Eur. Phys. J. C (2025) 85:265
https://doi.org/10.1140/epjc/s10052-025-13963-y

THE EUROPEAN PHYSICAL JOURNAL C

Regular Article - Computing, Software and Data Science

AUDITOR

Accounting data handling toolbox for opportunistic resources

Michael Boehler[1,a], **Ralf Florian von Cube**[2], **Max Fischer**[2], **Manuel Giffels**[2], **Raphael Kleinemühl**[3], **Stefan Kroboth**[1], **Benjamin Rottler**[1], **Dirk Sammel**[1], **Matthias Schnepf**[2], **Markus Schumacher**[1], **Raghuvar Vijayakumar**[1]

[1] Physikalisches Institut, Albert-Ludwigs-Universität Freiburg, Hermann-Herder-Str. 3, 79104 Freiburg, Germany
[2] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, Germany
[3] Fakultät 4, Bergische Universität Wuppertal, Gaußstraße 20, 42119 Wuppertal, Germany

Received: 11 February 2025 / Accepted: 13 February 2025

Abstract Increasing computing demands and concerns about energy efficiency in high-performance and high-throughput computing are driving forces in the search for more efficient ways to use available resources. Sharing resources of an underutilised cluster with a high workload cluster increases the efficiency of the underutilised cluster. The software COBalD/TARDIS can dynamically and transparently integrate and disintegrate such resources. However, sharing resources also requires accounting. AUDITOR (Accounting Data Handling Toolbox for Opportunistic Resources) is a modular accounting ecosystem that is able to cover a wide range of use cases and infrastructures. Accounting data are gathered via so-called collectors, which are designed to monitor batch systems, COBalD/TARDIS, cloud schedulers, or other sources of information. The data is stored in a database, and access to the data is handled by the core component of AUDITOR, which provides a REST API along both Rust and a Python client libraries. So-called plugins can take actions based on accounting records. Depending on the use case, one simply selects a suitable collector and plugin from a growing ecosystem of collectors and plugins. To facilitate the development of collectors and plugins for yet uncovered use cases by the community, libraries for interacting with AUDITOR are provided.

1 Introduction

The increasing computational demand in High Energy Physics (HEP) and other research areas as well as increasing concerns about energy efficiency in high-performance and high-throughput computing are driving forces in the search for more efficient ways to utilise available resources. One way to increase efficiency is to weaken the borders between providers of computing resources, such as HPC (High-Performance Computing), HTC (High-Throughput Computing) clusters or even cloud providers, by sharing resources across these boundaries, effectively shifting unused resources to overbooked resources.

It is vital that this sharing of resources is performed both dynamically and opportunistically, meaning that resources are only integrated from cluster A into cluster B when there is demand in B and availability in A, and vice versa. Since demand and availability can change rapidly, the integration and disintegration process must be automated. Ideally, the integration of resources from A into B is transparent, such that users of cluster B are unaware that their jobs are executed on cluster A. The software COBalD/TARDIS [1,2] manages opportunistic resource integration in a minimally invasive manner with the use of virtualisation or containerisation and by utilising available infrastructure, such as the (batch system or cloud) schedulers of the respective resource providers.

Sharing of resources between resource providers naturally raises the desire for an accounting system. For example, involved stakeholders may need to bill each other or resources may be exchanged for certain benefits (e.g. higher priorities on another cluster). Usually, the schedulers of the common batch systems, such as Slurm [3], MOAB [4] or HTCondor [5], have an integrated accounting mechanism. But whenever a cluster is dynamically integrated into another cluster, an independent instance is required, which can gather the accounting information of all involved resources. The soft-

[a] e-mail: michael.boehler@physik.uni-freiburg.de (corresponding author)

ware ecosystem that is presented here will fill this technology gap.

Computing sites participating in the Worldwide LHC Computing Grid (WLCG) are connected to the Grid with a so-called Compute Element (CE). There are two independent implementations, ARC-CE [6] and HTCondor-CE [7], which accept computing jobs from the WLCG and forward them to the local compute cluster. These sites guarantee a certain contribution to the WLCG in the form of computing time, so-called pledges. They prove the fulfilment of their respective pledges by tracking the amount of resources used by the jobs running on their hardware. This task is usually performed by the CEs, which transmit accounting data to a central accounting service either by means of an integrated accounting service, such as ARC-CE [6], or via the Accounting Processor for Event Logs (APEL) client [8] in the case of HTCondor-CE. In the case of clusters that are integrated with each other via COBalD/TARDIS, the available accounting systems have no way to transmit the resources provided by each individual cluster.

Given the wide range of use cases and software environments present in the field, an accounting system must be both flexible and extensible enough to cover current and potential future use cases. This work introduces the open-source software AUDITOR [9] (Accounting Data Handling Toolbox for Opportunistic Resources), which provides a solution to this problem.

At its core, AUDITOR contains a database, which stores so-called *records*. *Collectors* obtain information relevant for accounting from a source (for example a scheduler) and send this information to AUDITOR in the form of a record. *Plugins* request a subset of records from AUDITOR and act based on the information stored in the records. For example, a plugin may process the retrieved records and forward them to another accounting system in the appropriate format. Alternatively, a plugin could use the information stored in the records to tune parameters of a system, create a bill or compute the CO_2 footprint. The combination of collectors and plugins is chosen based on the use case and software environment, and multiple collectors and plugins can be used with a single AUDITOR instance. Various collectors and plugins are readily available, making it suitable for a diverse range of use cases. Collectors and plugins communicate with AUDITOR via a REST interface. Client libraries, which simplify the interaction with AUDITOR and, hence, the development of collectors and plugins, are provided for two programming languages: Rust [10] and Python [11].

The Sect. 2 introduces the design and interfaces of AUDITOR, its implementation, as well as the available client libraries, collectors and plugins. A special use case of AUDITOR is illustrated in the Sect. 3, which also presents data that has been recorded over a certain period of time in productive use. The Sect. 4 summarises the findings and gives an outlook on future developments.

2 Components of the ecosystem

This section introduces AUDITOR, its design and interfaces, as well as various collectors and plugins.

2.1 AUDITOR

AUDITOR is the central component of an ecosystem (Fig. 1) and is designed to accept records from collectors and to provide records to plugins.

Collectors and plugins interact with AUDITOR via a REST API. Client libraries in the programming languages Rust and Python allow to embed the REST calls in either one of them. AUDITOR stores all its data in a PostgreSQL[1] database and is in itself stateless. This makes it well suited for high-availability setups where multiple instances are placed behind a load balancer.

2.1.1 Records

A *record* describes a single entity of accountable information. It can be created and pushed to AUDITOR either via a direct call to the REST API or through the AUDITOR client tools. The *record* is then stored as an entry in a database. Each *record* is identified by a unique `record_id`. Context can be provided via the `meta` field, which is a *hashmap* that maps string keys to arrays of string values. Zero or more `components` indicate the types and amounts of resources that need to be accounted for. These could for example be CPU cores, RAM, disk space, GPUs, etc. Each `component` can have zero or more `scores` attached, which allow one to relate `components` of the same kind with different performance. For example, benchmarking scores such as HEP-SPEC06 [12] or HEPScore23 [13] can be used to quantify the performance of CPUs. Each score is identified by a `name` and a `value`. The timestamps for the start and end of the availability of resources are specified by `start_time` and `stop_time` respectively. *Records* received from AUDITOR also carry a `runtime` field, which is the difference between `stop_time` and `start_time` in seconds. All dates and times in the database are in UTC.

The following listing shows a summary of all fields. Square brackets (`[ ]`) indicate that this field can hold zero or more elements. The type and other additional information are given in parentheses.

– `record_id` (String, unique)

[1] https://www.postgresql.org/.

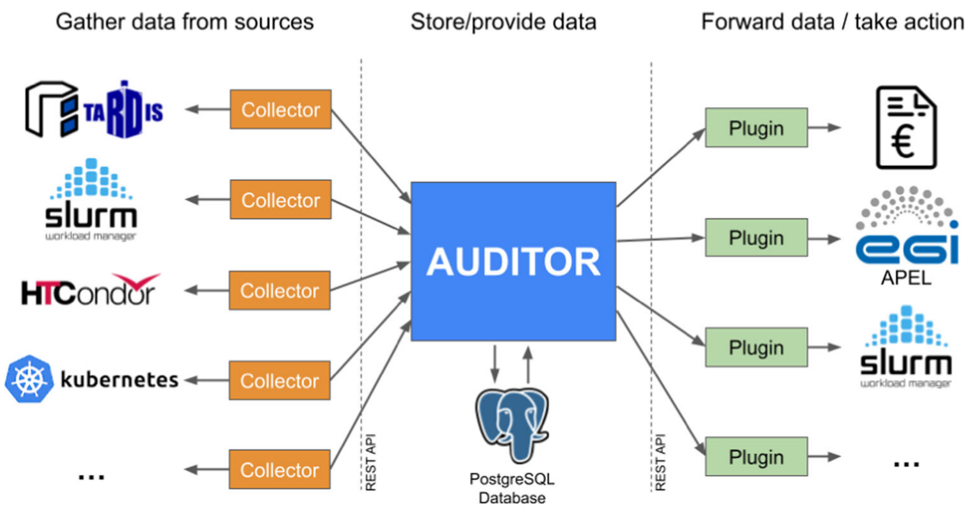

Fig. 1 Overview of the AUDITOR ecosystem. AUDITOR accepts *records* from *collectors*, stores them in a PostgreSQL database and offers these records to *plugins*, which act based on the records

- `[meta]` (HashMap[String → [String]])
- `[components]` (Array)
  - `name` (String)
  - `amount` (Integer)
  - `[scores]` (Array)
    - `name` (String)
    - `value` (Float)
- `start_time` (Datetime, UTC)
- `stop_time` (Datetime, UTC)
- `runtime` (Integer,seconds, output-only)

2.1.2 *Interface*

Records can be sent to AUDITOR via the `POST` method of the REST endpoint `/record`, which primarily accepts complete records, but also records without `stop_time`, which can be provided later. This is useful for certain collectors that, due to the nature of their source, can only register state changes i.e. the resource became available and the resource became unavailable.

In particular, this avoids the requirement of a collector to keep track of the state in between state changes. Instead, AUDITOR manages the state by allowing records to be finalised at a later point in time. This is not only more convenient but also improves the crash resilience of collectors. Records are transferred to AUDITOR as JSON payload in an HTTP request. Upon receiving, AUDITOR performs input validation on each record during the de-serialisation of the JSON object, and it rejects records with strings containing forbidden characters or with non-positive values. This avoids accidental wrong input and malicious attacks such as SQL injections. The provided client libraries perform exactly the same input validation locally, meaning that unacceptable input will be rejected even before being sent to AUDITOR.

Plugins can request a list of records via the `GET` method of the endpoint `/records` which receives all records from the database. Alternatively, the range of records received can be limited based on `stop_time` or `start_time` with `/records?start_time[gt]=<timestamp1>&start_time[lt]=<timestamp2>`, respectively.

The `<timestamp>` is required to be in RFC3339 [14] format, and the time zone is assumed to be UTC if not specified otherwise. Furthermore, AUDITOR offers advanced filtering options to refine record queries. Users can combine any record fields, such as `start_time`, `stop_time`, `meta`, `component`, and `runtime`, to narrow down their search. Operators are used in conjunction with these fields to define ranges for time fields (`gt`, `gte`, `lt`, `lte`) and to specify inclusion or exclusion criteria for meta and component fields (`contains`, `does not contain`). This comprehensive filtering capability allows for highly specific and targeted queries, enhancing the speed of data retrieval.

2.1.3 *Implementation*

AUDITOR is written in the RUST programming language and uses the ACTIX- WEB library[2] to implement the REST interface. Interaction with the PostgreSQL database is managed by the `sqlx`[3] library, which offers compile-time SQL query verification and SQL table migration. In order to better observe changes, logging is performed with the `tracing` library,[4] which outputs machine-readable traces with spans such that even performance-related metrics can be derived from the traces.

Metrics are exported to a Prometheus [15] scraper via the `/metrics` endpoint. These metrics include HTTP request metrics as well as a diverse set of database metrics. The latter can be partially or fully disabled to avoid potential negative performance impacts for large databases.

[2] https://actix.rs/.
[3] https://github.com/launchbadge/sqlx.
[4] https://github.com/tokio-rs/tracing.

2.2 AUDITOR client

Collectors and plugins can use AUDITOR's REST API directly. However, to simplify the communication, client libraries are provided for Rust and Python. These libraries offer high-level, idiomatic ways of constructing records to interact with AUDITOR and handle errors without the need for the developer to manually build the HTTP requests. These interfaces also validate the payload before sending it. This avoids accidentally sending records that will be rejected by AUDITOR. The Python client library is a slim Python layer on top of the Rust client library, implemented using the PyO3 library.[5]

2.3 Collectors

The purpose of collectors is to obtain relevant data for accounting from a data source (e.g. a batch system), create a record, and send it to AUDITOR endpoint `/record`. This section describes collectors that have already been implemented. However, as Fig. 1 shows, a kubernetes collector is already being worked on and will be released soon.

2.3.1 Slurm epilog collector

The *Slurm Epilog Collector* is a collector for the Slurm batch system and is executed at the end of a batch job during the so-called *epilog* (either on the head node or on the worker node). The information needed for constructing a record is obtained by parsing the job information output from `scontrol show job <jobid>` of the respective job. The collector is compiled into a portable, statically linked binary with no system dependencies. It is extensively configurable, and is hence well-suited for various use cases and environments. This is a simple yet effective way to send Slurm job information to AUDITOR. However, this approach comes with caveats inherent to execution in the epilog. For example, depending on the execution time of the job itself, executing `scontrol` can substantially extend the overall run time of the job. Trying to keep the epilog's run time low limits the available options for recovering from errors (for example, failures during sending) as every retry undesirably increases the elapsed time. Furthermore, the information about the job, which can be obtained from Slurm at the time of the *epilog*, is limited and, as such, may exclude certain use cases. For use cases where these limitations are not acceptable, the *Slurm Collector*, described in the following section, serves as an alternative.

2.3.2 Slurm collector

The *Slurm Collector* queries the Slurm accounting database via the command-line tool `sacct` for new jobs at regular intervals. For each new job, a record is formed, which is placed on a queue for sending. At regular configurable intervals, a worker thread sends the records in the queue to AUDITOR. In case sending fails, the record will be placed on the queue again, and sending is retried at the next interval. The queue is persistently stored to disk in a SQLite3[6] database for increased crash resilience. The collector is also compiled to a portable, statically linked binary with no system dependencies. It only relies on Slurm and, as such, can run wherever `sacct` is available. This collector can also be extensively configured.

To compare the two collectors for Slurm, we can briefly summarize: the *SLURM Epilog Collector* is a simple script that, once executed, can save the metadata of a Slurm job as a record in the AUDITOR database. The *Slurm Collector* is operated as a stand-alone Linux service. If you want little metadata that is fully available at the end of a Slurm job, the *Slurm Epilog Collector* as a simple script is much leaner. If you need metadata that is only available after the job has been completed, you have to use the *Slurm Collector* instead.

2.3.3 HTCondor collector

The collector for the HTCondor batch system regularly queries HTCondor to retrieve information about newly finished jobs from the history. The interval can be configured and defaults to 15 min. For each job, a record is produced and sent to AUDITOR using the Python client libraries. The HTCondor internal unique identifier of the latest processed job is stored in an SQLite3 database. By saving the status, the collector and/or the host on which the collector is running can be restarted. The same unique identifier is mapped to the `record_id` in the AUDITORrecord. The attributes of the HTCondor jobs, which are referred to as ClassAds [16], are mapped to proper components and scores of the corresponding data records. This is fully configurable. Start-, stop-, and runtime are inferred from the corresponding job attributes.

2.3.4 TARDIS collector

COBalD/TARDIS manages a fleet of so-called *drones*, which represent the opportunistic resources, typically in the form of virtual machines or containers. Each drone can be in one of a finite number of states (*booting*, *running*, *stopped*, etc.). Transitions between these states are tracked by TARDIS. Every state transition, for example when a drone starts or

5 https://pyo3.rs.

6 https://sqlite.org/.

stops being available, is also forwarded to a plugin interface of TARDIS. The *TARDIS Collector* connects to the TARDIS plugin interface and gets notified for each state transition. Since only transitions are available, a full record can only be formed after both the start and stop events of a particular drone are received by the collector. In principle, this requires the collector to keep track of the individual drones until they become unavailable. To avoid this, the TARDIS collector uses AUDITOR's `POST` method to send an incomplete record (without the `stop_time`) when a drone becomes available. Later it sends an updated record via the `PUT` method when the drone becomes unavailable. This shifts the burden of keeping track of the drones' state to the database of AUDITOR, which therefore reduces the complexity of the collector and makes it less prone to data loss in case of a crash. This collector is shipped with TARDIS.

2.4 Plugins

AUDITOR plugins can serve different purposes, either allowing only the tracking of resource usage, or providing the ability to perform additional operations based on accounting information. For this purpose, they request a list of AUDITOR records and use this data to take action.

2.4.1 Priority plugin

The *Priority Plugin* uses records in AUDITOR to regularly tune parameters of a system, particularly the priorities of different users or groups on a batch system. Since the command that is executed by the plugin is freely definable, the plugin can be used for any manipulation, e.g. of a batch system based on the calculated priority. A use case for this plugin is when multiple resource providers (e.g. groups) combine their resources in an opportunistic manner. The combined resources are served by a single *overlay batch system* (OBS) to which the users of the individual clusters submit their jobs. Such a setup effectively increases the cluster's usage efficiency beyond its boundaries. However, the involved providers typically do not contribute the same amount of resources to the aggregated cluster, leading to potentially unfair situations. Based on the amount of resources provided by each provider (which are tracked by AUDITOR), the Priority plugin prioritises users belonging to each provider in the OBS. Hence, users of a provider who contributed more resources than others, will have a higher priority, leading to their jobs being scheduled earlier.

The priority plugin runs as a service and regularly updates the priorities at configurable intervals. As such, changing contributions over time will continuously be reflected in the priorities. The provided resources c_i for each provider i are computed as

$$c_i = \int_{t_{\text{now}}-x}^{t_{\text{now}}} N_i(t) s_i dt, \tag{1}$$

where $N_i(t)$ is the number of chosen components of the respective records at time t of provider i, and s_i is a chosen score associated with the record. Only records that finished within x seconds before the current time (t_{now}) are considered.

The provided resources are then mapped to priorities within the configurable range $[p_{\min}, p_{\max}]$ according to one of two possible computation modes. The `FullSpread` mode computes the priorities such that the entire range is exploited by assigning the lowest and highest priority $p_{\min}$ and $p_{\max}$ to the users belonging to the provider contributing the least and most resources, respectively. All other priorities are placed in between according to their contribution. Given $c_{\min}$ and $c_{\max}$ as the minimum and maximum of $c_i \; \forall \, i$, respectively, the formula for computing the priority of a user belonging to provider i using the `FullSpread` mode is given as:

$$p_i = \frac{c_i - c_{\min}}{c_{\max} - c_{\min}} (p_{\max} - p_{\min}) + p_{\min}. \tag{2}$$

This approach can lead to large undesirable changes in priorities when each party provides roughly the same amount of resources. The second mode, `ScaledBySum`, on the other hand, maps the sum of all provided resources to the maximum priority according to the formula:

$$p_i = \frac{c_i}{\sum_i c_i} (p_{\max} - p_{\min}) + p_{\min}. \tag{3}$$

This typically leads to smoother changes in the priorities of the individual providers for two consecutive runs of the plugin. This approach, inherently, reaches the maximum priority only if the resources are provided by a single provider.

The computed priorities can be used in a user-definable shell command, which is executed after computing the priorities. Among other use cases, the shell command can be used to change priorities in a batch system.

Furthermore, the plugin can export the provided resources c_i and the computed priorities p_i to an HTTP endpoint (`/metrics`) that can be scraped by Prometheus. This allows for near real-time monitoring of the priority plugin using a Grafana dashboard.[7]

[7] https://grafana.com/.

2.4.2 APEL plugin

The APEL plugin is designed to forward accounting data from an AUDITOR instance to the Accounting Processor for Event Logs (APEL) accounting service, operated by the European Grid Infrastructure (EGI).

APEL is an accounting tool that collects accounting data from sites participating in the EGI and WLCG infrastructures, as well as from sites belonging to other Grid organisations that are collaborating with EGI. The accounting information is gathered from sources, processed, and published on the EGI/WLCG Accounting Portal. Compute Elements (CE) of the WLCG, such as HTCondor-CE, report the accounting data to APEL via the APEL client.

The *APEL plugin* of AUDITOR adds another possibility. The (current) advantage of the APEL plugin over the other tools is the possibility to generate and send reports for several sites. This is necessary for sites that operate an *Overlay Batch System* (OBS). Such a system distributes incoming jobs to several connected compute sites according to the availability of resources. In this scenario, not the OBS should be accounted for, but the resources that were provided by the individual sites. This information can be included in the meta information of AUDITOR records by *Collectors* (see Sect. 2.3) and is later used by the APEL plugin.

The APEL plugin provides two CLI commands:
`auditor-apel-publish` and
`auditor-apel-republish`.

The first command continuously sends *summary messages* containing information about the finished jobs of the current month to APEL. This happens at a given frequency, e.g. once per day. In addition, a *sync message* containing the total number of jobs for the current month is sent. With the second command, a manual republishing for a specified site and period of time is triggered, which creates a single summary message and sends it to APEL. The option to report a list of individual finished jobs instead of a summary message will be implemented in a future release.

All important settings for the APEL plugin are defined in a configuration file, such as the list of sites to report to APEL, connection- and authentication details, and the names of the relevant fields in the AUDITOR records.

To ensure that the APEL plugin is able to correctly communicate with the APEL server infrastructure, a local test environment has been deployed that recreates the APEL infrastructure. It sets up an ARGO Messaging Service [17] that accepts messages from the plugin. The messages are passed to a local APEL server via the so-called Secure STOMP Messenger, which is also part of the APEL project [8]. By using this environment a successful authentication with the ARGO broker could be tested, the validity of the transmitted ARGO messages cloud investigated and it also could be ensure that the wrapped APEL records are well-formed.

3 Example use case

This section illustrates a particular use case that utilises AUDITOR (Sect. 2.1), the TARDIS Collector (Sect. 2.3.4) and the Priority plugin (Sect. 2.4.1).

At the University of Freiburg, three HEP groups have access to two computing clusters. One of those clusters is a HTC cluster the ATLAS- BFG, operated by one of the HEP groups and served by the Slurm batch system. The second cluster is NEMO [18], operated by the computing center of the university and served by the MOAB batch system [4]. ATLAS- BFG is optimised for HEP workloads and, as such, is the preferred choice for HEP researchers. Each HEP group also has a dedicated quota in NEMO. However, switching between clusters depending on available resources is tedious due to the different batch systems and disjoint data storage. Instead, each individual group's quota in NEMO is made available in ATLAS- BFG as opportunistic resources. This is achieved by automatically and dynamically (depending on utilisation of NEMO and demand on ATLAS- BFG) spawning virtual machines (VMs) on NEMO that resemble worker nodes of ATLAS- BFG and are integrated as such. Therefore, ATLAS- BFG virtually appears to have more worker nodes available. Jobs submitted to ATLAS- BFG then transparently run on NEMO. Spawning and management of VMs is handled by a dedicated COBalD/TARDIS instance per HEP group [19]. Even though the VMs are associated with a single group, each VM can be used by the members of all groups in order to use the resources more efficiently. Therefore, each HEP group's quota in NEMO is effectively shared among all HEP groups.

Due to various factors, the amount of resources provided by each group is not equal. For example, users can still directly use NEMO, which effectively reduces the fraction of the group's quota available for integration into ATLAS- BFG. This can lead to potentially unfair situations since groups may not contribute equally to the aggregated cluster.

One option to restore fairness is to match the job priority of groups to the amount of resources provided. Therefore, jobs of a group that provides more resources will be scheduled earlier than jobs of other groups.

This can be achieved using AUDITOR, the TARDIS collector, and the Priority plugin. The TARDIS collector tracks the provided resources and forwards this information to an AUDITOR instance. Each hour, the Priority plugin requests all records that finished in the past 14 days. From this data it computes the new priorities according to the `ScaledBySum` mode described in Sect. 2.4.1 and updates the group priorities in Slurm accordingly.

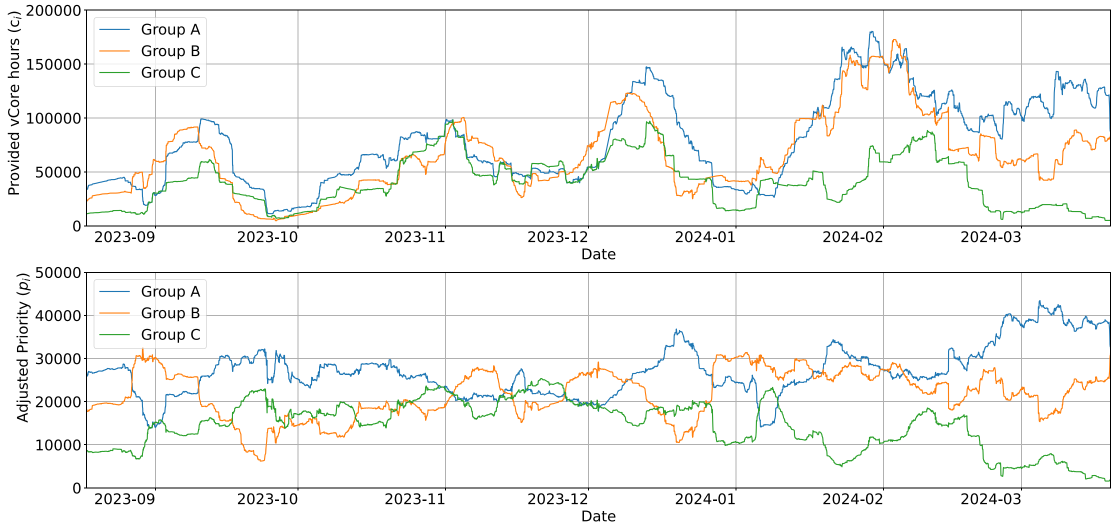

Fig. 2 The top plot shows the integral over the CPU core hours $(c_i(t))$ of the last 14 days per group (i) according to Eq. 1. The bottom plot shows the computed priority $(p_i(t))$ per group (i) using the `ScaledBySum` mode of the priority plugin according to Eq. 3

Figure 2 shows how this setup operates over several months. The top plot shows the computed core hours $c_i(t)$ according to Eq. 1, where $N_i(t)$ is the number of cores provided for all records associated with group $i \in \{A, B, C\}$ at a time instance t and s_i is the corresponding score (since NEMO is an HPC cluster with exactly one CPU architecture, s_i is a constant value, with $s_i = 10.0$ HEPScore23/core). Note that each point in the top graph is an integral over the resources provided within the last 14 days. The bottom graph shows the priority computed for each group with the `ScaledBySum` mode as defined in Eq. 3. The minimum and maximum priority values are set to $p_{\text{min}} = 1$ and $p_{\text{max}} = 65535$, respectively.

The priority curve (Fig. 2 lower plot) follows the behaviour of the curve of the resources provided (Fig. 2 upper plot). If more resources are contributed, the priority is adjusted accordingly. This is particularly evident where, for example, group A in blue provides the most resources towards the end of December and receives the highest priority shortly thereafter.

4 Discussion

While designing the accounting ecosystem AUDITOR, the emphasis was put on flexibility and extensibility in order to accommodate for a wide range of current and future use cases. Even though AUDITOR and its ecosystem is still being actively developed, it has reached a level of maturity, making it suitable for production use. A set of collectors and plugins is provided, and more are currently being developed by the core team and contributors.

The clients for Rust and Python facilitate the ecosystem's growth by simplifying the development of collectors and plugins for collaborators. By offering a high-level API via the client libraries, the underlying REST API can easily be updated and extended without breaking user code.

AUDITOR is lightweight and therefore has a small footprint in terms of resource consumption. Multiple instances for different use cases can rely on the same PostgreSQL instance without any interference. Since AUDITOR is completely stateless, multiple instances can be operated in parallel behind a load balancer for high-availability setups.

Development on AUDITOR continues, and future developments will include methods for authentication and authorisation, and a plugin for tracking the CO_2 footprint of utilised compute resources. Furthermore, a collector for Kubernetes[8] is currently under active development.

AUDITOR is an open source project characterised by a highly flexible design and collaborative development across multiple scientific disciplines and communities, making it easily adaptable to new use cases.

Funding This work was supported by the Federal Ministry of Education and Research (BMBF) within the joint project "Federated Digital Infrastructures for Research on Universe and Matter" (FIDIUM) and its subprojects (05H21VFRC2, 05H21VKRC2 and 05H21PXRC2). The HPC-cluster NEMO in Freiburg is supported by the Ministry of Science, Research and the Arts Baden-Württemberg through the bwHPC grant and by the German Research Foundation (DFG) through grant no INST 39/963-1 FUGG.

Data Availability Statement Data will be made available on reasonable request. [Author's comment: The data of the example use case is real data, analyzed and evaluated with AUDITOR v0.5.0. There are restrictions on the availability of the data, so it is not publicly avail-

[8] https://kubernetes.io/.

able. Data will be made available by the author [Michael Boehler] upon reasonable request.]

Code Availability Statement My manuscript has associated code/software in a data repository. [Author's comment: The software described in this paper is the accounting ecosystem AUDITOR, it is available on the github repository https://github.com/ALU-Schumacher/AUDITOR.]

Funded by SCOAP^3.

References

1. M. Fischer, E. Kuehn, M. Giffels et al., Matterminers/cobald: v0.14.0 (2023). https://doi.org/10.5281/ZENODO.1887872. https://zenodo.org/record/1887872
2. M. Giffels, M. Fischer, A. Haas et al., Matterminers/tardis: 0.8.2 (2024). https://doi.org/10.5281/ZENODO.2240605
3. A.B. Yoo, M.A. Jette, M. Grondona, *SLURM: Simple Linux Utility for Resource Management* (Springer, Berlin, 2003), pp. 44–60. https://doi.org/10.1007/10968987_3
4. Adaptive Computing Enterprises, Moab workload manager (2018). http://docs.adaptivecomputing.com/9-1-3/MWM/Moab-9.1.3.pdf. Accessed 09 July 2024
5. D. Thain, T. Tannenbaum, M. Livny, Concurr. Pract. Exp. **17**(2–4), 323 (2005)
6. M. Ellert, M. Grønager, A. Konstantinov et al., Advanced resource connector middleware for lightweight computational grids (2007). https://doi.org/10.1016/j.future.2006.05.008
7. B. Lin, B.P. Bockelman, M. Selmeci et al., htcondor/htcondor-ce: Htcondor-ce 24.1.2 (2024). https://doi.org/10.5281/ZENODO.3856680
8. M. Jiang, C.D.C. Novales, G. Mathieu et al., An APEL tool based CPU usage accounting infrastructure for large scale computing grids (2011). https://doi.org/10.1007/978-1-4419-8014-4_14
9. M. Boehler, F. von Cube, M. Fischer et al., The accounting ecosystem AUDITOR (2025). https://doi.org/10.5281/zenodo.14755407
10. N.D. Matsakis, F.S. Klock, ACM SIGAda Ada Lett. **34**(3), 103 (2014). https://doi.org/10.1145/2692956.2663188
11. G. Van Rossum, F.L. Drake, *Python 3 Reference Manual* (CreateSpace, Scotts Valley, 2009)
12. HEPiX Benchmarking Working Group, HEPSPEC06 benchmark (2024). http://w3.hepix.org/benchmarking.html. Online; Accessed 09 July 2024
13. D. Giordano, J.M. Barbet, T. Boccali et al., HEPScore: a new CPU benchmark for the WLCG (2024). https://doi.org/10.1051/epjconf/202429507024
14. C. Newman, G. Klyne, Date and time on the Internet: timestamps. RFC 3339 (2002). https://doi.org/10.17487/RFC3339. https://www.rfc-editor.org/info/rfc3339
15. B. Rabenstein, J. Volz, Prometheus: A next-generation monitoring system (Talk). (USENIX Association, 2015, 5)
16. R. Raman, M. Livny, M.H. Solomon, Matchmaking: distributed resource management for high throughput computing (1998). https://api.semanticscholar.org/CorpusID:390038. Accessed 09 July 2024
17. D. Vrcic, T. Zamani, A. Tsalapatis et al., ARGOeu/argo-ams-library: version 0.6.2 (2024). https://doi.org/10.5281/ZENODO.11504422
18. B. Wiebelt, K. Meier, M. Janczyk et al., Flexible HPC: bwForCluster NEMO (2017). https://doi.org/10.11588/HEIBOOKS.308.C3729. https://books.ub.uni-heidelberg.de/index.php/heibooks/catalog/book/308/c3729
19. S. Kroboth, M. Böhler, A.J. Gamel et al., Opportunistic extension of a local compute cluster with NEMO resources for HEP workflows (2022). https://doi.org/10.18725/OPARU-46064