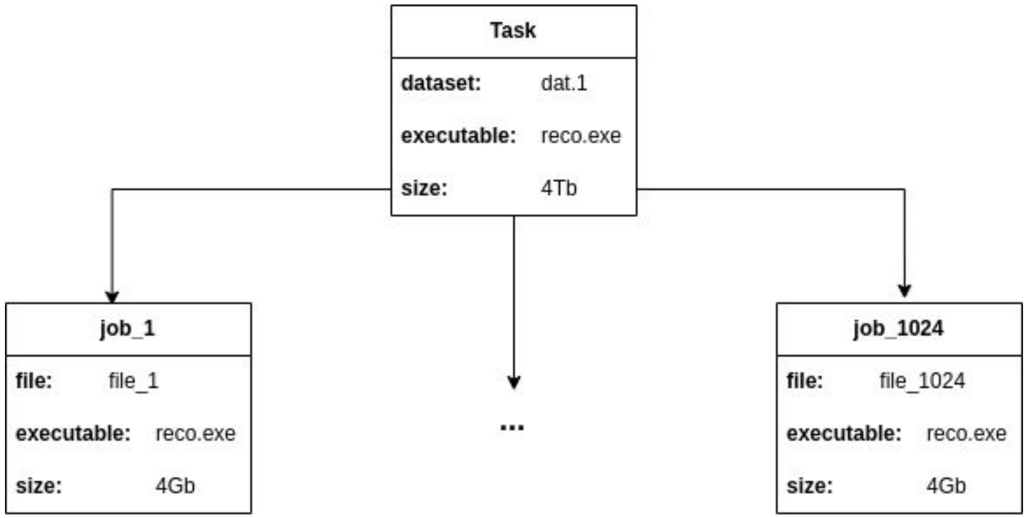
- ➢ DAQ provide data organized in time frames which placed in **files** with reasonable size (a few GB).
- ➢ Each of these file may be processed independently as a part of top-level **workflow chain**.
- ➢ No needs to exchange of any information during handling of each initial file, but results of may be used as input for next step of processing.

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**



Data processing work chain example

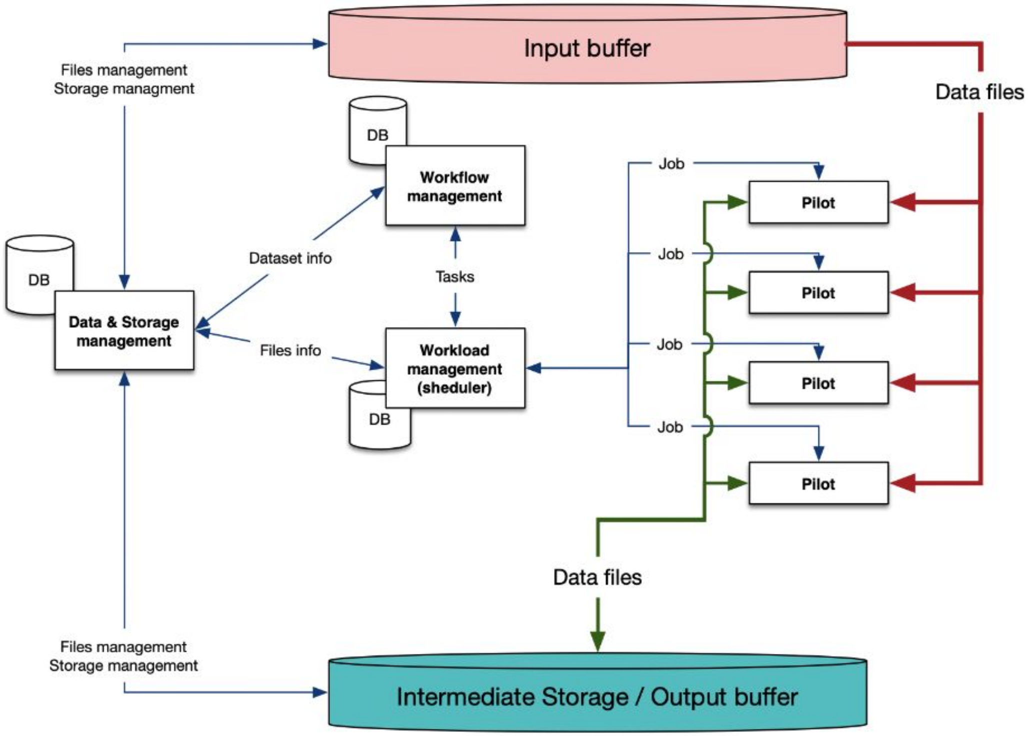


Task-job relationship

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 (one PhD student and one master student)**
  - Data lifecycle support (data catalog, consistency check, cleanup, storage);
- **Workflow Management System (master student)**
  - Define and execute processing chains by generating the required number of computational tasks;
- **Workload management system:**
  - Create the required number of processing jobs to perform the task;
  - Control job execution through pilots working on compute nodes;



Architecture of SPD Online Filter

Workload management system requirements

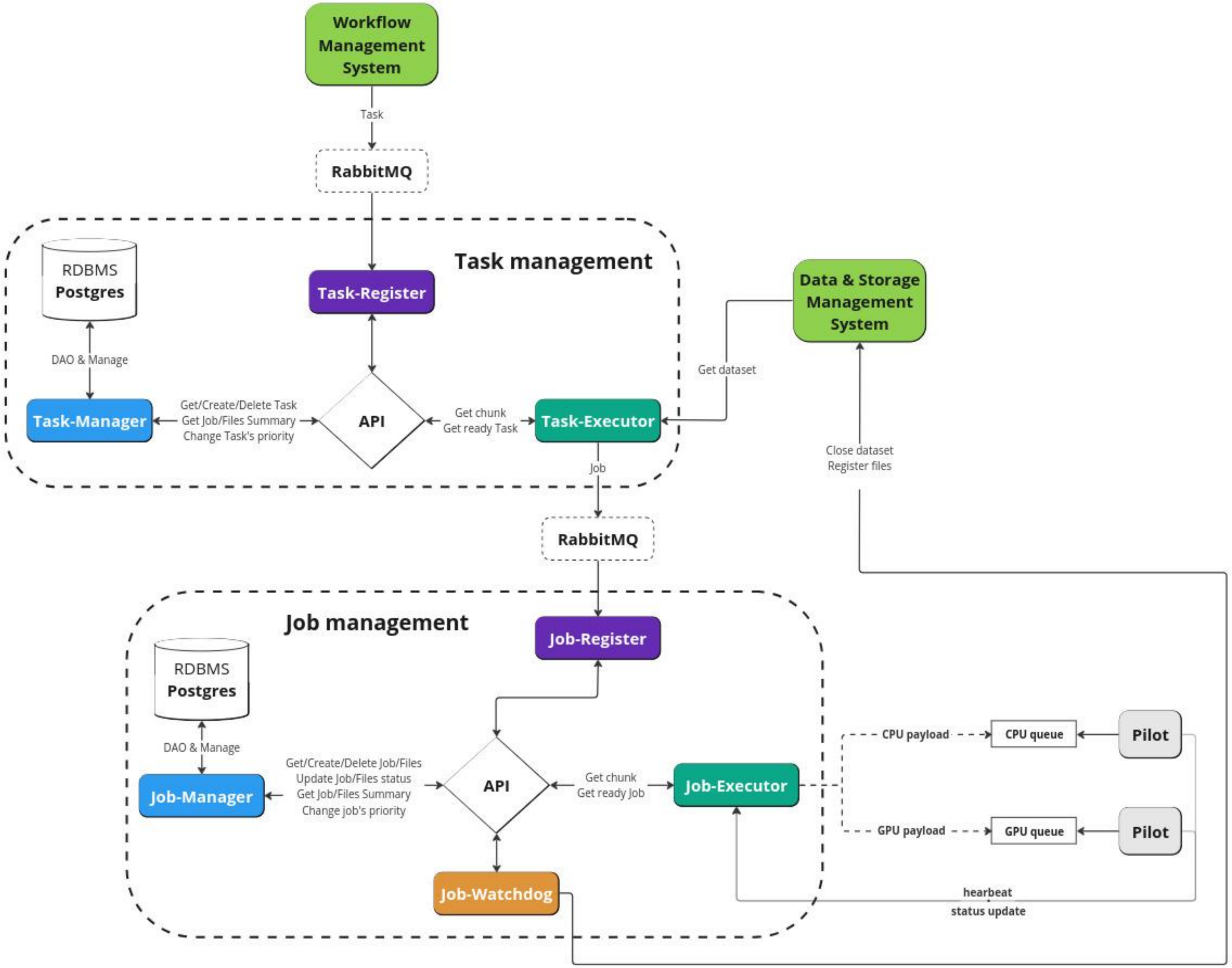
The key requirement - systems must meet the **high-throughput paradigm**.

- ❑ **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;
- ❑ **Consistency control**: control of the consistency of information in relation to the tasks, files and jobs;
- ❑ **Scheduling:** implementing a scheduling principle for task/job distribution;

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
- **pilot** – responsible for running jobs on compute nodes, organizing their execution, and communicating various information about their progress and status.



SPD Workload Management System High Level Architecture

Pilot Agent

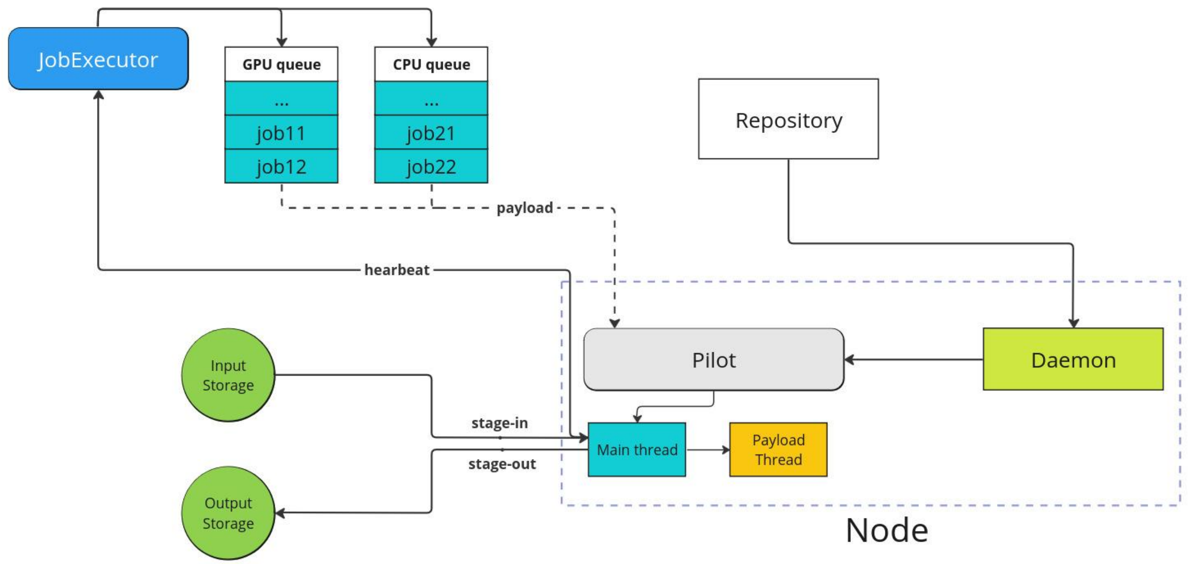
- The agent application is deployed on a compute node and consists of the following two components: a UNIX daemon and the pilot itself.
- The UNIX daemon's objective is to run the next pilot by downloading an up-to-date version from the repository.
- Pilot itself is a multi-threaded Python application 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;

Two types of nodes:

- Multi-CPU
- Multi-CPU + GPU

Two communication channels:

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

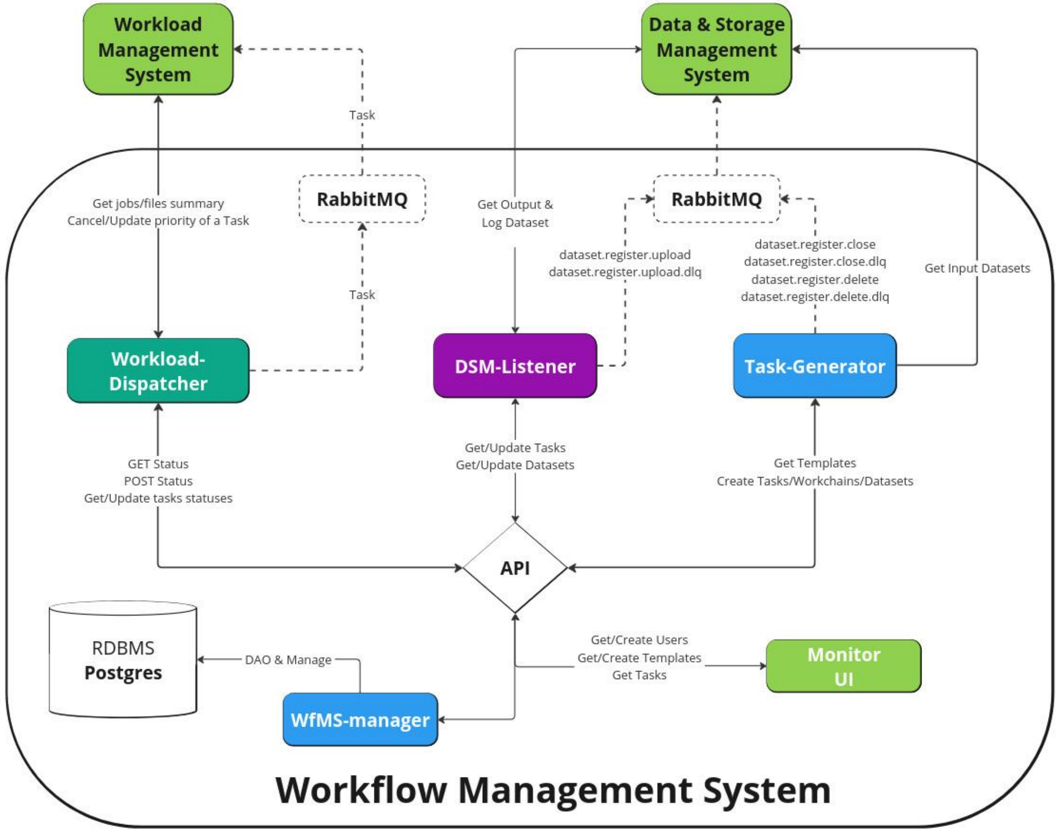


- ✓ A detailed job status model has been described;
- ✓ Error codes introduced;
- ✓ Pilot ran through all stages of the job execution (**D**irected **A**cyclic **G**raph);
- ✓ Pilot at this stage runs a script that does a basic hash compute;
- ✓ UNIX Daemon is implemented and currently running;

Interaction with Workflow Management System

The following interaction scenarios have been identified with the Workflow Management System

- Registration of a task for processing: **WfMS** passes the task description into the 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*) on operator side;
- Change priority of a task: is used to accelerate the rate at which the corresponding dataset is being processed;



Interaction with Data Management System

Routing Key	Msg	Algo
dataset.close	Dataset info • Dataset UID • File check list (file names)	Request the registered files in the dataset. If they match the checklist, set the status to **CLOSED**. Otherwise, return the messages back to the queue for deferred execution.
dataset.upload	Dataset UID	Marking dataset for uploading (**TO_UPLOAD**)
dataset.delete	Dataset UID	Marking dataset for deletion (**TO_DELETE**)

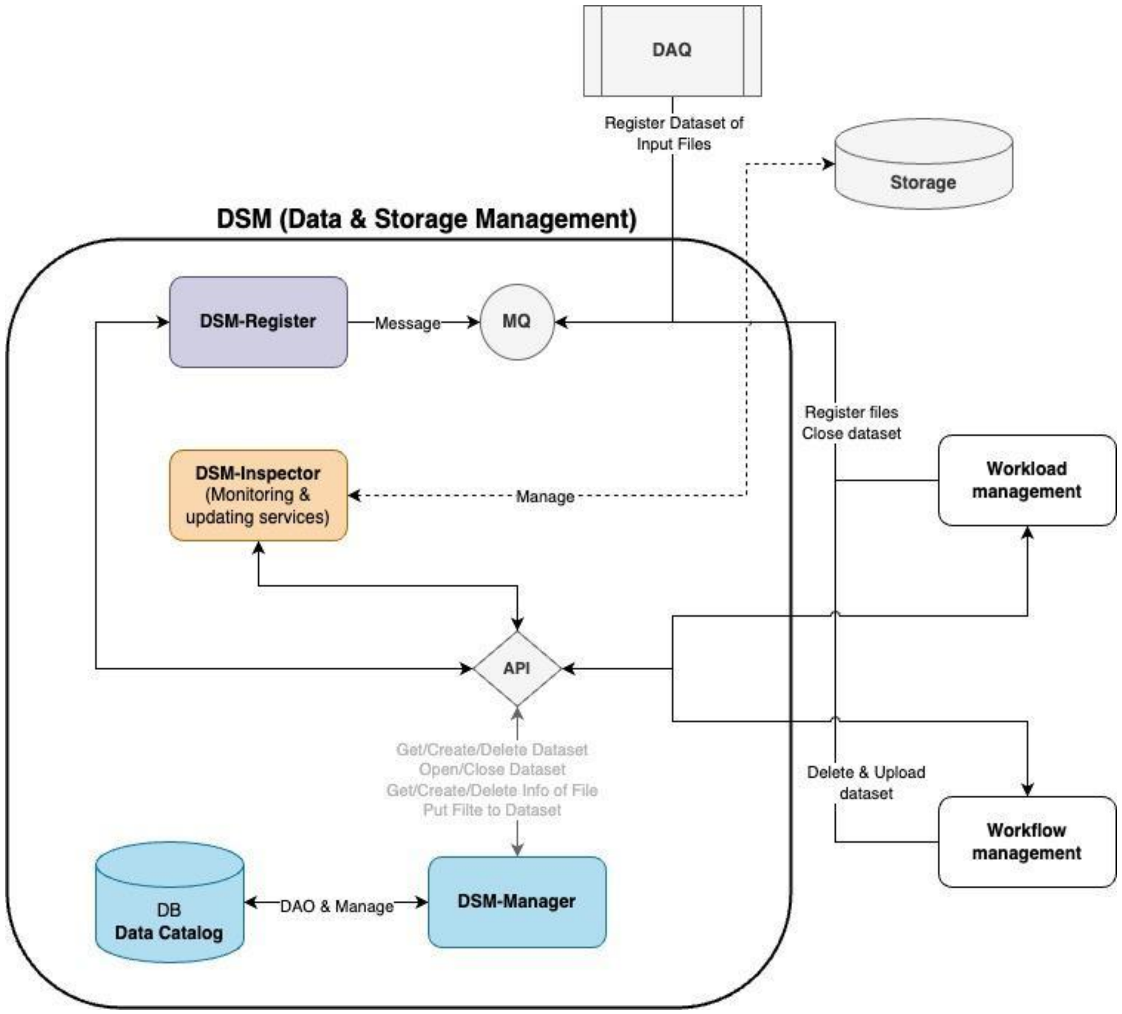
Signature and algorithm of message receiving gateways for the **dsm-register** service

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

Tereschenko, D., Ponomarev, E., Oleynik, D. *et al.* SPD On-Line Filter: Workflow and Data Management Systems. *Phys. Part. Nuclei* **55**, 603–605 (2024).

Database design

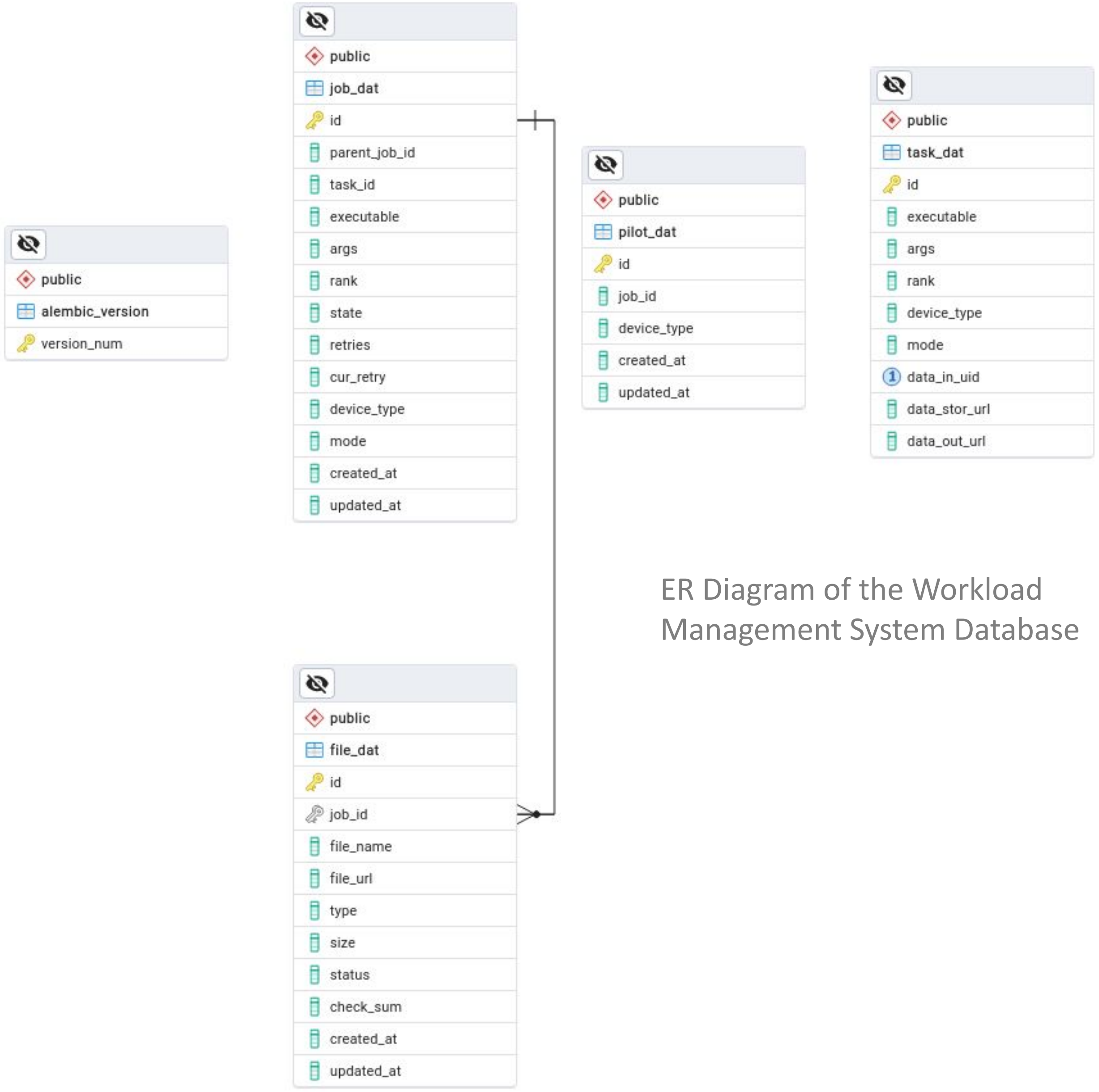
RDBMS - PostgreSQL 16

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 (B-tree);
- **Procedures** – task and job generation for test purposes;
- **Triggers** – rank update logic;
- **Decomposition** – single database per microservice (Postgres in Docker initially)



ER Diagram of the Workload Management System Database

Tech stack

Common	Frameworks
➢ Python 3.12 ➢ docker compose - running multi-container applications	➢ aio-pika (RabbitMQ + asyncio) - asynchronous API with RabbitMQ ➢ FastAPI + uvicorn
DB	**Extra**
➢ PostgreSQL - RDBMS ➢ Alembic (Migration) ➢ SQLAlchemy 2.0 ➢ asyncpg - Postgres DBAPI	➢ aiohttp - asynchronous HTTP client/server framework ➢ Pydantic - validate and serialize data schemes ➢ pytest-asyncio - test purposes

Current Status

Design of services:

- ✓ Designed and implemented a list of required REST API methods and their signatures;
- ✓ Implemented a mechanism for declaring the data model in the database based on ORM and migration scripts;
- ✓ Configured CD tools (build and deployment) on the JINR LIT infrastructure;
- ✓ Designed inter-service interaction scenarios – defined API contracts;
- ✓ Designed Pilot internal architecture;
- ✓ Workload Management System - Pilot Interaction Models in Finite State Machine.

Prototype of services:

- ✓ Most microservices implemented;
- ✓ Job management subsystem is the most advanced: most interactions implemented and being tested;
- ✓ Task partitioning is being implemented;
- ✓ Pilot and Pilot Daemon is currently working;
- ✓ Pilot handles all stages of job execution on the given workload.

Next major steps

- **Task processing**
  - Execute the entire workchain set up on the level of **WfMS**.
- **Middleware and applied software integration**
  - Requires prototyped applied software and simulated data.
- **Logging**
  - Currently, each microservice 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 (*Consul, Etcd*).
- **Documentation**
  - Given the increasing complexity of the internal logic of the software, it is necessary to document each step of the development.
- **Metrics and monitoring**
  - For example, service query-per-second, API responsiveness, service latency etc. (*InfluxDB, Prometheus, Graphana*)

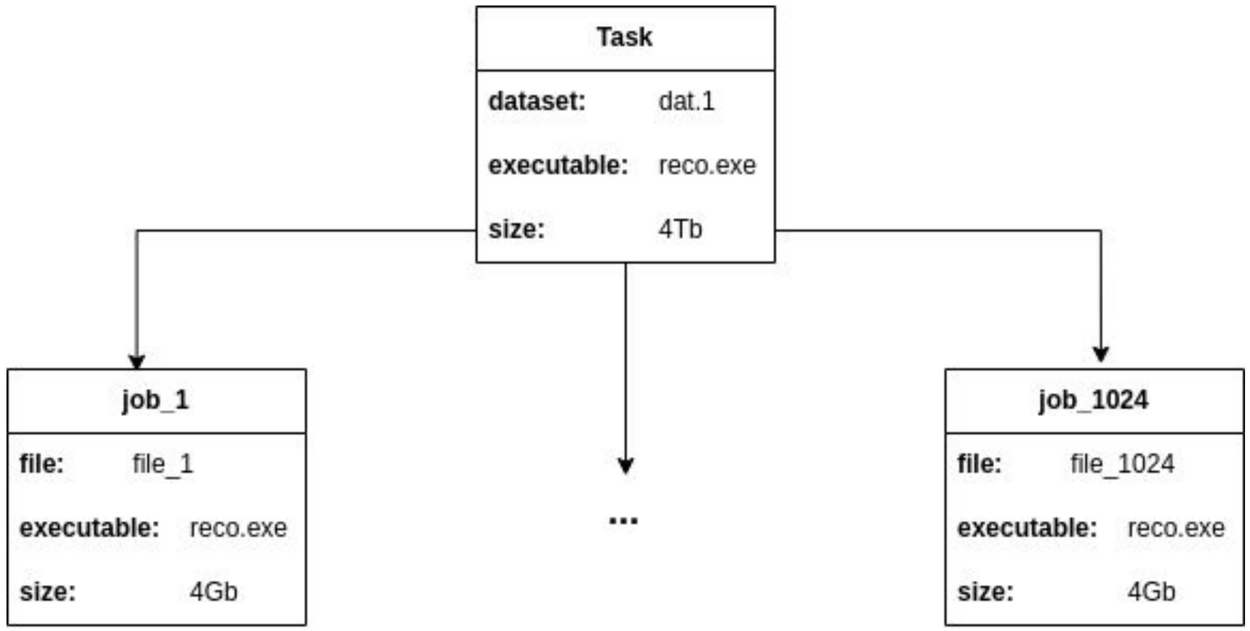
Thank you for your attention!

Backup slides

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 criterion for the completion of the task is the processing of the entire block of data.
- ➢ 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** is responsible for generating **jobs**, sending them to compute nodes, and executing them.

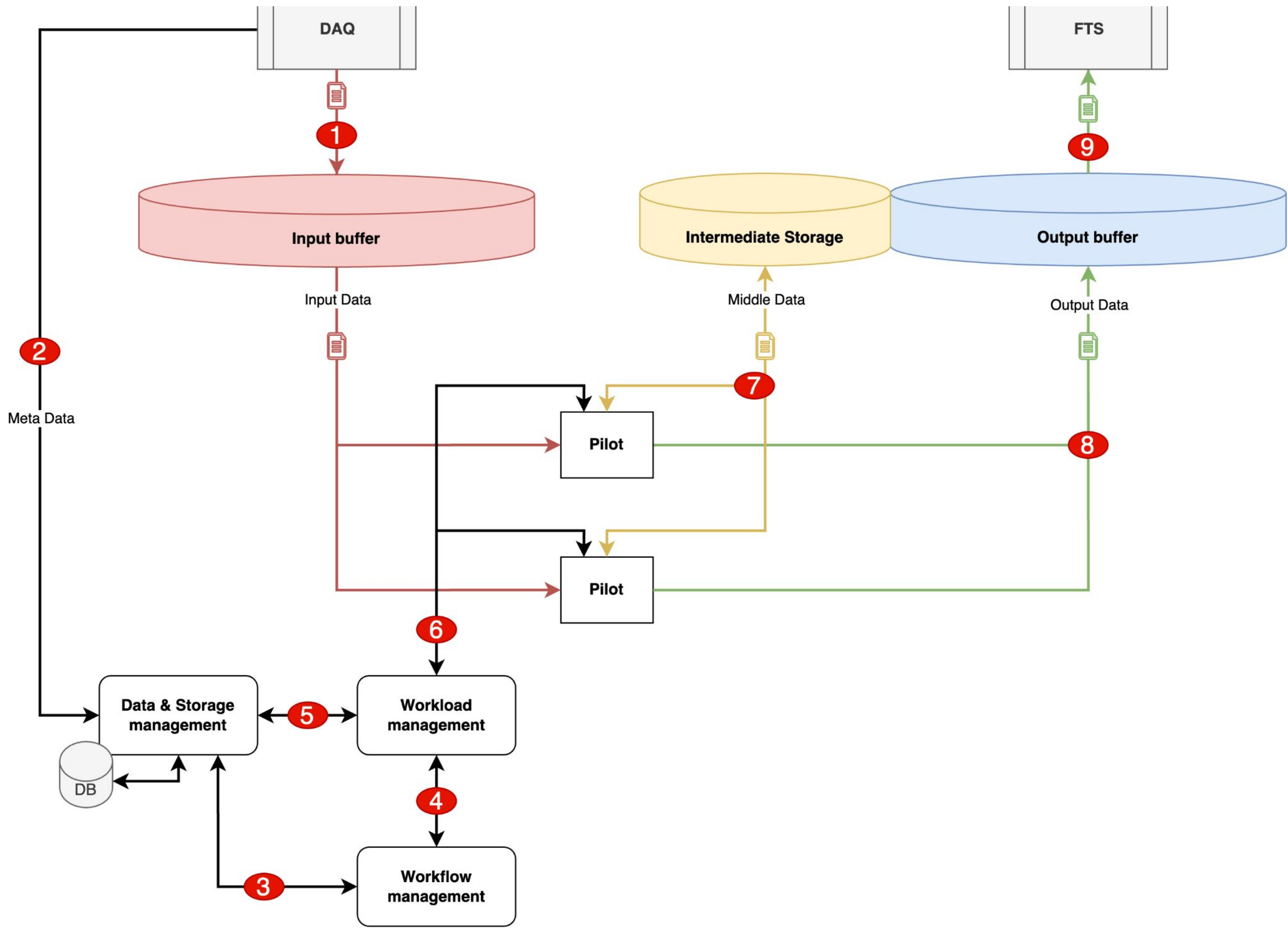


Task-job relationship

Dataflow and data processing concept

Main data streams:

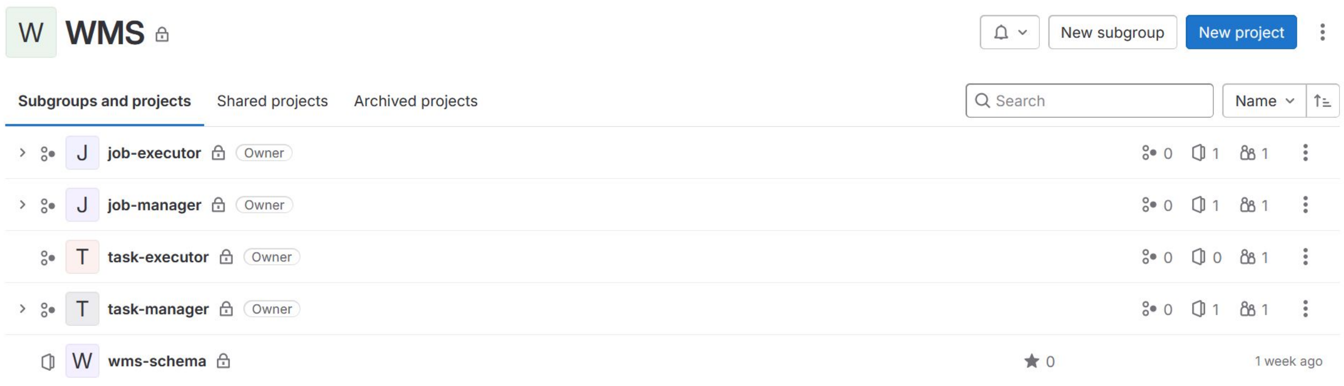
- SPD DAQs, after dividing sensor signals into time blocks, send data to the SPD Online Filter input buffer as files of a consistent size.
- The workflow management system creates and deletes intermediate and final data sets
- The **workload management system** “populates” the data sets with information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

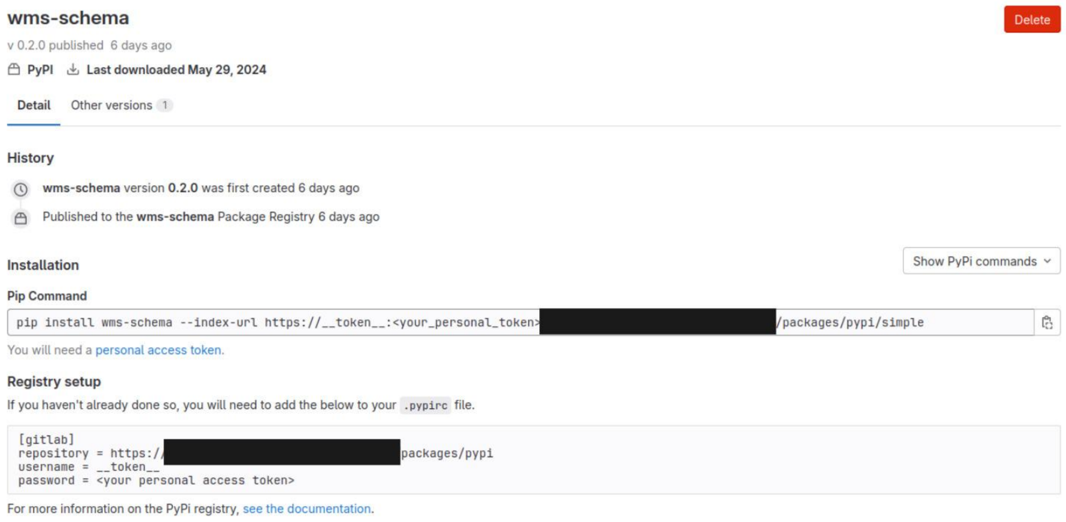


Modularization: deploying and using own packages

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

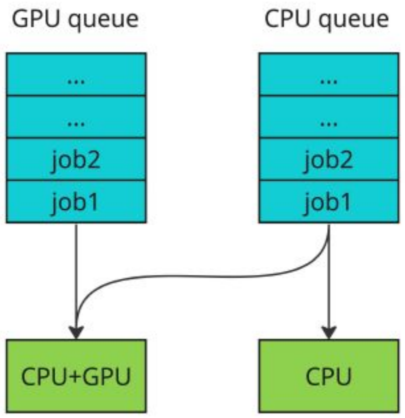


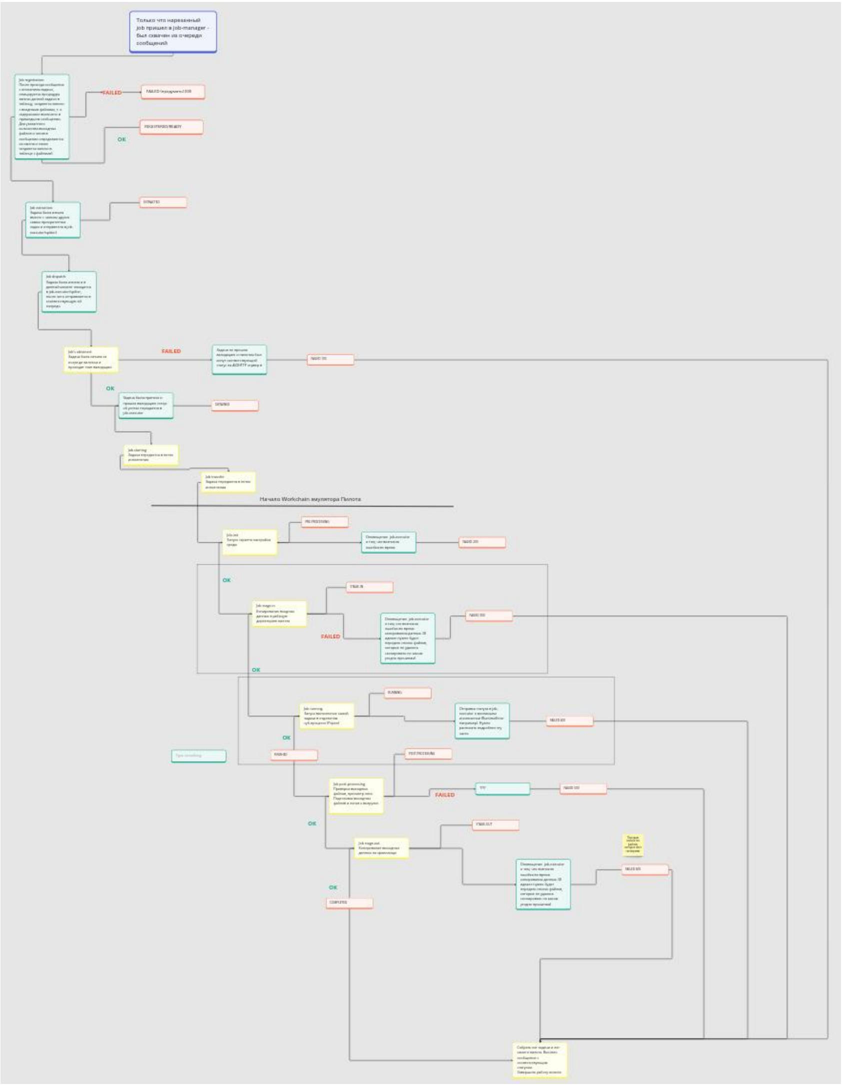


wms-schema is a package that contains a scheme for task and job data that is used in almost every other service

Interaction with the Pilot Agent

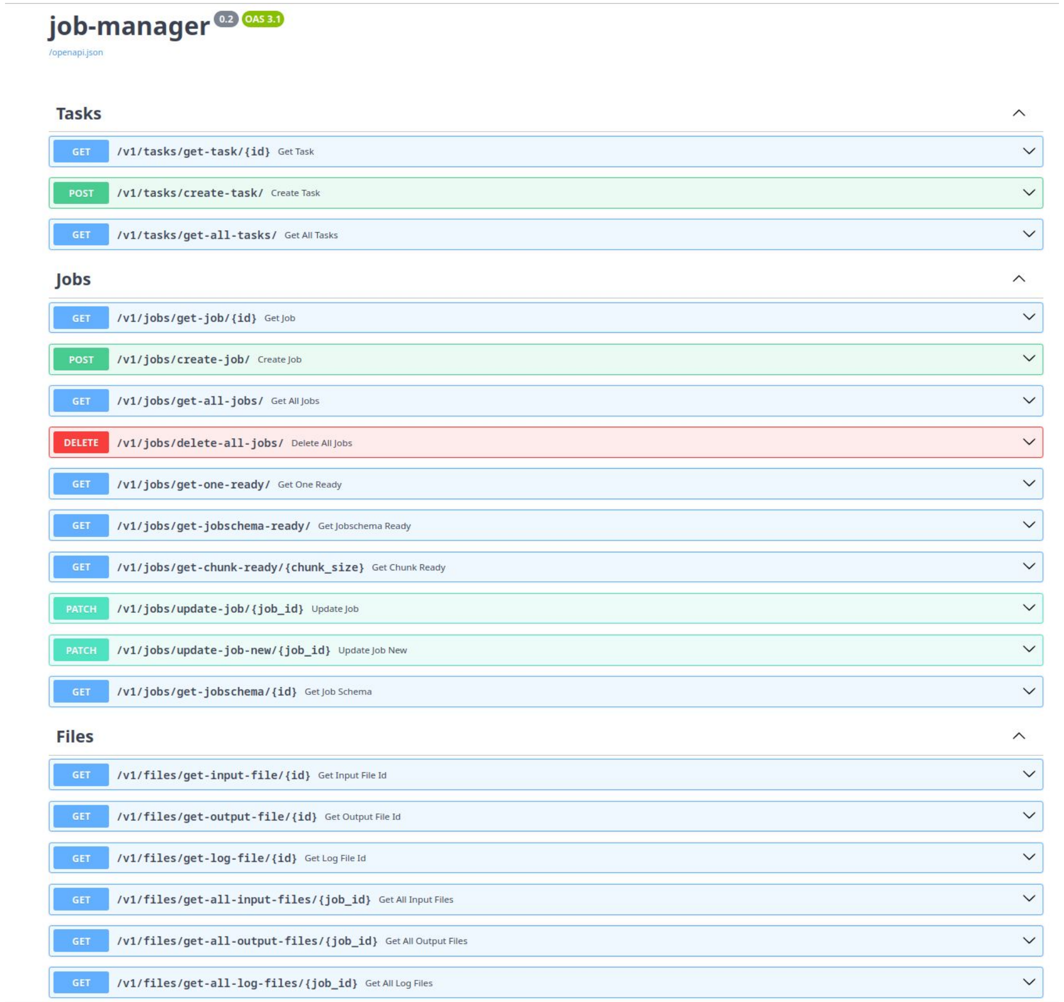
- Pilot has a series of prepossessing stages before running a job itself:
  a. start logging
  b. read configuration
  c. getting a job from message queue
  d. validation
- After those steps the Pilot launches another thread where it does
  a. environment setup script
  b. copying files locally from 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.



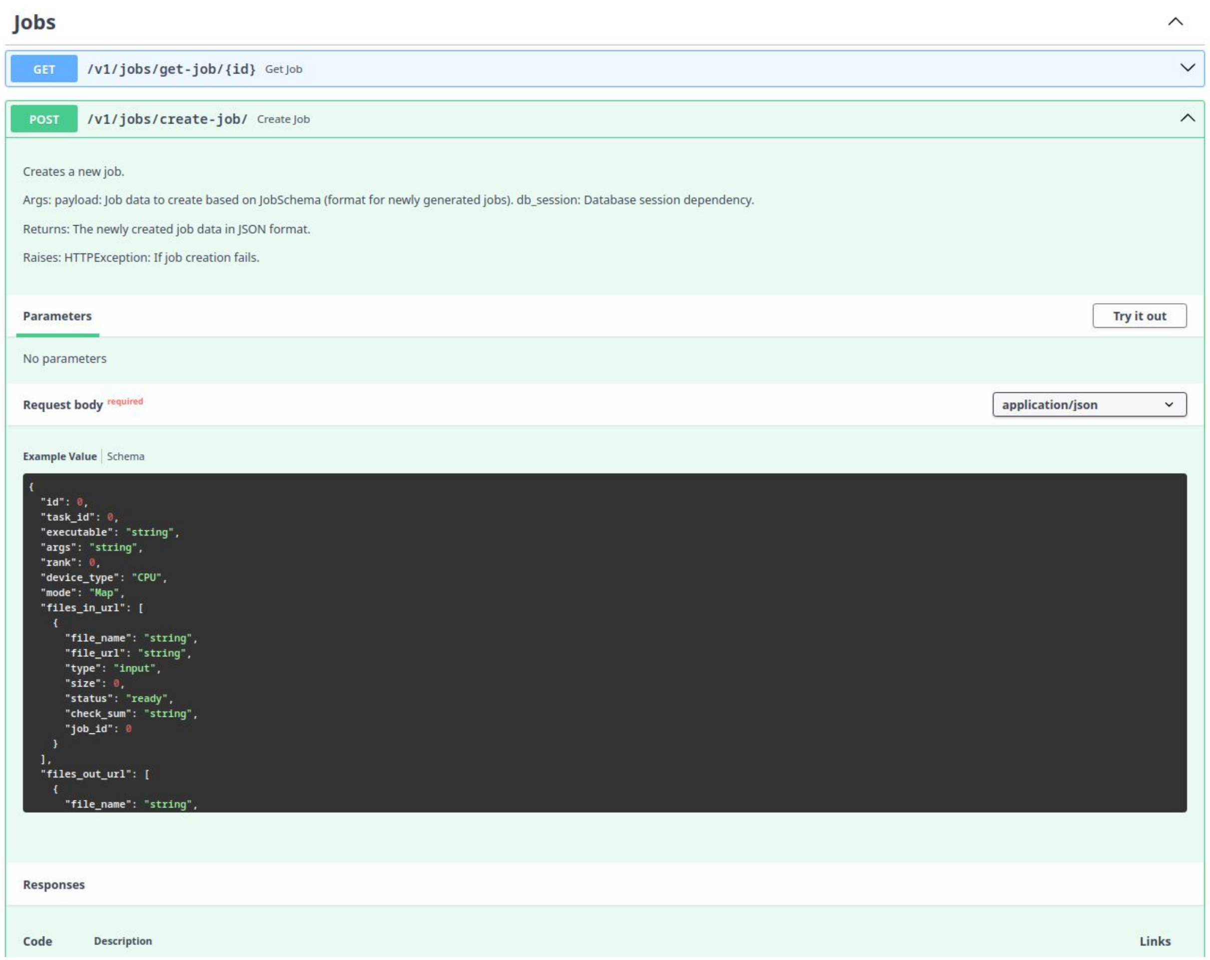


Prototyping Job-Manager (API)

- 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)



Swagger UI with job-manager service API description

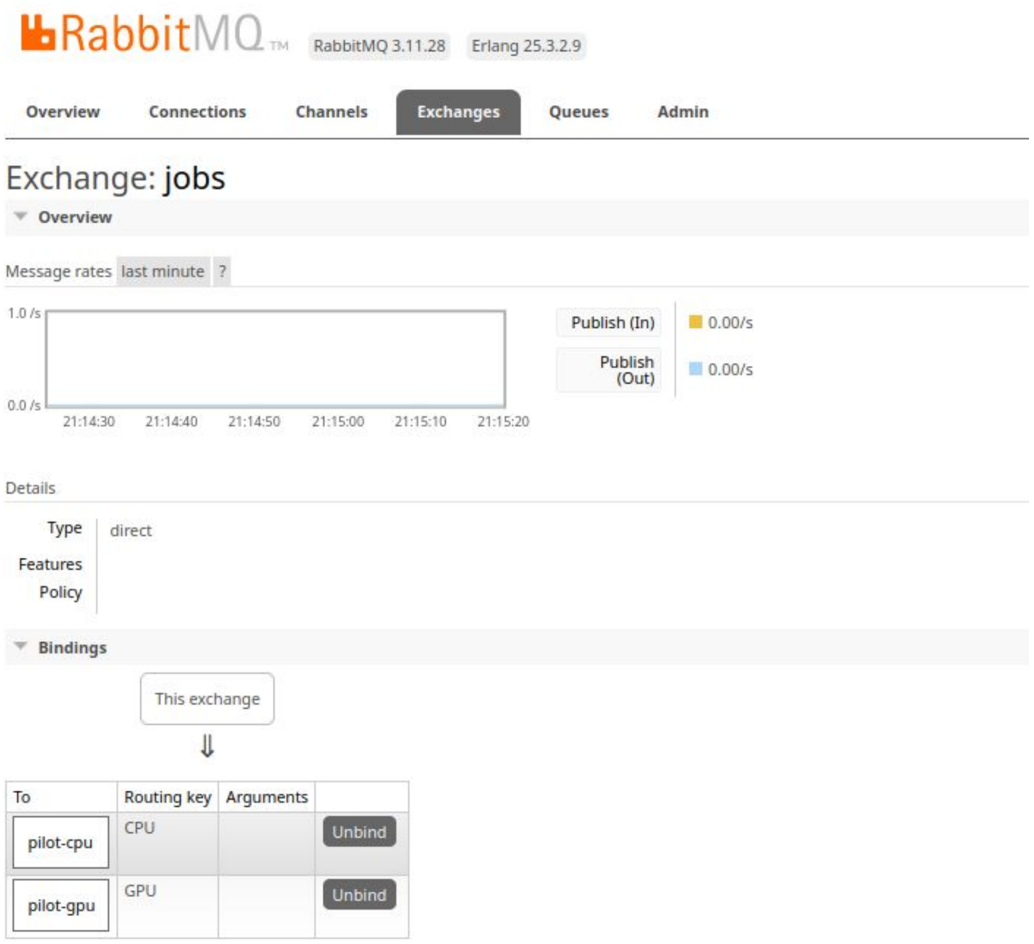


Example of a service call to post a new job

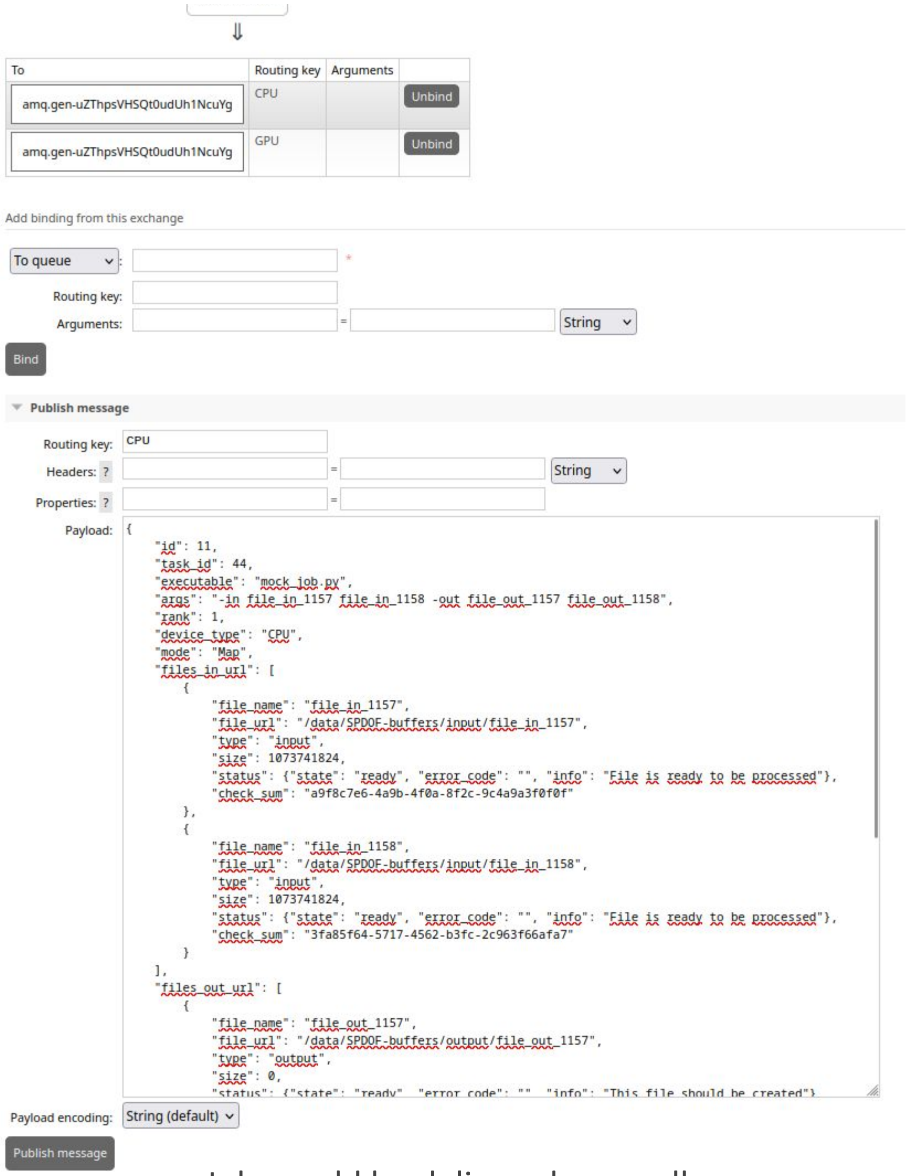
Prototyping Job-Executor - Pilot (RabbitMQ queues)

SPD NICA

- 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



Jobs could be delivered manually

Examples of Templates and Tasks

NICA SPD

- Registration and authorization
- Template and task output
- CWL template creation by user
- Preliminary validation and writing of CWL templates to the database

Template Manager Templates Tasks a@aaa.aaa Logout

Create template

template_id	name	inner_dataset_mask	description	status
1	template1	.test.	{"steps": {"decoding": {"run": {"class": "CommandLineTool", "baseCommand": "echo", "inputs": {"dataset_name": {"type": "string"}, "processing_program": {"type": "string"}, "processing_program_version": {"type": "string"}, "cable_map": {"type": "File"}, "input_params": {"type": "File"}}, "outputs": {"output_dataset": {"type": "File"}, "log_dataset": {"type": "File"}}}, "in": {"dataset_name": ".test.", "processing_program": "processing_program", "processing_program_version": "processing_program_version", "cable_map": "cable_map", "input_params": "input_params"}, "out": "[output_dataset, log_dataset]"}, "reconstruction": {"run": {"class": "CommandLineTool", "baseCommand": "echo", "inputs": {"dataset_name": {"type": "string"}, "processing_program": {"type": "string"}, "processing_program_version": {"type": "string"}, "cable_map": {"type": "File"}, "input_params": {"type": "File"}}, "outputs": {"output_dataset": {"type": "File"}, "log_dataset": {"type": "File"}}}, "in": {"dataset_name": ".test.", "processing_program": "processing_program", "processing_program_version": "processing_program_version", "cable_map": "cable_map", "input_params": "input_params"}, "out": "[output_dataset, log_dataset]"}}}	ACTUAL
2	template2	.test.	{"steps": {"decoding": {"run": {"class": "CommandLineTool", "baseCommand": "echo", "inputs": {"dataset_name": {"type": "string"}, "processing_program": {"type": "string"}, "processing_program_version": {"type": "string"}, "cable_map": {"type": "File"}, "input_params": {"type": "File"}}, "outputs": {"output_dataset": {"type": "File"}, "log_dataset": {"type": "File"}}}, "in": {"dataset_name": ".test.", "processing_program": "processing_program", "processing_program_version": "processing_program_version", "cable_map": "cable_map", "input_params": "input_params"}, "out": "[output_dataset, log_dataset]"}}}	ARCHIVED

Created template

Template Manager Templates Tasks a@aaa.aaa Logout

task_id	wflow_id	exec	args	rank	device	mode	retry	datas_in_id	datas_out_id	datas_log_id	status
11	6	processing_program	cable_map	1	CPU	map	5	26	27	28	IN_PROGRESS
12	6	processing_program	cable_map	1	CPU	map	5	27	29	30	IN_PROGRESS
13	7	processing_program	cable_map	1	CPU	map	5	31	32	33	IN_PROGRESS
14	7	processing_program	cable_map	1	CPU	map	5	32	34	35	IN_PROGRESS
15	8	processing_program	cable_map	1	CPU	map	5	36	37	38	IN_PROGRESS
16	8	processing_program	cable_map	1	CPU	map	5	37	39	40	IN_PROGRESS
17	9	processing_program	cable_map	1	CPU	map	5	41	42	43	IN_PROGRESS
18	9	processing_program	cable_map	1	CPU	map	5	42	44	45	IN_PROGRESS
19	10	processing_program	cable_map	1	CPU	map	5	46	47	48	IN_PROGRESS
20	10	processing_program	cable_map	1	CPU	map	5	47	49	50	IN_PROGRESS
21	11	processing_program	cable_map	1	CPU	map	5	51	52	53	IN_PROGRESS
22	11	processing_program	cable_map	1	CPU	map	5	52	54	55	IN_PROGRESS
23	12	processing_program	cable_map	1	CPU	map	5	56	57	58	IN_PROGRESS
24	12	processing_program	cable_map	1	CPU	map	5	57	59	60	IN_PROGRESS

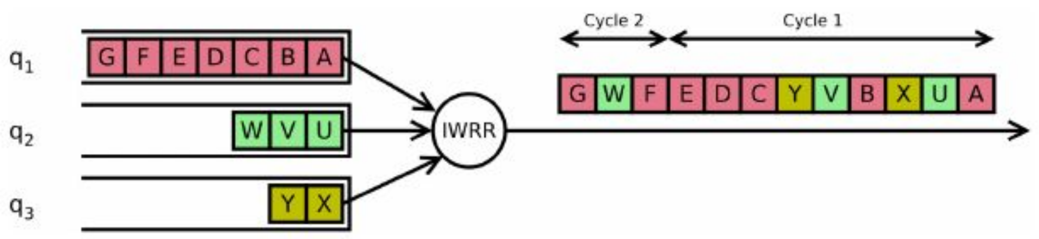
WfMS task description

R&D

- Jobs scheduling algorithm
- 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:

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

$$x_i - aging,\ y_i - retries$$



Interleaved Weighted round-robin

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:
initilize_datasets(dataset)
build_heap(rank)
while true do
    rank_max = heap.top()
    for r = 1 to rank_max do
        for i = 1 to N do
            if not dataset[i].chunk.empty() and rank[i] ≥ r then
                await dataset[i].chunk.cur_item
                update(dataset[i].chunk – ¿ cur_item)
            else if dataset[i].chunk.empty() then
                if dataset[i].chunk.cur_item then
                    dataset[i] = global_queue.head()
                end if
                update(rank[i])
                update(heap)
            end if
        end for
    end for
end while
```

Proposed task-partitioning algorithm