Eur. Phys. J. C (2026) 86:66
https://doi.org/10.1140/epjc/s10052-025-15275-7

THE EUROPEAN PHYSICAL JOURNAL C

Regular Article - Computing, Software and Data Science

iDDS: intelligent distributed dispatch and scheduling for workflow orchestration

Wen Guan[1,a], **Tadashi Maeno**[1], **Aleksandr Alekseev**[2], **Fernando Harald Barreiro Megino**[2], **Kaushik De**[2], **Edward Karavakis**[1], **Alexei Klimentov**[1], **Tatiana Korchuganova**[3], **FaHui Lin**[2], **Paul Nilsson**[1], **Torre Wenaus**[1], **Zhaoyu Yang**[1], **Xin Zhao**[1]

[1] Brookhaven National Laboratory, Upton, NY, USA
[2] University of Texas at Arlington, Arlington, TX, USA
[3] University of Pittsburgh, Pittsburgh, PA, USA

Received: 19 December 2025 / Accepted: 23 December 2025

Abstract The *intelligent distributed dispatch and scheduling (iDDS)* service is a versatile workflow orchestration system designed for large-scale, distributed scientific computing. iDDS extends traditional workload and data management by integrating data-aware execution, conditional logic, and programmable workflows, enabling automation of complex and dynamic processing pipelines. Originally developed for the ATLAS experiment at the large hadron collider, iDDS has evolved into an experiment-agnostic platform that supports both template-driven workflows and a Function-as-a-Task model for Python-based orchestration. This paper presents the architecture and core components of iDDS, highlighting its scalability, modular message-driven design, and integration with systems such as PanDA and Rucio. We demonstrate its versatility through real-world use cases: fine-grained tape resource optimization for ATLAS, orchestration of large Directed Acyclic Graph (DAG) workflows for the Rubin Observatory, distributed hyperparameter optimization for machine learning applications, active learning for physics analyses, and AI-assisted detector design at the electron–ion collider. By unifying workload scheduling, data movement, and adaptive decision-making, iDDS reduces operational overhead and enables reproducible, high-throughput workflows across heterogeneous infrastructures. We conclude with current challenges and future directions, including interactive, cloud-native, and serverless workflow support.

1 Introduction

The growing complexity of scientific computing poses major challenges for workflow orchestration. Modern large-scale experiments in high-energy physics, astronomy, and related domains routinely combine heterogeneous tasks – data management, simulation, machine learning, and analysis – executed across geographically distributed infrastructures. Traditional workflow and workload management systems provide essential scheduling and data handling capabilities, but they are often limited in their ability to express dynamic dependencies, integrate data availability directly into execution logic, or support iterative and adaptive workloads such as hyperparameter optimization and active learning.

The *intelligent distributed dispatch and scheduling (iDDS)* service was designed to address these gaps. iDDS provides a unified orchestration framework that combines data awareness, conditional execution, and flexible workflow representation to enable fine-grained automation at scale. Unlike conventional workload managers, iDDS treats workflows as programmable objects, supporting both template-based orchestration for well-structured pipelines and a Function-as-a-Task model that allows users to express workflows directly in Python. This dual approach makes iDDS suitable for both production-grade data processing campaigns and rapidly evolving machine learning pipelines.

The key capability of iDDS is to coordinate diverse tasks and activities, reducing operational overhead and increasing automation to improve efficiency. Its main features include: (1) integrating fine-grained data availability and movement into workflow logic for data-aware orchestration, (2) supporting complex workflow management such as Directed Acyclic

[a] e-mail: wguan2@bnl.gov (corresponding author)

Published online: 24 January 2026

Graphs (DAGs), conditional branching, and polymorphic workflows, (3) enabling iterative execution with parameter sweeps, iterative sequences, and distributed hyperparameter optimization, (4) providing a code-driven design that allows workflows to be expressed programmatically using Python functions, and (5) integrating with large-scale, geographically distributed workload management systems to ensure scalability.

Originally developed for the ATLAS [1] experiment at the large hadron collider (LHC) [2], iDDS has since evolved into a general-purpose orchestration platform adopted by multiple projects, including the Rubin Observatory [3] and the electron–ion collider (EIC) [4]. It has been successfully integrated with the production and distributed analysis (PanDA) [5] system for workload management and with Rucio [6] for data management. PanDA handles the scheduling of workloads across large-scale, heterogeneous distributed computing resources, while Rucio manages data movement among collaborating institutions. Its applications range from large-scale data reprocessing on tape, to management of complex DAG workflows, to distributed hyperparameter optimization and AI-assisted detector design.

In this paper, we present the concepts, architecture, and implementation of iDDS, and illustrate its versatility through real-world use cases. We conclude with a discussion of achievements, challenges, and future directions.

2 Concepts

2.1 Core concepts

The iDDS system is built around four fundamental concepts that collectively define the structure and logic of how workflows are represented, managed, and executed.

Work.
A *Work* unit is the atomic executable entity within a workflow. Each Work unit encapsulates a self-contained task (such as data transformation, inference, simulation, and filtering) and carries metadata describing its execution state, dependencies, inputs, and outputs. Each task consists of a group of jobs with similar attributes, which serve as the actual units of execution. Work units can be run independently or composed into a larger workflow, with their progress and state tracked throughout their lifecycle.

Workflow.
A *Workflow* is a collection of Work units connected through well-defined dependency relationships. This is represented as a Directed Acyclic Graph (DAG) that encodes the sequence and logic of execution, including parallelization, ordering, and data transfer. Workflows can be specified statically at submission time or dynamically expanded in response to runtime conditions. The DAG-based representation enables users to express complex processing logic while allowing iDDS to optimize overall execution strategies.

Condition.
A *Condition* is a control structure that guides the execution of a workflow by evaluating runtime information, such as the output of previous *Work* units or system metrics. Based on this evaluation, it determines whether and how subsequent Work units are executed. Conditions allow for branching, delays, failure handling, and adaptive behavior within workflows.

Parameter.
Parameters are key-value pairs that are passed into Work units and Workflows to influence their execution behavior. They may define runtime settings, dataset identifiers, model configurations, or execution thresholds. Parameters can be hierarchical and dynamically generated during workflow execution, supporting advanced techniques such as hyperparameter search or data-driven configuration.

2.2 Workflow representation styles

To support a wide variety of use cases, iDDS provides two primary styles of workflow representation, each addressing different user requirements and levels of abstraction. These styles can be used independently or combined, providing users with flexibility to balance between abstraction and control in their workflow design.

Template-based representation.
In this style, users define workflows declaratively using templates. These templates describe the structure and dependencies among Work units, along with the associated Conditions and Parameters. While templates can be dynamically generated, they become static or semi-static once submitted to iDDS. This approach is particularly suited for well-defined workflows such as data production chains, where the workflow logic remains constant but the parameters (e.g., input datasets) vary at runtime. Template-based representation offers high reusability and can be validated prior to execution.

Code-based representation (a.k.a Function-as-a-Task).
For more dynamic and logic-heavy workflows, iDDS supports a code-driven style where users define their workflows programmatically using Python functions. Each function acts as a Task, and its outputs can be analyzed at runtime to determine subsequent execution steps. This style, inspired by the Function-as-a-Service [7] paradigm, allows for complex runtime decisions, loop constructs, and integration with external APIs. It is particularly well-suited for use cases like machine learning pipelines or real-time analysis, where execution logic must adapt based on intermediate results.

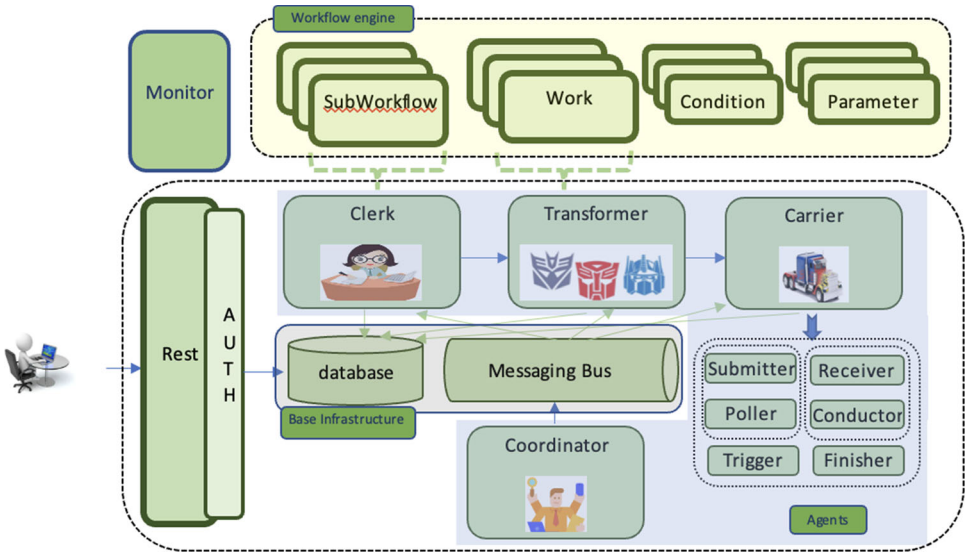


Fig. 1 Schematic overview of the iDDS architecture, illustrating its main components: (1) Workflow engine, (2) base infrastructure, (3) RESTful service, (4) agents, and (5) monitors

3 System architecture

This section describes how the core concepts of iDDS are implemented in practice, including the system's modular design, data flow, and key components responsible for orchestration, communication, and execution.

Figure 1 provides a schematic overview of the iDDS architecture, which comprises five main components: the workflow engine, base infrastructure, RESTful service, agents, and monitors.

3.1 Workflow engine

The workflow engine implements the core abstractions defined in Sect. 2.1 – *Work*, *Workflow*, *Condition*, and *Parameter* – as concrete classes. Each class encapsulates a *Template* (static logic) and a *Metadata* object (dynamic runtime context), together forming the foundation for reusable and adaptive workflows.

- **Templates:** define reusable workflow blueprints with partially fixed structures and parameters.
- **Metadata:** capture dynamic runtime information, enabling workflows to evolve adaptively based on execution context.

These objects are persisted in the database, executed by agents, and tracked throughout their lifecycle.

3.1.1 Directed Acyclic Graph (DAG)

iDDS supports both Directed Acyclic Graph (DAG) and cyclic graph structures at the task and job levels, integrating seamlessly with workload management systems such as PanDA to orchestrate large-scale processing workloads.

At the task level, iDDS implements a Directed Graph (DG) engine that manages acyclic and cyclic dependencies. *Templates* define DAG- and loop-based workflows, while *Metadata* and custom conditions control branching and execution logic. This enables dynamic, adaptive execution paths based on runtime status.

At the job level, DAG support manages fine-grained job dependencies. iDDS automatically evaluates these relationships and incrementally releases downstream jobs as their dependencies are satisfied.

3.1.2 Workflow execution and tracking

Each *Work* unit and *Workflow* is represented in the database with attributes such as status, timestamps, dependencies, input/output datasets, parameter bindings, and *Metadata*. iDDS employs a state machine to track the lifecycle of each *Work* unit, from submission through execution to completion or failure. Execution is initiated by agent components (see Sect. 3.4) and monitored through a combination of periodic polling and event-driven updates.

For execution, each *Work* unit is serialized and submitted to workload management systems such as PanDA or HTCondor [8], where it runs as a distributed job. On the compute node, a lightweight wrapper reconstructs the *Work* object and executes its logic. iDDS agents then monitor job progress, evaluate *Conditions*, propagate and bind *Parameters*, and track results to ensure correct orchestration across the workflow.

3.1.3 Function-as-a-Task

The core idea of Function-as-a-Task is to transparently convert functions into *Work* objects using Python decorators, which are then submitted as *Tasks* to remote workers via

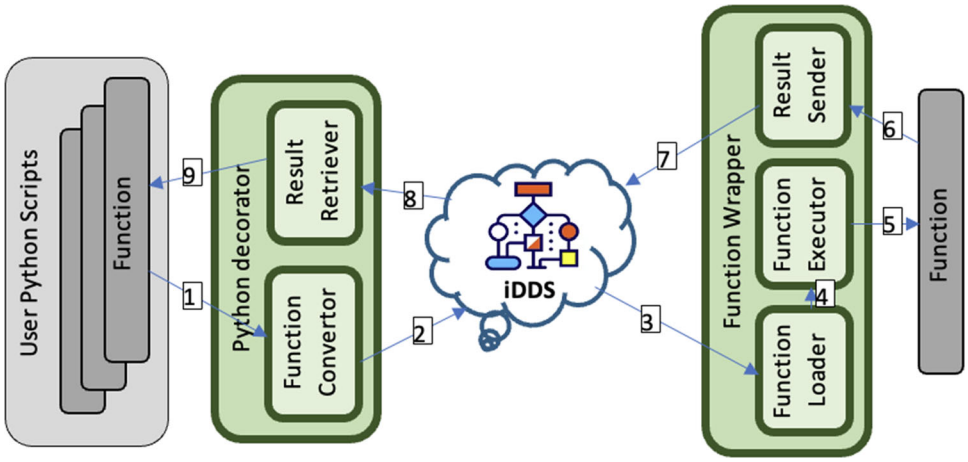


Fig. 2 Function-as-a-Task workflow: (1) Local Python functions are serialized and converted into executable *Work* units using Python decorators; (2) iDDS submits the *Work* units as *Tasks* to remote workers via a workload system; (3) A function wrapper loads the *Work* units and executes the function, collects the results, and sends them back to iDDS; (4) The local Python decorator retrieves and returns the results to the function caller

a workload management system. The user-defined Python script serves as the *Workflow* that controls execution logic (Fig. 2). *Workflow* scripts can run locally or on iDDS-managed clusters. For security, iDDS executes them on a sandboxed HTCondor cluster, isolating untrusted code from the main server infrastructure. This model simplifies the construction of complex workflows – such as machine learning pipelines – by allowing users to express intricate logic and conditional behavior directly in Python.

The *Workflow* unit can be executed either on the client side or on the iDDS server side:

- **Client side**: The *Workflow* may be embedded in a user's main script (e.g., a machine learning script) and executed directly on the client side. If the user submits the main script to PanDA as a PanDA job, the *Workflow* is executed as part of that job.
- **iDDS server side**: Lightweight workflows can be executed on the iDDS server to reduce latency and improve responsiveness. The server hosts a small HTCondor cluster dedicated to running *Workflow* scripts. HTCondor ensures script isolation and prevents excessive concurrent workflow executions that could overload the system.

Work unit execution proceeds in two stages:

- **Serialization and distribution:** Annotated functions are serialized and wrapped as *Work* objects, allowing transparent submission to workload management systems as Tasks. During this stage, the source code and execution environment are packaged into a ZIP archive and uploaded to a centrally managed HTTP cache, where access is protected through x509/OIDC-based authorization. The *Work* objects are then submitted as tasks or jobs, with hooks tracking their execution status and results.
- **Execution:** The *Work* object is deployed across distributed resources through workload management systems, together with its source code and execution environment retrieved from the HTTP cache. An enhanced wrapper reconstructs the *Work* object and executes the function, while results are returned asynchronously via STOMP [9] or RESTful HTTP.

This design enables seamless, scalable execution of complex Python functions on distributed infrastructures such as PanDA, requiring no significant code adaptation while ensuring reliable and responsive result retrieval.

3.2 Base infrastructure: database and event bus

The database and the event bus are the backbone of iDDS, serving as the base infrastructure upon which all other system components are built, as shown in Fig. 3.

3.2.1 Database

iDDS employs a relational database as its central repository for persisting workflow states, work unit dependencies, data collections, and scheduling metadata. The schema incorporates versioning to support both backward compatibility and forward extensibility.

The database serves two critical roles. First, it records user-submitted workflow requests and captures the relationships among workflow objects, ensuring persistence and traceability. Second, it maintains the status of workflow objects for organizing and coordinating system operations. This structured status tracking enables iDDS agents to efficiently identify and operate on tasks that are ready for execution, ensuring reliable and scalable workflow management.

iDDS leverages SQLAlchemy [10] for object–relational mapping, enabling automatic mapping of Python objects to relational tables. SQLAlchemy supports dynamic schema creation and teardown, simplifying testing and deployment, while its compatibility with multiple backends – such as Oracle, PostgreSQL, MySQL, and SQLite – allows iDDS to operate across diverse platforms and migrate between

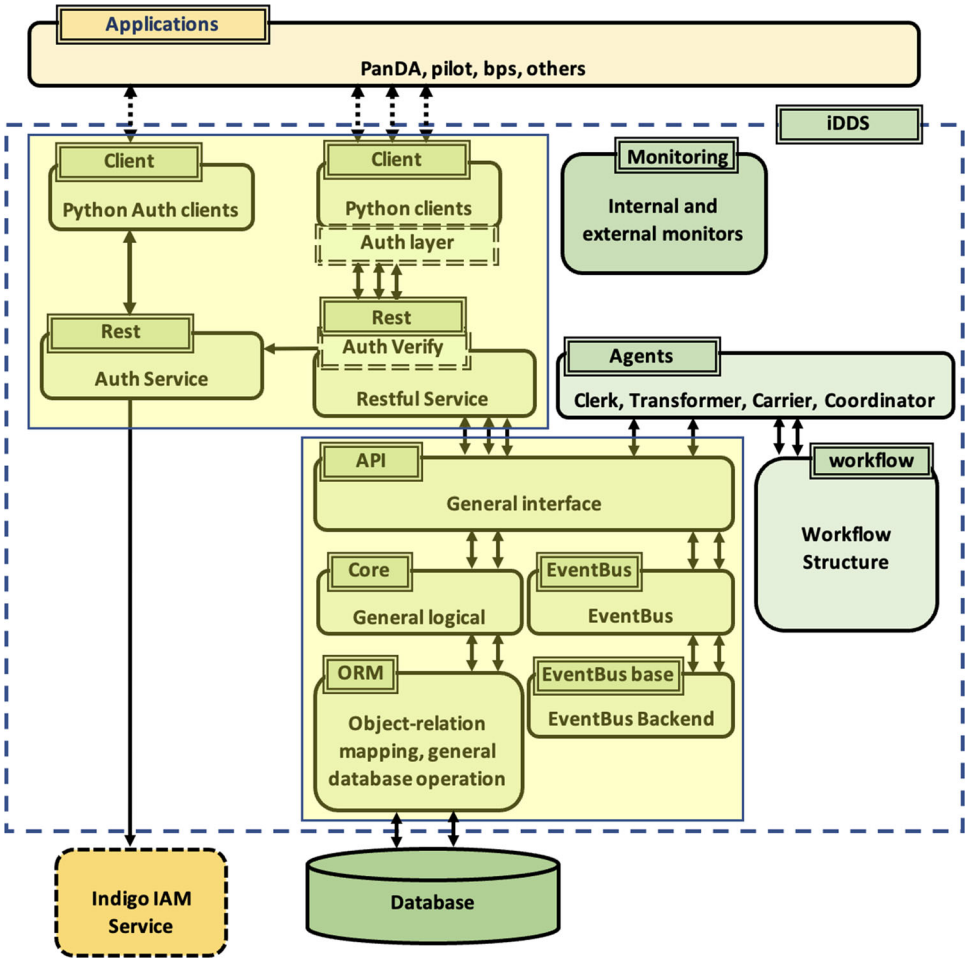


Fig. 3 iDDS architecture

databases with minimal effort. Database schema versioning is managed by Alembic [11], which automates upgrades and downgrades to ensure the schema evolves consistently with the codebase. This approach streamlines maintenance, supports agile development, and preserves data integrity across deployments.

3.2.2 Event bus

iDDS employs an event bus based on the publish-subscribe model to enable asynchronous, decoupled communication among system components. This central communication infrastructure allows event producers and consumers to interact without being directly linked. Events – such as task completions, data availability, error signals, and status updates – are published to the bus and consumed by agents or orchestrators to trigger subsequent execution stages. This design enhances responsiveness and modularity, allowing agents to react immediately to workflow state changes and significantly accelerating system operations.

iDDS implements multiple event bus backends to support various deployment needs (external messaging services like ActiveMQ [12] can also be integrated as backends):

- **LocalEventBus:** A lightweight implementation based on a Python dictionary, enabling fast in-process event delivery. It is suitable for single-process deployments but not usable when multiple iDDS server processes are active.
- **DBEventBus:** A database-backed event bus that stores events persistently, enabling distributed delivery across agents on different hosts. Performance depends on the underlying database system.
- **MsgEventBus:** A high-throughput, distributed event bus built on the ZeroMQ [13] messaging library. While efficient, it requires application-level logic to handle message routing and delivery guarantees.

3.3 RESTful service

The RESTful service offers a standardized interface for both users and external systems, exposing endpoints for workflow submission, status queries, data caching, catalog access, and more. Designed according to REST principles, these APIs ensure ease of integration and broad compatibility with external platforms.

The service is implemented as a Flask [14] application deployed using WSGI [15] daemons behind an Apache

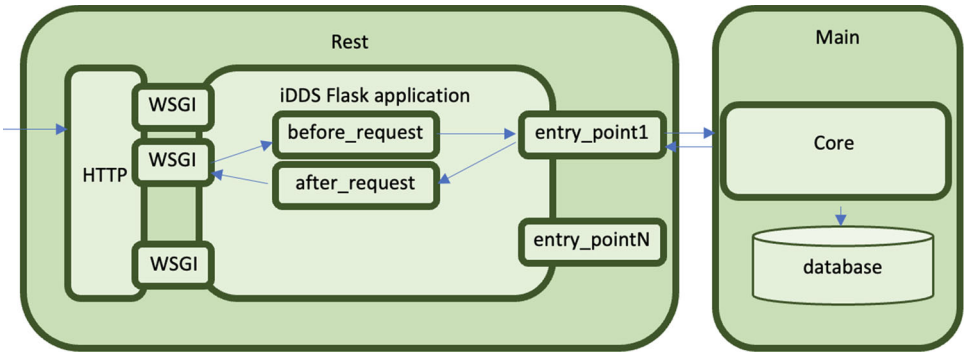


Fig. 4 iDDS RESTful service architecture: (1) the iDDS application is implemented using Flask and served via WSGI daemons behind an Apache HTTP server; (2) authentication and authorization are enforced through filters applied in the *before_request* hook; (3) validated requests are dispatched to corresponding entry points

HTTP server, as shown in Fig. 4. The HTTP server passively listens for incoming HTTPS requests and relays them to WSGI daemons, which invoke the Flask application. Within Flask, authentication and authorization filters are applied via the *before_request* hook. Upon successful authorization, requests are dispatched to the appropriate entry points, which invoke iDDS core functionalities.

3.3.1 RESTful entry points

The RESTful API is organized into multiple logical groups:

- **authentication**: Handles identity flows such as OIDC token generation.
- **ping**: Simple health check endpoint to verify server availability.
- **request**: Submit, update, or query the status of workflow requests.
- **cache**: Upload user-defined code or payloads to the iDDS server.
- **catalog**: Query data availability and status for workflow-related datasets.
- **monitor**: Retrieve monitoring information for workflows and their associated tasks.
- **message**: Send control messages (e.g., abort workflow).
- **log**: Access logs for workflows and work items.

This modular organization ensures a scalable and user-friendly interface for interacting with iDDS.

3.3.2 Authentication and authorization

iDDS supports both OpenID Connect (OIDC) tokens and X.509 certificates for flexible, secure authentication and authorization.

OIDC-based authentication and authorization.
OIDC is supported via integration with Indigo IAM [16], enabling identity federation through providers like Google, CILogon [17], or dex [18]. The flow (illustrated in Fig. 5) consists of three stages:

- **Registration**: Users register with Indigo IAM through a trusted identity provider. During registration, group memberships are assigned to control access rights.
- **Authentication**: The user initiates the flow by contacting the authentication entry point. The system responds with a login URL. The user authenticates through the identity provider, and a token containing identity and group information is returned.
- **Authorization**: When users access protected resources, their token is parsed by the authorization logic in Flask's *before_request*. The token is validated against Indigo IAM, and access permissions are determined based on group roles embedded in the token. The resolved roles and permissions are then cached for some time to optimize later access.

X.509-based authentication and authorization.
For legacy and grid-based environments, iDDS also supports X.509 certificates. The Apache HTTP server integrates with the GridSite library [19] to validate user certificates and enforce access control based on certificate attributes.

3.4 Agents

Agents are stateless, autonomous components responsible for executing and coordinating iDDS workflows. Each agent specializes in a specific role and interacts with the central database and event bus to receive tasks, report progress, and trigger downstream operations. Agents are horizontally scalable and operate asynchronously to support high-throughput processing.

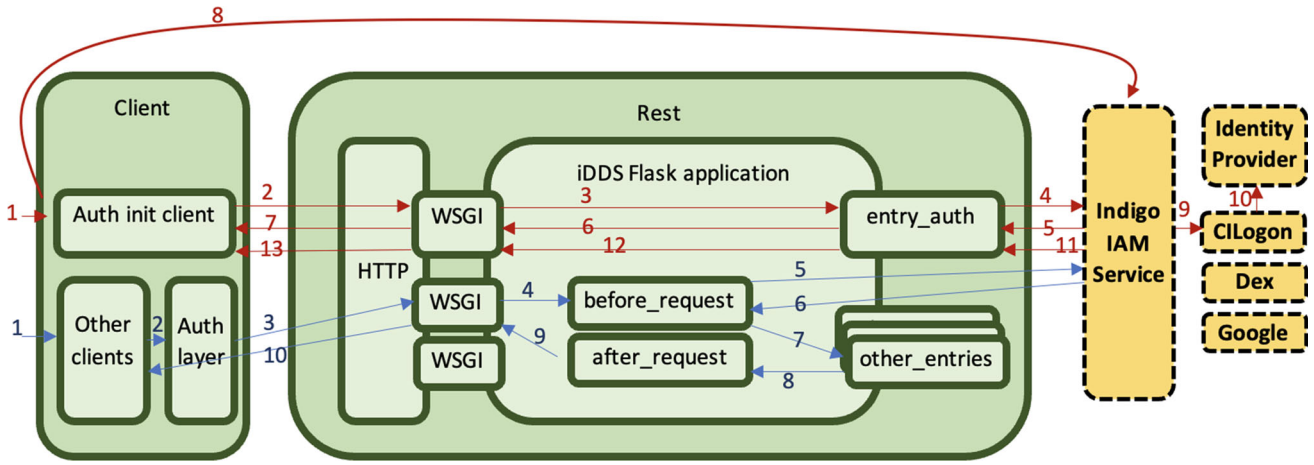


Fig. 5 iDDS OIDC-based IAM: (1) authentication stage (red); (2) authorization stage (blue)

3.4.1 Agent roles

Workflows are initiated via the RESTful service and registered in the central database. The agents then process each workflow as follows:

- **Clerk**: Decomposes Workflow by creating and managing *Work* objects based on defined *Conditions* and *Parameters*.
- **Transformer**: Prepares *Work* objects for execution, ensuring all prerequisites (e.g., input data) are met and selecting appropriate execution environments.
- **Carrier**: Manages communication with external workload management systems, including submission of *Work* objects and monitoring their execution status.
- **Coordinator**: Optimizes event delivery within the event bus by aggregating and prioritizing messages to prevent bottlenecks.

3.4.2 Agent details

Clerk

The Clerk agent decomposes Workflow and generates Work objects. During workflow execution, it evaluates *Condition* objects to determine if new *Work* objects should be created or if the workflow should terminate. When a new *Work* object is needed, the Clerk references *Parameter* objects to generate inputs. It continuously monitors the state of active *Work* objects and, upon their completion, updates related *Condition* and *Parameter* objects to drive downstream execution.

Transformer

The Transformer agent coordinates the execution of *Work* objects. It verifies that all execution prerequisites – such as input data – are met and selects the appropriate workload system based on availability, efficiency, and policy constraints. The Transformer ensures that each *Work* object is executed in an optimal environment, improving resource utilization and minimizing delays.

Carrier

The Carrier agent interfaces with external workload management systems to handle the submission and tracking of the *Work* execution. It uses a message-based channel to exchange status and control information efficiently. The Carrier comprises several sub-agents, each performing specific tasks:

- **Submitter**: Submits *Work* objects to the workload management system and returns tracking metadata.
- **Poller**: Monitors execution status of submitted *Work* objects.
- **Finisher**: Finalizes *Work* objects upon completion or failure.
- **Conductor**: Sends execution status updates to external systems.
- **Receiver**: Consumes status messages from the workload system and updates internal records.
- **Trigger**: Evaluates dependency graphs and triggers downstream *Work* objects when conditions are satisfied.

Coordinator

The Coordinator agent enhances the efficiency of the event bus, which enables asynchronous communication across the iDDS system. In large-scale workloads, the event bus may become congested due to high message volumes (e.g., thousands of parallel job updates). The Coordinator addresses this challenge by:

- **Merging events**: Consolidates similar or redundant messages to avoid unnecessary overhead.
- **Priority management**: Assigns higher priority to critical operations (e.g., *Work* objects completion) over lower-priority updates.

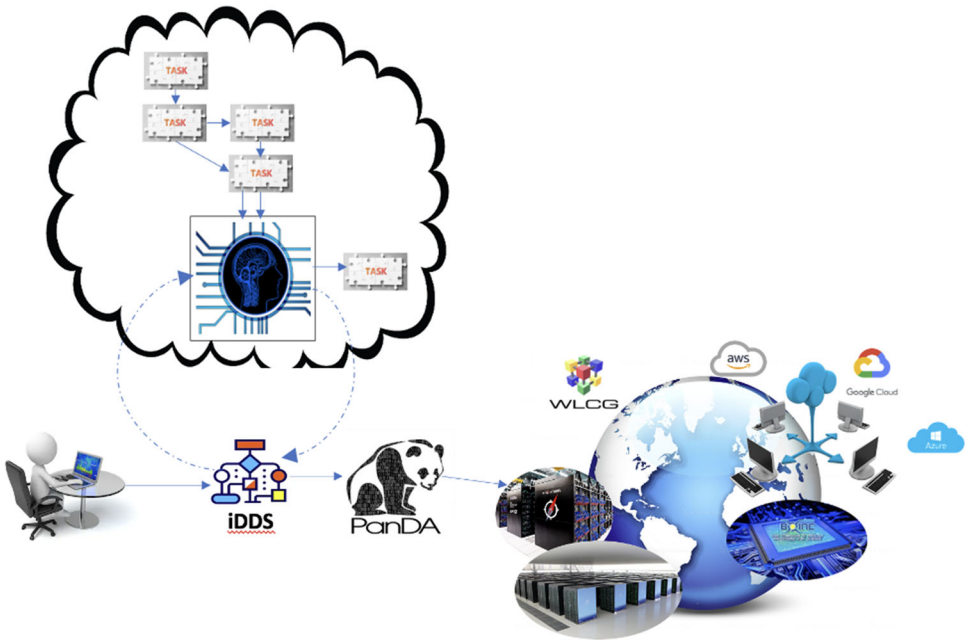


Fig. 6 An integrated workflow with PanDA and iDDS: iDDS automates complex, dynamic workflows while PanDA schedules workloads across large-scale, distributed, and heterogeneous computing resources

By regulating event flow, the Coordinator ensures agents remain responsive and system operations are not delayed by excessive message traffic.

3.4.3 Operation scheduling

Operations in iDDS are fully distributed, with no central scheduler. Specialized agents manage different types of operations and periodically poll the database to identify tasks that have remained idle beyond a configured threshold. Upon detection, they initiate the corresponding actions.

To enhance responsiveness, iDDS integrates an optional event bus that supports asynchronous, event-driven communication. When an agent completes an operation, it emits an event to the bus, triggering other agents responsible for the next steps. This reduces latency and enables prompt reactions to state changes.

To prevent duplicate execution, agents update the status and timestamp of tasks upon triggering, ensuring they are not reprocessed by other agents during polling. While event triggering is the primary mechanism for rapid response, some events may be lost due to network or system issues. To address this, database polling operates in a fallback or "lazy" mode, ensuring that missed events are eventually handled. This hybrid design allows iDDS to balance efficiency and reliability, with the flexibility to disable the event bus when not required.

3.5 PanDA integration

iDDS is tightly integrated with the PanDA Workload Management System [5], enabling distributed job submission and execution across heterogeneous computing resources within the WLCG and other infrastructures. In this integration, *Work* objects from iDDS workflows are translated into PanDA tasks or jobs, allowing iDDS to handle orchestration while PanDA manages large-scale execution.

PanDA is a robust, production-grade workload management system designed for high-throughput, distributed computing. It excels at managing large-scale and heterogeneous resources across institutions and regions. One of PanDA's key strengths lies in its abstraction of underlying compute infrastructure, presenting users with a unified interface for job submission and monitoring. This abstraction allows users to deploy complex workloads without needing detailed knowledge of the underlying systems.

By leveraging PanDA, iDDS gains access to transparent, policy-driven scheduling and execution capabilities across a wide array of computing sites. This is particularly beneficial for large-scale machine learning and data processing workflows, where coordination of compute and data placement is critical.

Figure 6 illustrates a typical integrated workflow, where iDDS manages the dynamic orchestration of tasks, while PanDA schedules and dispatches jobs to globally distributed resources.

3.6 Monitoring

iDDS provides comprehensive monitoring capabilities to ensure the health, transparency, and efficiency of workflow execution. Monitoring tools are used to track system status, workflow progress, and performance metrics such as throughput, latency, failure rates, and resource usage. Logs from agents and services are collected and visualized in real-time dashboards, with integration support for observability

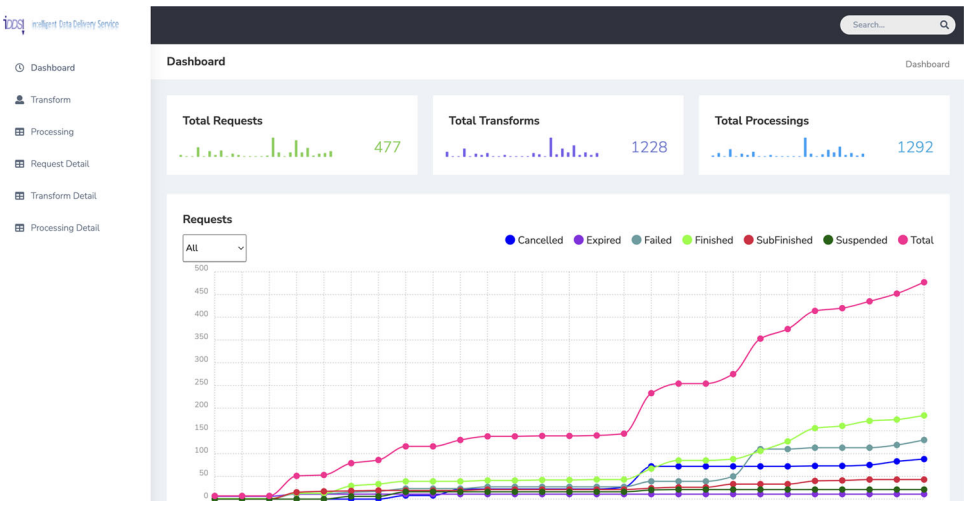


Fig. 7 Example visualization from the iDDS internal monitor, showing the state of Work and Workflow objects

tools like Loki and Grafana to facilitate debugging and performance tuning.

Internally, iDDS includes a built-in monitoring system that continuously tracks the state of both *Workflow* and *Work* objects. This internal monitor offers real-time visibility into the orchestration lifecycle and supports operational diagnostics, as illustrated in Fig. 7.

In addition to the native monitor, iDDS integrates with the PanDA monitoring system to correlate workflow-level metadata with job execution status. This integration provides end-to-end visibility into distributed workloads, enabling users to track both high-level workflow orchestration and low-level job execution metrics, as shown in Fig. 8.

4 Use cases

iDDS has demonstrated its flexibility and scalability across diverse scientific domains to orchestrate complex workflows, manage large-scale data operations, and facilitate AI-driven scientific discovery. This section highlights use cases that illustrate these capabilities in practice.

4.1 Optimization of tape resource utilization for ATLAS

The ATLAS Data Carousel [20,21] aims to maximize the use of cost-effective tape storage over more expensive disk resources.

Traditionally the Data Carousel operated at the dataset level due to limitations in the workflow management (WFM) and distributed data management (DDM) systems, which resulted in significant overhead and required large disk pools to cache entire datasets before processing could begin. In contrast, iDDS enhances the WFM system with file-level granularity, enabling input data to be processed incrementally as it becomes available from tape [22,23]. This fine-grained awareness significantly reduces data staging overhead and eliminates redundant data transfers and caching. This approach allows iDDS to maintain a minimal input data footprint on disk, optimizing the end-to-end resource usage.

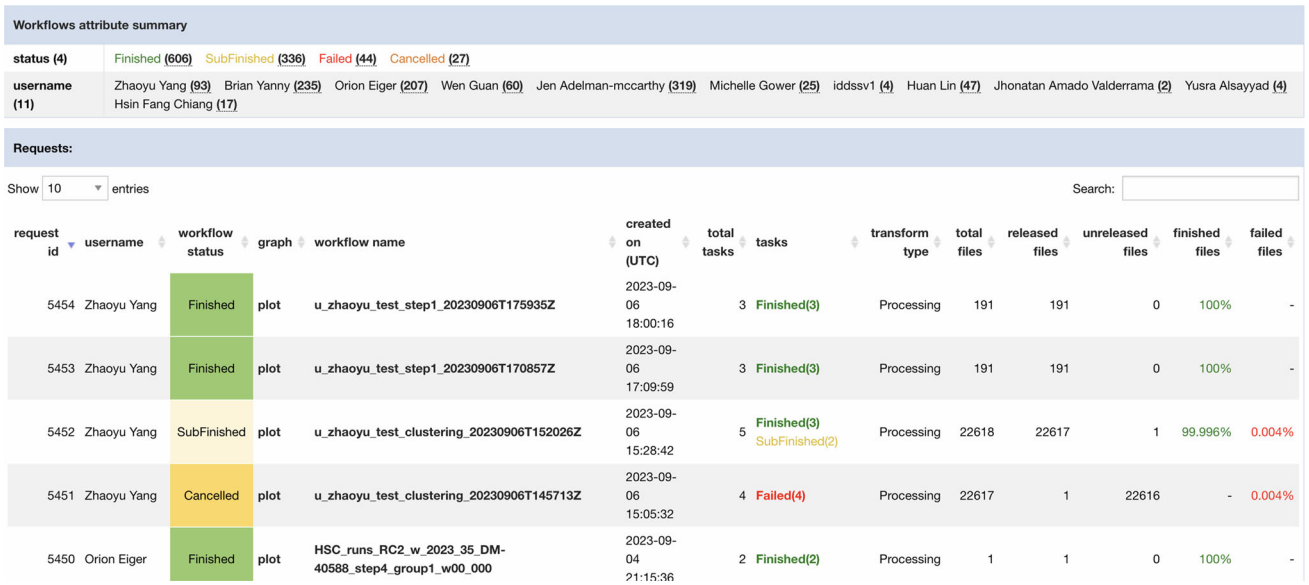
Fully integrated into the ATLAS computing infrastructure since mid-2020, iDDS has played a key role in large-scale data reprocessing campaigns. Figure 9 presents the number of processed requests (one per dataset) from 2021 to 2025, illustrating the sustained throughput achieved with the iDDS-enhanced Data Carousel.

4.2 Complex workflows for Rubin Observatory

The Rubin Observatory (LSST) leverages PanDA as both its workflow and workload management system [24], with iDDS integrated to manage complex task and job dependencies. For each submitted payload, Rubin middleware dynamically generates a workflow graph containing job-level dependencies. These workflows can comprise over 100,000 jobs, forming the vertices of a large DAG.

iDDS plays a central role by incrementally releasing jobs based on dependency resolution and messaging triggers. When a job completes, iDDS triggers agents to evaluate dependent jobs and release them as appropriate. At the task level, iDDS manages the execution and release of finalizing steps such as merge tasks.

This system has been in production since mid-2021, with a dedicated PanDA–iDDS instance deployed at SLAC to support Rubin Observatory's data production campaigns. Over the past few years, it has processed numerous tasks across multiple sites for Rubin ComCam (Commissioning Camera) and DRP (Data Release Production) workflows, as illustrated in Fig. 10. An example task-level DAG from a Rubin workflow is shown in Fig. 11.



Workflows attribute summary

status (4)	Finished (606) SubFinished (336) Failed (44) Cancelled (27)
username (11)	Zhaoyu Yang (93) Brian Yanny (235) Orion Eiger (207) Wen Guan (60) Jen Adelman-mccarthy (319) Michelle Gower (25) iddssv1 (4) Huan Lin (47) Jhonatan Amado Valderrama (2) Yusra Alsayyad (4) Hsin Fang Chiang (17)

Requests:

Show 10 entries Search:

request id	username	workflow status	graph	workflow name	created on (UTC)	total tasks	tasks	transform type	total files	released files	unreleased files	finished files	failed files
5454	Zhaoyu Yang	Finished	plot	u_zhaoyu_test_step1_20230906T175935Z	2023-09-06 18:00:16	3	Finished(3)	Processing	191	191	0	100%	-
5453	Zhaoyu Yang	Finished	plot	u_zhaoyu_test_step1_20230906T170857Z	2023-09-06 17:09:59	3	Finished(3)	Processing	191	191	0	100%	-
5452	Zhaoyu Yang	SubFinished	plot	u_zhaoyu_test_clustering_20230906T152026Z	2023-09-06 15:28:42	5	Finished(3) SubFinished(2)	Processing	22618	22617	1	99.996%	0.004%
5451	Zhaoyu Yang	Cancelled	plot	u_zhaoyu_test_clustering_20230906T145713Z	2023-09-06 15:05:32	4	Failed(4)	Processing	22617	1	22616	-	0.004%
5450	Orion Eiger	Finished	plot	HSC_runs_RC2_w_2023_35_DM-40588_step4_group1_w00_000	2023-09-04 21:15:36	2	Finished(2)	Processing	1	1	0	100%	-

Fig. 8 Example visualization from the PanDA-integrated monitor, correlating iDDS workflows with PanDA job execution

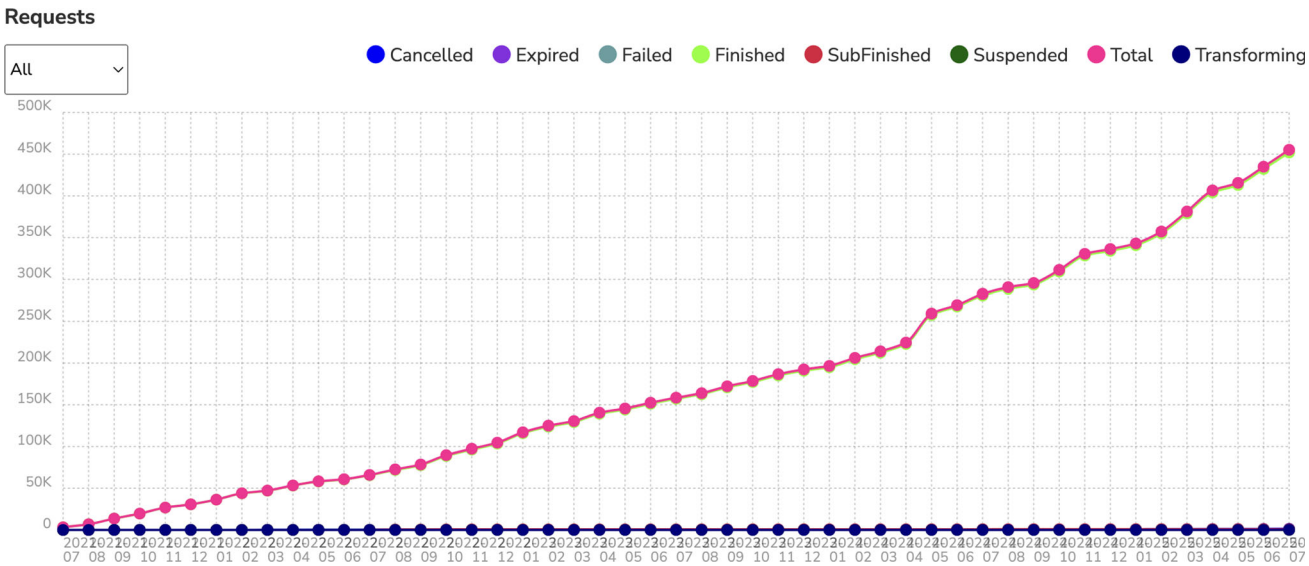


Fig. 9 Data reprocessing throughput (2021–2025) using the fine-grained Data Carousel with iDDS. The y-axis represents the number of processed requests, each corresponding to a dataset that may span hundreds of gigabytes or more

4.3 Distributed hyperparameter optimization

Hyperparameter optimization (HPO) [25] is a fundamental task in machine learning (ML), aimed at tuning the parameters that govern the training process to achieve optimal model performance. Effective HPO often requires launching and evaluating a large number of training jobs in parallel, making it computationally expensive and technically complex – especially in geographically distributed environments such as grids, HPC systems, and cloud platforms.

iDDS addresses these challenges by providing a fully automated, scalable platform for distributed HPO [26]. It seamlessly orchestrates workloads across heterogeneous CPU/GPU resources, and efficiently collects and integrates upstream training results to guide subsequent iterations. iDDS is particularly well-suited for ML workflows involving iterative computation and dynamic decision-making (Fig. 12).

A single iteration of HPO workflow consists of the following steps:

1. **Candidate sampling**: iDDS centrally explores the hyperparameter space using advanced search strategies such

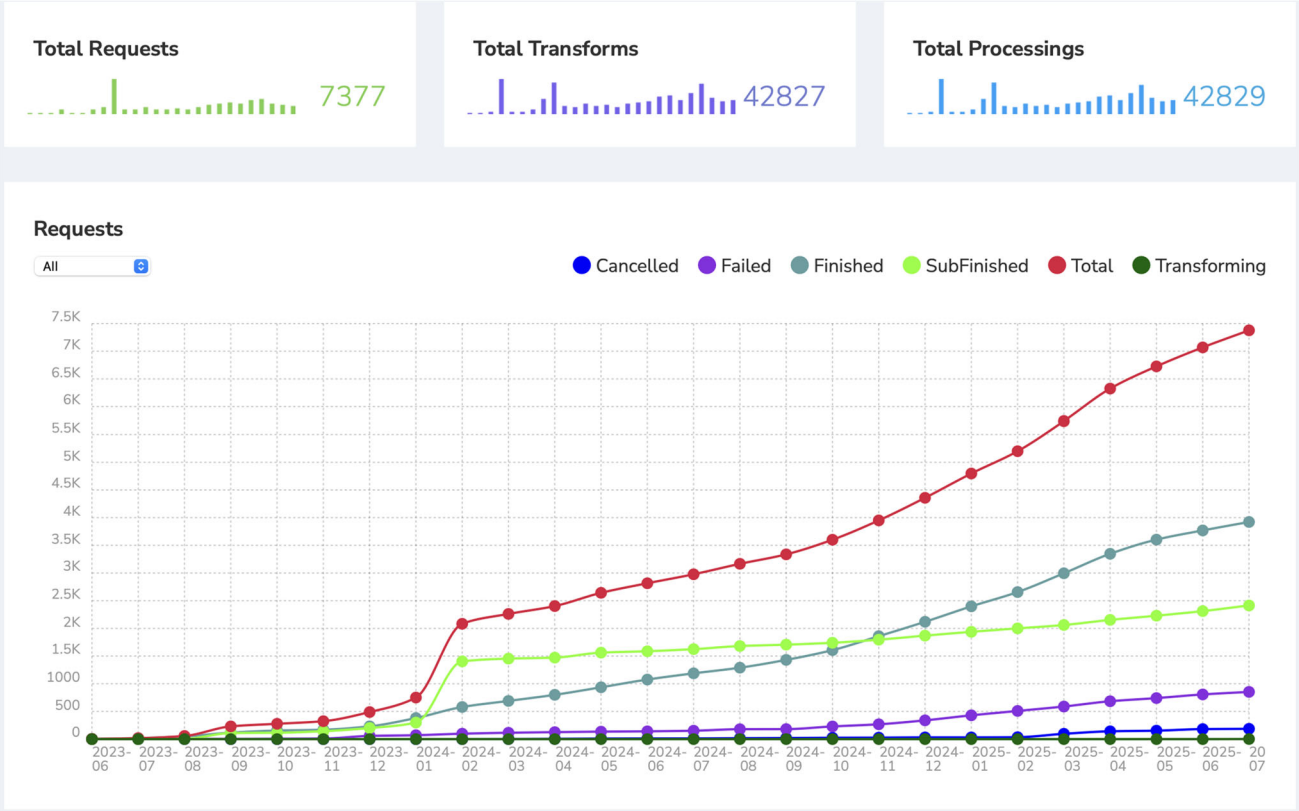


Fig. 10 Rubin Observatory production activity using the iDDS–PanDA system since late 2021. The monitor displays data from mid-2023 onward, as earlier data have been archived. Each request corresponds to a workflow, and each transform represents a task comprising hundreds of thousands of jobs

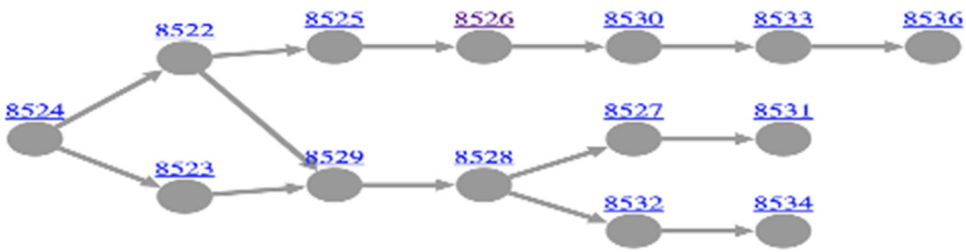


Fig. 11 Visualization of a DAG showing task-level dependencies within a Rubin Observatory workflow

as Bayesian optimization [27], and chooses candidate parameter sets.

2. **Training with each candidate and evaluation**: The generated candidates are asynchronously dispatched to distributed computing sites via PanDA for model training and calculating performance metrics for each candidate.
3. **Search space refinement**: iDDS collects performance metrics to refine the search space and initiates the next iteration based on the refined space.

This iterative process continues until the best-performing hyperparameters and corresponding trained models are identified. To further improve efficiency, iDDS supports segmented HPO, enabling the simultaneous optimization of multiple machine learning models. This approach increases throughput, reduces bias, and is well suited for ensemble learning and comparative model studies.

This HPO service is currently in production for ATLAS machine learning projects such as FastCaloGAN [28], where it has demonstrated strong scalability and performance. Although originally developed for ATLAS, the system is designed to be experiment-agnostic and can be readily extended to support HPO in a broad range of scientific and industrial ML applications.

4.4 Active learning

Active learning (AL) [26,29,30] is an iterative machine learning technique that strategically refines the parameter space by incorporating feedback from previous results. Rather than processing the entire dataset in a single pass,

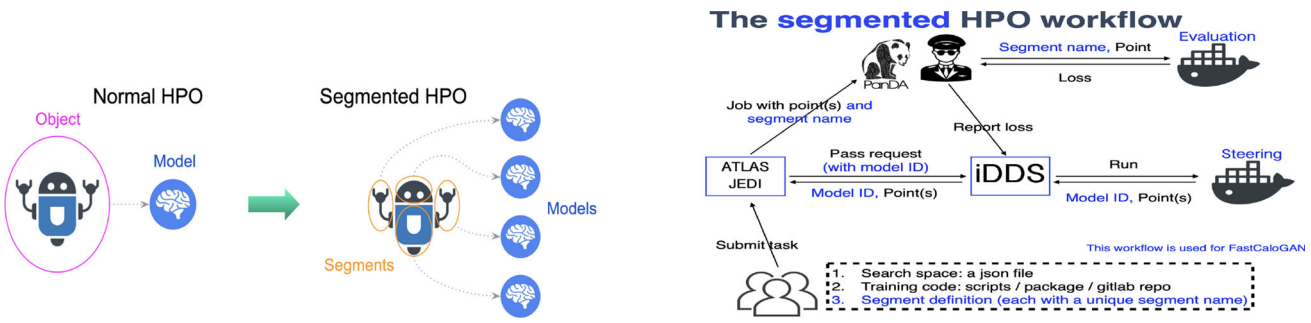


Fig. 12 Architecture of the iDDS hyperparameter optimization service

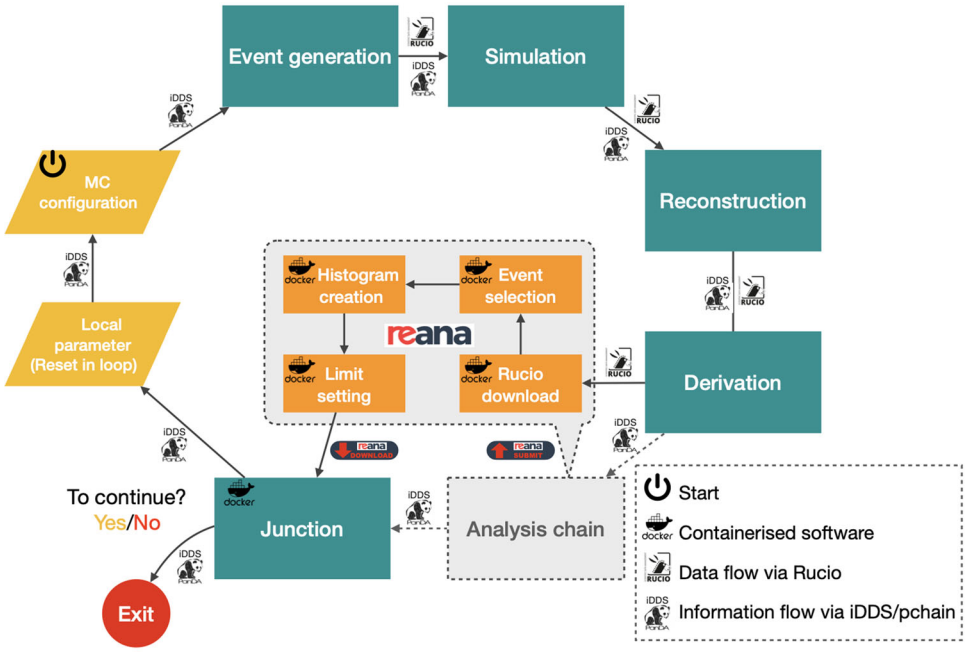


Fig. 13 A schematic view of the active learning workflow used in the $H \to ZZ_d \to 4l$ analysis

AL selectively samples and labels new data points based on model uncertainty, enabling more efficient searches – especially in domains such as new physics analyses.

iDDS, in coordination with PanDA, enables fully automated, intelligent AL workflows by orchestrating iterations of training, uncertainty estimation, data selection, and retraining. Its conditional logic and dynamic branching capabilities make it particularly well-suited for AL use cases that demand tight feedback loops and scalable computation with minimal human intervention.

A representative AL workflow has been implemented for the optimized search of $H \to ZZ_d \to 4l$ [31], as illustrated in Fig. 13. The workflow comprises two primary stages: the production chain and the analysis chain. The *production chain* starts from a template Monte Carlo (MC) configuration and proceeds through simulation and reconstruction to generate a Derived AOD (DAOD) sample for each physics parameter point. Once the DAOD is available, the *analysis chain* is triggered, in which PanDA jobs execute a REANA workflow [32] for data analysis and run Bayesian optimization to evaluate statistical significance, identify regions of interest (e.g., excesses between expected and observed limits), and propose new parameter points.

Throughout this process, iDDS manages all orchestration steps, ensuring coordination between components, data flow, and dynamic generation of new workloads – without human intervention. The workflow has successfully demonstrated AL-driven re-analysis capabilities and was published in an ATLAS Public Note [31]. A second AL workflow is currently under development for a generic Heavy Higgs $\to WW$ search.

4.5 AI-assisted detector design at EIC (AID2E)

The AI-assisted detector design for the electron-ion collider (AID2E [33]) project showcases the effective use of iDDS in orchestrating iterative workflows involving geometry generation, simulation, reconstruction, and analysis. Metadata and performance metrics from previous runs are incorpo-

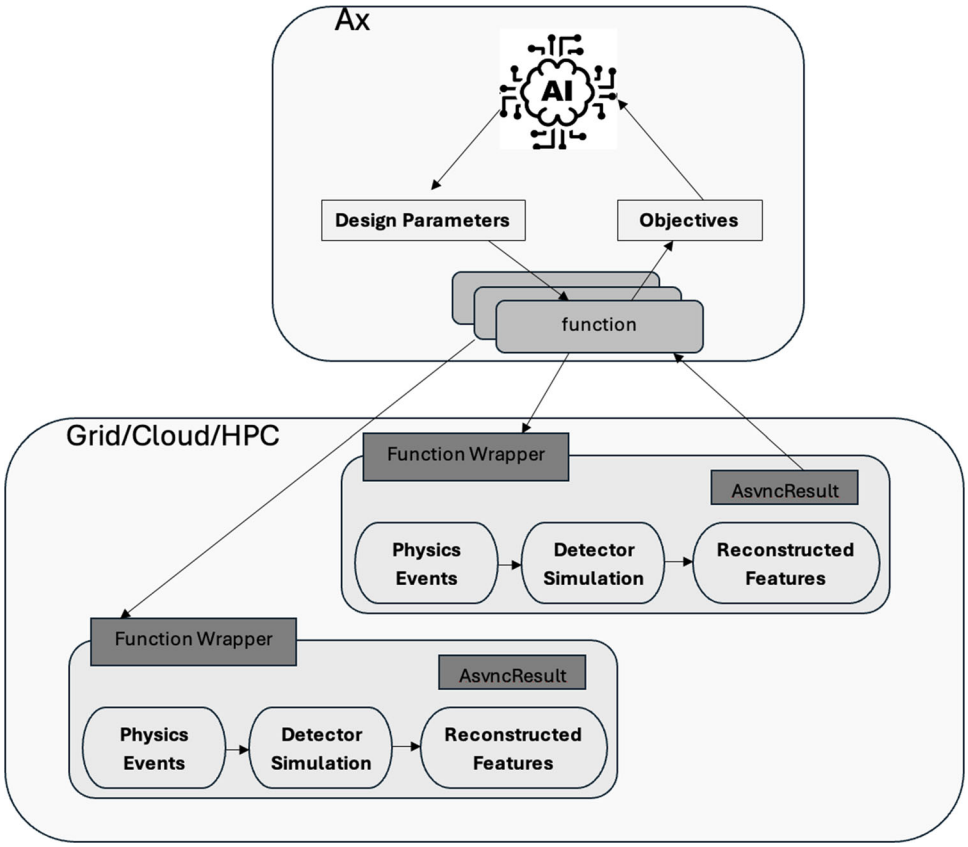


Fig. 14 AID2E: the Function-as-a-Task model maps local evaluation logic to distributed computing resources using a Python decorator that transforms a local function into a distributed task

rated to guide the subsequent iterations, enabling efficient exploration of detector configurations (Fig. 14).

Due to the complexity of AID2E, directly converting its workflow into distributed PanDA jobs is non-trivial and typically requires substantial restructuring. To simplify this process, we apply the iDDS Function-as-a-Task model, which streamlines the transformation of local workflows into distributed applications. Function-as-a-Task uses Python decorators to convert local functions in the AID2E code into distributed tasks composed of multiple concurrent jobs. With minimal code changes, users can transparently offload computations to PanDA-managed resources, enabling scalable parallel execution. Building on this capability, we have integrated Function-as-a-Task into AID2E. Each optimization cycle evaluates multiple detector configurations through simulation and assesses performance metrics such as resolution. These results guide the generation of new configurations in subsequent iterations, continuing until performance targets are met. Throughout the process, iDDS ensures reproducibility and scalability by managing parameter sets, tracking metadata, and maintaining dependencies across iterations. This enables optimization algorithms to iteratively refine detector designs based on data-driven feedback.

By leveraging iDDS and PanDA, AID2E achieves scalable, high-throughput execution across diverse computing infrastructures – including Grid, Cloud, and HPC environments – significantly accelerating the detector design optimization process.

5 Conclusion and outlook

iDDS provides a unified, scalable, and programmable platform for managing distributed workloads and data orchestration in large-scale scientific environments. The template-based and code-based workflow representations provide users with great flexibility to define workflows that accommodate dynamic runtime conditions and resource constraints. Its support for conditional logic, parameter passing, and result tracking makes it particularly well-suited for modern scientific workflows involving simulation, reconstruction, data analysis, and AI.

Integration with established infrastructures, such as PanDA and Rucio, allows iDDS to minimize the need for new resource configurations and leverage existing diverse resources. Its asynchronous result retrieval service, metadata handling, and execution tracking provide a reliable foundation for reproducible and efficient computation.

Looking ahead, iDDS will continue to evolve to support emerging computational paradigms. Ongoing developments aim to enhance the user experience, broaden support for interactive and serverless workflows, and integrate with cloud-native ecosystems.

As scientific computing becomes increasingly data-driven and iterative, iDDS is positioned to be a cornerstone of intelligent workflow orchestration, enabling new discoveries through automation, adaptability, and scalability.

Acknowledgements This work was done as part of the distributed computing research and development program within the ATLAS Collaboration. We thank our ATLAS colleagues for their support. In particular, we wish to acknowledge the contributions of the ATLAS Distributed Computing (ADC) team. Copyright 2024 CERN for the benefit of the ATLAS Collaboration. Reproduction of this article or parts of it is allowed as specified in the CC-BY-4.0 license. This manuscript has been authored by employees of Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy. The publisher by accepting the manuscript for publication acknowledges that the United States Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The iDDS project was initially supported by IRIS-HEP from 2020 to 2022 through the National Science Foundation under Cooperative Agreement OAC-1836650.

Data Availability Statement This manuscript has no associated data. [Authors' comment: Data sharing not applicable to this article as no datasets were generated or analysed during the current study.]

Code Availability Statement Code/software will be made available on reasonable request. [Authors' comment: The code/software generated during and/or analysed during the current study is available from the corresponding author on reasonable request.]

Funded by SCOAP[3].

References

1. ATLAS Collaboration, The ATLAS Experiment at the CERN large hadron collider. J. Inst. **3**, 08003 (2008)
2. L. Evans, P. Bryant (eds.), LHC machine. J. Inst. **3**, 08001 (2008)
3. Z. Ivezic et al., LSST: from science drivers to reference design and anticipated data products. Astrophys. J. **873**(2), 111 (2019)
4. F. Willeke, J. Beebe-Wang et al., Electron ion collider conceptual design report 2021. Technical report, Brookhaven National Lab. and Jefferson Lab. (2021). https://doi.org/10.2172/1765663
5. T. Maeno et al., PanDA: production and distributed analysis system. Comput. Softw. Big Sci. **8**, 4 (2024). https://doi.org/10.1007/s41781-024-00114-3
6. M. Barisits et al., Rucio: scientific data management. Comput. Softw. Big Sci. **3**, 11 (2019)
7. funcX – federated function as a service. https://funcx.readthedocs.io/en/latest/
8. HTCondor – a software system that creates a high-throughput computing environment. https://htcondor.org/
9. STOMP Protocol specification. https://stomp.github.io/. Accessed: 2025-07-14
10. M. Bayer, SQLAlchemy: Python SQL toolkit and ORM. https://www.sqlalchemy.org. Version 2.0 (2024)
11. M. Bayer, Alembic: database migration tool. https://alembic.sqlalchemy.org. Version 1.13 (2024)
12. Apache ActiveMQ – flexible and powerful open source multi-protocol messaging. https://activemq.apache.org/
13. ZeroMQ – an open-source universal messaging library. https://zeromq.org/
14. Flask – a lightweight WSGI web application framework. https://flask.palletsprojects.com/
15. Python Web Server Gateway Interface (WSGI). https://peps.python.org/pep-3333/
16. Indigo Identity and Access Management (IAM) Service. https://indigo-iam.github.io/v/current/docs/
17. CILogon – an integrated identity and access management platform for science. https://www.cilogon.org/
18. Dex – a federated OpenID connect provider. https://dexidp.io/
19. A. McNab, The GridSite Web/Grid security system. J. Phys. Conf. Ser. **219**, 062058 (2010)
20. M. Barisits et al., ATLAS Data Carousel. EPJ Web Conf. **245**, 04035 (2020). https://doi.org/10.1051/epjconf/202024504035
21. M. Borodin et al., The ATLAS Data Carousel Project status. EPJ Web Conf. **251**, 02006 (2021). https://doi.org/10.1051/epjconf/202125102006
22. W. Guan et al., Towards an intelligent data delivery service. EPJ Web Conf. **245**, 04015 (2020). https://doi.org/10.1051/epjconf/202024504015
23. W. Guan et al., An intelligent data delivery service for and beyond the ATLAS experiment. EPJ Web Conf. **251**, 02007 (2021). https://doi.org/10.1051/epjconf/202125102007
24. E. Karavakis et al., Integrating the PanDA workload management system with the Vera C. Rubin Observatory. EPJ Web Conf. **295**, 04026 (2024). https://doi.org/10.1051/epjconf/202429504026
25. J. Bergstra et al., Algorithms for hyper-parameter optimization. Adv. Neural. Inf. Process. Syst. **24**, 2546 (2011)
26. W. Guan et al., Distributed machine learning workflow with PanDA and iDDS in LHC ATLAS. EPJ Web Conf. **295**, 04019 (2024). https://doi.org/10.1051/epjconf/202429504019
27. J. Snoek, Practical Bayesian optimization of machine learning algorithms. (2012). https://doi.org/10.48550/arXiv.1206.2944. arXiv:1206.2944
28. M. Javurkoval, The fast simulation chain in the ATLAS experiment. EPJ Web Conf. **251**, 03012 (2021). https://doi.org/10.1051/epjconf/202125103012
29. K. Cranmer, Active learning for excursion set estimation. (2019). https://indico.cern.ch/event/708041/contributions/3269754/
30. B. Settles, *Active Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning* (Morgan & Claypool Publishers, 2012)
31. ATLAS Collaboration, Demonstrating an active learning driven pipeline for optimised analysis reinterpretation: an extended search for Higgs bosons decaying into four-lepton final states via an intermediate dark Z boson. (2023). https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/PUBNOTES/ATL-PHYS-PUB-2023-010/
32. REANA. https://reanahub.io/
33. M. Diefenthaler, C. Fanelli, L.O. Gerlach, W. Guan, T. Horn, A. Jentsch, M. Lin, K. Nagai, H. Nayak, C. Pecar, K. Suresh, A. Vossen, T. Wang, T. Wenaus, AI-assisted detector design for the EIC (AID(2)E). (2024). https://arxiv.org/abs/2405.16279