D6.3 Simulation API, post-processing functions and database schema

Version 1.0

Document Information

Contract Number	952181
Project Website	http://www.coec-project.eu
Contractual Deadline	31/03/2022 (M18)
Dissemination Level	Public
Nature	Other
Author	Christian Witzler
Contributors	Jens Henrik Göbbert
Reviewers	Alexandros Katsinos

Center of Excellence in Combustion

Change log

Version	Author	Description of Change
V0.5	Christian Witzler	Initial version
V0.6	Jens Henrik Göbbert	
V0.7	Christian Witzler	Incorporate internal review feedback
V1.0	Christian Witzler	Final formatting

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

Center of Excellence in Combustion

Outline

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

1. Introduction

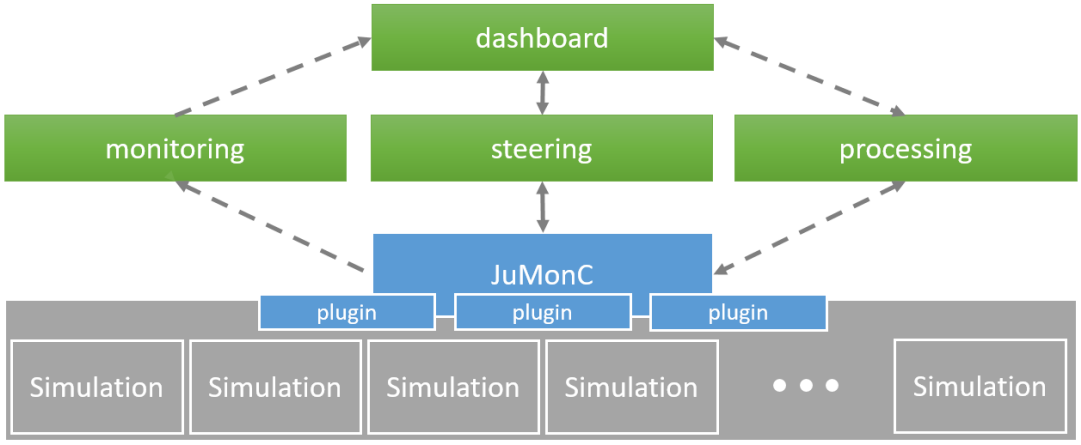

Figure 1. JuMonC forms the software basis for monitoring, steering and processing functions up to the planned development of CoEC dashboards.

The present CoEC Deliverable D6.3 describes the implementation of the software component, which as core and interface forms the technical basis for the implementation of the central goals of the WP6 work package (described in D6.2[1]). A solution has been programmed and described here for the CoEC that simplifies and generalises direct access to runtime information and simulation control of the eleven high-performance CoEC simulation codes and to counteract the increasing complexity of the overall systems.
As shown in Figure 1, the software solution thus enables code-independent development of monitoring, steering and processing functions for the simulation. It is thus the basis up to the development of the planned CoEC dashboards.

With the implemented solution, the CoEC faces the challenge of providing a common, unified and forward-looking API that can flexibly adapt to the different codes on the simulation side, while ensuring a stable data structure on the information evaluation side. This increases the development speed and the code quality for tools and functions for data evaluation.

The summary of D6.2 described the goal as follows:
"As we move towards exascale supercomputers, we aim to improve access to runtime information and simulation control. It is our goal to provide the best possible insight into the current state at any point in time of a simulation. In addition, despite different codes, it is intended to develop a unified interface for simulation control that can be used to send additional control commands at simulation runtime. Both will also pave the way to a CoEC dashboard."
The implemented JuMonC forms this unified interface.

[1] CoEC Deliverable 6.2

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

2. Implementation

To fulfil the goals described in D6.2 as a bridge between classical HPC and a very dynamic interactive and modern supercomputing or automated workflows, an **easy access** to runtime information was required to monitor and control the running simulations in CoEC. JuMonC ("Jülich Monitoring and Control") was developed to be this enabler from simulation to post-processing.

JuMonC	https://gitlab.jsc.fz-juelich.de/coec/jumonc

2.1 Aims

An important goal of JuMonC is to significantly facilitate the collection and evaluation of **runtime information**. This allows users to track issues related to the code development stage or to the simulation setup, as well as to monitor large-scale simulation runs. For example, up-to-date and easily accessible runtime information can be used to detect load imbalances. Therefore, under/over-utilizing of the available resources could be avoided by noticing the possible bottlenecks that each simulation might face, as well as to detect nodes that are underperforming due to hardware-related issues, so that instant actions could be taken to resolve them.

Besides monitoring, JuMonC provides the API-basis to enable simulation control to **influence the simulation** at runtime. The degree to which this influence will finally be possible depends on the support of this functionality by the simulation. Technically, JuMonC does not set any limits. One can trigger a simple data dump, for a later restart, termination of the simulation, rebalancing to unload certain nodes or to remove them completely from the simulation.

Simulation control is a key feature to enable in situ processing. This type of **post-processing** the simulation output, provides live access to the data to different user groups. JuMonC enables easy setup of in-situ tools, like Catalyst[2], and provides all the information needed to connect to a running simulation. This is possible because one can now tell Catalyst through the simulation that there is a (additional) data consumer, who then gets access to this VTK[3] data in ParaView[4].

To perform these tasks, JuMonC collects monitoring information and makes it available to the user in the form of a **REST-API**. REST stands for "Representational State Transfer" and can generally be accessed via network requests. A REST-API is characterised by the stateless communication, simple interfaces and communication using HTTP(S) features that it incorporates. In addition, a REST-API should be self-

[2] https://www.paraview.org/in-situ/
[3] https://vtk.org/
[4] https://www.paraview.org/

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

explanatory, pointing to other parts of the API so that any functionality can be found beginning from a starting point. However, it is not a fixed standard, but only guidelines, so that a REST-API is easy to implement, fast and less cumbersome. In addition, user endpoints do not have to meet additional protocol requirements, which allows 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 make this possible in JuMonC's REST-API, information is organised in a **hierarchical tree structure.** A tree structure lends itself to a web resource, since we use paths like "network/status/bytes". This defines the hierarchy, in the example path we ask for information about the network, more precisely about the status information of the sent and received bytes. In the case of the leaves of this tree, you can make a further specification of the operation by influencing the execution with parameters. For example, you can request the data from a specific node or average the data over a specific time.

In Figure 2 the multiples levels of the REST-API are shown, each level provides information about the functionality of the lower levels, both with a short description, and a list of all possible parameters that can change this behaviour, including the explanation of the influence of all these parameters. In order to improve orientation, a distinction is made for all information as to whether it is static or dynamic. All static information is always located in the "config" substructure. All values that can change dynamically are located in the "status" substructure.

This results in a structure where we have the different hardware components like CPU, GPU, network, main memory and storage as different information sources. For each of these types we can then query static data like the maximum clock rate of the CPU or GPU, or determine dynamic values like the network traffic in the next 10s. The information that can be categorised very well in this way includes the information that the schedule manager (e.g. SLURM[5]) provides, even if this information is more related to the "hardware" HPC system, such as the time remaining until the allocation for this calculation expires or allocation size.

In addition, JuMonC is the framework to provide flexible information of the individual simulations depending on the simulation application used. JuMonC comes with a flexible and extendable plugin structure, so that each simulation application can have its own specific monitoring information and control abilities. The **plugins** are light-weighted and easy to implement Python scripts where the simulation applications report data to and inquire about requests such as aggregating data.
As described in D6.2, JuMonC was developed for simulation control and runtime information collection. A flexible plugin structure allows both of these ideas to be realised in a way that ensures that the developers of the simulation codes can quickly and easily get involved with their requests and ideas. Consequently, JuMonC is started together with the simulation and the exact characteristics are determined by the plugins used.

[5] https://slurm.schedmd.com/

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

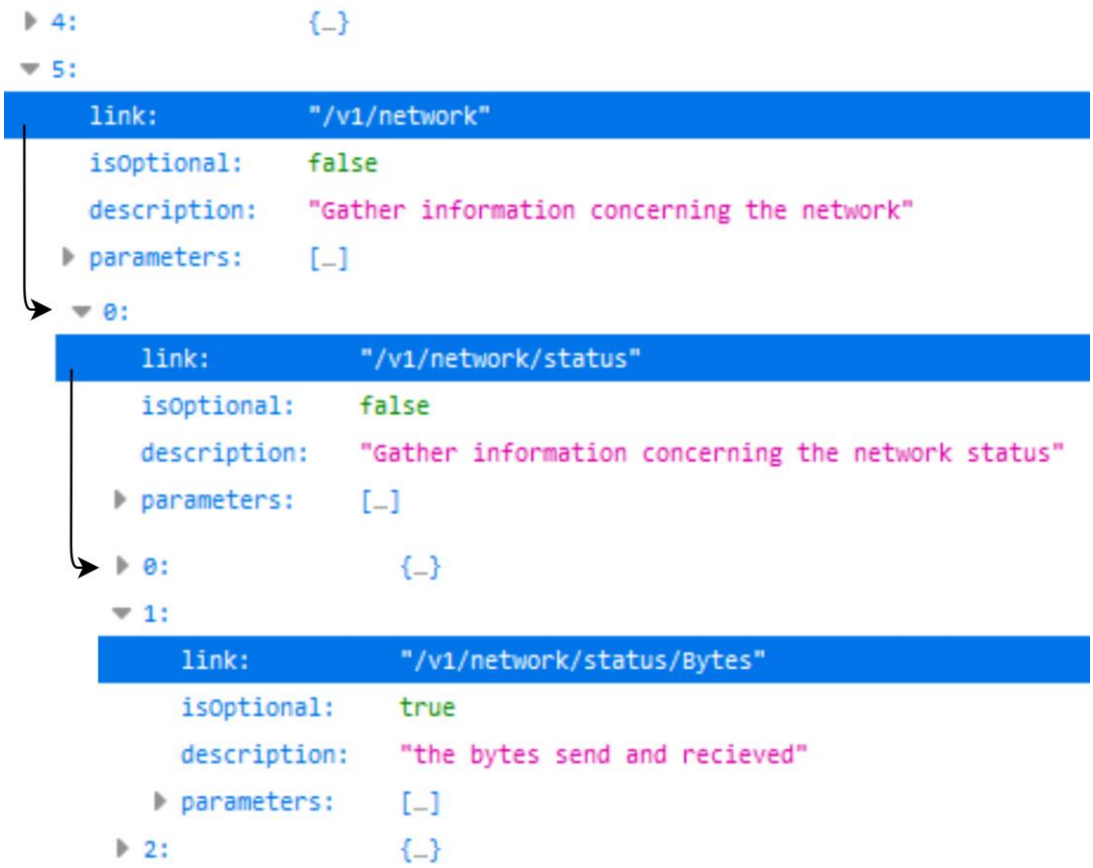

Figure 2. This figure shows multiple levels of the tree structure of the JSON responses by JuMonCs REST-API

In CoEC different supercomputing centres provide compute time to the developers and users of the combustion codes. Therefore, it is important that JuMonC is used solely at the user level without the need for consultation with administrators. This is ensured as JuMonC starts with the simulation code from within the job script under user control.

For an openly accessible interface, it is imperative to plan a consistent **security concept** from the very beginning. JuMonC's REST-API provides access to, insights into the simulation, influence on the performance, up to changes of the settings and thus results. Hence it needs to be secured.
The first hurdle that prevents misuse is that all HPC centres protect their computers with a firewall, which, if functioning correctly, significantly limits the number of people with access to this web resource. Therefore, it is more likely to be accidentally misused and not actively targeted, for example if someone wants to interact with their simulation but tries to contact a wrong node. But to limit the impact of both dangers, it will be possible to block access with missing authorization. As an additional part of an authorization you can introduce different rights levels by having different tokens, each allowing a certain scope of functions. This allows different users to access different parts of the REST-API. These scopes are ordered, and each level grants the access of the level before it and some additional authorization (see Table 1).

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

scope token name	authorised functionality
see_links	see available functionality of this instance of JuMonC
retrieve_data	see available functionality, retrieve already computed runtime information
compute_data	see available functionality, retrieve and compute runtime information
retrieve_simulation_data	see available functionality, retrieve and compute runtime information, retrieve already computed simulation information
compute_simulation_data	see available functionality, retrieve and compute runtime/simulation information
full	see available functionality, retrieve and compute runtime/simulation information, administrative JuMonC access

Table 1. All available authorization scopes with an overview of authorised functionality

The reason for the different scopes of authorization that can be configured in JuMonC, is to give control over what can be accessed. This allows to restrict the control and influence someone can exert on the simulation, based on trust and needed access to fulfil a role. Therefore, they are ordered with a need for greater access for the functionality that has higher performance impacts. The lowest impact has someone just exploring the capabilities of JuMonC in a specific instance, while the next levels allows one to access and gather runtime information. A higher need for confidentiality is possible for simulation data, while ordering the simulation to calculate specific values can have a serious impact on the simulation performance if done imprudently.

2.2 Requirements

In order to achieve these goals, several decisions had to be made regarding the components to be used as a basis. JuMonC has been developed in **Python**. This is an obvious choice for tool development in HPC and science, and provides the advantages of high flexibility and the possibilities for dynamic change that Python offers. In addition, Python makes it possible for users to easily use JuMonC afterwards, as python supports virtual environments, which allow an easy installation of the required dependencies via a package manager. This facilitates users to set up and use the application quickly and without significant effort. Of course Python as a programming language in a centre of excellence project has a downside, as being an interpreted language it cannot compete with compiled languages in terms of pure performance. However, since JuMonC is not

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

intended to handle computationally intensive tasks itself, the advantages of Python outweighed this disadvantage, as it does not affect JuMonC.

Next, a web framework had to be chosen, where besides **Flask**[6] also Django[7] and Tornado[8], were available. Flask was chosen because it is well supported, easy to use, and its structure is well suited for implementing a REST-API. It provides developers complete control over the codebase and allows flexible selection of the desired components.

In addition, internal communication is required. For the time being, **MPI** is the most suitable communication base, as it is very widespread in the HPC area and is therefore available at all HPC centres. To keep JuMonC future proof and prevent deadlocks, all communication is internally funnelled through one file that implements all needed communication routines based on MPI, but keeps the chance to change the communication in one central place.

Core dependency	**Provides …**
Python	programming framework
Flask, Flask-Login	RESTful design
MPI, mpi4py	communication
Plugin dependency	**Provides …**
psutil	CPU-, IO-, network- and main memory information
pynvml	GPU information
PAPI, python_papi	CPU hardware counters
LIKWID, pylikwid	CPU hardware counters

Table 2. Overview over the functions each dependency is needed for

This clarifies the mandatory dependencies, Python, MPI with mpi4py and Flask with flask-login, that must be available to use JuMonC. In addition to these dependencies, there are also others that enable certain functionalities. But these are not so critical, because in their absence only certain hardware-related information cannot be collected. These optional dependencies are currently psutil[9] for general hardware information like

[6] https://flask.palletsprojects.com/en/2.0.x/
[7] https://www.djangoproject.com/
[8] https://www.tornadoweb.org/en/stable/
[9] https://github.com/giampaolo/psutil

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

CPU data , pynvml[10] as the Python bindings to the NVIDIA Management Library, and PAPI[11] and LIKWID[12] for direct access to CPU hardware counters.

It is possible to use **different sources** for similar data in parallel, if each source provides its own advantages. This data only has to be ordered in a sufficient structure in the REST-API, so that the users can choose which data source (or both) they want to use. This allows users of JuMonC to only provide the dependencies for one data source and still access this type of data.

2.3 Software architecture

A solid base has been created so that it is flexible enough to handle the communication with the simulation and the collection of runtime information. This is achieved by separating the different functional components, an overview of which can be seen in Figure 3. The interaction always starts with the consumer accessing it from the left side. The consumer has the possibility to access the **REST-API** with a tool of his choice, which enables web requests. Figure 3 lists a few examples on the left. This can be the direct access in a terminal with simple tools like curl[13]. The user is not limited in his choice of the endpoint and can use much more interactive possibilities like a Jupyter Notebook[14] or dashboard to access and visualise the data directly.

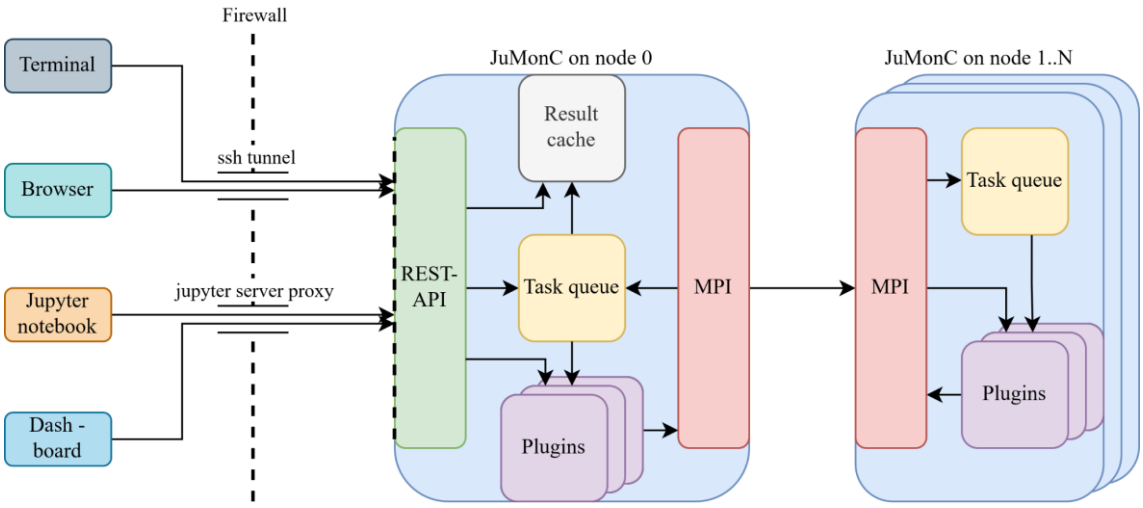

Figure 3. This sketch shows the general structure and interaction of the components. The arrows always indicate the initial communication direction, i.e. which component addresses which other component and then awaits a response.

[10] https://pypi.org/project/pynvml/
[11] https://icl.utk.edu/papi/
[12] https://github.com/RRZE-HPC/likwid
[13] https://curl.se/
[14] https://jupyter.org/

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

In Figure 3 the firewall is sketched as an entrance gate. It is the first **security** level used by the data centres to protect and restrict network access from outside. However, authorised users are not affected by this restriction if using, for example, an ssh tunnel or, if available, a server proxy. Both take over the same functionality, they forward the requests from outside to certain points within the HPC system.
After starting JuMonC, the user interaction only takes place via the REST-API. This provides a central point to further authorise users and restrict access (see chapter 2.1) through the verification of scope tokens. Each time JuMonC is started a scope token, which is created and issued individually, must be provided. While all tokens (see Table 1) are generated randomly on JuMonCs start, they can be set to a prefered value when starting JuMonC.

JuMonC starts simultaneously with the simulation as a separate process on each node once. It communicates via its REST-API from node 0. This process serves as the central communication interface with the user. User requests are received here, and answered depending on what is requested. In the simplest case, where it is only a request for some part of JuMonC's API tree, it is answered directly by the REST-API. The next more difficult case for JuMonC requests previous results, these can simply be requested from the result cache and returned to the user. When new results are requested, they must be actively collected. Again, there are different levels of complexity for JuMonC, the easiest is 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 0 of JuMonC. For all the previous options, in the case of dynamic results, these are cached.

The most complex case from the point of view of JuMonC is when a process is to be queried over a longer period of time over several nodes, such as average network traffic in the next minute. In this case relevant information is stored in the task queue on rank 0. From there it is communicated using MPI and executed on all nodes in separate threads so that in the meantime the MPI communication is not blocked.

A link and an estimated time are returned as the response, so that the user does not have to wait a long time without a response. With the given link results can be retrieved via the cache as soon as they are available.
The capability of handling user responses in two different ways was made because the processing times of the 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, as 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 normal time, so that the user gets feedback.

After the goals for D6.3 were defined and the prerequisites were set with Python, Flask and MPI, the first implementation was made for the ideas from D6.2 from October. The

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

choice of these components allowed us to focus on the functionality. The last decision to be made was in which data format to return data on requests. To achieve the goals for D6.2 and D6.3, a format was needed that found a compromise between computers and humans. A good compromise is the **JSON** format, a simple format that allows computers to extract the critical information very easily, but remains descriptive and formattable enough for a human to keep track of it.

2.3.1 Plugin structure

To support simulations with significantly different requirements, JuMonC uses a plugin model internally, that allows the loading of separate plugins at the startup, and which inherits and defines certain entry points of a class. Thus, JuMonC is easy to extend and allows plugins not only for the communication with simulations, but also for the collection of system data that may be of interest. Thus, it is possible to include the gathering of new data by including alternative plugins, and without the need for direct development of JuMonC. Moreover, it allows for an extension of the tasks that a simulation can perform through the REST-API, but provides a similar interface through the use of JuMonC. Using a plugin structure makes it easier to engage users and developers of simulation codes, because they get an easy possibility to fit a plugin to their special needs. Another advantage that JuMonC gains from the plugin structure, is that it can be easily used in other projects outside of CoEC, which increases the chance for a widespread and long lasting success.

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

3. Sustainability

Software implementation is one thing, but in a large project like CoEC it is important that a software is developed sustainably. Therefore, in this first implementation, a code basis is created that allows continuous further development and improvement. This allows for more than one developer to work on JuMonC simultaneously. As a cooperation tool, which builds up a version history, git was chosen, because it is widely used and therefore known to many developers. To create easy access for all possible developers to this project, it is available on GitLab, where everyone can create an account to contribute to JuMonC.

To allow long-term development with backward compatibility, each REST-API function will contain a reference to the REST-API version as part of its path in the tree structure. This allows the development of new features that break the old way of JuMonC in a new REST-API version, without dropping the support for the established versions. Then it is a conscious user decision to switch to the new version of the REST-API.

3.1 Continuous integration

Continuous integration(**CI)**, are automatic processes that are started on GitLab and allow to inspect the code for certain criteria. This allows developers to detect bugs early in the development and creates confidence that newer JuMonC versions have not accidentally created bugs in old functions. An overview of the used CI in JuMonC can be seen in Table 3.

Here we use three so-called **static tests**, so these tests check the source code without executing it. For example, the code style is analysed here by **Prospector**. This enforces the development to follow general best-practice rules, which improves and ensures the readability of the code. This is important if you want to support JuMonC over a longer period with different developers, because old code that was developed without such guidelines quickly becomes confusing.

The second static test that is performed uses **mypy**. In python there is no fixed requirement to assign a type to variables, because this happens dynamically and can change while the application is running. Not having to worry about this increases the speed of development. Unfortunately, this can lead to hard to find errors. In order not to have a fixed typing, but still to create the possibility of not accidentally passing unexpected data types to functions, mypy allows you to add an annotation to variables and functions that specifies the variable type, that then can be checked for inconsistencies by mypy as part of the CI.

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

CI Component	Functionality
mypy[15]	tests variable typing
Prospector[16]	enforces rules for uniform code style
Bandit[17]	tests for security issues
pytest[18]	executes developer provided tests of JuMonC's functions
coverage	provides a percentage value of code testet by pytest
REST-API link tester	generates a recursive links of all available links in the REST-API and tests them for availability and removed links since last commit
CMD arguments	Updates documentation for start parameters of JuMonC

Table 3. Overview of the used components in the CI pipeline for JuMonC

As a final static analysis, **Bandit** is used. Bandit is a program that analyses code to find bugs that make the built program more vulnerable to attack. Bandit is particularly useful in this project because it checks and reports several security-related errors that can be made when using Flask. Especially such a check of flask is very valuable, because flask is the communication interface to the outside, even if there are additional safeguards.

Another important component of the automatic tests are the **unit tests,** which are performed using **pytest**. This allows us to call the developed functions in isolation and check if the results are as expected. If there is a high **coverage** of tests, it can be prevented that old functionality is accidentally broken when adding new functionality. For an API that is to be continuously developed, this can prevent unpleasant surprises and thus maintain and strengthen the trust of the users.
pytest can be used to determine how high the test coverage of the existing source code is, which subsequently gives you a percentage value of the tested lines of code in relation to the existing lines of code. Here, of course, complete coverage is the goal. To make this more possible, the tests are stored in a separate folder that is not part of the actual API, so that the static tests can be omitted. This is justifiable, since each individual test calls a function and checks the results. Therefore, there is only a low complexity for the tests themselves, which makes tests for variable type and security vulnerabilities unnecessary.

[15] https://mypy.readthedocs.io/en/stable/
[16] https://prospector.landscape.io/en/master/
[17] https://bandit.readthedocs.io/en/latest/
[18] https://docs.pytest.org/en/7.1.x/

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

A last test to control the code is a self-developed test, which checks that all links given to the user are available and do not point to (yet) unimplemented endpoints. This checks the self-describing component of the REST-API, recursively for all links specified by JuMonC. In addition, a list of the existing links is created and stored in GitLab as well, so you can check if there are differences to the previous list and especially if there are deleted links. This allows to notice if by mistake functionality is no longer available or findable.

Another automatic function that has been integrated is an automatic documentation of the start-up parameters that can be used. This transforms the help page that can be displayed at program start-up using "-h", or is automatically displayed in case of invalid parameters, into a more readable file that is then automatically added to the documentation on gitlab. This automatically ensures that this part of the documentation is always kept up to date and reduces the effort of changes, since these only have to be made in one place.

3.2 Documentation

For long term development and usage the documentation is important, so that functionality can be found and used. As a simple form of documentation, the self-describing linking of the REST-API can be used. Since the goal for a REST-API is that all functionality can be found and used easily and intuitively, further documentation may be unnecessary. But this documentation is only available when JuMonC is started, and does not contain references to functionality that is not available due to missing dependencies when the application is launched. Therefore, another documentation is created using **OpenAPI**. OpenAPI is a good choice because it was created specifically for the documentation of web interfaces and there is a very clear overview of the web interface on GitLab (for example see Figure 4). For this project it is advantageous that OpenAPI supports JSON schema. Therefore, it is possible to describe JuMonC's responses with **SCHEMA** and this description can be used directly in OpenAPI, so we can use the desired description with SCHEMA in CoEC for all available data directly here as well.

As further documentation for users, there is a general **readme**, which clearly explains the installation and dependencies and serves as a starting point for documentation. From here the user is then referred to the further documentation in the form of SCHEMA descriptions, OpenAPI and documentation of the start parameters. It is therefore a great starting point to learn about JuMonC, an overview over the available documentation from within the readme can be seen in Figure 5.

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

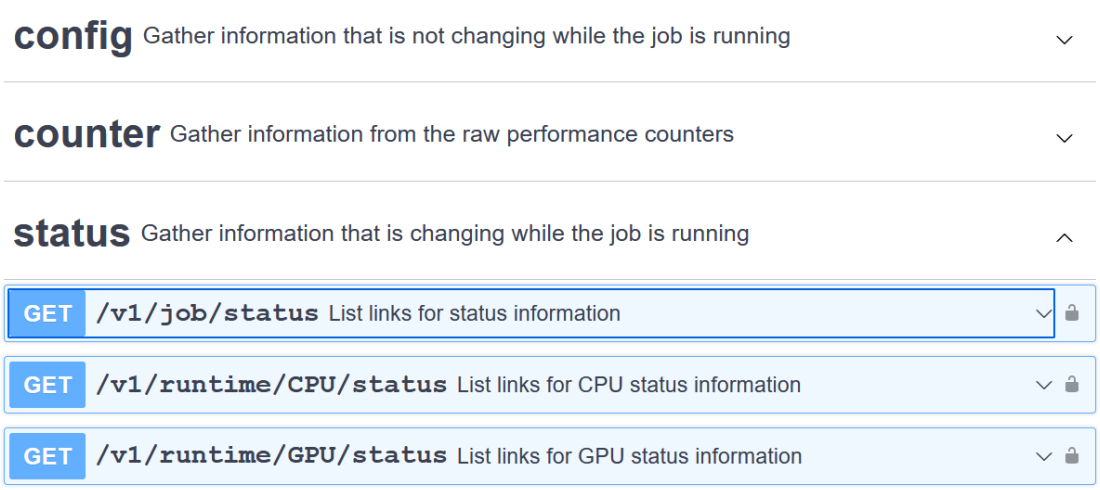

Figure 4. Excerpt of JuMonC's OpenAPI documentation

Additional documentation is available for developers who want to help with the development, which explains the internal communication and task queue and how to best **contribute**.

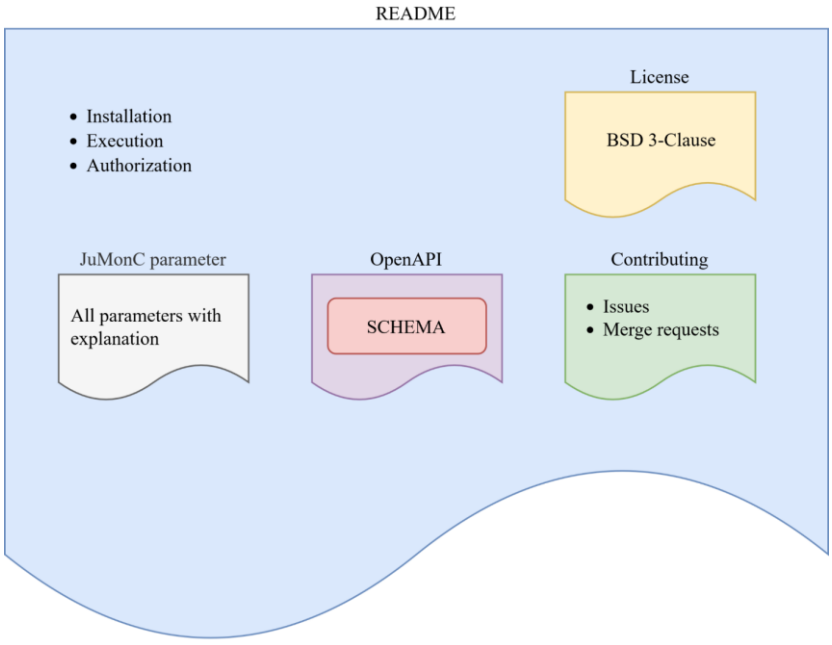

Figure 5. Overview over the documentation available from within JuMonC's readme

3.3 Contributing

For a project to last over time, it must be possible for more developers to contribute, and it has to be possible for normal users to report problems and improvement requests. Without these capabilities, a software project can quickly bypass the wishes of the users,

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

or bugs that have already been reported by users can persist because they are not known to any developer.

One advantage of **GitLab** is that it makes it easy for users to further contribute to its development, by reporting bugs and requests using so-called issues. In this way, problems and solutions can be commented on and discussed, and it is possible for new users to follow up on these decisions. As a further possibility to contribute, JuMonC can be forked into user owned projects. Therefore, you can make individual changes there, which hopefully can be transferred to the main repository in the form of a merge request in case of changes with general interest. Then it is of course important that these changes pass the automatic tests and adhere to the general structure of the repository.

At present, **the repository** is divided into four main parts: the definition of the automatic CI functions, the documentation, the tests performed by pytest, and the actual source code. There is a further division, especially for the actual source code, between the various tasks to be performed, the data management, and the presentation of this functionality with the REST-API. This separation enables very clear code, since these three main components are separated from each other and thus function more independently of each other via clear internal interfaces. This allows for easier testing, as these separate functionalities do not need to be tested all at once. Additionally, it allows one to easily track where changes of a merge request are focused and if more important aspects for overall functionality and security like authorization were changed.

Another way to contribute indirectly to this project is through **plugins**. Since these are separate from JuMonC and only integrated, they can be developed in their own projects without a direct connection to JuMonC. JuMonC is of course connected to the developed plugin, because it’s API is necessary to use the plugin. Depending on the wishes of the plugin developers, the plugin can remain independent or, depending on the development status, be completely integrated.

The CoEC project has received funding from the European Union’s Horizon 2020 research and innovation program under grant agreement No 952181.

4. Next Steps

As the next steps to improve JuMonC, there are some general improvements. It is important to round out JuMonC's capabilities even further to create a complete workflow.

Another enhancement is to **extend the data** provided and its variety to cover more usage scenarios. The aggregation of data from multiple nodes will be extended to increase flexibility, so that each user can request exactly the data he needs.

The next two refinements go hand in hand, namely the **data cache** needs to be optimised so that clear statements can be made about the **work memory footprint** of JuMonC. Here one could store the cache of the results with the help of a SQL based database persistently on the hard disk and take advantage of the fact that one can set a cache size in the database software. This directly allows the second enhancement of **persistent** data. It would then be possible to get access to this data, even after JuMonC has been terminated. This is a significant further enhancement, as it will make it easier to compare data with older simulation runs.

To complete the functionality of JuMonC even further, one can add a possibility to **automatically query** certain values, which are then available in the result cache, and in the case of persistent data can be evaluated even after the simulation has run. Or if this is not useful, you can define base values that can be used for further analysis of comparative values from the past.

Another important point will be the **cooperation** with the existing simulations in **CoEC** and to ensure that there are good usable plugins with suitable functionality for some of these simulations, so that JuMonC can find its way into a productive use.

This goes together with the creation of examples, such as one or more jupyter notebooks and **dashboards** that use this data and present it clearly. These then serve as a good example of what is possible and can function as additional documentation on how best to access the necessary data from a dashboard, for example.

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.

5. Summary & Conclusion

In summary, JuMonC provides access to a wide variety of data on and about the simulation, which can be expanded to include new capabilities in the future. The data is made available as JSON via a REST-API, which gives the user a great deal of freedom as to how he or she can best access the data. Through a basic structure of the REST-API provided by JuMonC, it is possible to find all information without further documentation. By using established and flexible dependencies like Python and Flask and an internal plugin structure, it remains possible to adapt and extend JuMonC to new circumstances in the future. The communication to other compute nodes is kept flexible in JuMonC, so that it is possible to detach the communication from MPI in the future if necessary. This and the low communication requirements make JuMonC well suited for monitoring tasks, enabling post-processing and dashboards in CoEC. This close (insitu-) monitoring is ever more important, with increasing simulation sizes, to prevent the unnecessary usage of computing resources.

In general, the development of JuMonC was focused on the fact that it is future-proof and can be continuously developed by all interested parties over a long period of time. For this purpose, there is a clear structuring of the repository and a multi-layered documentation for both users and developers. The integration of various automatic tests, known as continuous integration, lays the foundations for sustainable joint development. Since the source code of JuMonC is publicly available, any interested user can report problems as well as contribute to solutions and improvements.

This results in a good foundation for further development for more interactivity in all functions of large simulation runs.

The CoEC project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 952181.