Computing and Software for Big Science (2024) 8:4
https://doi.org/10.1007/s41781-024-00114-3

REVIEW

PanDA: Production and Distributed Analysis System

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

Received: 15 August 2023 / Accepted: 17 January 2024

Abstract
The Production and Distributed Analysis (PanDA) system is a data-driven workload management system engineered to operate at the LHC data processing scale. The PanDA system provides a solution for scientific experiments to fully leverage their distributed heterogeneous resources, showcasing scalability, usability, flexibility, and robustness. The system has successfully proven itself through nearly two decades of steady operation in the ATLAS experiment, addressing the intricate requirements such as diverse resources distributed worldwide at about 200 sites, thousands of scientists analyzing the data remotely, the volume of processed data beyond the exabyte scale, dozens of scientific applications to support, and data processing over several billion hours of computing usage per year. PanDA's flexibility and scalability make it suitable for the High Energy Physics community and wider science domains at the Exascale. Beyond High Energy Physics, PanDA's relevance extends to other big data sciences, as evidenced by its adoption in the Vera C. Rubin Observatory and the sPHENIX experiment. As the significance of advanced workflows continues to grow, PanDA has transformed into a comprehensive ecosystem, effectively tackling challenges associated with emerging workflows and evolving computing technologies. The paper discusses PanDA's prominent role in the scientific landscape, detailing its architecture, functionality, deployment strategies, project management approaches, results, and evolution into an ecosystem.

Keywords Workload management · Distributed computing · Exascale · Heterogeneous computing

Introduction

The Production and Distributed Analysis (PanDA) system is a data-driven workload management system engineered to operate at the LHC [1] data processing scale. It has been developed to meet the intricate requirements for data (re) processing, detector simulation, and physics analysis in the ATLAS experiment [2], where diverse resources are distributed at about 200 computing centers spanning more than 40 countries, thousands of scientists analyze the data remotely, the volume of processed data is beyond the exabyte scale, dozens of scientific applications and emerging workflows are supported, and data processing requires several billion hours of computing usage per year. A design principle of PanDA was the tight integration of workload and data management, i.e., the tight integration with the distributed data management system Rucio [3], to allow ATLAS to manage and process close to 100 petabytes of data per year in an efficient way. PanDA was also designed to have the flexibility to adapt to emerging computing technologies in processing, storage, networking, and distributed computing middleware. The system has seamlessly integrated a wide range of computing resources, while the spectrum of computing options keeps increasing across the Worldwide LHC Computing Grid (WLCG) [4], volunteer computing, High-Performance Computing (HPC) operating independently of WLCG, Leadership Computing Facilities (LCFs) [5, 6], and commercial clouds. The scalability and flexibility make PanDA well-suited for adoption by various exabyte-scale scientific communities. The interest in PanDA by other big data sciences, such as the Vera C. Rubin Observatory [7] and the sPHENIX experiment [8], brought the primary motivation to generalize PanDA for the High Energy Physics community,

✉ Tadashi Maeno
tmaeno@bnl.gov

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

other data-intensive sciences, and wider Exascale scientific domains.

Many other experiments and scientific programs also have broad needs for large-volume data processing on diverse and geographically distributed resources. While some such experiments have adopted PanDA, many other systems have been developed as well over the years and are in use across the community. Prominent examples include JAliEn [9] for the ALICE experiment [10], GlideinWMS [11] for the CMS experiment [12], DIRAC [13] for the LHCb [14], Belle II [15], and CTA [16] experiments, and Pegasus WMS [17] for the LIGO experiment [18]. Signature features of PanDA within this range of systems include its extreme scalability, grounded in its use of still-current technologies such as REST interfaces fronting horizontally scalable web services; tight integration with data management, in particular Rucio, enabling sophisticated choreography of data-intensive workflows; modular encapsulation of resource specifics; an integral workflow management system enabling highly complex workflows to be defined and executed; and comprehensive monitoring ranging from high-level system overviews to drill-down details serving analysis users, operations experts, and system developers and debuggers.

This paper is structured as follows: In Sect. "Concepts" we outline the main concepts for the rest of this paper. Section "System Architecture" describes the system architecture of the PanDA system with implementation highlights. Functionality details of PanDA and its adaptability are discussed in Sect. "Functionality Details", followed by descriptions of typical PanDA deployment in Sect. "Infrastructure and Installation" and project management strategies in Sect. "Project Management". Section "Results and Experience" is dedicated to presenting results and sharing valuable experiences, while Sect. "The evolving PanDA Ecosystem" focuses on tracing the evolution of the PanDA ecosystem. Finally, in Sect. "Conclusions", we close the paper with a summary and an outlook on future challenges to expand PanDA for the next generation of big data science experiments.

Concepts

The main concepts of the PanDA system are *Computing and storage resources*, *Worker node*, *PanDA queues*, *Virtual Organization*, *User*, *Workflow*, *Workload*, *Task*, *Job*, *Worker*, *Priority*, and *Global share*.

Computing and Storage Resources and Worker Node

Computing resource providers offer computing resources with various processing capabilities, such as the grid, HPC centers, and commercial cloud services. A worker node represents various entities depending on the workload or resource configuration, such as a (virtual) host, a cluster of multiple hosts, or a slot on a host. It encompasses a combination of CPUs, GPUs/accelerators/co-processors, memory, and disk space.

Storage resource providers accommodate data storage needs. A storage resource is composed of a persistent data storage with disk or tape, and a storage management service running on top of it. The association between computing and storage resources can be arbitrary, but in most cases resources from the same provider are associated with each other. PanDA integrates diverse and geographically distributed computing and storage resources to provide a consistent interface to users.

PanDA Queues

A PanDA queue represents a group of worker nodes in a computing resource with a certain set of attributes describing their specifications and requirements, such as CPU type and power, memory, disk space, and walltime limit. For example, certain resource providers make computing resources available through batch queues in the underlying batch systems. In such cases, one PanDA queue is typically defined for each batch queue. It is also possible to define multiple PanDA queues for a single batch queue, allowing them to share worker nodes underneath. A PanDA queue is associated with one or more storage resources for automatic data motion, which is described in Sect. "Automatic Data Distribution and Aggregation".

Virtual Organization

A virtual organization (VO) refers to a dynamic set of individuals defined around a set of resource-sharing rules and conditions. Its members are geographically apart but work for a common objective, such as a scientific experiment, for example, the ATLAS collaboration, or research program. VOs are implemented in VOMS [19] or Indigo IAM [20], and can define groups with certain criteria, such as affiliation, role, and scientific objective, e.g. a working group describes a group of people who collaborate together to achieve a specific objective in the VO.

User

A user is a person interacting with PanDA and is uniquely identified by a username. PanDA authenticates and authorizes users to allow or reject access to computing and storage resources based on their profile information. PanDA has an Identity and Access Management (IAM) scheme fully compliant with OIDC/OAuth2.0 [21, 22], capable of identity federation among scientific and academic identity providers.

PanDA also supports legacy X.509 authentication. More details of the authentication and authorization mechanism can be found in Sect. "Authentication and Authorization".

Each user has the flexibility to belong to one or more VOs. Furthermore, a user can hold one or more roles within each VO, allowing them to have different responsibilities and privileges. There are six roles:

- *Scientist*s run private workloads for their analysis on PanDA without any special privileges. The difference between private and managed workloads is described in Sect. "Workflow and Workload".
- *Production manager*s submit coordinated, large-scale managed workloads on behalf of the VO groups. For example, this could be campaigns for Monte Carlo simulation, reprocessing and derivations, or testing and validating new releases of the experiment's software.
- *Coordinator*s define resource allocations among groups and activities in the VO. They are also responsible for providing regular reports to auditors and funding agencies.
- *Computing center administrators* are administrators at computing centers providing computing resources.
- *Experts* are developers of the PanDA system.
- *Shift team members* supervise the system operation and execution of workloads on their duties for a period of time, taking first-level actions, e.g. contacting computing center administrators and developers, in case of failures or anomalies.

Workflow and Workload

Figure 1 represents the different concepts in the workflow hierarchy, which will be described in more detail in this section about workflows and workloads, and in the upcoming Sects. "Task" and "Job and Worker" about tasks and jobs.

A workflow is a top-level work entity in the hierarchical organization of the processing. One workflow corresponds to a scientific objective defining intermediate steps for the user to accomplish. Illustrative examples of scientific objectives include the search for new particles, fine-tuning detectors in multi-dimensional parameter spaces, conducting sky surveys to identify astronomical objects, and executing bulk data processing for experiment collaborations.

A workload is a software program or application that utilizes computing resources to achieve one of the intermediate steps in a workflow. A workflow comprises a group of constituent workloads and their topological relationship. The user runs all workloads or a subset of workloads in a workflow on PanDA. It is possible to run managed workloads that are organized and coordinated centrally at the VO level to utilize a substantial amount of computing resources within the VO. Individual users can also run workloads independently, using their own allocations on behalf of VO groups with specific objectives, such as studying particular physics processes. A workflow may comprise both managed workloads and those initiated by individual users.

While the workflow in Fig. 1 is a simple linear sequence of steps, the dependencies and relations between steps can be more complex. For example, it is possible for each workload to take multiple output data from upstream workloads as input and feed its output data to multiple downstream workloads. The initiation of a downstream workload occurs either (1) when the upstream workload is completed, ensuring that all input data are ready, or (2) while the upstream workload is still running and partial input data are available. The configuration for this behavior is flexible and can be tailored to specific workflow requirements. In scenario

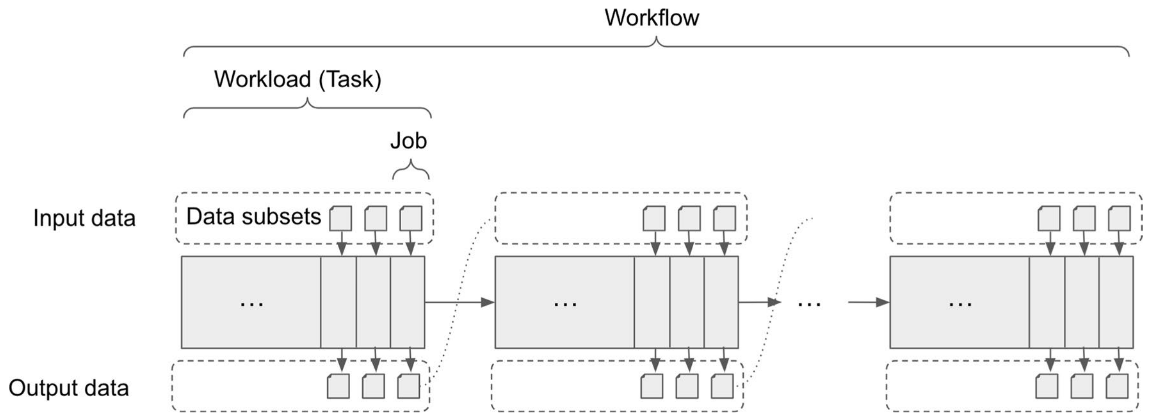


Fig. 1 Representation of a basic workflow example composed of a sequence of workloads that depend on the previous step. Each workload is divided into jobs, which process an input data subset (e.g. a set of input files) and generate an output data subset (e.g. one output file). The output data of one workload are fed as input into the next workload in the sequence

(2), both the upstream and downstream workloads can run concurrently, minimizing delays associated with waiting for complete input data.

Task

A task is a PanDA object with distinct states mapping to a single workload. A task takes input data and produces output data. The purpose of the task is to process the input data entirely. Generally, the input and output data are collections of files, but there are also other formats, such as a group of sequence numbers, metadata, a collection of sub-file data, or notifications.

Job and Worker

A job is a PanDA object with distinct states that corresponds a sub-unit of a workload partitioned from a task. A single task consists of multiple jobs, and each job is executed by a worker, which is an abstraction of the execution point, such as a worker node and CPU cluster. A worker refers to a software program or a collection of software programs that assumes the responsibility of executing one or more jobs on computing resources. Each job is tailored based on the user's preference (if any) and/or constraints of the computing resource. For example, if jobs are flexible in terms of duration and disk usage, they are generated to have a short execution time and produce small output files when processed on resources with limited time slots and local disk space. The task input is logically split into multiple subsets, and each job gets a subset to produce output. The collection of job outputs is the task output.

Global Share

Global shares define the allocation of computing resources among various VO groups and/or user activities. The aggregation of available computing resources is dynamically partitioned into multiple global shares. Each task is mapped to a global share according to its VO group and activity type. Many PanDA system components internally work with global shares.

Priority

The priority of a task or job determines which task or job has precedence over other competing tasks or jobs in the same global share. Their priorities are relevant in each global share, i.e. high-priority tasks in a global share do not interfere with low-priority tasks in another global share. Generally, jobs inherit the priority of their task, but the first several jobs in each task have higher priorities to collect various metrics of the task as soon as possible. Task and job state transitions, as well as access to external systems like data transfers, are prioritized according to their priorities.

System Architecture

Figure 2 shows a schematic view of the PanDA system. There are five main components in the system:

- *JEDI* is a high-level engine to tailor jobs for optimal usages of diverse and geographically distributed resources dynamically.
- *PanDA server* is the central job pool implemented as a stateless web service to allow asynchronous communication from users, pilot, and Harvester over HTTPS.
- *Pilot* is a transient agent to execute a job on a worker node, periodically reporting various metrics to the PanDA server throughout its lifetime.
- *Harvester* generates and submits pilots using the appropriate communication protocol for each resource provider and communicates with the PanDA server on behalf of the pilot as necessary.
- *PanDA monitor* is a web-based monitoring of tasks and jobs processed by PanDA, providing a common interface for end users, the central operations team, and remote site administrators.

JEDI and the PanDA server share the central database to manage tasks and jobs. PanDA monitor reads from the central database to offer various views to users. Harvester has a more lightweight database, which can be either consolidated or distributed depending on the deployment model. Users interact with the PanDA system using client tools through the PanDA server. Details of PanDA components, database, authentication and authorization mechanism, and client tools are explained in the following sections.

JEDI

JEDI (Job Execution and Definition Interface) processes tasks and generates jobs for the PanDA server. The main functions are as follows:

- To receive and parse task specifications, which users submit through the PanDA server.
- To collect information about task input data.
- To decide the destination for each task output.
- To choose the computing resources based on the characteristics and requirements of each task. The brokerage algorithm is described in Sect. "Brokerage".
- To tailor jobs as described in Sect. "Job Sizing" and assign jobs to computing resources by taking global shares into account.

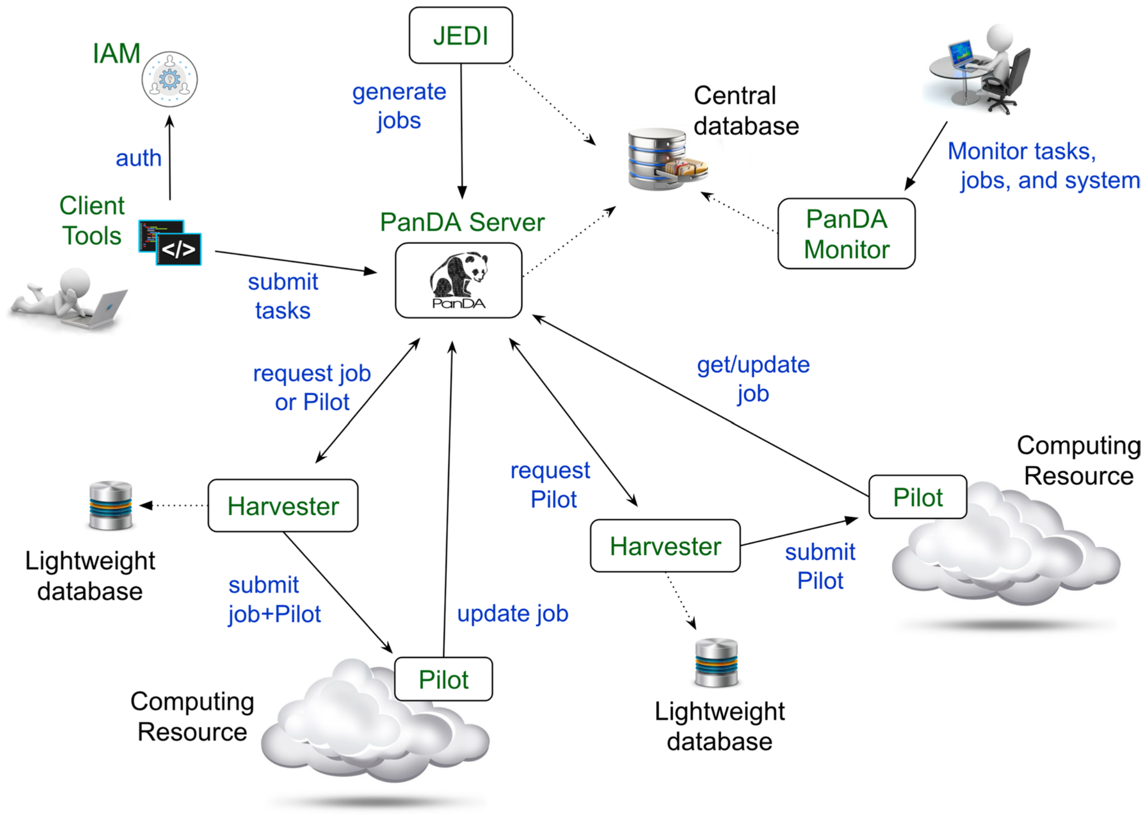


Fig. 2 A schematic view of the PanDA system with white boxes representing the main components. Dashed arrows show connections to databases, while solid arrows describe interactions between components or from users

- To optimize task parameters for each task based on results of prior jobs in the same task, as described in Sect. "Dynamic Optimization of Task Parameters".
- To reassign jobs if workload distribution becomes unbalanced among computing resources.
- To take actions on tasks according to various timeout configurations or user commands.
- To finalize tasks once their input data are fully processed.

JEDI comprises a master process, stateless agents running on multiple threads/processes, and a fine-grained exclusive lock mechanism. Agents run independently and do not directly communicate with each other. They take work entities such as tasks and jobs from the database, perform actions on them, and update the database. Each agent is designed around a plugin structure with the experiment-agnostic core and plugins. Plugins are software add-ons to enhance JEDI's capabilities to perform experiment-specific functions, such as interfacing with experiment-specific external services and performing actions required for the experiment-specific use-cases. It is possible to configure JEDI instances to load new plugins and add new features.

The exclusive lock mechanism allows operations to be distributed across threads, processes, and instances, so that JEDI horizontally scales with multiple instances. For example, while one agent process is working on a particular task, the task is locked, and other agent processes are prevented from updating the task. This is typically useful to avoid inconsistent modifications caused by concurrently running processes.

PanDA Server

The PanDA server is the central job pool. It consists of Web applications with a RESTful interface running on Apache HTTP servers [23] and time-based process schedulers. It takes care of jobs throughout their lifetime. The main functions are as follows:

- To receive jobs from JEDI and other job sources that directly generate jobs mainly for testing purposes.
- To guarantee input data availability in the storage resources associated with the computing resources where jobs are scheduled to run.
- To dispatch jobs to worker nodes once the input data are transferred or when the input data are already available.
- To monitor jobs while they are running on worker nodes.

- To dispatch metadata of sub-file data and keep track of the processing at the sub-file level, if the job is configured to perform the fine-grained processing [24].
- To post-process output data once jobs are done on worker nodes.
- To take actions on jobs according to various timeout configurations or user commands.
- To report job states to JEDI if those jobs were generated by JEDI.

The PanDA server scales horizontally by adding instances since Web servers are stateless and time-based processes are fine-grained.

Figure 3 below shows the architecture of a single PanDA server instance. PanDA Web applications are embedded in Web Server Gateway Interface (WSGI) [25] daemons running behind an Apache HTTP server. The parent Apache HTTP process spawns WSGI daemons via mod_wsgi [26] in addition to child HTTP processes. The number of WSGI daemons is static, while the number of child processes dynamically changes depending on the load to optimize resource usage of the instance. Child processes receive requests from actors such as users and pilots. The requests are passed to PanDA Web applications through an internal request queue and WSGI daemons. There are two types of requests:

- Synchronous requests: Actors are blocked for a while and receive responses when PanDA Web applications complete processing the requests.
- Asynchronous requests: Actors immediately receive a response and the requests are asynchronously processed. This is typically done when the requests invoke heavy procedures like access to external services. This mode prevents the HTTP server from being clogged.

The time-based process scheduler, the so-called PanDA daemon, is a daemon in charge of launching various scripts periodically. Its functionalities are very similar to the standard cron daemon, but it has the following advantages:

- No need to maintain an extra crontab configuration file.
- On each instance, the same script runs sequentially, i.e. only one process for each script, which is especially helpful when the script may run longer than the period configured. No new process will spawn until the existing one finishes, while the cron daemon blindly launches processes so that one has to fine-tune the frequency or let the script itself kill old processes to avoid duplicated execution.
- There is an exclusive control mechanism to prevent multiple instances from running the same script concurrently. It is also useful for sharing results between instances and avoiding repeating the same action multiple times. If this is enabled for a script, only one instance can run the script at a time, which is useful for long-running scripts that can run on any instance.
- Better system resource usages, e.g. limited total processes n_proc to run scripts, reduction of the overhead to launch processes, and sharing of database connections

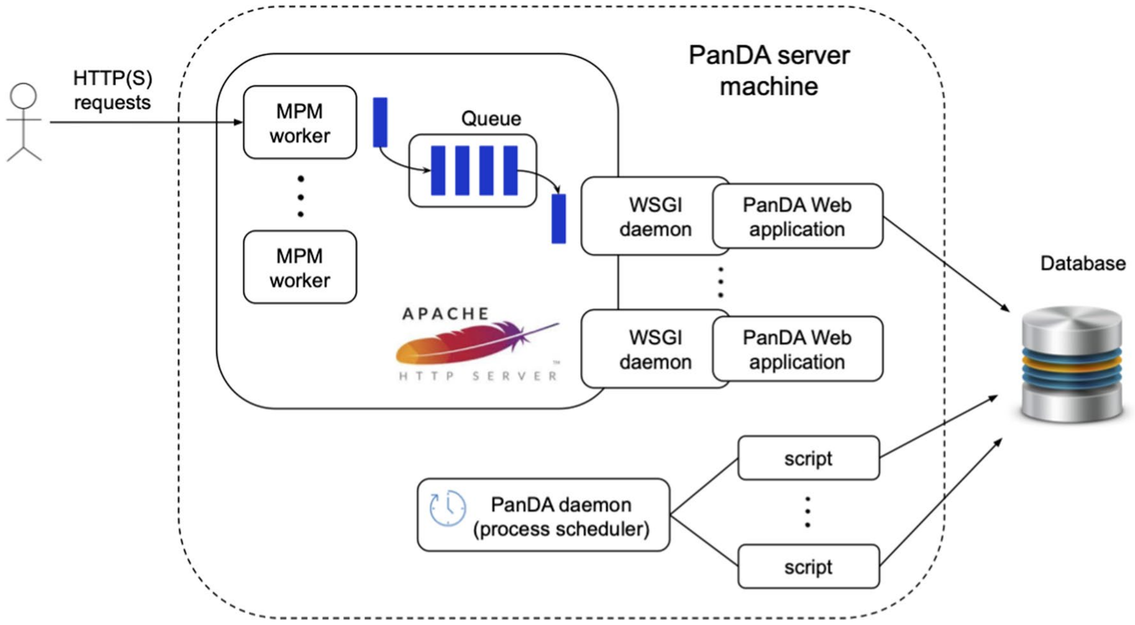


Fig. 3 The architecture of a single PanDA server instance

among scripts to avoid making a new database connection in every run.

Pilot

The pilot is a transient agent to execute and monitor grid jobs on a worker node. A grid job contains a payload that a user wants to execute. The payload has certain requirements, e.g. input and output files, that are staged by the pilot, and needs a working environment that is set up by the pilot. The payload may require running inside a container, which is also set up by the pilot, or in some cases by the wrapper script that launches the pilot. On the fine-grained processing, the pilot launches and feeds a payload with a metadata set of sub-file data downloaded from the PanDA server.

Pilot Highlights

The pilot runs on the worker nodes on local resources, grids, clouds, HPCs, and volunteer computers. It is downloaded and executed by wrapper scripts that are sent by Harvester 3.5 to the worker nodes via batch systems. The wrapper selects the pilot version after checking with the CRIC information system [27]. The pilot interacts with the PanDA server either directly, via the ARC Control Tower [28], or with the resource-facing Harvester service. Some of the pilot highlights are listed below:

- The pilot is responsible for running payloads created by users, while monitoring all steps and keeping the PanDA server updated.
  - Any necessary input files, and produced output files will be transferred from/to the relevant storage element.
  - Input files may be accessed directly from storage by the payload, in which case the pilot will not transfer the file.
  - The job may consist of a suite of pre-, co- and post-processes as well as the main payload itself.
  - The pilot can execute special utility processes, e.g. memory monitoring tools, running in parallel with the payload.
  - All processes can be executed in their own containers, either predetermined or set by the users.
- All communications between pilot and external services (e.g. PanDA, Rucio, and others) are done with HTTPS.
- File transfers are handled by dedicated copy tools.
  - Currently supported copy tools include Rucio (using the Rucio API), `xrdcp`, `gfal`, `gs`, `s3`, `mv/cp/ln`, `objectstore`, `lsm` (locally defined site mover).
  - For each file transfer (as well as for directly accessed files), the pilot can send a detailed report, such as access protocol, timestamps, and hostname, to the Rucio trace server.
- HPCs with no outbound network are supported by delegating communications to proxy services.
- Identification and reporting of 130+ unique errors, including detailed error diagnostics whenever available. It is possible to customize error handling and messages.
- Multiple options exist for debugging troublesome payloads.
  - Debug mode can be activated when the job or task is created, or after it has started.
  - An individual job running in debug mode can be followed on the PanDA monitor via the tail of the latest modified file uploaded on each server update (every 5 minutes in debug mode) or live via near real-time logging of the payload with the help of Google Cloud Logging [29], Fluentd [30] and Logstash [31].
  - The pilot can execute `ls` on a requested work directory and `ps` for a given process id, as well as report on disk usage, and make the output available on the PanDA monitor. It can also run the `gdb` debugger to generate core files and place them in the job log for later retrieval.
- The pilot may run in staging mode, in which it only performs file staging.
- A single pilot can run multiple jobs sequentially until it runs out of time.
  - It also has support for prompt processing, where the pilot immediately starts processing a payload if it receives job information from ActiveMQ [32]. When the current job has finished, the pilot resumes the listening mode, waiting for another payload.
- The pilot is user ("experiment") independent with the user code stored in plugins.
  - The plugins encapsulate all the code that is relevant for each pilot user.
  - Plugins currently exist for the ATLAS, sPHENIX and Vera C. Rubin experiments.
  - A joining VO can add their code specifics (e.g. how the payload and other tools should be executed, special checks and algorithms) starting from a generic plugin
  - Fully implemented pilot plugins range from 1,700 to over 8,000 lines of code (the generic plugin has 1,200 lines of code).
- The current pilot version is Python 3 compliant.

– A pull request to the pilot GitHub repository triggers unit tests and runs Flake8 verification for Python 3.7, 3.8 and 3.9 versions.

PanDA Monitor

The PanDA monitor (BigPanDAmon) is a web application that has been developed to serve the monitoring needs of all PanDA users according to their roles and behavioral characteristics. It provides a set of aggregated reports, dashboards and graphical representations of the PanDA system objects such as jobs, tasks, sites, and users. The interface allows users to drill down into the reason for a job failure or observe the broad picture such as tracking the computing site performance or the progress of a whole production campaign.

PanDA Monitor Architecture

The BigPanDAmon system is a Django-based [33] web application consisting of a set of core views and separate modules which can be plugged in. It collects data from different sources in particular the PanDA database, the CRIC information system, Rucio providing payload logs of jobs, the Elasticsearch [34] cluster where PanDA/JEDI application logs are exported, and the MONIT Grafana [35] instance where accounting data are available for the ATLAS instance.

Due to the Django object-relational mapping layer, the system supports Oracle and PostgreSQL databases. The BigPanDAmon uses the OAuth2 protocol for the authorization of users and supports the following third-party single sign-on services: CERN, GitHub and Google. To improve response time, the system has advanced caching mechanisms and scheduled preprocessing tasks, which are shown in the data-flow diagram (Fig. 4). The database-level caching is used to store the prepared-to-render data and not rendered pages themselves. This allows one to treat the general and user-specific data separately. Also, a common CephFS storage is mounted on all application nodes for data like job logs or rendered plots from Grafana.

The front-end of the application is implemented using the Django built-in server-side rendering of static content and AngularJS for interactive features. For visualizations, several libraries are used, in particular D3.js [36] as the default one, and Chart.js [37], based on HTML5 features for plots with a large number of objects, for example, a profile of a task with a few thousand jobs.

PanDA Monitor Views

In total, there are 70 different views, all of which can return information in the form of an interactive web page or as JSON output. In addition, there are 54 APIs for delivering complementary information on demand. The core views provide essential information about the principal objects of the PanDA system listed in Sect. "Concepts". Therefore, they cover various monitoring needs of users according to their roles.

Core Views

This group of views consists of three types: summary, info, and regional summary. The summary view aggregates attributes of objects by counting the number of occurrences for each property value and represents it in a table where the number of occurrences is a link to the same view filtered by it (Fig. 5). This way, a user can obtain a detailed view

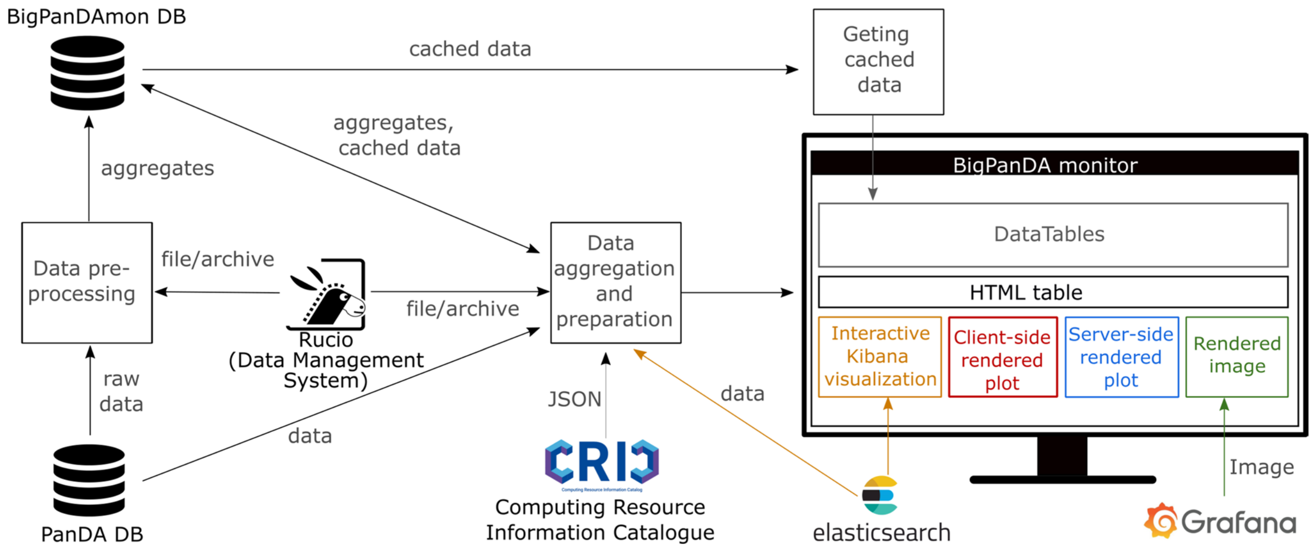
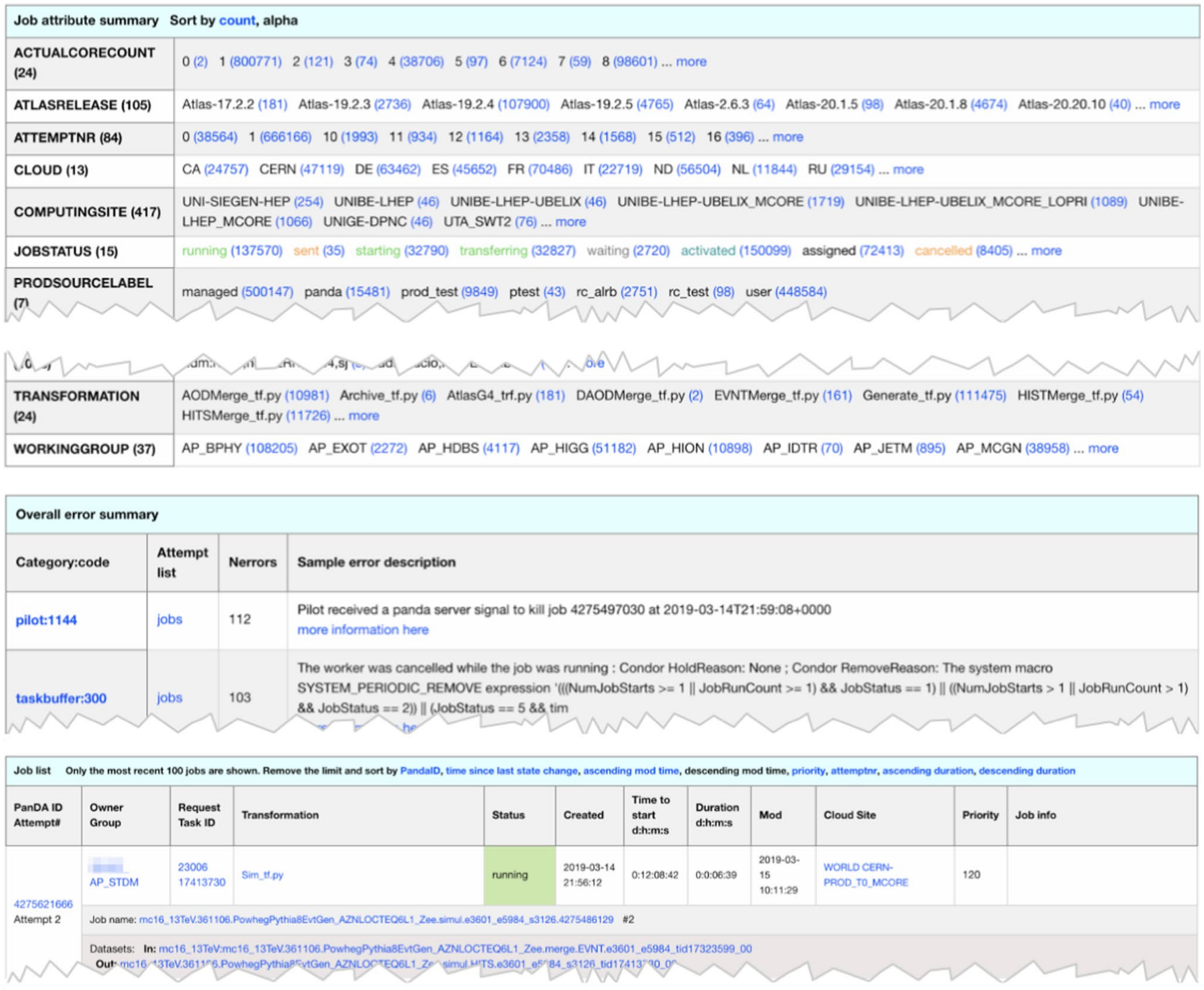


Fig. 4 BigPanDAmon data flow diagram



Job attribute summary Sort by count, alpha	
ACTUALCORECOUNT (24)	0 (2) 1 (800771) 2 (121) 3 (74) 4 (38706) 5 (97) 6 (7124) 7 (59) 8 (98601) ... more
ATLASRELEASE (105)	Atlas-17.2.2 (181) Atlas-19.2.3 (2736) Atlas-19.2.4 (107900) Atlas-19.2.5 (4765) Atlas-2.6.3 (64) Atlas-20.1.5 (98) Atlas-20.1.8 (4674) Atlas-20.20.10 (40) ... more
ATTEMPTNR (84)	0 (38564) 1 (666166) 10 (1993) 11 (934) 12 (1164) 13 (2358) 14 (1568) 15 (512) 16 (396) ... more
CLOUD (13)	CA (24757) CERN (47119) DE (63462) ES (45652) FR (70486) IT (22719) ND (56504) NL (11844) RU (29154) ... more
COMPUTINGSITE (417)	UNI-SIEGEN-HEP (254) UNIBE-LHEP (46) UNIBE-LHEP-UBELIX (46) UNIBE-LHEP-UBELIX_MCORE (1719) UNIBE-LHEP-UBELIX_MCORE_LOPRI (1089) UNIBE-LHEP_MCORE (1066) UNIGE-DPNC (46) UTA_SWT2 (76) ... more
JOBSTATUS (15)	running (137570) sent (35) starting (32790) transferring (32827) waiting (2720) activated (150099) assigned (72413) cancelled (8405) ... more
PRODSOURCELABEL (7)	managed (500147) panda (15481) prod_test (9849) ptest (43) rc_alrb (2751) rc_test (98) user (448584)
TRANSFORMATION (24)	AODMerge_tf.py (10981) Archive_tf.py (6) AtlasG4_trf.py (181) DAODMerge_tf.py (2) EVNTMerge_tf.py (161) Generate_tf.py (111475) HISTMerge_tf.py (54) HITSMerge_tf.py (11726) ... more
WORKINGGROUP (37)	AP_BPHY (108205) AP_EXOT (2272) AP_HDBS (4117) AP_HIGG (51182) AP_HION (10898) AP_IDTR (70) AP_JETM (895) AP_MCGN (38958) ... more

Overall error summary			
Category:code	Attempt list	Nerrors	Sample error description
pilot:1144	jobs	112	Pilot received a panda server signal to kill job 4275497030 at 2019-03-14T21:59:08+0000 more information here
taskbuffer:300	jobs	103	The worker was cancelled while the job was running : Condor HoldReason: None ; Condor RemoveReason: The system macro SYSTEM_PERIODIC_REMOVE expression '(((NumJobStarts >= 1 || JobRunCount >= 1) && JobStatus == 1) || ((NumJobStarts > 1 || JobRunCount > 1) && JobStatus == 2)) || (JobStatus == 5 && tim

Job list Only the most recent 100 jobs are shown. Remove the limit and sort by PandaID, time since last state change, ascending mod time, descending mod time, priority, attemptnr, ascending duration, descending duration

PanDA ID Attempt#	Owner Group	Request Task ID	Transformation	Status	Created	Time to start d:h:m:s	Duration d:h:m:s	Mod	Cloud Site	Priority	Job info
4275621666 Attempt 2	[redacted] AP_STDM	23006 17413730	Sim_tf.py	running	2019-03-14 21:56:12	0:12:08:42	0:0:06:39	2019-03-15 10:11:29	WORLD CERN-PROD_T0_MCORE	120	

Job name: mc16_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.simul.e3601_e5984_s3126.4275486129 #2

Datasets: In: mc16_13TeV:mc16_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.merge.EVNT.e3601_e5984_tid17323599_00
Out: mc16_[illegible]3TeV.3611[illegible]6.PowhegPythia8[illegible]vtGen_AZNLO[illegible]TEQ6L1_Z[illegible] simul.[illegible]TS.e3601_e5[illegible]84_s3126_tid17413[illegible]0_0

Fig. 5 Example of jobs summary view

of certain objects of interest or observe the broader picture. This type of view exists for jobs, tasks, PanDA queues, and computing resources.

The info view collects information related to a particular object. For example, in a task info view, there is a list of all task parameters, a summary of its job states, a list of associated datasets, processing progress in the number of files, and staging progress from the tape storage if it is applicable for the task. There are info views for jobs, tasks, datasets, files, PanDA queues, computing resources, and users.

The regional summary view represents the number of job states for each PanDA queue, computing resource, and a group of computing resources. This is an essential view for system administrators of computing sites and operators.

Modules

It is possible to customize a BigPanDAmon instance with various modules. Each module provides optimal views to diagnose issues on a specific workflow, system components/objects (e.g. Harvester and global shares), or campaigns, with a helpful interface to track them down. The experiment-specific modules compose the information of PanDA objects and other systems used in experiments. For example, the ATLAS instance installs the ATLAS release tester [38] module to monitor nightly tests running as PanDA jobs that verify newly added merge requests on the main repository of the ATLAS offline software framework, Athena [39].

Harvester

Harvester is a resource-facing service between the PanDA server and a collection of pilots for resource provisioning and workload shaping. It is a lightweight stateless service running on virtual machines for the virtual organization or edge nodes of HPC centers, to provide a uniform view for various resources, with a modular design to support different resource types and workflows.

Harvester Workload Mapping

There are various types of computing resources, including but not limited to:

- Grid resources are dedicated to the experiment, hence available 24/7 with stable throughput. Grid sites are rather homogeneous, i.e. they have similar architecture, policies, and behavior.
- HPC and opportunistic resources usually provide a huge amount of slots but at irregular intervals, which yield bursty and intermittent throughput. Each HPC center can have different edge services and operational policies.
- Cloud resources, including industrial clouds in research infrastructures and various commercial clouds, have gained importance in scientific research. Different clouds have different policies and require different APIs to access.

Thus, the Harvester needs to face diverse resources with common machinery for pilot provisioning and to have the capability of dynamically optimizing resource allocation among PanDA queues (for example, single-core, multi-core, and high-memory). To achieve this, the Server-Harvester-Pilot model has been adopted. The concept of job-worker mapping is introduced in Harvester, where Job and Worker are described in Sect. "Job and Worker". A worker is a representative of a collection of pilots that run on a designated computing resource such as a worker node and CPU cluster. A worker corresponds to a pilot in most Grid usecases.

Harvester supports various types of Worker provisioning schemes with push/pull modes and different job-worker mapping. Here are the types of Worker provisioning schemes.

Pull

Figure 6 shows the worker provisioning scheme in simple pull mode. Harvester submits workers without jobs. Once acquiring a computing resource, each worker pulls a job directly from the PanDA server. Harvester decides the quantity of workers to submit for a PanDA queue according to Harvester's configuration.

Pull Unified-Pilot-Streaming (Pull UPS)

In simple pull mode, the worker submission is typically made independently from the job generation. The disadvantage of this mode is that the blind submission of workers is inefficient and can waste worker node resources when there are no jobs available for a worker. The Unified Pilot Streaming (UPS) mode intelligently submits workers to solve the above issue. Figure 7 shows the worker provisioning scheme in pull UPS mode. It works as follows:

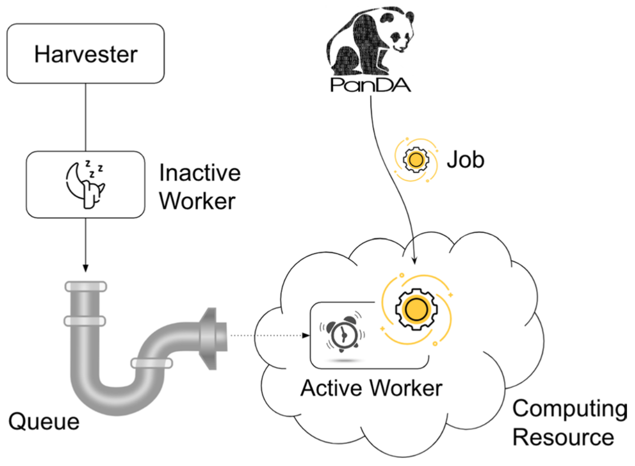


Fig. 6 Worker provisioning with simple pull mode

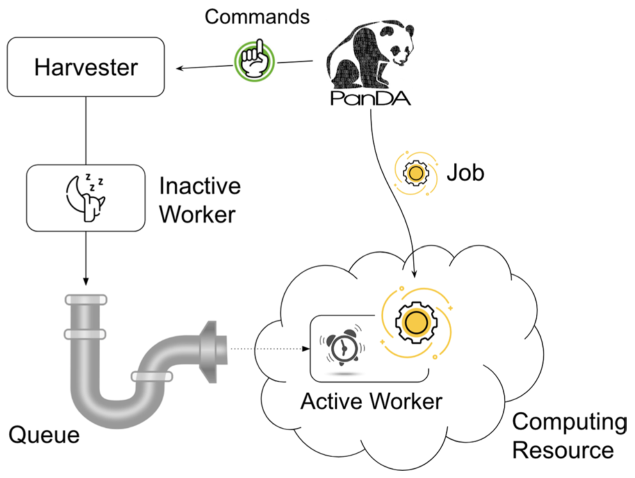


Fig. 7 Worker provisioning with pull UPS mode

1. PanDA calculates how many workers of each type are needed based on the queued and running jobs at a computing resource, and the overall Global Share preferences. The types are generic (i.e. not job-specific) categories: single-core, multi-core, and high-memory.
2. PanDA sends commands based on this information to Harvester.
3. Harvester submits the workers with the proper requirements to the site.

4. When a worker starts, it will ask for a job matching the requirements.
5. PanDA dispatches again the job according to the Global Shares.

PanDA decides how many workers of each type are submitted, and the decision is not randomly left to Harvester. This mode is used for most Grid resources.

1-to-1 Push

Figure 8 shows the worker provisioning scheme with 1-to-1 push mode. Jobs are prefetched and bound to workers before Harvester submits them. Each worker has one job. This workflow is used for some Grid and HPC resources.

1-to-1 Push and Job Late-Binding

Jobs are prefetched while workers are submitted to scheduling systems asynchronously. Harvester assigns one job to each worker once the worker gets a computing resource. This workflow is used for HPCs with long-waiting batch queues.

1-to-Many Push (Multi-worker)

Jobs are prefetched, and multiple workers are submitted for each single job. Harvester collects and combines information from all associated workers for each job. Typically this mode is used for HPCs to fill unused slots using special jobs. The job must have the capability of allowing multiple workers to collaboratively process a part of the workload in the job while being aware of what others are doing through an external mechanism, such as a parameter serve and an exclusive file lock.

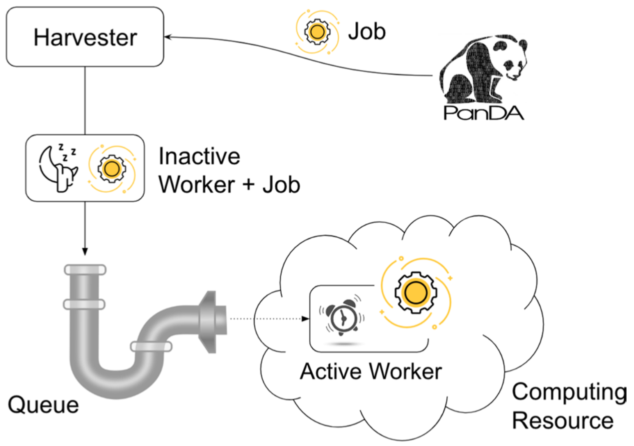


Fig. 8 Worker provisioning with 1-to-1 push mode

Harvester Architecture

Figure 9 shows a schematic view of the Harvester architecture. Harvester comprises a database, agents, and plugins.

Harvester Database

Harvester stores the states of jobs and workers, and other cached information from external sources into its database. The engine of the Harvester database can be MySQL, MariaDB, and SQLite.

Harvester Agents

Harvester has several stateless and multi-threaded agents, one for each action (e.g. worker submission, job fetching, pre-staging, and credential renewal). Harvester takes action based on the state transition of jobs and workers. The information which the agents require is stored and updated in the database. There is no direct messaging between agents. The functions of agents are explained below.

- *Submitter* periodically looks up queues to fill, creates entries of new workers in the database, and submits workers for the queues to the computing resources via submitter plugins.
- *Monitor* checks and updates the state of workers by communicating with the resources via monitor plugins.
- *Sweeper* kills the workers at the resource via sweeper plugins, and cleans up entries of terminated workers in the database.
- *Credential Manager* calls credential manager plugins to check and renew credentials, e.g. VOMS proxy certificate and JSON web tokens (JWTs) [40], to be used for authentication and authorization with computing resources.
- *Propagator* periodically reports information on jobs and workers in Harvester to the PanDA server.
- *Job Fetcher* prefetches the jobs from the PanDA server and stored them in the Harvester database. The jobs are later bound to workers and submitted.
- *Command Manager* fetches commands from the PanDA server and stores them in the Harvester database. Other agents take action according to the commands.
- *Cacher* periodically downloads information from external data sources, such as PanDA queue configurations from the CRIC information system, and caches them into the database.

Harvester Plugins

Harvester plugins are developed to communicate with diverse resources. Functions in the plugin are called by their corresponding Harvester agent. For example, the Submitter

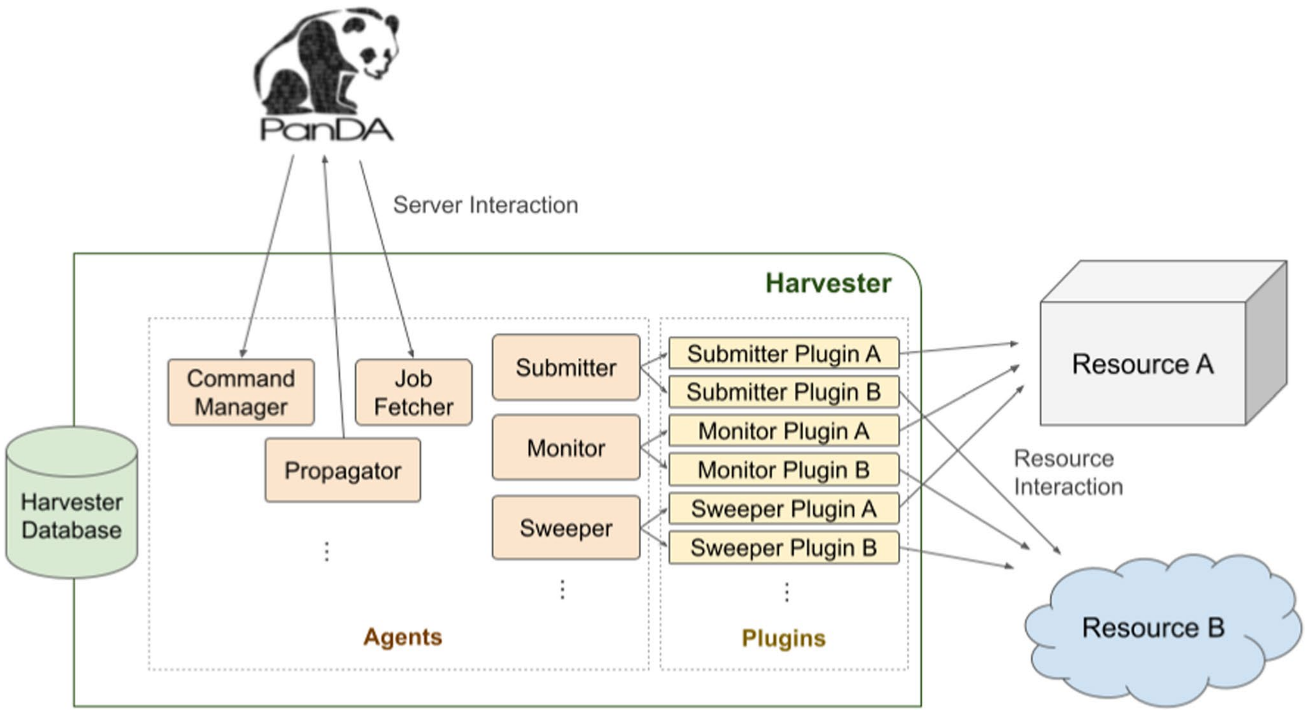


Fig. 9 Harvester architecture diagram

agent calls HTCondor [41] submitter plugin to submit pilots to a HTCondor batch system, HTCondor CE [42], or ARC CE [43]. The plugins to be used for each PanDA queue are specified in the Harvester configuration. Experts of each resource develop their dedicated plugins and submit the code to the Harvester GitHub repository. Currently, there are plugins available for HTCondor, ARC Control Tower, Kubernetes [44], SLURM [45], PBS [46], and Lancium [47].

PanDA Database

Operations are check-pointed and persisted on a transactional database shared by all the PanDA and JEDI servers. The database contains the state for all the jobs and tasks, related metadata for files and datasets, as well as information for other entities like users and sites. PanDA supports Oracle and PostgreSQL as database backends. The database interface module contains SQL code written and optimized for Oracle, since it has historically been the main database technology. When using a PostgreSQL database, queries are translated to the PostgreSQL dialect through an in-house layer.

The database usage has been highly optimized to keep up with the transaction rate required by big science experiments. In addition to the usage of targeted indexes, Oracle partitioning is also used to optimize data access. Depending on the size and relevance of the tables, older entries can be deleted through a sliding window procedure. However, for many tables the information is kept over extended periods of time in order to be able to browse the data in the monitoring. This leads to very large tables with many million rows.

In the case of ATLAS, all task and job definitions are kept since 2006. The current policy is to keep the current and the last two years on the main database. Completed jobs are moved to an archive table, in order to control the size of the active jobs table. Once per year, the archived jobs older than 2 years are manually moved to a separate archival database with less performant storage and CPUs.

PanDA frequently requires to calculate aggregations of jobs (sums, averages) to gather share, site or user statistics. It is very costly to run independent aggregations of the tables within seconds. For this reason, background procedures precompute and store aggregations on auxiliary tables every couple of minutes. This reduces the load on the database significantly.

In addition, central tables are replicated to Elasticsearch to off-load analytics and accounting procedures to external infrastructure.

Authentication and Authorization

PanDA has an Identity and Access Management (IAM) scheme fully compliant with OIDC/OAuth2.0, capable of identity federation among scientific and academic identity providers.

PanDA IAM consists of Indigo IAM, CILogon [48], and identity providers. Indigo IAM is an account and group membership management service to define VOs and groups, to add/remove users to/from VOs and groups, and issue ID tokens once users are authenticated. CILogon is a federated ID broker to delegate authentication to ID providers such as CERN [49], BNL SDCC [50], and Google Identity Platform [51].

Figure 10 shows the procedure of user authentication and authorization, where the device code flow is used to allow users to run command-line tools. First, the user invokes a command-line tool that checks if a valid ID token is locally available. If not, the command-line tool sends an authentication request to Indigo IAM on behalf of the user and retrieves a verification URL. Then, the user opens a web browser to go to the verification URL and is eventually redirected to their ID provider through CILogon. Once the user successfully logs on, a couple of tokens are exchanged between CILogon and Indigo IAM, and an ID token is issued. The command-line tool gets the ID token and places it in the HTTP request header when accessing the PanDA server. The PanDA server decodes the token and authorizes the user based on OIDC claims such as name, username, and groups.

Legacy X.509-based user authentication is also supported and is performed via the GridSite library [52] that is loaded inside the Apache HTTP processes of the PanDA server.

Resource Authentication and Authorization

The pilot is authenticated and authorized via JWTs or X.509 certificates. If a computing resource provider accepts JWTs, Harvester fetches access tokens from the token issuer deployed by each experiment, and submits the worker with the access tokens to the gateway services of the computing resources. Access tokens have a very short lifetime and specify the gateway services in the audience claim, to avoid being abused for access to other resources. On the other hand, if a computing resource provider works only with X.509 certificates, Harvester obtains proxy certificates with X.509 certificates from VOMS servers, which add the authorization attributes, such as VO groups and roles, as extensions, submitting the pilot with proxy certificates.

Client Modules and Tools

There are two Python modules and client tools for users to send commands to the PanDA server through its RESTful interface using standard HTTP methods. One Python module

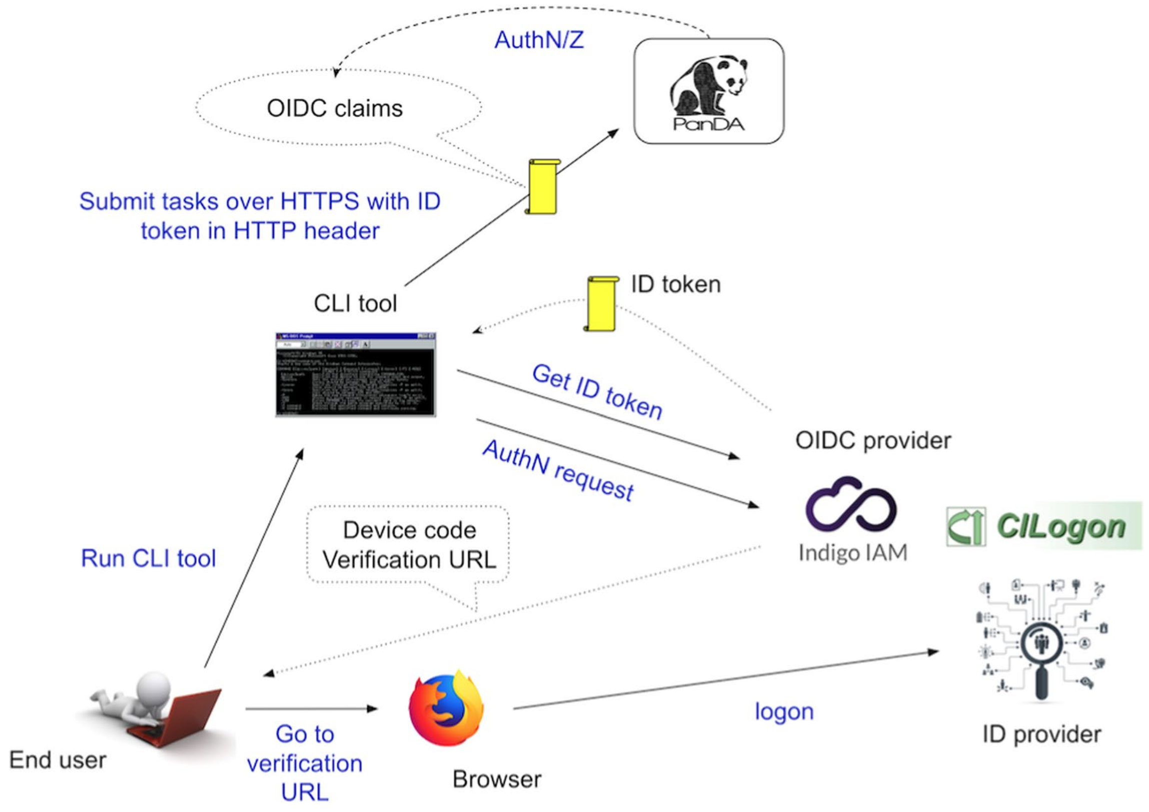


Fig. 10 The procedure of user authentication and authorization

provides a set of limited client APIs for non-administrative users to submit and manage tasks and jobs, go through the authentication procedure, and retrieve system information. The other Python module allows the execution of administrative commands, such as eliminating problematic users, or changing system configuration. Client tools are a variety of command-line applications built on top of client APIs for the execution of specific workloads, bookkeeping, and workflow management. Ordinary users typically use client tools to run their analysis on PanDA. The non-administrative Python module and client tools are part of a Python package, while the other module is included in a separate Python package.

Functionality Details

Brokerage

Brokerage is one of the most crucial functions in the system to distribute workload among computing resources. It has the following goals:

- To distribute jobs among all available computing resources.
- To minimize the waiting time for each job to produce output data.
- To execute jobs in such a way that the jobs respect their priorities and resource allocations.
- To choose computing resources for each job based on the characteristics of the job and constraints of the computing resources.

It is not straightforward to satisfy these goals for all kinds of jobs since some of them are logically contradictory. The brokerage has a plugin structure so that each VO can provide an algorithm according to its own needs and use cases, based on task and job priorities, global shares and allocations, data localities, costs for data transfers, data placement policies, matching between resource specifications and data processing requirements, constraints and downtime of computing resources, transfer backlog over the network, and so on. For example, when certain resource providers enforce stringent policies that restrict the execution of arbitrary jobs due to security concerns, the brokerage assigns only centrally managed jobs by the VO experts to the computing resources that are generated from managed workloads.

Dynamic Optimization of Task Parameters

JEDI automatically optimizes task parameters for compute/storage resource requirements and strategies to partition workload while running those tasks. In the early stage of task execution, JEDI generates several jobs for each task using only a small portion of input data, collects various metrics such as data processing rate and memory footprints, and adjusts the following task parameters. Those first jobs are called scout jobs. The automatic optimization is triggered twice for each task; when half of the scout jobs are finished, and when the first 100 jobs are finished while generating the remaining jobs after completion of the scout jobs.

Job Sizing

JEDI generates jobs based on the requirements of computing resources in terms of CPU core, disk space, RAM, and walltime, using the task parameters shown in Table 1.

If one of the first three parameters in Table 1 (*n**PerJob) is specified, jobs are generated accordingly to respect the parameter. If some computing resources cannot accept those jobs due to resource limitations, such as small disk spaces and short walltime, the brokerage avoids those resources.

If they are not specified, jobs are generated to meet the requirements of each computing resource. The number of

Table 1 Task parameters for JEDI to size jobs

Name	Description
nFilesPerJob	The number of input files per job
nGBPerJob	The total size of input/output files and working directory
nEventsPerJob	The number of events per job
cpuTime	CPU time per event
cpuEfficiency	CPU efficiency (0.9 by default)
baseTime	The part of the job execution time not scaling with CPU power, such as periods for payload initialization and finalization
outDiskCount	The expect output size per event
workDiskCount	The working directory size
coreCount	The number of CPU cores
ramCount	Resident set size per CPU core

events nEvents in each job must satisfy the following two formulae:

$$\begin{aligned} S \geq\ & \text{inputDiskCount} \\ & + \max(0.5\text{GB}, \text{outDiskCount} \times \text{nEvents}) \\ & + \text{workDiskCount} \end{aligned}$$

$$W \geq \frac{\text{cpuTime} \times \text{nEvents}}{C \times P \times \text{cpuEfficiency}} + \text{baseTime}$$

where S, W, C, and P are the disk space size, the walltime limit, the number of CPU cores, and the averaged benchmark score measuring CPU performance [53] at the computing resource, respectively. inputDiskCount is the total size of the job input files, a discrete function of nEvents. Note that inputDiskCount is zero if the computing resource is configured to read input files directly from the local storage resource.

Automatic Data Distribution and Aggregation

Data stage-in and stage-out can cause significant bottlenecks in data-intensive processing. Dynamic and asynchronous data placement is crucial for efficient usage of computing and storage resources. PanDA relies on Rucio for distributed data management. PanDA interacts with Rucio using the client API for name and location lookup, requesting transfers, retrieving attributes and metadata, organizing and deleting data, etc. Rucio sends notifications via STOMP-compatible message-queuing services, e.g. ActiveMQ, when asynchronous operations are fulfilled, and PanDA consumes the notifications to take subsequent actions.

Temporary input data replicas are dynamically created for jobs when all the following conditions are met:

- I/O intensities to process those jobs are not significant, since I/O-intensive jobs usually require more data transfers than CPU-intensive jobs to fill computing resources over the same period. Replicating data for the latter is more optimal in terms of network usage.
- Computing resources, associated with the storage resource where the original data are available, are busy.
- Other computing resources are idle.
- The input data are unavailable at the storage resources associated with the idle resources.

PanDA makes transfer requests to Rucio, Rucio notifies PanDA once the transfers are done, and then PanDA dispatches the jobs to the idle computing resources, so that they do not have to wait on the computing resources while input data are being staged. Jobs release the CPUs as soon as they upload the output data to the storage resource associated with the computing resource where the jobs are running. Then, PanDA requests Rucio to transfer the output data to the final destination.

In most cases, computing and storage resources are located within the same provider, meaning that the storage resource is locally connected to the computing resource. This setup allows for efficient data transfers between the two components using a local area network. However, it is possible to associate remote storage resources with computing resources. In such cases, stage-in and stage-out operations between the computing and storage resources are carried out over a wide area network. The entire data are transferred to computing resources, or alternatively, data can be streamed over the network based on job requirements.

Task and Job Retry Strategies

It is possible to retry tasks if a part of the input data were not successfully processed or new data were added to the input data. The task state changes from finished or done back to running, and output data are appended to the same output data collection. Tasks cannot be retried if they end up with a fatal state, such as broken and failed, since they are typically configured wrongly or run problematic jobs, and not worth retrying.

On the other hand, the job state is irreversible, i.e. jobs do not change their states once they go to a final state. JEDI generates new jobs to re-process the input data portion, which was not successfully processed by previous jobs. The configuration of retried jobs can be optimized based on experiences from previous jobs (e.g. increased memory requirements). It is also possible to configure rules to avoid the job retrial for fatal error codes or messages.

The Job Retry Module greatly simplifies operations by taking actions based on error codes and messages. Several actions are currently available, and new actions can be implemented and registered in the database.

Customizing PanDA

Each experiment can configure PanDA components to utilize only the necessary features according to its requirements and available resources. Most PanDA components have structures composed of the experiment-agnostic core and plugins. Plugins are software add-ons to enable particular functionalities dynamically loaded when PanDA components are launched. Plugins inherit base API classes that define standard abstract functions, to implement experiment- and resource-specific actions. There are default plugins implementing generalized actions. It is also possible to create and install new plugins to enhance existing capabilities. Each PanDA component hosts a small collection of plugins, with each individual plugin consisting of several hundred to thousands of lines of code. For example, if an experiment requires the execution of large-scale parallel multi-node jobs, particularly those that require specialized optimizations, tailored schemes could be implemented in new plugins. The PanDA and Harvester documentation [56, 56] describe how to customize PanDA components in detail.

Although experiments have different requirements, they can use the default plugins in many cases. For example, both the Vera C. Rubin Observatory and the sPHENIX experiment have deployed their own PanDA services with the default plugins at SLAC [56] and BNL SDCC, respectively. One of the main differences between their deployment is that Harvester for Vera C. Rubin installs Kubernetes and HTCondor plugins to utilize Google cloud resources in addition to the facility resources available through ARC CEs, while Harvester for sPHENIX installs only HTCondor plugins to utilize BNL resources on a HTCondor batch system.

While tight integration with Rucio is a PanDA's advantage, it is configurable to operate without Rucio. Such a configuration is particularly valuable for experiments that do not necessitate advanced data management capabilities.

Infrastructure and Installation

Infrastructure Sizing

The largest PanDA deployment to date serves the ATLAS experiment and is handled centrally at the CERN data center, including the databases and servers. It is important to understand that most of the ATLAS activities (except Event Generation and some Analysis) run multi-core jobs, for which the ATLAS Software is capable of managing multiple processes across typically eight cores. The fraction of single- and multi-core jobs changes as the different activities and campaigns are scheduled. So far in 2023, in average 85% of the resources have been managed in multi-core mode. While the main purpose of running multi-core jobs is to optimize the memory consumption on the worker nodes, another positive side-effect is that it reduces the load on the PanDA infrastructure considerably. In practice, ATLAS runs a few hundred thousand jobs concurrently and around one million jobs per day. The JEDI and PanDA server clusters consist of 9 virtual machines each, with individual specifications of the virtual machine comprising 8 cores and 16 GB of RAM. The ATLAS PanDA server cluster needs to handle typical request rates of 200 Hz, coming predominantly from the pilots that are retrieving and updating jobs. On the first half of November 2023, the mean time to handle requests was 0.04 s and the 90th percentile was 0.07 s. JEDI, on the other hand, needs to handle sufficient job generation to keep the Grid fully utilized. The other critical component is the PanDA database, which acts centrally to all the servers. ATLAS' central applications like PanDA and Rucio share an Oracle cluster, where each application runs on a separate node with 16 cores and 768 GB of RAM. The large memory is important so that the active tables are kept in memory and to avoid disk I/O. The database server has enough headroom to handle spikes and future growth.

Service Deployment with Kubernetes and Helm Charts

All PanDA components are containerized and can be deployed using Helm [57] charts on Kubernetes clusters. A single set of container images and Helm charts works both for vanilla Kubernetes and OKD [58]. The difference between these two services is that OKD forbids running containers as root and the usage of lower privileged ports. Every time a new version of a PanDA component is released on GitHub [59], a new container image is published automatically in the GitHub registry. While all the container images and the Helm charts are publicly available, the Helm secrets are used to deploy sensitive information securely, such as usernames and passwords for the databases, or client ID for IAM. The secrets can be stored in a private repository or the same repository but encrypted. They are typically deployed only once and need to be updated only if there is a new password or update.

When deploying PanDA on Kubernetes, it is possible to deploy only a specific PanDA component based on the requirements of each experiment using experiment-specific description files and secrets. It is also possible to deploy the PanDA system without the CRIC information system by providing a couple of JSON files that define PanDA queues, sites, storages, etc.

For the PanDA deployment at SLAC for the Vera C. Rubin Observatory, we have the following Kubernetes configuration with each pod having 4 CPU Cores and 16GB of RAM: 2 pods for PanDA server, 2 pods for PanDA JEDI, 2 pods for iDDS, and 3 pods for Harvester.

The Kubernetes-based deployment significantly simplifies the deployment of the PanDA components, and introduces new possibilities, such as the automatic scaling of the PanDA components based on the load and the continuous integration and testing framework based on Kubernetes.

Database Schema Installation, Upgrade, and Versioning

The PanDA database schema is available inside the `schema` folder in the `panda-database` GitHub module. For a new database installation, a user can execute the SQL statements found in the `schema` folder for either Oracle or PostgreSQL. When deploying PanDA on a new database, the schema installation script creates a new versioning table holding the schema version, e.g. 0.0.42.

Whenever there is a new version of the PanDA database schema, the version number increases to reflect the change. Whenever there is a schema change that increases the version number, a diff file is provided within the `upgrade` folder for all the schema changes between the previous and current versions. At the end of each diff file, there is an entry to update the version number in the versioning table, e.g. from 0.0.41 to 0.0.42.

When the PanDA server runs, it checks if the version in the versioning table is the minimum required for the PanDA server to work and fully function. If the database schema version is lower than the one required, the PanDA server will exit with a warning message.

Project Management

PanDA is an open-source project developed under the Apache V2 license [61]. PanDA benefits from the contributions of key developers from multiple experiments in Europe, the United States, and Asia, who regularly report their progress in the PanDA core team meetings and their experiment technical meetings. Jira [61] keeps track of roadmaps and milestones in the project, and GitHub hosts the PanDA codebase for version control, allowing coherent contributions from multiple developers. Each PanDA component has its own GitHub repository.

The development workflow is based on branches or forks in GitHub repositories. Developers make changes in their own branches or forks without affecting the main branches. Changes must be incremental to guarantee that they do not break existing production PanDA instances. It is the responsibility of each developer to check the changes on integration PanDA instances with the complete flow of tasks and jobs. A test suite runs on the integration instances with the Oracle database. In many cases, end-to-end tests with actual tasks and jobs are required since it is hard to detect problems through unit tests due to the complexity of the PanDA system and use-cases.

Once developers are comfortable with the changes, they submit pull requests to ask the repository managers for review. The whole review process is done through GitHub, which retains the history of communication in the pull request to help future contributors understand the changes. Once a pull request is approved, the repository managers merge it and tag a new release version in the repository, depending on the criticality of the changes. There is no time constraint between release versions. It is possible to tag many release versions per day for a single repository if necessary. Tagging of new versions automatically triggers GitHub actions to publish new packages in PyPI [62] and register new container images in the GitHub packages registry.

New versions of packages are usually deployed on production instances as soon as they are published. It is recommended that production instances deploy the latest versions of packages, although each experiment can follow its own deployment policy, e.g. they could upgrade PanDA instances only during idle periods.

Results and Experience

Figure 11 illustrates PanDA's scalability and flexibility, which shows the evolution in the number of CPU cores over the last decade for ATLAS. During this period, the managed resources became more diverse. In the early years, PanDA managed exclusively WLCG resources,

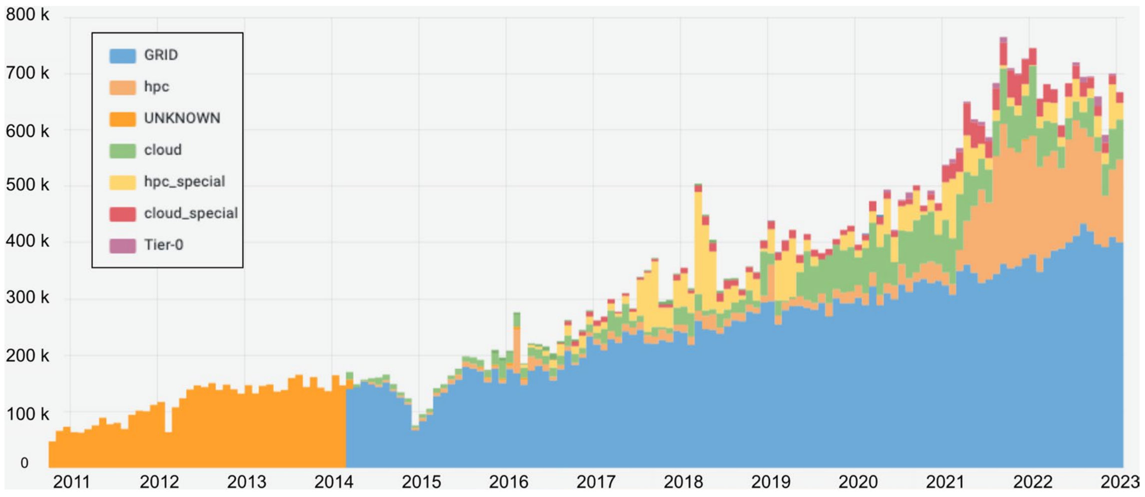


Fig. 11 Monthly average CPU cores managed by PanDA for ATLAS by resource types between 2011 and 2023. The accounting did not track the resource type until 2014, since the usage of non-Grid resources was negligible (labeled "UNKNOWN")

while in the last years, the amount of HPC and Cloud resources has been increasing steadily. Each type of resource can have a different interface and behavior, sometimes requiring customized worker submission through Harvester. Several major HPCs, such as Titan [63], Theta [64], Cori [65], Perlmutter [66], MareNostrum 4 [67], and Vega [68], have been successfully integrated with PanDA. Some grid-like HPCs (labeled "hpc") have provided external network connectivity and operational policies to allow the execution of standard ATLAS jobs through the central ATLAS software repository. Other HPCs (labeled "hpc_special") have required local ATLAS software installation and typically executed only specific types of ATLAS jobs. PanDA has also been integrated and used at scale with the most common commercial Cloud service providers, such as Amazon and Google [69]. Some cloud service providers (labeled "cloud") have used the central ATLAS software repository, while other Cloud service providers (labeled "cloud_special") have used local ATLAS software due to technical difficulties to access the repository. The different resources are distributed worldwide around more than 40 countries and 150 data centers hosted at universities and laboratories. These sites come in very different sizes, nowadays ranging anywhere between a few hundred cores to a few hundred thousand cores. PanDA is able to tailor jobs to any size and keep all resources full. Some types of resources also are limited to certain types of jobs (typically Monte Carlo simulation) due to I/O restrictions. These imbalances are handled at the worldwide level by applying the Global Shares described in Sect. "Global Share", and prioritizing less-favored activities at other sites.

Since 2022, Vera C. Rubin has been using PanDA for Data Release Production (DRP) campaigns. During Phase 2 of the Rubin Observatory’s Data Preview 0 (DP0.2) in 2021, PanDA demonstrated the capability to run 16 million jobs at the Google-based Interim Data Facility (IDF). Most jobs were processed on a cluster with approximately 4,000 cores, up to 14GB/core RAM with a total CPU usage of 2.5M core-hours. Eight million jobs were also processed for the Hyper Suprime Cam (HSC) reprocessing at the US Data Facility (USDF) at SLAC. Figure 12 shows the accumulation of the number of Vera C. Rubin jobs managed by PanDA since May 2021.

The successful processing of DP0.2 drove the decision to endorse PanDA for the DRP campaigns. The 2023 DRP campaigns are estimated to have around 36 million jobs for the HSC Public Data Release 2 (HSC-PDR2) and around 8 M for the HSC reprocessing. The entire PanDA infrastructure has been deployed [70] on a Kubernetes cluster at SLAC, integrating data facilities in the UK and France in addition to USDF and Google IDF.

The sPHENIX experiment at BNL’s Relativistic Heavy Ion Collider (RHIC) [71] also decided in 2021 to adopt PanDA for offline production and took advantage of the Kubernetes-based approach to the PanDA infrastructure deployment. The PanDA service has been running on an OKD cluster at BNL SDCC, stress tests were conducted in the Summer of 2023, and the entire system has been prepared for data-taking in the Spring of 2024.

The Evolving PanDA Ecosystem

Needs for emerging workflows and computing technologies have driven the steady growth of PanDA’s scope beyond “a layer over distributed batch queues". The integration with the intelligent Data Delivery Service (iDDS) [72] has prompted the evolution of PanDA into an ecosystem to address challenging issues that were not foreseen in traditional workflows. iDDS is an open-source software designed to integrate with various workload management systems, including PanDA. It provides essential functionalities for efficiently managing workflows at different granularities. The following examples demonstrate the expansion of the PanDA ecosystem. PanDA capabilities and flexibilities to manage diverse resources and dynamically tailor jobs for optimal workload processing have played a key role in accomplishing the goals in these examples, while leveraging iDDS for high-level workflow scheduling.

The High Luminosity upgrade to the LHC is expected to start operation in early 2029 [73] and will deliver an unprecedented volume of scientific data at the multi-exabyte scale.

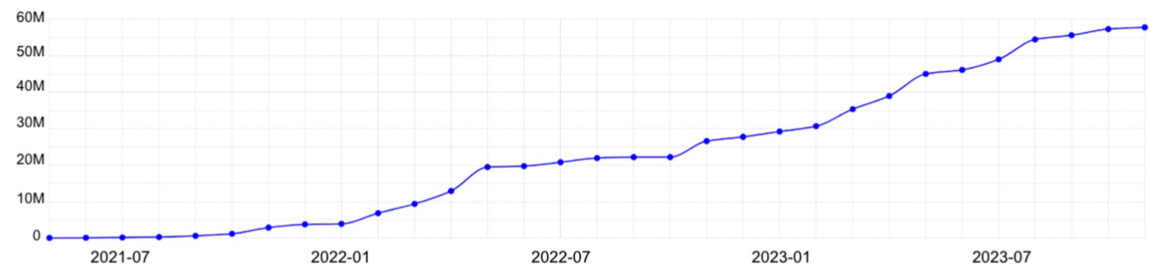


Fig. 12 Accumulated number of Vera C. Rubin jobs managed by PanDA since May 2021

The present LHC computing and data management model will not be viable to ensure fast and reliable data delivery for processing by scientific groups distributed all over the world. Therefore, more efficient and dynamic data access strategies need to be developed. The ATLAS experiment launched the Data Carousel R &D project [74] in 2018 to study the feasibility of getting input data from tape directly for various ATLAS workflows. PanDA and iDDS have enabled a bulk production campaign, with input data resident on tape, to be executed by staging and promptly processing a sliding window of input data onto disk buffer, which helps to decrease the amount of disk storage usage at any one time.

Machine learning is becoming an important tool for data analysis in science experiments. A hyperparameter is a parameter to control the training process in machine learning. Hyperparameter Optimization (HPO) is a resource-intensive procedure to choose a set of optimal hyperparameters for a machine learning algorithm. A fully automated platform has been built with PanDA and iDDS on top of geographically distributed GPU resources provided by the Grid, HPCs, and clouds, such that a huge computing power can be applied to large-scale HPO sessions. This platform incorporates iterative approaches to search hyperparameters, such as Bayesian Optimization through Nevergrad [75] or Scikit-learn [76]. Those approaches require a master application to collect results and trigger successive rounds of hyperparameter generation and evaluation based on previous results.

Active learning is a technique to guide the sampling of the multi-dimensional phase space to find the exclusion contours in an iterative process: the sampled theory phase space points are selected such that the vicinity of the exclusion region is prioritized, reducing the sampling density in the less-interesting areas. It allows searching in a larger space at the same precision while reducing resource usage under the same search space. Users can process, through the PanDA ecosystem, analysis workflows composed of integrated pipelines and active learning to achieve comprehensive exclusion.

HPCs and LCFs are large resources, and future generations of these facilities are expected to have artificial intelligence and machine learning as principal application targets. Considerable effort has been devoted to bringing HPCs and LCFs into PanDA, leading to the development of the fine-grained work scheduling capability in the PanDA ecosystem to enable the high-efficient utilization of these resources. Harvester has played a key role in dealing with various HPC/LCF-specific requirements and operational constraints. The rapidly growing Function-as-a-Service (FaaS) area has been identified as a possible solution to leverage HPCs and LCFs effectively. A high-performance FaaS system, funcX [77], orchestrates scientific workloads across diverse resources, operating as a persistent gateway service in front of an HPC/LCF to route workloads to the resource. New Harvester plugins are under development to integrate funcX with the PanDA ecosystem, such that funcX provides a secure and capable access path for PanDA workloads to reach HPC/LCF resources and be executed by a trusted and locally resident service.

As illustrated by the previous examples, advanced and complex workflows have gained increasing importance in scientific experiments. Support for these workflows allows users to exploit remote computing resources and service providers distributed worldwide, overcoming limitations on local resources and services. The computing options keep increasing across traditional resource providers as well as emerging service levels like FaaS, Platform-as-a-Service (PaaS), and Container-as-a-Service (CaaS), each one providing new advantages and constraints. Users can significantly benefit from these providers, but at the same time, it is cumbersome to deal with multiple providers even in a single analysis workflow with fine-grained requirements coming from their applications' nature and characteristics. There are quite a lot of issues to address, such as the isolation of users from the complexities of distributed heterogeneous providers, resource provisioning for CPU and GPU hybrid applications, integration of FaaS, PaaS, and CaaS providers, smart workload routing, knowledge-based automatic data placement, seamless execution of complex workflows, interoperability between pledged and user resources, interactivity for users on top of asynchronous resources, and on-demand data production. The PanDA ecosystem has been ready to cope with these issues and develop solutions for present and future data-intensive experiments.

Conclusions

It is an extremely difficult challenge to manage data-intensive workloads in terms of effective resource usage, proactive placement of large volumes of data, and quick delivery of analysis results. PanDA provides a solution for scientific experiments to fully exploit their distributed heterogeneous resources fully with demonstrated scalability, usability, flexibility, and robustness. The modular and horizontally scalable architecture of PanDA has been successfully proven over nearly two decades of steady operation in ATLAS, allowing incremental system evolution to adapt to emerging computing technologies in processing, storage, networking, and distributed computing middleware.

The PanDA system has performed very well for ATLAS including the LHC Run 1 and Run 2 data-taking periods while expanding the scope to two big data experiments, the Vera C. Rubin Observatory and the sPHENIX experiment.

Advanced and complex workflows have gained increasing importance in science experiments, driving the steady growth of PanDA evolving into an ecosystem. There are a great number of issues to support emerging workflows and computing technologies, and new developments and challenges are still coming. The PanDA ecosystem is ready to cope with those issues and develop solutions for present and future data-intensive experiments.

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 2023 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.

Author Contributions TM wrote the main manuscript text and other authors contributed relevant sections. All authors reviewed the manuscript.

Data availability The authors confirm that the data supporting the findings of this study are available within the article.

Declarations

Conflict of Interest The authors declare no competing interests.

References

1. Evans, L, Bryant, P (eds.) (2008) LHC Machine. J Inst 3: 08001. https://doi.org/10.1088/1748-0221/3/08/S08001
2. ATLAS Collaboration (2008) The ATLAS Experiment at the CERN Large Hadron Collider. J Inst 3:08003. https://doi.org/10.1088/1748-0221/3/08/S08003
3. Barisits M et al (2019) Rucio: scientific data management. Comput Softw Big Sci 3:11. https://doi.org/10.1007/s41781-019-0026-3
4. Worldwide LHC Computing Grid (WLCG). https://wlcg.web.cern.ch/. Accessed 13 Nov 2023
5. Argonne Leadership Computing Facility. https://www.alcf.anl.gov/. Accessed 13 Nov 2023
6. Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/. Accessed 13 Nov 2023
7. Ivezic Z et al (2019) LSST: from science drivers to reference design and anticipated data products. Astrophys J 873(2):111. https://doi.org/10.3847/1538-4357/ab042c
8. Adare A, et al An upgrade proposal from the PHENIX Collaboration arXiv:1501.06197
9. Grigoras AG et al (2014) JAliEn—a new interface between the AliEn jobs and the central services. J Phys Conf Ser 523(1):012010. https://doi.org/10.1088/1742-6596/523/1/012010
10. ALICE Collaboration (2008) The ALICE experiment at the CERN LHC. A large ion collider experiment. JINST 3: 08002. https://doi.org/10.1088/1748-0221/3/08/S08002
11. Sfiligoi I (2008) glideinWMS-a generic pilot-based workload management system. J Phys Conf Ser 119(6):062044. https://doi.org/10.1088/1742-6596/119/6/062044
12. Collaboration CMS (2008) The CMS experiment at the CERN LHC. JINST 3:08004. https://doi.org/10.1088/1748-0221/3/08/S08004
13. Stagni F et al (2020) The DIRAC interware: current, upcoming and planned capabilities and technologies. EPJ Web Conf. 245:03035. https://doi.org/10.1051/epjconf/202024503035
14. LHCb Collaboration (2008) The LHCb detector at the LHC. JINST 3: 08005. https://doi.org/10.1088/1748-0221/3/08/S08005
15. Kou E, et al The Belle II Physics Book arXiv:1808.10567
16. The CTA Consortium (2011) Design concepts for the Cherenkov Telescope Array CTA: An advanced facility for ground-based high-energy gamma-ray astronomy. Exp Astron 32:193. https://doi.org/10.1007/s10686-011-9247-0
17. Deelman E et al (2019) The evolution of the Pegasus Workflow Management Software. Comput Sci Eng 21(4):22–36. https://doi.org/10.1109/MCSE.2019.2919690
18. LIGO–Virgo–KAGRA Collaboration (2020) Prospects for observing and localizing gravitational-wave transients with advanced LIGO, advanced Virgo and KAGRA. Living Rev Relat 23: 3. https://doi.org/10.1007/s41114-020-00026-9
19. VOMS—Virtual Organization Membership Service in Grid computing. https://italiangrid.github.io/voms/. Accessed 13 Nov 2023
20. Ceccanti E et al (2017) The INDIGO-Datacloud authentication and authorization infrastructure. J Phys Conf Ser 898(10): 10201. https://doi.org/10.1088/1742-6596/898/10/102016
21. OpenID Connect (OIDC). https://openid.net/connect/. Accessed 13 Nov 2023
22. OAuth 2.0. https://oauth.net/2/. Accessed 13 Nov 2023
23. Apache HTTP Server—an open-source HTTP server for modern operating systems. https://httpd.apache.org/. Accessed 13 Nov 2023
24. Fullana Torregrosa E et al (2019) Grid production with the ATLAS Event Service. EPJ Web Conf. 214:04016. https://doi.org/10.1051/epjconf/201921404016
25. Python Web Server Gateway Interface (WSGI). https://peps.python.org/pep-3333/. Accessed 13 Nov 2023
26. Apache module supporting the Python WSGI specification. https://modwsgi.readthedocs.io/en/master/. Accessed 13 Nov 2023
27. Anisenkov A et al (2020) CRIC: computing resource information catalogue as a unified topology system for a large scale, heterogeneous and dynamic computing infrastructure. EPJ Web Conf. 245:03032. https://doi.org/10.1051/epjconf/202024503032
28. Nilsen JK et al (2015) ARC control tower: a flexible generic distributed job management framework. J Phys Conf Ser 664:03032. https://doi.org/10.1088/1742-6596/664/6/062042
29. Google Cloud Logging. https://cloud.google.com/logging/. Accessed 13 Nov 2023
30. Fluentd—an open source data collector for unified logging layer. https://www.fluentd.org. Accessed 13 Nov 2023

31. Logstash—a server-side data processing pipeline. https://www.elastic.co/logstash/. Accessed 13 Nov 2023
32. Apache ActiveMQ—flexible and powerful open source multi-protocol messaging. https://activemq.apache.org/. Accessed 13 Nov 2023
33. Django Framework. https://www.djangoproject.com. Accessed 13 Nov 2023
34. ElasticSearch—an open search and analytics solution. https://www.elastic.co/. Accessed 13 Nov 2023
35. Karavakis E et al (2017) Unified monitoring architecture for IT and grid services. J Phys Conf Ser 898:092033. https://doi.org/10.1088/1742-6596/898/9/092033
36. Data-Driven Documents. https://d3js.org. Accessed 13 Nov 2023
37. Chart.js —Open source HTML5 Charts. https://www.chartjs.org. Accessed 13 Nov 2023
38. Cuhadar Donszelmann T et al (2020) ART—ATLAS Release Tester using the Grid. EPJ Web Conf. 245:05015. https://doi.org/10.1051/epjconf/202024505015
39. The ATLAS Experiment's main offline software repository. https://gitlab.cern.ch/atlas/athena. Accessed 13 Nov 2023
40. JSON Web Token (JWT)—A compact URL-safe means of representing claims to be transferred between two parties. https://jwt.io/. Accessed 13 Nov 2023
41. HTCondor—a software system that creates a high-throughput computing environment. https://htcondor.org/. Accessed 13 Nov 2023
42. Bockelman B et al (2021) Principles, technologies, and time: the translational journey of the HTCondor-CE. J Comput Sci 52: 101213. https://doi.org/10.1016/j.jocs.2020.101213
43. Ellert M et al (2007) Advanced Resource Connector middleware for lightweight computational Grids. Future Gener Comput Syst 23(2):219–240. https://doi.org/10.1016/j.future.2006.05.008
44. Kubernetes—Production-Grade Container Orchestration. https://kubernetes.io/. Accessed 13 Nov 2023
45. Jette M, et al (2003) SLURM: Simple linux utility for resource management. https://doi.org/10.1007/10968987_3
46. Portable Batch System (PBS). http://www.pbspro.org/. Accessed 13 Nov 2023
47. Lancium—Power Orchestration for Energy-Intensive Industries. https://lancium.com/. Accessed 13 Nov 2023
48. CILogon—An Integrated Identity and Access Management Platform for Science. https://www.cilogon.org/. Accessed 13 Nov 2023
49. European Organization for Nuclear Research (CERN). https://www.home.cern/. Accessed 13 Nov 2023
50. The Scientific Data and Computing Center (SDCC) at Brookhaven National Laboratory (BNL). https://www.sdcc.bnl.gov/. Accessed 13 Nov 2023
51. Google Identity Platform. https://cloud.google.com/identity-platform. Accessed 13 Nov 2023
52. McNab A (2010) The GridSite Web/Grid security system. J Phys Conf Ser 219:062058. https://doi.org/10.1088/1742-6596/219/6/062058
53. HEP-SPEC06 (HS06) benchmarking. http://w3.hepix.org/benchmarking. Accessed 13 Nov 2023
54. PanDA documentation. https://panda-wms.readthedocs.io/en/latest/index.html. Accessed 13 Nov 2023
55. Harvester documentation. https://github.com/HSF/harvester/wiki. Accessed 13 Nov 2023
56. Stanford Linear Accelerator Center (SLAC). https://www6.slac.stanford.edu/. Accessed 13 Nov 2023
57. Helm—the package manager for Kubernetes. https://helm.sh/. Accessed 13 Nov 2023
58. OKD—The Community Distribution of Kubernetes that powers Red Hat OpenShift. https://www.okd.io/. Accessed 13 Nov 2023
59. GitHub—a code hosting platform for version control and collaboration. https://github.com/. Accessed 13 Nov 2023
60. Apache License, Version 2.0. https://www.apache.org/licenses/LICENSE-2.0. Accessed 13 Nov 2023
61. Jira—Issue & Project Tracking Software. https://www.atlassian.com/software/jira. Accessed 13 Nov 2023
62. The Python Package Index (PyPI). https://pypi.org/. Accessed 13 Nov 2023
63. Titan at Oak Ridge National Laboratory. https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/. Accessed 13 Nov 2023
64. Theta at Argonne Leadership Computing Facility. https://www.alcf.anl.gov/alcf-resources/theta. Accessed 13 Nov 2023
65. Cori at National Energy Research Scientific Computing Center (NERSC). https://docs.nersc.gov/systems/cori/. Accessed 13 Nov 2023
66. Perlmutter at NERSC. https://docs.nersc.gov/systems/perlmutter/. Accessed 13 Nov 2023
67. MareNostrum 4 Supercomputer at the Barcelona Supercomputing Center. https://www.bsc.es/marenostrum. Accessed 13 Nov 2023
68. Vega at the Institute of Information Science. https://www.izum.si/en/vega-en/. Accessed 13 Nov 2023
69. Barreiro Megino FH et al (2021) Seamless Integration of Commercial Clouds with ATLAS Distributed Computing. EPJ Web Conf. 251:02005. https://doi.org/10.1051/epjconf/202125102005
70. Karavakis E, et al (2023) Integrating the PanDA Workload Management System with the Vera C. Rubin Observatory. Proceedings of 26th International Conference on Computing in High Energy and Nuclear Physics (CHEP) (to appear)
71. Harrison M, Ludlam T, Ozaki S (2003) RHIC project overview. Nucl Instrum Meth A 499:235–244. https://doi.org/10.1016/S0168-9002(02)01937-X
72. Guan W et al (2021) An intelligent Data Delivery Service for and beyond the ATLAS experiment. EPJ Web Conf. 251:02007. https://doi.org/10.1051/epjconf/202125102007
73. LHC long shutdown schedule change. https://hilumilhc.web.cern.ch/article/ls3-schedule-change. Accessed 13 Nov 2023
74. Borodin M et al (2021) The ATLAS Data Carousel Project Status. EPJ Web Conf. 251:02006. https://doi.org/10.1051/epjconf/202125102006
75. Bennet P et al (2021) Nevergrad: black-box optimization platform. ACM SIGEVOlution 14:8. https://doi.org/10.1145/3460310.3460312
76. Pedregosa F et al (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
77. Chard R, et al (2020) funcX: a federated function serving fabric for science. Proceedings of 29th international symposium on high-performance parallel and distributed computing, 65. https://doi.org/10.1145/3369583.3392683