Future Generation Computer Systems 164 (2025) 107541

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# JuMonC: A RESTful tool for enabling monitoring and control of simulations at scale

Christian Witzler [a], Filipe Souza Mendes Guimarães [a], Daniel Mira [b], Hartwig Anzt [c,d], Jens Henrik Göbbert [a], Wolfgang Frings [a], Mathis Bode [a,*]

[a] Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich GmbH, Wilhelm-Johnen-Straße, 52428 Jülich, Germany
[b] Barcelona Supercomputing Center (BSC), Plaça Eusebi Güell, 1-308034, Barcelona, Spain
[c] School of Computation, Information and Technology, Technical University of Munich, Munich, Germany
[d] Innovative Computing Lab (ICL), University of Tennessee, Knoxville, TN, USA

ARTICLE INFO

*Keywords:*
High-performance computing
Monitoring
Computational steering
REST-API
NekRS
ICON

ABSTRACT

As systems and simulations grow in size and complexity, it is challenging to maintain efficient use of resources and avoid failures. In this scenario, monitoring becomes even more important and mandatory. This paper describes and discusses the benefits of the advanced monitoring and control tool JuMonC, which runs under user control alongside HPC simulations and provides valuable metrics via REST-API. In addition, plugin extensibility allows JuMonC to go a step further and provide computational steering of the simulation itself. To demonstrate the benefits and usability of JuMonC for large-scale simulations, two use cases are described employing nekRS and ICON on JURECA-DC, a supercomputer located at the Jülich Supercomputing Centre (JSC). Furthermore, a large-scale use case with nekRS on JSC's flagship system JUWELS Booster is described. Finally, the interplay between JuMonC and LLview (a standard monitoring tool for HPC systems) is presented using a simple and secure JuMonC-LLview plugin, which collects performance metrics and enables their analysis in LLview. Overall, the portability and usefulness of JuMonC, together with its low performance impact, make it an important application for both current and future generations of exascale HPC systems.

## 1. Introduction

Exascale supercomputers enable completely new scientific insights by significantly expanding the parameter space accessible to simulations. However, these extremely large simulations very often also result in increasingly complex workflows that are becoming more and more difficult to manage. This is further complicated by the increasingly heterogeneous architecture (CPUs and GPUs) of supercomputers [1,2].

As part of the preparations for the first European exascale supercomputer, JUPITER, which will be based at Jülich Supercomputing Centre (JSC) from 2024, exascale-enabled workflows for complex simulations are also being developed. Such a workflow could, for example, consist of a simulation code, an in-transit visualization service, an on-the-fly machine learning (ML) framework, and a tool for simulation monitoring and control. In this context, a visualization service using SENSEI, ADIOS2, and ParaView [3,4] and the ML framework JuLES [5,6] have been established at the JSC as an application example around the nekRS/nekCRF simulation code [7,8] – a leading HPC code framework for computational fluid dynamics (CFD). This workflow is extended by the tool in this article in order to be optimally prepared for exascale workflows.

More specifically, the exascale workflow can be supported by interactive in-situ monitoring and control, allowing a human-in-the-loop to supervise simulation performance and progress. With this in mind, JuMonC (Juelich Monitoring and Control) has been developed to greatly simplify the collection and analysis of runtime information and is documented and discussed in this work.

### 1.1. Novel contribution

The main goal of JuMonC is to allow users to monitor jobs, track their progress, and identify potential problems related to code development, job setup, and job execution. This information is of great interest and relevance to both users and system operators. For example, up-to-date and easily accessible runtime information can be used to detect load imbalances. This can help avoid under- or over-utilization of available resources by identifying potential bottlenecks that each simulation

* Corresponding author.
*E-mail address:* m.bode@fz-juelich.de (M. Bode).

https://doi.org/10.1016/j.future.2024.107541
Received 31 December 2023; Received in revised form 13 September 2024; Accepted 17 September 2024
Available online 23 September 2024

may face, as well as detecting nodes that are under-performing due to hardware-related issues, so that immediate action can be taken.

In addition to monitoring, another goal is to provide an API base that allows the job controller to influence the simulation at runtime. The extent to which this control is ultimately possible depends on the application's implementation of this functionality. Examples of what could be achieved range from a simple data dump (which could be induced to be used for a later restart) or clean job termination, up to rebalancing the load of certain nodes – or even removing them from the simulation altogether.

Simulation control is a key feature that enables in-situ processing. This type of live handling of the output provides real-time access to the data for various use cases. JuMonC enables easy setup of in-situ tools and provides all the information needed to connect to a running job. An example of such a tool is Catalyst, which can be setup with an (additional) data consumer to access the VTK data in Paraview.

JuMonC is designed to run in user space, under user control. This has the advantage that it runs parallel to the application, reducing the complexity of the interaction between JuMonC and the simulations. Together with a simple extension support, this can allow custom functionality such as performing specific analyses, changing simulation parameters, or data dumps.

To perform these tasks, data must to be available to the end users. The goal here is to collect monitoring information and make it accessible in the form of a REST-API. REST stands for 'Representational State Transfer' and is generally accessible via network requests. A REST-API is characterized by stateless communication, simple interfaces, and communication using HTTP(S) features that it incorporates. Furthermore, a REST-API should be self-explanatory, pointing to other parts of the API so that any functionality can be found from a single starting point. Finally, user endpoints do not need to comply with additional protocol requirements, allowing for a more flexible choice of software to access this API.

To make the REST-API user-friendly, it is important that the functionality is logically structured and easy to use. To achieve this, JuMonC's REST-API organizes information in a hierarchical tree structure. This organization is usually appropriate for a web resource due to the use of slash-separated paths, e. g., 'network/status/bytes', which defines the hierarchy: information about the 'network', restricted to the 'status' information of the 'bytes'. The entire REST-API is structured like this in a very similar and descriptive way. In the case of the leaves of this tree, it is possible to further specify the operation by influencing the execution with parameters. For example, it is possible to provide arguments to request the data from a specific node or average the data over a specific time.

To summarize, JuMonC is unique in the sense that it is flexible, extensible and scalable, enables simulation monitoring and control, and integrates easily into complex usage workflows on supercomputers. None of these features alone is groundbreaking, but the combination makes JuMonC a novel tool for HPC systems.

### 1.2. Outline

To give a complete view of the capabilities of JuMonC, this paper is organized as follows: Section 2 describes other monitoring tools that are used in the high-performance computing (HPC) community, and highlights their differences from JuMonC. Section 3 is devoted to detailing the background of the technology used by JuMonC. A comprehensive description of JuMonC is then given in Section 4. To further demonstrate the functionality of JuMonC and to provide use cases for performance measurements, a nekRS example and an ICON application are described in Section 5. Integration with the LLview monitoring tool is also discussed. Measurements and functionality are further elaborated in Section 6. Finally, a summary and an outlook for future work is given in Section 7.

## 2. Related work

While monitoring is a common task in HPC, there is no single tool that fits all purposes. With a vast number of different applications, including many that are developed by and for the computing centers themselves, it is only to be expected that they will have different strengths and weaknesses. Some of the most well-known examples of monitoring tools are Prometheus [9], which is open source and provides many metrics collectors (or exporters) of data that is typically visualized with Grafana [10], Elasticsearch [11], and Kibana [12,13]. Examples of self-developed monitoring tools are MPCDF, developed at the Max–Planck-Institute [14,15], and the HPC Report from the Leibniz-Rechenzentrum (LRZ) [16].

At the Jülich Supercomputing Centre, the monitoring tool of choice is LLview [17], which has been developed internally for several decades [18] and provides fast and unobtrusive job reports that are generated for each job running on the supercomputers. LLview automatically collects and processes various performance metrics per minute and allows users to access a summary of the collected data in near real time a few minutes later.

Other monitoring tools for other sites such as ClusterCockpit [19], LIMITLESS [20], DiMMon [21], or Ganglia [22] are all options for scalable monitoring systems. While they have different architectures and different strengths, they have in common that they are set up by site administrators and run without direct user interaction. This allows for the collection of system monitoring information, but makes it difficult to monitor application-specific parameters.

In contrast to all of these tools, JuMonC is intended to be run by the user, i. e., under user control, with one running instance per job. The goal of JuMonC is to focus more on ad-hoc data collection to provide more flexibility to the user. There is still some support for automated data collection, but the main idea behind JuMonC is that users can query data when they need it with the preferred frequency. JuMonC aims to be independent of any other complex tool (but of course has other common libraries as dependencies), while still offering the possibility of interoperability, by providing the data through an easily accessible REST-API as JSON files that are readable by both humans and other programs. A distinct advantage of JuMonC is motivated by user control, allowing easy extension with user-specific modules via plugins. Each user can decide and develop small or large extensions for specific data and accessibility needs. In addition, JuMonC can be used across HPC sites because it is installed by the user without special permission requirements. This increases code portability and allows easier reuse of plugins, scripts, and collected data.

Finally, other approaches monitor application performance using 'heartbeats' included in the actual simulation that are collected and evaluated [23]. This is something that can also be incorporated into JuMonC through specific user-controlled plugins – taking advantage of the easy extensibility mentioned above. Compared to tools like Scalasca, JuMonC aims for a higher level of monitoring without going into the actual details of application specific reasons for the performance [24].

## 3. Background

Monitoring and computational steering operates in a broad field of tension. On the one hand, these tools must be so close to the hardware that they are able to access hardware counters and other system metrics. On the other hand, they must be able to interact with applications to be monitored and exchange information. A third layer results from the local software environment of the HPC centers and includes schedulers, accounting programs and other monitoring and

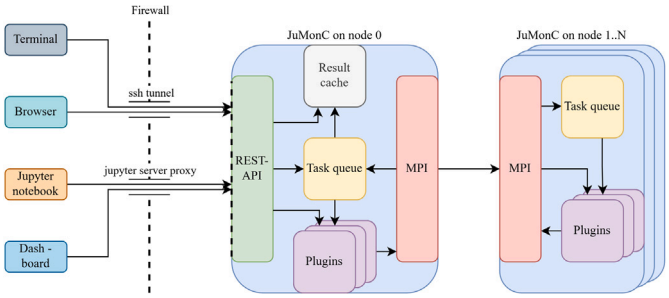


**Fig. 1.** JuMonC overview, showing user access from the left and the components that could be used, depending on the query.

analysis libraries. As described in Section 2, there are many different libraries that perform different tasks in this area of conflict. Overall, however, no uniform software stack seems to have developed on the large supercomputers in this area and many centers use tailor-made or in-house solutions. At least in the European HPC ecosystem, there are efforts to consolidate the software stacks in this area and provide users on all Tier-0 supercomputers with a standardized software environment for monitoring and control. However, this is likely to take a few more years and will come too late for the first European exascale supercomputers, for example.

In order to support the widest possible range of applications, JuMonC uses existing standards wherever possible.. The main standards used by JuMonC are described below.

REST is a commonly used communication pattern for web-based applications. It works by breaking communication into small sub-problems that are stateless. Accepted markers for a RESTful API are [25]:

- Uniform interface;
- Clear boundary between client and server;
- Stateless operations;
- RESTful resource caching;
- REST allows for multiple tiers of servers.

The Message Passing Interface (MPI) is a portable message passing standard that defines library routines. There are several implementations of this standard for different programming languages. Key concepts include point-to-point communication, collective (subset) communication, and parallel I/O [26].

JSON is a data exchange file format that uses human-readable text. It associates the data with field names in a file. Because it uses human-readable text but a clear syntax, it is easily read by humans and computers alike [27]. For an example of the JSON used by JuMonC, see the Listing 1.

```
{
  "step":20,
  "time":0.1,
  "dt":0.005,
  "CFL":0.0
}
```

Listing 1: Example JSON returned by the JuMonC-nekRS plugin.

A JSON schema is a standard for describing the contents of other files and adding additional metadata. It also allows a file to be checked for correctness, such as whether all required values are present, and whether data types and value ranges match. The schema file is written as a JSON file to take advantage of its clear syntax [28]. An example JSON Schema used by JuMonC can be seen in Listing 2.

```
{
  "$schema": "http://json-schema.org/draft
      ↪ -04/schema#",
  "type": "object",
  "properties": {
    "step": {
      "type": "integer",
      "minimum": 0,
      "description": "The␣number␣of␣steps
          ↪ ␣already␣completed"
    },
    "time": {
    .......
  },
  "required": [
    "step",
    "time",
    "dt",
    "CFL"
  ]
}
```

Listing 2: Excerpts of an example JSON schema, describing the results of a valid response. An example for a valid result for this description can be seen in Listing 1.

## 4. Software architecture

This section focuses on the software architecture of JuMonC. JuMonC is to be started as a separate process once on each node in parallel with the simulation. It communicates via its REST-API from the process on node 0, which is the central communication interface with the user. This is where user requests are received and responded to, depending on the request. An overview of JuMonC's request workflow can be seen in Fig. 1. It can be seen that there are several ways to access JuMonC, even through a firewall that is commonly used on HPC systems. By using a web-access-based API, there are many different ways that users can access the data provided by JuMonC. The internal data flows are also visible in Fig. 1, and will be discussed in more detail in the following sections.

JuMonC is open source to allow for further development and to support the HPC community. The source code is available on GitLab (https://gitlab.jsc.fz-juelich.de/coec/jumonc) and it can be easily installed using pip (https://pypi.org/project/jumonc/).

### 4.1. Task-based approach

JuMonC is task-based to allow parallel execution of different queries, which is especially important because JuMonC allows queries that collect data over long periods of time, such as the average network throughput for a given interface. In the simplest case, where the query is only to a part of JuMonC's API tree, it is answered directly by the REST-API. The next more difficult case for JuMonC is requesting previous results, which can simply be retrieved from the results cache and returned to the user.

Another case is when new results are requested. These need to be actively collected. Again, there are different levels of complexity for JuMonC, the simplest being a short query to a local plugin. If not only local information is needed, but the answer is expected to be available in a short time, the plugins can communicate with each other via MPI and then provide the result on node zero of JuMonC. JuMonC uses its own internal MPI communication, separated from the simulation use of MPI.

The most complex case from JuMonC's point of view is when a process needs to be queried over a longer period of time across multiple nodes, such as the average network traffic in the next minute. In this

case, the relevant information is stored in the task queue at rank zero. From there it is communicated via MPI and executed on all nodes in separate threads so that the MPI communication is not blocked in the meantime. The response is a link and an estimated time, so that the user does not have to wait long for an answer. Using the given link or search queries for the database, the results can be retrieved from the cache as soon as they are available.

The ability to handle user responses in two different ways was made because the processing time of requests can vary significantly. For very short requests it is convenient to get an answer directly without having to make an extra request. This could block the internal MPI communication depending on the request itself, so this is only possible for short requests. Since long requests can be arbitrarily long in some cases, since the length of a time average can be freely chosen by the user, the internal communication must be freed from this blocking and a response from the REST-API must be delivered in a reasonable time, so that the user gets feedback. The decision as to whether a job is short or long is mainly influenced by the averaging time set for the relevant queries and compared to a configurable value. A standard asynchronous REST call was not used because of the need to provide the same interface and support for many user interfaces. However, not all interfaces have tools readily available to use this, so a new method of callbacks in the form of links that can be accessed by the user to retrieve the results at a later time was added.

A more detailed view of the decision tree for this task-based approach can be seen in Fig. 2 for the node with rank 0 (root node). Rank 0 has a special role and could be a bottleneck, but for the intended use case of a background task, the amount of data is considered to be small. No problems caused by this have been seen in any testing. This includes the REST-API interface, which is always the initiator of tasks, with no requests JuMonC remaining in the background. While the flow for the root node involves many parts, it is very simple for all other nodes. As shown in Fig. 3 all other nodes have a very simple logic. All tasks arrive with the MPI broadcast and are then executed. This can either trigger a data gathering using a plugin or be a data collection process. If it is a data gathering task, it will gather the data and store it locally with an ID so that it can be found and returned in a later collection operation.

### 4.2. Plugins

To support simulations with significantly different requirements, JuMonC uses a plugin model internally, allowing user-specific functions and separate plugins to be loaded at startup, inheriting certain entry points of a class. This makes JuMonC easily extensible and allows plugins not only to communicate with simulations, but also to gather other system data that may be of interest. Thus, it is possible to collect new data by including alternative plugins, and without having to directly develop JuMonC. Furthermore, it allows to extend the tasks that a can be performed by the simulation through the REST-API, but provides a similar interface through the use of JuMonC. Using a plugin structure makes it easier for users and developers of simulation code to adapt a plugin to their specific needs.

As a starting point for specific plugins, there is a plugin for JuMonC that adds generic log parsing to the REST-API configured when JuMonC is started. This provides a simple plugin that can be easily customized for specific log files, as well as copied as a base for a plugin that provides more simulation-specific functionality.

When JuMonC is started, it allows the use of parameters that are passed to the specific plugin. Additional arguments, which can be provided to all plugins and are independent of plugin development, allow setting the base REST-API path for plugin functionality, as well as disabling plugins that are not needed. By default, plugins can request REST-API paths for their use, and only receive reassigned paths in case of a conflict; by setting a different path for a plugin, the user can prevent automatic renaming.

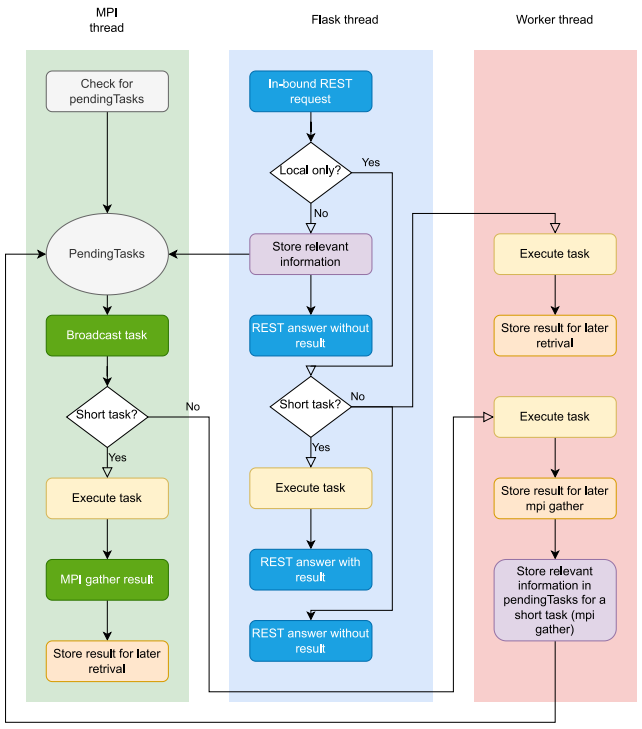


**Fig. 2.** JuMonC process flow on the root node. The REST-API logic is shown on the left, going through the task queue and being passed to the other nodes via MPI depending on the task.

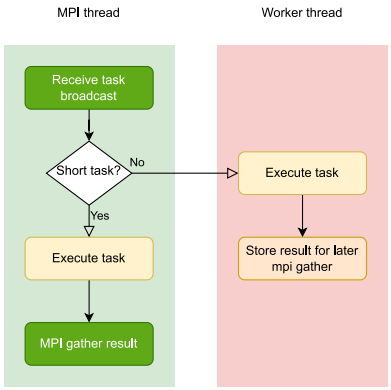


**Fig. 3.** JuMonC's other nodes process flow (root in Fig. 2), when data from different nodes is needed.

### 4.3. JuMonC API

JuMonC's API is a REST-API which has the advantage of providing a variety of access options, even through the firewalls typically deployed at HPC sites to protect compute nodes from direct Internet access. These options range from simple command line tools such as 'curl' directly on the login or compute nodes, to generic or specific dashboards accessed through a protected tunnel. Depending on the compute site, access may be simplified by using a service such as Jupyter-JSC [29], which runs on the HPC system but allows access from browsers directly over the Internet.

JuMonC 's REST-API follows the standards for REST-API's, so it is stateless and can be setup to deliver its data over either http or https. This means that each path can be accessed independently of all other paths, passing arguments and then retrieving the appropriate result. The only state that can be used is a login functionality, which eliminates the need to send the access token on each access and instead validates

against the cookie set at login. This is a convenience feature, everything can still be accessed without using the login by simply adding the token to the arguments for each REST-API call that requires authentication.

Also, all the functionality of JuMonC can be found by following the links, as JuMonC is self-describing. The REST-API is a hierarchical tree structure, where each branch contains links to other branches or leaves that contain the actual functionality. This follows a general structure where each branch has a description, possible parameters, and the path for any direct links from that branch. The JSON description of all links uses the same schema to allow automatic discovery of links and the parameters needed for those links. The monitoring functionality provided by JuMonC is organized by hardware component, each with two sub-branches, one for status and one for configuration information. Configuration refers to values that are static during a running job, such as the total amount of system memory available. Status information refers to values that are expected to change during the runtime of the job, such as the amount of system memory currently in use. All status information is cached by JuMonC and can be retrieved later using the REST-API.

JuMonC provides a REST-API that can, in principle, be accessed from anywhere. Therefore, there are security considerations to prevent accidental or malicious access by unauthorized parties. As a first level of control, JuMonC requires access to it's REST-API to be token authenticated. All communication to JuMonC 's main functionality requires a token to validate itself. To allow limited access, JuMonC knows several token levels, restricting some functionality to higher access tokens. The simplest token allows access only to links that provide no data and are therefore fast to execute, and only provides a way to explore the available functionality from the descriptions for all links, without the ability to execute any of that functionality. The next level of access then allows you to retrieve information from the cache, i. e., the ability to see the results of other users' queries, without the ability to trigger any data collection. In total, there are five access levels by default, but for even finer granularity, plugins can also introduce their own levels that fit in between.

The security concept allows granular sharing of access to the data and functionality provided by JuMonC, so that researchers can collaborate without giving away full access. By default, a token for each level is generated at startup and can be found on JuMonC's stdout. Additional tokens can be set at startup or using JuMonC's initialization file.

As an additional security measure, JuMonC supports the use of SSL for use cases where the network between the communication partners is not trusted. SSL support is implemented in two ways, either using an ad-hoc certificate generated by JuMonC at startup that must be accepted by the user, or using a supplied certificate that can be used by JuMonC.

In addition, JuMonC can be set to only accept network connections on a specific network interface, with the most restrictive version limiting it to localhost access, i. e., only accepting access from the same host running JuMonC.

For better automated parsing of REST-API results, JuMonC includes a unified command to request a JSON schema describing the usual data fields present. While this is an optional feature, and therefore may not be present for all plugins, when present it facilitates data usage by automated programs by adding context and error checking capabilities to the normal data responses. This allows automatic checks for completeness, validity range, and useful descriptions for automatically generated plots.

### 4.4. Deployment

Since the goal is for JuMonC to be run by the user, it is important that it is easy to deploy. Therefore, it is provided as a pip package, which makes it easy to install and ensures that all dependencies are installed. Since JuMonC also has optional dependencies (cf. Section 4.5) to enable some of the features, pips' mechanism for extras can be used to automatically install all dependencies for the required features.

**Table 1**
Mandatory dependencies needed to run JuMonC.

| Dependency | Functionality |
|---|---|
| At least python3.6 | Runs JuMonC |
| mpi4py [30,31] | Node-to-node communication |
| Flask [32,33] | Manages REST-API |
| Flask-Login [34] | Allows cookie based login |
| Flask-SQLAlchemy [35] | Enables a SQL based cache |
| Pluggy [36] | Manages plugin extensions |
| Typing-extensions [37] | Only needed in python $\leq$ 3.9 to allow newer python functionality |

**Table 2**
Optional dependencies for additional functionality in JuMonC.

| Dependency | Functionality |
|---|---|
| pynvml [38] | Monitoring for NVIDA GPUs |
| psutil [39] | Monitoring for I/O, main memory and CPU |
| pyOpenSSL [40] | SSL encrypted REST-API access |

JuMonC allows extensions in the form of plugins, for user-specific functionality that can be targeted to a specific simulation. These plugins can be provided in two different ways, either as a Python file to be loaded by JuMonC at startup, or by setting a correct entry point so that JuMonC is able to find the plugin automatically at startup. Setting a correct entry point allows plugins to be installed using pip and then made available via JuMonC's REST-API without any further configuration.

For data persistence, JuMonC comes with an internal database that logs the information it retrieves. The information of cached results can be accessed through the REST-API. It is also supported to start JuMonC with an old database to access and compare with older simulation runs. The database is also available to plugins so that these results can be stored along with all other information. For the basic functionality of JuMonC, only dynamically changing data is added to the database, static information like the maximum memory is not added.

### 4.5. Dependencies

JuMonC depends on other common libraries. Some dependencies are mandatory and others allow additional functionality. These dependencies are handled by pip dependency management, making this a simple process for users, only mpi4py needs a system, where an MPI installation is present. The mandatory and optional dependencies are summarized in Tables 1 and 2.

### 4.6. Automatic testing

To increase user confidence in the ongoing development of JuMonC, automated testing is included. These tests start with static analysis to ensure coding standards and correct variable typing, and look for security vulnerabilities. Further tests include unit tests and finally JuMonC is executed and responses for all REST-API paths are tested. Finally, JuMonC is bundled and tested for different Python versions to ensure compatibility with older Python versions.

### 4.7. Latency

JuMonC should run in the background with minimal resource usage, especially when not needed. Since some MPI implementations, especially when used with mpi4py, continuously check for new messages, causing additional CPU load, JuMonC uses non-blocking MPI communication. This allows fine-grained control over the behavior between

checks. In the case of JuMonC, this means inserting a sleep between checks to free up processor resources. Since this also increases latency for JuMonC's REST-API responses that require data from other nodes, the default sleep timer of one ms can be configured by users to suit their needs. The average increase in latency $\overline{\Delta t_L}$ depends on the sleep time $t_s$ and the execution time of JuMonC's functionality $t_e$. To allow each task to run independently of other concurrent tasks, there are two command broadcasts for each functionality. The first one to trigger the execution and the second one to trigger the reporting of the results. Assuming a random access time for the REST-API, waiting on average half the wait time is expected for the first access. Depending on the configured sleep time and the access duration, there are two extreme cases for the latency increase. Either $t_e$ is much larger than $t_s$, where the average latency increase is given by

$$t_e \gg t_s : \overline{\Delta t_L} = 2 \cdot \frac{1}{2} \cdot t_e = t_e \tag{1}$$

or vice versa with the average latency increase as

$$t_s \gg t_e : \overline{\Delta t_L} = t_e + \frac{1}{2} \cdot t_e = \frac{3}{2} \cdot t_e. \tag{2}$$

### 4.8. Data granularity

JuMonC allows data collection on demand, and there are no specific limits on the data collection rates enforced by JuMonC. Therefore, multiple data points per second are possible, but increase the possible performance impact. No limit is enforced. However, there is a limit based on the internal blocking of MPI communication. This limitation depends in part on the size of the job allocation, but even for large allocations, 10 requests per second are possible.

## 5. Applications

To further evaluate JuMonC, use cases are needed. This section describes two use cases, a CFD application computed with nekRS and an Earth experiment simulated with ICON. The coupling of JuMonC with LLview is also discussed. nekRS, ICON, and LLview will be important contributors to beyond the state-of-the-art applications planned to be run on the first European exascale supercomputer JUPITER, which is planned for 2025 and will be hosted at JSC. Both nekRS and ICON are part of the JuBench benchmark suite [41] that was used for the JUPITER procurement.

### 5.1. JuMonC integration into use cases

This section briefly describes the two use cases used to measure performance and demonstrate JuMonC.

#### 5.1.1. Mesoscale convection with nekRS

CFD is a very common application for supercomputers with examples ranging from atmospheric boundary layers and geological flows to energy devices. A particular example is to better understand natural convection in turbulent conditions, characterized by a high Reynolds number (Re), e. g., corresponding to a low Prandtl number (Pr) and a high Rayleigh number (Ra) [42–44]. Due to the resulting scales and driving mechanisms, one resulting regime is called mesoscale convection. High Re simulations result in very large simulations due to the large scale separation, making it a good example case for JuMonC and exascale workflows in general. The code used here is nekRS that features an optimized GPU backend, as well as a CPU backend.

nekRS is a high-order spectral element solver that uses the same input and output mechanism as nek5000. The difference is that nekRS is a GPU-enabled code, using occa [45] as a layer to port to different architectures. It allows customization through the use of user-defined functions that can also be executed as GPU kernels on the GPU [7].

The physical conditions are set in the simulation using the dimensionless numbers Pr and Ra, where Pr is the ratio of momentum diffusivity to thermal diffusivity [46]. Ra is by definition the ratio of the timescales for diffusive heat transport to convective heat transport at a given velocity [47].

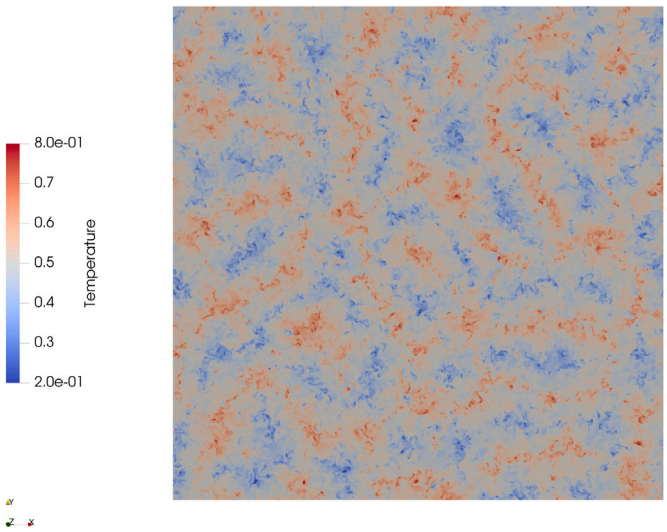


**Fig. 4.** A slice at half height for natural convection at Ra = $10^7$ and Pr = 0.7, showing the normalized temperature distribution with visible superstructures.

The Re starts with low values because it depends on the velocity, which is initialized to zero. Only the temperature is initialized as a gradient from top to bottom. Periodic boundary conditions (BC) are used in the width and length directions. The height BCs use no-slip conditions for the velocity and Dirichlet conditions for the temperature. Fig. 4 shows the temperature field for a slice at half height of the simulated volume after thermalization. Superstructures for the convective heat plumes are clearly visible.

To integrate data from nekRS into JuMonC, a separate plugin [48] has been developed that can be easily installed using pip, and is then automatically used by JuMonC. This plugin is able to access data from nekRS's logfile, for static information like the nekRS version used. Dynamic information that can then be used via JuMonC's REST-API includes simulation time, time step length, time steps, and Courant number (CFL number). The steering component depends on code integration with nekRS and, e. g., allows to trigger additional writes of the simulation data and enables additional statistics. Another usage example is to change the in-situ data frequency (cf. Section 6).

#### 5.1.2. Earth experiments with ICON

As an additional simulation code, ICON (Icosahedral Nonhydrostatic) [49,50] was used. This is a weather/climate modeling code that uses an icosahedral grid and numerically calculates separate modeling levels for different time and length scales. The example simulation starts with a measured world wide weather data and goes forward in the time from here. This allows detailed results to be viewed for many physical quantities. One example is the surface air pressure, which can be seen in Fig. 5 and, due to pressure differences due to altitude, allows an easy geographical orientation due to the visible continents.

### 5.2. Using JuMonC with LLview

LLview is the monitor tool of choice at JSC. It provides users with easy access to important hardware information, such as memory or GPU usage, and generates detailed job reports after each job has been completed. Coupling JuMonC and LLview is therefore a benefit for both tools and is described in more detail below.

To make the metrics collected by JuMonC available automatically, the existing LLview infrastructure is used to add new graphs to the

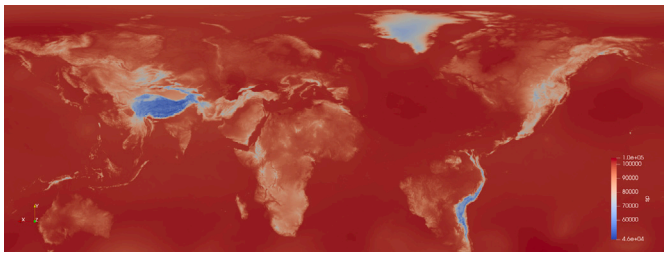

**Fig. 5.** ICON's simulated surface air pressure.

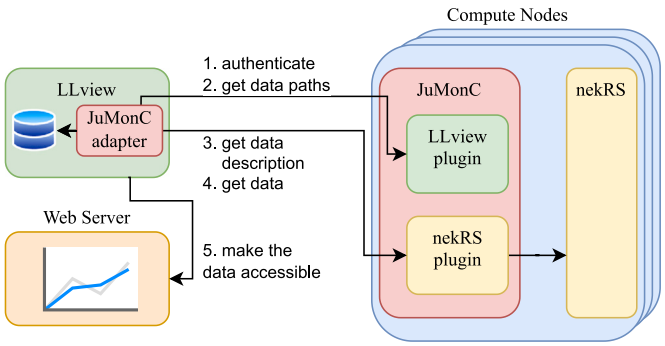


**Fig. 6.** Interplay of LLview and JuMonC. Data is requested from the JuMonC adapter (component of LLview) to the plugins, processed, and added to the job reports.

reports of jobs using JuMonC. This is achieved by adding new modules to each of the programs: a plugin for JuMonC, where the authentication and default settings are set, and an adapter for LLview, which is used to query and parse the data.

Simply put, LLview and JuMonC complement each other, because LLview lacks the ability to access user/simulation specific data to add to its plots, while JuMonC has no plotting utilities included, and LLview offers such a choice, while being able to add data locked behind admin access, which JuMonC lacks.

#### 5.2.1. LLview plugin for JuMonC

To allow a better and more consistent interaction between JuMonC and LLview, an additional plugin [51] has been created. It is mainly designed to be a single entry point where LLview can connect to the variable configurations of different JuMonC plugins. The plugin is able to automatically gather information about which REST-API paths provide information that can be used by LLview. Although the JuMonC-LLview interaction is automatically configured, the user can customize it via a configuration file or job startup parameters.

An important feature of the plugin is to provide secure access to the metrics for LLview only. Because JuMonC runs as a user process and accesses the user data (e. g., to retrieve information from logfiles), there is no direct way to give LLview– and LLview only – a unique access token. To obtain the token, LLview uses secure public/private key authentication with JuMonC. The procedure is as follows:

1. LLview requests a random message and the source node from JuMonC's REST-API;
2. To avoid a 'man-in-the-middle' type of attack, the node is confirmed to be the one, where the job is running (from SLURM output);
3. LLview signs the message (including the node name) with its private key and sends it back to JuMonC;
4. JuMonC verifies that the message was signed by LLview using the public key;
5. JuMonC returns a token for further access.

The current implementation uses the Edwards-curve Digital Signature Algorithm (EdDSA) [52] for keys and also allows LLview to be granted a limited token.

After receiving the token, LLview is able to use JuMonC's REST-API to query all paths that are accessible with that token and proceed to add specific simulation data to the job report.

#### 5.2.2. JuMonC adapter for LLview

The new JuMonC adapter has been added to the LLview workflow, which collects data by running every minute. The first step is to get the job ID of all jobs running on the monitored system, which is done using the data from the base Slurm adapter. The current timestamp is also stored. Slurm is also currently used to provide LLview with additional information about the current JuMonC instance: The adapter checks whether JuMonC is running or has a custom configuration set in the `--comment` field. If JuMonC is active for a given job, the following information is stored: the node on which the job is running, as well as any custom port and path.

Using the information collected for all jobs running JuMonC, LLview proceeds to query the available metrics as shown in Fig. 6. To be able to perform further queries, the authentication process to request a valid token (as described in Section 5.2.1) is performed first. In possession of the token, LLview asks to loop through the available paths or use a custom path. Each of the available paths is contacted to collect all metrics and their provided properties (such as name, description, type, minimum or maximum, etc.) using the description JSON schema if available.

The last step of the JuMonC adapter is to create an internal file that is further processed and stored in the database. To avoid a constantly growing database, the metrics are stored with generic names associated with their properties, and a maximum number of metrics can be set in the configuration of LLview (set to 10 by default).

#### 5.2.3. Presentation

After the metrics are collected and added to the database, LLview can generate new graphs that are included in the job reports generated internally for all jobs running on the system. The reports are provided in PDF and HTML formats, the latter being the preferred format because it uses Plotly [53] for interactive graphs. In this case, the users can select which of the collected metrics are plotted on each of the axes (cf. Section 6).

#### 5.2.4. Perspectives

JuMonC proved to be an extremely useful tool for getting data from the user space into the LLview database, which helps in the development of future LLview extensions. For example, other plugins could provide detailed metrics from instrumented jobs that are not available in the standard LLview reports. These could then be extended to provide all the collected information in a single place. Another possible use is to define 'calibration' jobs, where predefined metrics are set and then compared to those obtained by the monitoring tools.

With LLview as a standard tool for monitoring and job reporting at JSC and the customizability of JuMonC, important foundations have been laid to support a wide range of applications on the future exascale supercomputer JUPITER.

## 6. Results and discussion

This section presents and discusses the qualitative and quantitative results of JuMonC.

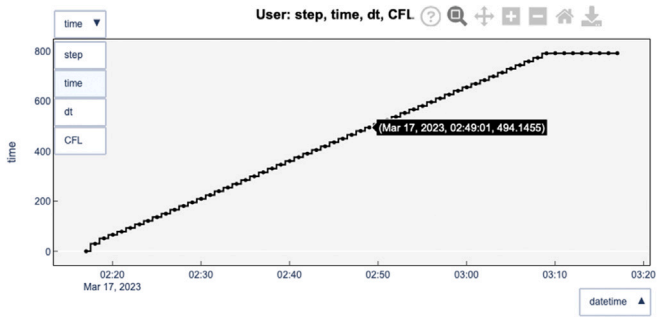


**Fig. 7.** Graph on the job report of LLview with metrics obtained from nekRS via JuMonC. The user can select the desired metrics on both axes via drop-down boxes.

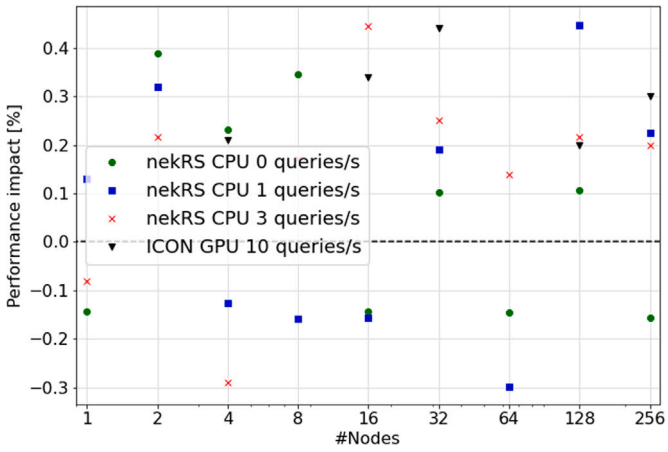


**Fig. 8.** Performance impact by JuMonC on nekRS running on the CPU backend and using all CPU cores and ICON running on GPUs. Performance is compared as the average of 3 runs, with and without JuMonC running. Using different workloads for JuMonC, with zero requests, or one and three requests per second or ten requests per second for ICON.

### 6.1. Workflows

JuMonC with its lightweight, flexible, secure, and user-friendly design was developed to support exascale-enabled workflows and data usage at scale. It enables interactive data collection, system usage information, and other simulation-specific data. With JuMonC, it is also possible to use plugins to extend simulations with very specific code that allows the use of JuMonC's control capabilities. This allows a user not only to monitor his simulations for problems, but also to provide and use specific problem solutions.

The data from JuMonC can easily be used in several cases, one of which is LLview a job reporting tool running on Jülich Supercomputing Centre. This closes the circle from a new on-demand approach to monitoring simulations and systems from the user space, with access to simulation-specific data, to a data source for established job monitoring. For LLview, this is a way to access simulation-specific data that is transparent to the user and easy to configure.

With simulation-specific data available in LLview, it is possible to identify potential problems with a running nekRS simulation. One such problem can be seen in Fig. 7, which shows the time history of the nekRS simulation, plotted against the system clock time. In this case, a numerical artifact caused nekRS to keep reducing its time step length, there is no longer a visible change in the progress. Using only standard system metrics, this problem will not be visible, because nekRS will still work correctly and perform calculations that show a normal system load. If the user notices this, the simulation can be aborted or other actions can be taken, such as triggering a full data dump (using JuMonC) for more in-depth analysis without wasting more system resources before nekRS slows down even more and aborts itself.

An important design constraint for JuMonC was the goal to have minimal impact on the actual simulation, so that the monitoring does not affect the simulation performance. Of course, this can only be true for the actual monitoring, if it is used for control purposes, it depends on the triggered functionality if there will be an impact, e. g., a complete data dump will take some time.

### 6.2. Measurements

The performance and usability of JuMonC is approached quantitatively next. Two different application codes are considered. Furthermore, measurements on CPU and GPU backends are shown. To avoid unnecessary length, measurements are presented for nekRS with its CPU backend and ICON running on GPUs. The measurements were performed on JURECA-DC [54]. In addition, the next section demonstrates JuMonC's steering for an even larger nekRS setup computed on up to 3360 GPUs on JUWELS Booster [55] in the next section.

The nekRS tests on JURECA-DC used a weakly scaling workload and all available CPU cores, overlapping with the CPU cores used for JuMonC. The performance impact of solving the test case on the same node allocations with and without JuMonC was compared. Because resources are shared with other users, such as the network, there is some variation between runs. To reduce the impact of this variation, three runs were used for each configuration. This test uses data collected from all nodes, so JuMonC must use its internal communication to all nodes for each of these queries. To better understand the impact, JuMonC was used with different query loads, just running in the background with no active usage, one query per second, or three queries per second. The overall results show a very small impact with no visible scaling behavior for the impact, as can be seen in Fig. 8. The overall average for all sizes shows an impact of less than 0.1% in the no query case, and using one query per second and three queries per second results in an impact of 0.14%.

As an additional test, the performance impact of JuMonC on ICON running on the GPUs was tested. Since in this case CPU resources are unused and available for JuMonC, the combination of ICON with JuMonC was tested with ten queries per second and the results were added to Fig. 8. Even with ten queries with data gathering on all nodes of the job, the average of the impact on total execution time for all ICON cases remains below 0.3%, without showing any negative scaling behavior. Another important measure is the memory footprint. It is plotted for the ICON application in Fig. 9 as a function of the number of nodes used. The memory footprint increases with more nodes, due to the larger amount of collected data that scales with the increasing number of MPI processes. Overall, the total memory usage of JuMonC is not critical.

For the exascale usability of JuMonC, the scaling behavior of the query latency is also important. Since this is influenced by the waiting time between MPI calls to JuMonC, these tests were done with the default value of 5 ms. This value can be changed according to the specific use case to achieve the required performance. The total latency can be seen in Fig. 10 and scales roughly with 5 ms per each doubling of the node count, that is needed due to the waiting time. So this is an important value to tune the performance according to your simulation and needs. To increase the responsiveness of JuMonC's API, this value can be changed during the runtime of JuMonC through the REST-API, to temporarily allow a larger impact on simulation performance in exchange for lower latency.

### 6.3. Steering

A natural convection case with $\mathrm{Ra} = 10^{12}$ and $\mathrm{Pr} = 0.7$ is used to demonstrate a steering use case with JuMonC. Due to the resulting

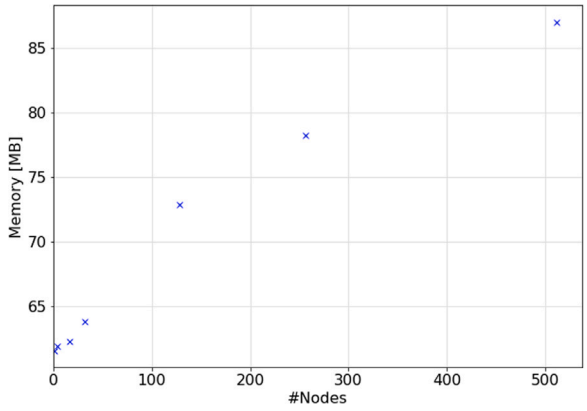


**Fig. 9.** Memory need of JuMonC measured for different scales, using ICON as simulation code to run parallel to JuMonC. The measurement is the maximum resident size on any one node, as reported by 'ps', showing how much memory is allocated for JuMonC.

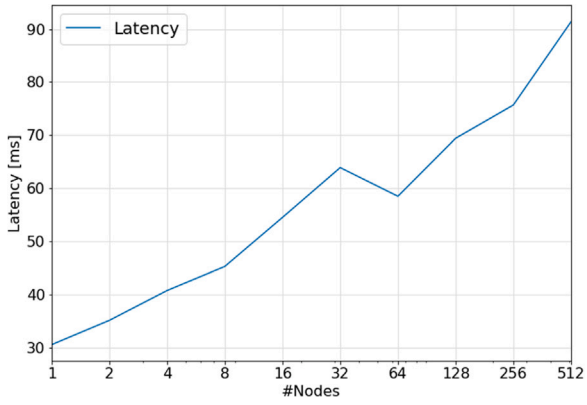


**Fig. 10.** Latency measured for requests from JuMonC that require MPI communication for different scales, using nekRS as simulation code to run parallel to JuMonC.

high Re, this case was run on 3360 GPUs on JUWELS Booster. Since JUWELS Booster has a total of 3744 GPUs, this is large enough to be considered a 'full system' run, considering that usually a fraction of GPUs are not available to users for various reasons, such as hardware problems. A complex in-situ/in-transit visualization workflow was applied to allow complex visualization of the enormous amount of data in parallel with the simulation run. One technical difficulty was to find the optimal operating point, which was defined by the number of GPUs for the simulation, the available GPUs for the visualization, and an optimal visualization interval. Due to the size of the simulation and the limited availability of supercomputers for simulations of such a size, JuMonC was used to monitor the status, in particular the average GPU utilization. Additionally, the steering function was used to allocate GPUs for either simulation or visualization to achieve optimal performance. This is complicated by the fact that the optimum is not static. Turbulence is only statically stationary, so the numerical complexity can also vary between individual time steps. For example with increasing simulation runtime, the solution time per time step can increase, so that a previously determined optimal operating point is no longer optimal in relation to the visualization interval.

The use case was simulated twice for a runtime of 6 h with an identical starting point. Checkpointing was performed at equal intervals for both cases after two dimensionless time units. Overall, it was possible to increase the average GPU utilization from 76.2% to 80.5% by monitoring and intervening in the simulation with JuMonC in parallel. This already includes the overhead caused by JuMonC, which was 0.2%.

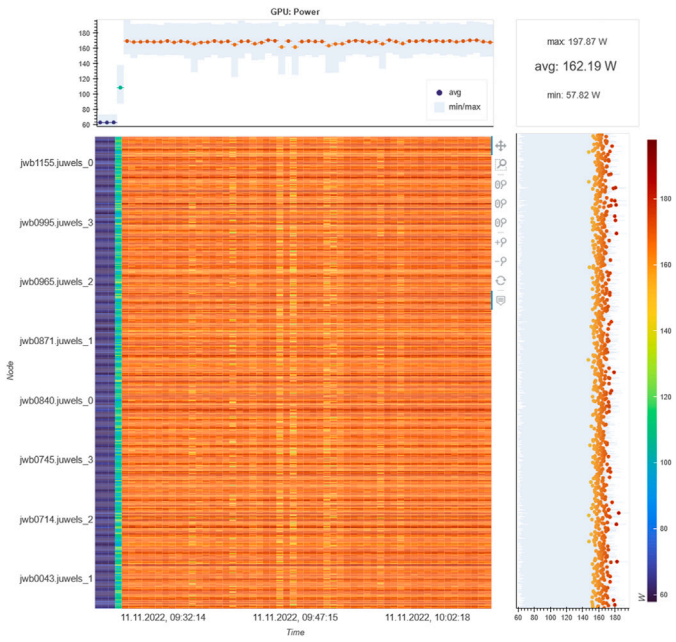


**Fig. 11.** Example visualization of data collected with JuMonC, here power draw of 800 GPUs over time. Data is gathered using JuMonC's scheduled tasks and then retrieved from the cache and visualized inside a Jupyter notebook [56] using Bokeh [57].

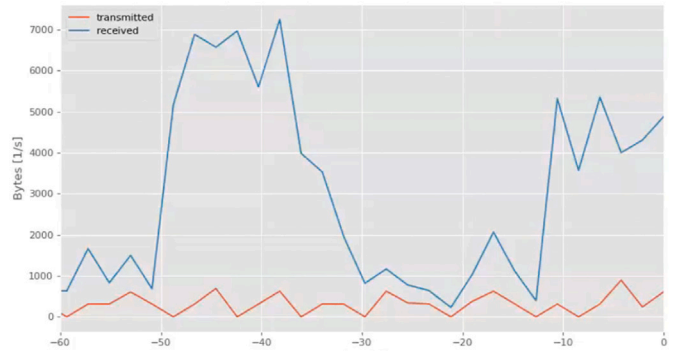


**Fig. 12.** Example visualization of data collected with JuMonC, here the total network activity over time. Data is gathered in a Juypter notebook using requests and plotted using matplotlib [58].

### 6.4. Plugins

JuMonC can be used as a data source for several different monitoring tasks. Two examples of use with dashboards are shown in Figs. 11 and 12. These are based on Jupyter notebooks that can be run as either a notebook or a dashboard, and are available as example in the JuMonC repository. The first has several components and is very similar to LLview job reports, with the main difference that JuMonC only provides the data on demand, but therefore with a potentially much higher refresh rate. The main area is used to showcase a color-coded view of GPU power consumption, in this case, showing all nodes for multiple timesteps. On the top is a plot of the node average over time, while on the right side is a plot of the time average over the nodes. The other example uses JuMonC to periodically update the total network traffic report for all nodes.

While JuMonC's approach of running in userspace under user control has many advantages for code portability and user extensibility, it also has a drawback. JuMonC is limited to the data available to the user. Therefore, if HPC sites limit the performance data available to users, JuMonC will not be able to access that data, while a more integrated,

site-specific tool might get exceptions. If the data is available, but in a non-standard way, this reduces the portability of JuMonC, but this can be reduced by using custom site-specific JuMonC plugins.

The usability of plugins is one of the major advantages of JuMonC, which is extensible with custom simulation and user-specific functionality. These can use the basic functionality provided for integration into the REST API, communication and plug into the configuration mechanism of JuMonC. To allow easy customization, development and sharing of plugins, JuMonC is able to automatically detect correctly configured plugins and use the path to the plugin as part of its configuration.

## 7. Conclusions and outlook

JuMonC is a tool for monitoring and steering of HPC simulations at scale. This paper demonstrates the use of JuMonC with the simulation codes nekRS and ICON, and how it interoperates with other monitoring tools such as LLview. Monitoring and influencing code evaluation through steering allows for more flexible and targeted use of HPC resources, preventing wasted computing time and potentially leading to better scientific results. This functionality is provided by JuMonC through a REST-API that allows flexible user access through a variety of different tools, with minimal performance impact on the simulation, further reduced by relying mostly on an on-demand approach. Performance and memory measurements show that JuMonC is suitable to be employed in large use cases and even towards exascale while providing a wide range of functionality.

An important point for future work is to extend and diversify the plugins, for example to increase the monitoring capabilities. Possible plugin extensions might include collecting more CPU hardware counters using libraries like PAPI [59,60] and LIKWID [61,62], both of which are accessible via Python bindings (python_papi [63] and pylikwid [64]). And while there is a plethora of data available for NVIDIA GPUs through the current implementation of GPU monitoring based on NVML [65], there is currently no support for monitoring non-NVIDIA GPUs, such as AMD and Intel.

Additional plugin support for other simulation codes is also needed, both to broaden the potential user base and to show what is possible with JuMonC. These plugins would also serve as a good starting point for users who want to develop their own plugins. In this sense, JuMonC could also be extended to provide broader support for data collection options, not just node-specific data or data from all nodes, but ready-to-use functionality through plugins for averaging, median, or obtaining the critical points and extrema.

Even for the particular example discussed in this paper, which combines JuMonC with nekRS/ICON and LLview, other improvements are also possible. For example, in the case of the nekRS plugin, these could include improvements to make the nekRS performance information available in the plugin, and thus in JuMonC's REST-API. Another possible improvement would be more advanced data analysis, but this might lead to a simulation and case-specific plugin. For the JuMonC-LLview interaction, one possible improvement would be an automatic way to inform LLview of a running JuMonC instance, which is done automatically without the need for the user to specify that it is running in a SLURM comment field.

In summary, although JuMonC is a newborn in the monitoring tools field, the existing examples already demonstrate some of its current and future uses. It has been designed to provide not only monitoring capabilities directly to the user, but also control knobs to steer his simulations. Because it is easy to install and extend, it has the potential to become a widely used tool in the HPC community, aided by its open source nature.

## CRediT authorship contribution statement

**Christian Witzler:** Writing – original draft, Software, Conceptualization. **Filipe Souza Mendes Guimarães:** Writing – review & editing, Software. **Daniel Mira:** Investigation. **Hartwig Anzt:** Investigation. **Jens Henrik Göbbert:** Investigation, Conceptualization. **Wolfgang Frings:** Software. **Mathis Bode:** Writing – review & editing, Supervision, Software, Investigation, Conceptualization.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

JuMonC is open source and sources are available.

## Acknowledgments

The authors gratefully acknowledge the computing time granted through JARA on the supercomputer JURECA and the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer JUWELS. This work was supported by the European Union's Horizon 2020 research and innovation program under grant agreements No. 952181 (Center of Excellence in Combustion), No. 955606 (DEEP-SEA), and No. 955811 (IO-SEA). Support by the Joint Laboratory for Extreme Scale Computing (JLESC, https://jlesc.github.io/) for traveling is acknowledged.

## References

[1] J. Dongarra, A. Geist, Report on the Oak Ridge National Laboratory's Frontier System, ICL Technical Report ICL-UT-22-05, 2022.
[2] S. Habib, R. Roser, R. Gerber, K. Antypas, K. Riley, T. Williams, J. Wells, T. Straatsma, A. Almgren, J. Amundson, S. Bailey, D. Bard, K. Bloom, B. Bockelman, A. Borgland, J. Borrill, R. Boughezal, R. Brower, B. Cowan, H. Finkel, N. Frontiere, S. Fuess, L. Ge, N. Gnedin, S. Gottlieb, O. Gutsche, T. Han, K. Heitmann, S. Hoeche, K. Ko, O. Kononenko, T. LeCompte, Z. Li, Z. Lukic, W. Mori, P. Nugent, C.-K. Ng, G. Oleynik, B. O'Shea, N. Padmanabhan, D. Petravick, F.J. Petriello, J. Power, J. Qiang, L. Reina, T.J. Rizzo, R. Ryne, M. Schram, P. Spentzouris, D. Toussaint, J.L. Vay, B. Viren, F. Wurthwein, L. Xiao, ASCR/HEP exascale requirements review report, 2016.
[3] V.A. Mateevitsi, M. Bode, N. Ferrier, P. Fischer, J.H. Göbbert, J.A. Insley, Y.-H. Lan, M. Min, M.E. Papka, S. Patel, S. Rizzi, J. Windgassen, Scaling computational fluid dynamics: In situ visualization of NekRS using SENSEI, in: Proceedings of the Supercomputing Conference Workshops (ISAV), 2023.
[4] M. Bode, et al., Deciphering boundary layer effects in high-Rayleigh-number convection using 3360 GPUs and a high-scaling in-situ workflow, arXiv (2024).
[5] M. Bode, AI super-resolution: Application to turbulence and combustion, in: N. Swaminathan, A. Parente (Eds.), Machine Learning and its Application To Reacting Flows, Lecture Notes in Energy 44, Springer, 2023.
[6] M. Bode, AI super-resolution subfilter modeling for multi-physics flows, in: Platform for Advanced Scientific Computing Conference (PASC '23), 2023.
[7] P. Fischer, S. Kerkemeier, M. Min, Y.-H. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, T. Warburton, NekRS, a GPU-accelerated spectral element Navier-Stokes solver, 2021, arXiv:2104.05829.
[8] S. Kerkemeier, C. Frouzakis, A. Tomboulides, P. Fischer, M. Bode, nekCRF: A GPU accelerated high-order reactive flow solver for direct numerical simulations, arXiv (2024).
[9] Prometheus, 2024, https://prometheus.io/.
[10] Grafana, 2024, https://grafana.com/.
[11] Elasticsearch, 2024, https://www.elastic.co/.
[12] Kibana, 2024, https://www.elastic.co/kibana/.
[13] M. Ott, W. Shin, N. Bourassa, T. Wilde, S. Ceballos, M. Romanus, N. Bates, Global experiences with HPC operational data measurement, collection and analysis, in: 2020 IEEE International Conference on Cluster Computing, CLUSTER, 2020, pp. 499–508.
[14] MPCDF, 2024, https://docs.mpcdf.mpg.de/doc/computing/performance-monitoring.html.
[15] L. Stanisic, K. Reuter, MPCDF HPC performance monitoring system: Enabling insight via job-specific analysis, 2019.

[16] HPC report, 2024, https://doku.lrz.de/display/PUBLIC/HPC+Report.
[17] LLview, 2024, http://llview.fz-juelich.de.
[18] W. Frings, M. Riedel, A. Streit, D. Mallmann, S. v.d.Berge, D. Snelling, V. Li, LLview: User-level monitoring in computational grids and e-science infrastructures, in: Proceedings of German E-Science Conference. - Baden-Baden, 2007. - Max Planck Digital Library. - ID 316542.0, in: Proceedings of German e-Science Conference, Baden-Baden, 2007.
[19] J. Eitzinger, T. Gruber, A. Afzal, T. Zeiser, G. Wellein, ClusterCockpit — A web application for job-specific performance monitoring, 2019, pp. 1–7.
[20] A. Cascajo, D. Singh, J. Carretero, LIMITLESS — Light-weight monitoring tool for large scale systems, Microprocess. Microsyst. 93 (2022) 104586.
[21] K. Stefanov, V. Voevodin, S. Zhumatiy, V. Voevodin, Dynamically reconfigurable distributed modular monitoring system for supercomputers (DiMMon), Procedia Comput. Sci. (ISSN: 1877-0509) 66 (2015) 625–634, 4th International Young Scientist Conference on Computational Science.
[22] M.L. Massie, B.N. Chun, D.E. Culler, The ganglia distributed monitoring system: design, implementation, and experience, Parallel Comput. (ISSN: 0167-8191) 30 (7) (2004) 817–840.
[23] S. Ramesh, S. Perarnau, S. Bhalachandra, A.D. Malony, P. Beckman, Understanding the impact of dynamic power capping on application progress, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2019, pp. 793–804.
[24] M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, B. Mohr, The scalasca performance toolset architecture, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 702–719.
[25] H. Subramanian, P. Raj, Hands-on RESTful Web API Design Patterns and Best Practices: Design, Develop, and Deploy Highly Adaptable, Scalable, and Secure RESTful web APIs / Harihara Subramanian, Pethuru Raj, Packt Publishing, Birmingham, UK, ISBN: 9781788992664, 2019.
[26] Message Passing Interface Forum, MPI: A message-passing interface standard version 4.0, 2021, URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.
[27] D. Crockford, C. Morningstar, Standard ECMA-404 the JSON data interchange syntax, 2017.
[28] F. Pezoa, J.L. Reutter, F. Suarez, M. Ugarte, D. Vrgoč, Foundations of JSON schema, in: Proceedings of the 25th International Conference on World Wide Web, WWW '16, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, ISBN: 9781450341431, 2016, pp. 263–273.
[29] Jupyter-JSC, 2024, https://jupyter-jsc.fz-juelich.de/.
[30] Mpi4py, 2024, https://pypi.org/project/mpi4py/.
[31] L. Dalcin, Y.-L.L. Fang, Mpi4py: Status update after 12 years of development, Comput. Sci. Eng. 23 (4) (2021) 47–54.
[32] Flask, 2024, https://pypi.org/project/Flask/.
[33] M. Grinberg, Flask Web Development: Developing Web Applications with Python, O'Reilly Media, Inc, 2018.
[34] Flask-Login, 2024, https://pypi.org/project/Flask-Login/.
[35] Flask-SQLAlchemy, 2024, https://pypi.org/project/Flask-SQLAlchemy/.
[36] Pluggy, 2024, https://pypi.org/project/pluggy/.
[37] Typing-extensions, 2024, https://pypi.org/project/typing-extensions/.
[38] Pynvml, 2024, https://pypi.org/project/pynvml/.
[39] Psutil, 2024, https://pypi.org/project/psutil/.
[40] pyOpenSSL, 2024, https://pypi.org/project/pyOpenSSL/.
[41] A. Herten, S. Achilles, D. Alvarez, J. Badwaik, E. Behle, M. Bode, T. Breuer, D. Caviedes-Voullième, M. Cherti, A. Dabah, S. El Sayed, W. Frings, A. Gonzalez-Nicolas, E.B. Gregory, K. Haghighi Mood, T. Hater, J. Jitsev, C. John, J.H. Meinke, C.I. Meyer, P. Mezentsev, J.-O. Mirus, S. Nassyr, C. Penke, M. Römmer, U. Sinha, B. von St. Vieth, O. Stein, E. Suarez, D. Willsch, I. Zhukov, Application-driven exascale: The JUPITER benchmark suite, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '24, Association for Computing Machinery, New York, NY, USA, 2024, accepted for publication, https://arxiv.org/abs/2408.17211.
[42] A. Brandenburg, K. Subramanian, Astrophysical magnetic fields and nonlinear dynamo theory, Phys. Rep. 417 (1–4) (2005) 1–209.
[43] A. Pandey, J. Schumacher, K.R. Sreenivasan, Non-Boussinesq convection at low Prandtl numbers relevant to the Sun, Phys. Rev. Fluids (ISSN: 2469-990X) 6 (10) (2021).
[44] R.J. Samuel, M. Bode, J.D. Scheel, K.R. Sreenivasan, J. Schumacher, No sustained mean velocity in the boundary region of plane thermal convection, J. Fluid Mech. 996 (2024) A49, http://dx.doi.org/10.1017/jfm.2024.853.
[45] D.S. Medina, A. St-Cyr, T. Warburton, OCCA: A unified approach to multi-threading languages, 2014, arXiv:1403.0968.
[46] Y.A. Çengel, Heat Transfer: A Practical Approach, internat. ed., WCB/McGraw-Hill Series in Mechanical Engineering, McGraw-Hill, Boston, Mass., ISBN: 0071152237, 1998.
[47] T.M. Squires, S.R. Quake, Microfluidics: Fluid physics at the nanoliter scale, Rev. Modern Phys. 77 (3) (2005) 977–1026.
[48] JuMonC-nekRS, 2024, https://pypi.org/project/jumonc-nekrs/.
[49] A. Dipankar, B. Stevens, R. Heinze, C. Moseley, G. Zängl, M. Giorgetta, S. Brdar, Large eddy simulation using the general circulation model ICON, J. Adv. Modelling Earth Syst. 7 (3) (2015) 963–986.
[50] G. Zängl, D. Reinert, P. Rípodas, M. Baldauf, The ICON (icosahedral non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core, Q. J. R. Meteorol. Soc. 141 (687) (2015) 563–579.
[51] JuMonC-LLview, 2024, https://pypi.org/project/jumonc-llview/.
[52] S. Josefsson, I. Liusvaara, Edwards-curve digital signature algorithm (EdDSA), 2017.
[53] Plotly Technologies Inc, Collaborative data science, 2015, URL https://plot.ly.
[54] Jülich Supercomputing Centre, JURECA: Data centric and booster modules implementing the modular supercomputing architecture at Jülich Supercomputing Centre, J. Large-Scale Res. Facil. 7 (A182) (2021).
[55] Jülich Supercomputing Centre, JUWELS cluster and booster: Exascale pathfinder with modular supercomputing architecture at Juelich Supercomputing Centre, J. Large-Scale Res. Facil. 7 (A138) (2021).
[56] Jupyter, 2024, https://jupyter.org/.
[57] Bokeh, 2024, https://bokeh.org/.
[58] Matplotlib, 2024, https://matplotlib.org/.
[59] PAPI, 2024, https://icl.utk.edu/papi/.
[60] H. Jagode, A. Danalis, J. Dongarra, Exa-PAPI: The exascale performance API with modern C++, 2020.
[61] LIKWID, 2024, https://hpc.fau.de/research/tools/likwid/.
[62] T. Röhl, J. Eitzinger, G. Hager, G. Wellein, LIKWID monitoring stack: A flexible framework enabling job specific performance monitoring for the masses, in: 2017 IEEE International Conference on Cluster Computing, CLUSTER, 2017, pp. 781–784.
[63] Python-papi, 2024, https://pypi.org/project/python_papi/.
[64] Pylikwid, 2024, https://pypi.org/project/pylikwid/.
[65] NVML, 2024, https://developer.nvidia.com/nvidia-management-library-nvml.

Mathis Bode is a researcher at the Jülich Supercomputing Centre (JSC) at Forschungszentrum Jülich GmbH. His work focusses on computational engineering and physics, fluid dynamics, and exascale.