JÜLICH
Forschungszentrum

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich
Fachbereich 9 – Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Interactive Visualization in the Cloud: Browser-based Visualization in JupyterLab with ParaView trame for HPC Use-cases

Bachelorarbeit von Jonathan Windgassen
3277745

04.09.2023

Erstprüfer:	Prof. Dr.-Ing. Andreas Terstegge
Zweitpüfer:	Dipl.-Ing. Jens Henrik Göbbert

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Name: ______________________________

Jülich, den _________________

____
Unterschrift der Studentin / des Studenten

Abstract

In this work, we present *JuViz*, a novel approach to browser-based interactive visualization. It bridges the cloud-based development environment *JupyterLab* with the visualization framework *ParaView*. With *JuViz*, users of the *high performance computing (HPC)* systems can access flexible visualization apps in the browser. It relies on the framework *trame* and is a part of the cloud platform *JupyterJSC*. Apps utilize *ParaView* as the visualization framework for remote rendering of the data on the HPC system. *JuViz* allows adding customized apps, which can be developed for certain communities or purposes. This empowers users to create web apps, tailored to their specific needs and requirements. It enables the comfortable, easily accessible, intuitive and flexible visualization of large data, an essential aspect to find wide adoption within the scientific community and be integrated into visualization workflows.

The setup is demonstrated by the development of the *trame ESM-Slicer* — a *trame* app for the *Earth System Modelling* community — which is capable of visualizing atmospheric and climate data. It is utilized to assess the strength and limits of *trame* and provide insight into the development process of apps made for *JuViz*.

Acknowledgements

First and foremost, I would like to express my gratitude towards Professor Andreas Terstegge, who agreed to supervise and assess this thesis.

I want to thank my supervisor Jens Henrik Göbbert, who not only also took the responsibility as the secondary examiner, but also helped me along all stages of the work, from defining the scope of the work to helping me with the software. Without him, this thesis would not have been possible.

I thank my friends, family and colleagues for their support, both mentally during the work and regarding the creation of this thesis.

I acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC) and funding from the European Union's Horizon 2020 research and innovation program under the Center of Excellence in Combustion (CoEC) project, grant agreement no. 952181.

Contents

Abstract

1 Introduction **1**

2 Visualizing Large Data for HPC **3**

2.1 Local Visualization . . . 4
2.2 Remote Desktop . . . 5
2.3 Remote Rendering . . . 5
2.4 In-situ Visualization . . . 6
2.5 Mixed Rendering . . . 8
2.6 Visualization as a Service . . . 8

3 Technical Background **11**

3.1 ParaView . . . 11
3.1.1 Catalyst . . . 12
3.1.2 trame . . . 13
3.2 JupyterLab . . . 14
3.2.1 JupyterJSC . . . 14
3.2.2 jupyter-server-proxy . . . 15
3.3 jupyter-trame-proxy . . . 15

4 Development **16**

4.1 JuViz Extension . . . 17
4.1.1 Implementation of the JupyterLab Extension . . . 18
4.1.2 Implementation of the JupyterLab Server Extension . . . 19
4.1.3 Loading user trame Apps . . . 23
4.2 JuViz App . . . 25
4.3 Installation and Distribution . . . 26

4.4 trame ESM-Slicer . . . 26
4.4.1 Requirements and Desired Features . . . 28
4.4.2 Development . . . 28
4.4.3 Challenges and their Solution . . . 30
4.4.4 Summary . . . 31

5 Discussion **33**
5.1 JSC Concerns . . . 33
5.1.1 HPC . . . 33
5.1.2 Maintainability . . . 35
5.1.3 Security . . . 36
5.2 User Concerns . . . 37
5.2.1 Usability . . . 38
5.2.2 Communities . . . 38
5.2.3 Flexibility and Compatibility . . . 39

6 Conclusion and Outlook **41**

Bibliography **43**

Appendix **45**

List of Figures

1.1 Structural overview over the software components of *JuViz* 2

2.1 Overview of the terms describing in-situ systems 7

3.1 Overview of the *ParaView* software architecture 11
3.2 The *ParaView* pipeline, representation in the client and as a tree 12

4.1 Figure 1.1 with steps outlining the *JuViz* functionality 16
4.2 Screenshots of *JuViz* Extension showing from a - f the different UI components for launching and pairing *ParaView Servers* and *trame* apps . . 20
4.3 Class diagram for the Configuration class 21
4.4 Directory tree for adding *trame* apps to *JuViz* 24
4.5 Visual comparison of the *Panoply* and *trame ESM-Slicer* User Interface . 27
4.6 Schematic overview of the *trame ESM-Slicer* UI 29
4.7 Pipeline structure of the *trame ESM-Slicer* 30

5.1 Diagram outlining the encryption and access restrictions for JuViz 36

Listings

4.1 *JupyterLab* extension enty point 18
4.2 User API Handler 22
4.3 *ParaView* template script (except) 23
4.4 *JuViz* app.yml 25
4.5 *JuViz* launch.sh 25

A.1 Configuration class Python slug with comments 45
A.2 Complete *ParaView* template script for the *JUWELS Booster* system . . 48

1 Introduction

In the realm of scientific simulation and HPC, visualization stands as an essential aspect, enabling scientist to gain critical insights into their data and generate meaningful visualizations for the processes at work. With the growth of HPC, the ability to visualize large datasets became even more challenging, as the complexity of rendering and access to the data made traditional local visualization impossible.

The *Jülich Supercomputing Centre (JSC)* at the *Forschungszentrum Jülich (FZJ)* is one of the largest supercomputing centers in Europe. It provides the systems and infrastructure needed for large simulations and regularly houses datasets that fall into this category. As part of the *Algorithms, Tools, and Methods Lab for Visualization and interactive HPC*, we provide the users of the systems with the software required to render and analyze the data generated on these systems. With the decline of traditional visualization workflows and the newly arising challenges for visualization, we often notice the insufficiency within the currently provided infrastructure. Thus, we aim to establish new visualization techniques, that are capable of handling large data and provide adequate performance for large exascale simulations. To achieve comfortable and accessible visualization for large datasets, a way to seamlessly access HPC-rendered visualizations without a complicated setup is needed.

In this thesis, we prepare the basic implementation of such a system — *JuViz* — with a focus on interactive visualization in the cloud. It provides browser-based visualization within the *JupyterLab* environment [23]. With the cloud-platform *JupyterJSC* [1], *JupyterLab* has been established at the JSC in the past, bridging the access between browsers and the HPC systems. It provides a complete, extensible development environment, running on the systems themselves and allowing access from a local web browser. By utilizing its extensibility, we gain the means for an accessible interface, sidestepping the necessity for in-depth knowledge. This interface is versatile, enabling customization and arbitrary visualizations, without resulting in overloaded user interfaces (UIs). Security remains a critical aspect, especially within multi-user environments, necessitating safety and shielding sensitive information. In the current workflows, *ParaView* [8] — a powerful framework for scientific visualization and analyzation — is used to take care of the main visualization tasks. It has a large popularity as a versatile visualization tool with HPC compatibility at the JSC. Thus, *ParaView* was selected as the visualization

[1] https://jupyter-jsc.fz-juelich.de

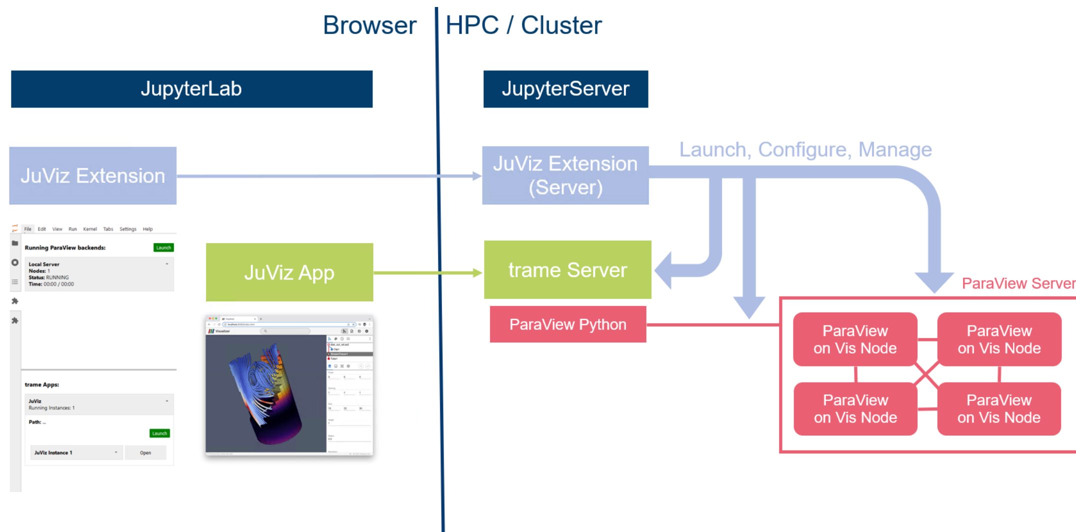


Figure 1.1: Structural overview over the software components of *JuViz*

framework for *JuViz*.

By combining *ParaView* with the *trame* [24] framework, it can be united with *JupyterLab*. *trame* is part of the *ParaView* ecosystem and capable of creating visualization-focused web application. The structure of this combination is shown in Figure 1.1 and is explained in-depth in chapter 4. This novel approach promises interactive visualization, simplifying the complexities of data access, rendering, and interaction in a customizable and accessible manner.

In the first part, this thesis examines different approaches for visualization, which are compared from the perspective of a user and a supercomputing center. We inspect the capabilities of the software products with the HPC infrastructure and their convenience for accessing and working. After a brief introduction of all the required software components, the development process is outlined. The development encompasses three products, that are combined at the end of the process. The main software component consists of an extension for *JupyterLab* — *JuViz* — that was integrated into the *JupyterJSC* platform. It serves as a control hub, capable of launching both *ParaView* and *trame*. It abstracts this control into a convenient user interface (UI). Alongside this extension, a default *trame* app is created. Additionally, a second, specialized *trame* app is used with the extension to assess strengths and limitations of the approach. Finally, we discuss the design decisions taken during the development.

2 Visualizing Large Data for HPC

The interactive visualization of HPC data in an accessible and flexible way is far from trivial. In the last years, numerous ways have been established with this goal. Each resulting workflow comes with its own advantages, but also requirements to the software architecture.

The following is supposed to be a comparison between a selection of different approaches, some of which are already provided at the JSC. However, it should be explicitly pointed out, that neither the selected workflows nor the criteria used are meant to be complete lists and were selected with our HPC infrastructure in mind. We split the criteria into two categories, ones that are primarily of interest to the JSC itself, and ones that concern the usability of the workflows to the end user.

JSC Concerns

JSC concerns are primarily from the viewpoint of a supercomputing centre. While they can, most definitely, be relevant for some users, most of them will only be secondary concerns to the average user.

HPC compatibility How well an approach is suited to the HPC architecture. This encompasses the performance of the system and its capability to work with large datasets. This also involves the capability to process the input data in an in-situ or in-transit manner, meaning processing the data without requiring it being written to disk and instead concurrently to the data generation (see section 2.4).

Maintainance How smoothly can software updates and patches be rolled-out to all users. This aspect is especially relevant to the software maintainers of the system, in situations where changes to the system or improved software should be enabled system-wide.

Security Since HPC systems are often multi-user systems by nature, many approaches must be adjusted to the subsequent increased security standards. It is crucial, that the used software guarantees authorized access to the data and shields foreign parties from reading it.

User Concerns

User concerns are, opposed to the JSC concerns, much more oriented towards usability and flexibility of the workflow. These aspects are important for the average user, as they determine the willingness for this approach to be integrated into workflows and find adoption withing the HPC community.

Usability Interactivity, Responsiveness, and Accessibility on different platforms, as well as simplicity of the installation and usage.

Communities Support for working in communities. This encompasses the sharing of code, workflows and visualization pipelines for collaborating.

Flexibility and Compatibility How flexible is the program towards changing requirements or integration into already existing workflows.

2.1 Local Visualization

The term "Local Visualization" it meant to encompass all workflows, where the program execution and visualization of data is bounded to a single-, often desktop machine, with no steps of the visualization happening remotely in the cloud or on the cluster. This approach is the most primitive workflow possible, where the Visualization is done by a program that was previously manually installed on the machine. The initial step in this process usually involves transferring the data to the local machine, which allows it to be further processed in upcoming steps.

This approach has numerous limitations, most notably the limited disk space for most machines and the performance of a single processor. In addition to this, the downloading process can further lead to significant time and effort expenses. Furthermore, this process does not allow the use of HPC resources and will therefore be infeasible for large visualizations. Relying on local software also does not allow for central management by a software maintaining team, making the user responsible to fix and update applications they rely on. Since significant parts of visualization workflows are executed locally, sharing them with colleagues is generally difficult.

In recent years, many communities have been developing their internal software suits for visualization. These are specialized for specific types of visualization or input data. Since many of these programs are designed by the communities themselves, they are tailored to their specific needs and conveniently provide the function required for the specific type of visualization, granting a comfortable user experience. With the rapid increase in size the HPC simulations experienced in recent years, however, these programs are

often not capable of working with massive sizes and rely on other software that scale down the input to a manageable size. This not only results in inexact visualizations, but also to additional preprocessing work, making the visualization more complicated. Lastly, the abundance of different visualization framework leads to a wide spectrum of requirements, formats and workflows presence, which complicates and obfuscates the visualization work between different areas.

2.2 Remote Desktop

Remote Desktop is the extension of desktop applications to servers. Here, the applications will be rendered on a virtual display and streamed to the client, in exchange for the inputs from the client.

On one hand, due to the fact that remote desktop is, by design, very similar to a native desktop, many of the complications with local visualization frameworks are still valid here: Software packages are often still be limited to a single machine (a single node on the cluster), however, the hardware specifications for most systems are significantly more spacious. Also, using a remote desktop comes with its own cost, namely the transmission of the virtual desktop. This stream includes the UI of the program and operating system, which is an unnecessary payload for the visualization. The transmission results in higher latency, making the user experience for remote desktop applications less welcoming.

On the other hand, the trouble of data transfer to the machine can often be avoided, given the fact that the remote desktop runs on the same machine (or file system). The same goes for the community aspect, meaning workflows can more easily be shared, as all parts of the visualization are executed remotely. Additionally, if the remote desktop is on the same interconnect as the simulation, in-transit visualization becomes available, provided it is supported by the software used. Most importantly, manual installations can often be avoided, considering the fact that most HPC architectures use a form of flexible module system, allowing much of the required data to be preinstalled to the system by the maintainers. This also allows the application of software updates of the software.

2.3 Remote Rendering

One prominent alternative to remote desktop is remote rendering. With remote rendering, the visualization will be split into two parts: the first is a server on a remote system, i.e., the HPC system, that is responsible for the large calculations and heavy lifting. The

other component is running on the local system, primarily designed for displaying the media generated by the backend and receiving user inputs.

While this setup still requires local installations (the client), we gain a lot of flexibility when selecting where we want to render. Since the interactive elements of the UI are executed locally, the increased latency from the transmission mentioned above does not apply here.

However, such an architecture is expensive to develop and can not be expected as a feature of many smaller visualization programs. If implementations for remote rendering exists, it is often capable of running on multiple machines, granting access to a much higher potential performance. Central software management is only partially possible, as only the software on the remote system can be updated. Thus, the user still retains a part of the duty of updating software manually.

2.4 In-situ Visualization

With the aforementioned increase in size of simulations and their resulting data, it has become exceedingly impossible to create visualizations based on data that has been written to disk. This does not only stem from the fact that the data might be too large to be written to disk, but also from the inherent writing times, that would need to extensively pause the simulation until the dump is complete.

This led to the new branch of HPC visualization called *in-situ* visualization. The central idea being, that the data will not be written to disk, but instead processed concurrently, while the simulation is run, with only the finished visualization being written. The exact structure of an in-situ setup can vary between architecture and framework. Childs et al. [4] showed a more granular scheme for describing in-situ setups, that split the different aspects of the pipeline into multiple axes, seen in Figure 2.1.

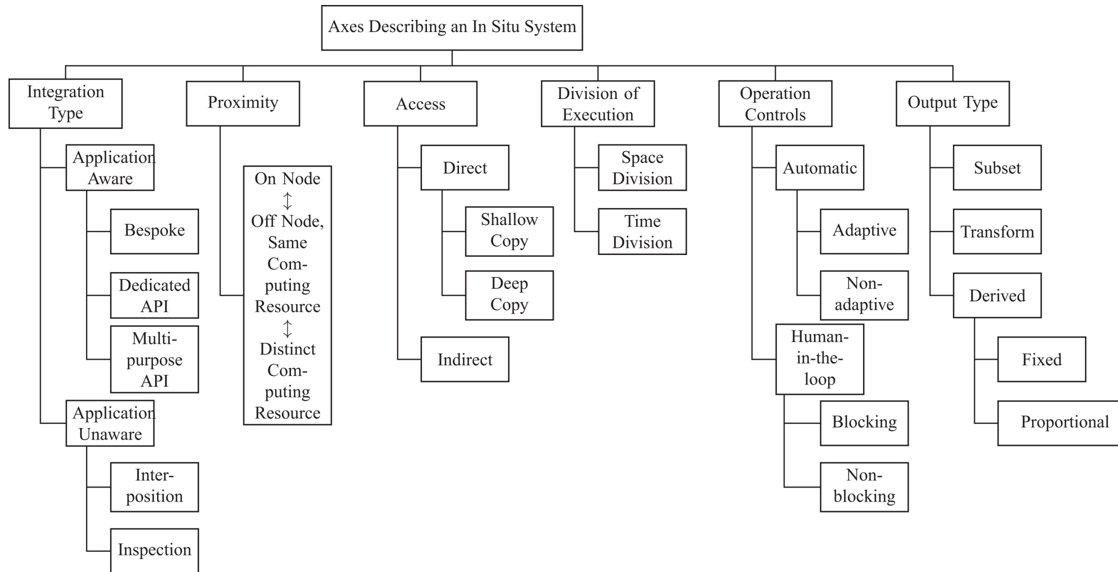


Figure 2.1: Overview of the terms describing in-situ systems [4]

In their work, they also define three common combinations with this terminology that are often encountered in the context of visualization:

Tightly-coupled The visualization runs tightly bound to the simulation, being called from the simulation at each step and executed on the same node. The "Division of Execution" is therefore timely, since at each execution, the simulation will be paused during the rendering. At the end of the rendering process, the media generated by the visualization will be written to disk. This setup provides little flexibility and reliability against software failures, but no off-node data transmission is necessary.

Loosely-coupled The visualization runs decoupled from the simulation on separate nodes. As the rendering is executed on different resources, with both visualization and computation nodes are exclusively dedicated to their respective work. In this case, the visualization will run asynchronously, waiting to receive new data from the simulation. Instead of waiting for the rendering to be finished, pausing the simulation is only necessary until the data transmission is completed.

Hybrid setup One downside of loosely-coupled in-situ pipelines are the enormous, often unnecessary, amounts of data sent to the visualization nodes, resulting in long transmission times. By using a hybrid setup, preprocessing and filtering the data is done next to the simulation, with this reduced data being transmitted to the visualization nodes. The primary visualization is executed on other nodes once the transmission is finished. This minimizes the idle time on the simulation side [1].

By these given definitions, loosely-coupled visualization is always split into simulation and rendering on different nodes. While this is often the case, a loosely-coupled setup can be further refined into **loosely-coupled in-situ** and **loosely-coupled in-transit** visualization. For the former, the visualization is executed on the same computational resources as the simulation (similar to a tightly-coupled setup), but is run as a separate executable from the simulation. Instead, the definition given above would be described as *loosely-coupled in-transit*, where *in-transit* indicates that the data is transmitted to another node. In practice, a *loosely-coupled in-transit* is most suited for large visualizations, as it is much more flexible and can better adapt to different simulations.

2.5 Mixed Rendering

The approaches discussed above separate the process of visualization in calculation / rendering and displaying, which might then be executed on different machines. "Mixed rendering" encompasses processes, where both clients and server are responsible for calculating significant portions of the visualization. In recent years, numerous setups were developed, that instead take a more complex approach, utilizing calculation on both server and client. While the bulk, computationally intensive calculations are still processed on the server, the final image output will instead be rendered on the client, with the transmitted data instead encompassing the preliminary data. By utilizing such infrastructure, fidelity and latency of the visualized data can be improved [7] [3].

Since the format of the intermediate data is heavily dependent on the type of visualization, transmission and conversion must be implemented on a use-case basis. This makes it very hard to abstract universal patterns in these visualizations. On one hand, this approach increases the size of data transmitted compared to full remote rendering, and requires non-trivial computations on the client. On the other hand, it often yields a much higher fidelity for interactive visualizations. This becomes increasingly relevant for VR/XR applications, who gained a lot of popularity with the advance of the required hardware. These setups react hypersensitive to frame drops and high latency, due to the high risk of motion sickness associated with it.

2.6 Visualization as a Service

The concept of *Visualization as a Service (VaaS)* has become increasingly present in recent years with the establishment of *Service Oriented Architecture (SOA)*, like *Infrastructure as a Server (Iaas)* and *Software as a Service (SaaS)*, as a prominent concept for providing services on the internet. The core idea is to provide a cloud-based service, that enables users to remotely and interactively perform their data visualization. The

access to this service, similar to *IaaS* and *SaaS*, is granted through the browser, while the service itself relies on remote rendering. This moves the complexities of data processing, rendering, and visualization to powerful backends, e.g., an HPC cluster, freeing users from the limitations of their local machines.

In their work, Pugmire et al. [21] propose two sets of abstraction, that are required for *VaaS*. The first three abstractions are set as ”Data Access”, ”Service Composition” and ”Data Interpretation”. They serve as the basis for visualization services. ”Data Access” is concerned with the necessity to load data from various formats and sources (e.g., as in-situ data). ”Service Composition” comprises the methods used for defining the visualization pipeline, defining the operation(s) executing on the data. ”Data Interpretation” bridges these components, by requiring knowledge about the input data format that can be used by the pipeline to properly process the data. Building on top of these abstractions is another set, that is concerned with the performance and flexibility of the service. These abstractions are defined as ”Portability”, ”Performance Models” and ”Declarative Visualization”. ”Portability” ensures the utilization of the service across different architectures and HPC structures. ”Performance Models” should be used to provide the user of the service with valuable information, like time estimates, to enable the confirmation that given visualizations are within the resource requirements. ”Declarative Visualization” separates the intentions of the user from the underlying implementations, abstracting the selection of an optimized implementation for the given data.

With the constraints given above, *VaaS* can become a powerful visualization platform, allowing for flexible and accessible visualization of large data on HPC infrastructure. If the abstractions are followed correctly, *VaaS* allows for an approach that is compatible with many formats and requirements. Its accessibility from the browser makes it comfortable to use, while the declarative configuration allows visualization, even for users without in-depth knowledge of the computations and architectures. *VaaS* can also be designed for collaboration, allowing scientist to share both the generated data and the visualization pipeline. This was previously only possible by sharing the data written to disk. For this system to properly work, though, it must strictly adhere to these concepts, requiring large software products and consequently development effort.

At the JSC, establishing a *Visualization as a Service* platform is a long-term goal for providing adequate visualization infrastructure for the upcoming exascale HPC system *JUPITER* [11]. With *JuViz*, we want to lay the foundations for such a service. This ground laying implementation, is primarily focused on the ”Service Composition”, ”Portability” and ”Declarative Visualization”, but also touches on the other aspects. To achieve ”Potability”, the already established cloud platform *JupyterJSC* is used. It allows the starting of a *JupyterLab* environment that is running on the HPC systems at the JSC. As the *JupyterLab* environment is accessed from the browser, *JuViz* becomes detached from operating systems and platforms, purely relying on a web server. *ParaView* is used to provide a software framework, capable of handling many different file formats and visualization tasks. We thus achieve the ”Service Composition” aspect, by

allowing users to create visualization pipelines in *ParaView* that can be loaded and used in *JuViz*. With the *trame* framework, these pipelines can be designed to be flexible and adaptive, allowing users to define a pipeline suitable for their specific need. In combination, *trame* and *ParaView* satisfy the "Declarative Visualization" requirement, where the developers of an app are able to abstract the construction of the workflows, with *ParaView* providing optimized implementations for the executed calculations.

3 Technical Background

The following section provides a brief overview over the software components, which are used as part of *JuViz*. These primarily consist of *ParaView*, the visualization framework used for the rendering of the data, and *JupyterLab* for the cloud platform *JupyterJSC*.

3.1 ParaView

ParaView [8] is a popular open-source software for scientific visualization, developed by *Kitware*. Programs from the *ParaView* ecosystem are prominent candidates when looking for a visualization framework. It is an established software at the JSC and is therefore a convenient choice as the visualization framework for *JuViz*.

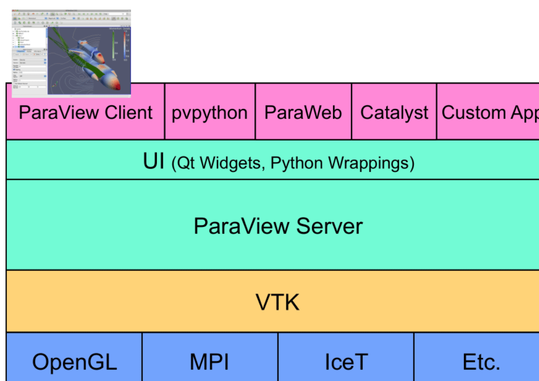


Figure 3.1: Overview of the *ParaView* software architecture [1]

ParaView is designed in a client/server architecture, where the software is split into a server, responsible for the heavy lifting of rendering the data, and a client to communicate with the server and display the rendering results. The *ParaView Server* is based on the *Visualization Toolkit (VTK)* [22]. The structure of *ParaView* can be seen in Figure 3.1. While the *ParaView Server* will be started locally with the client, it can also be run on another system, with the client connecting to it and utilizing the processing power of the remote system. The *ParaView Server* is able to utilize different architectures and processors (like GPUs) and is designed to run on multiple Nodes of a cluster through *Message Passing Interface (MPI)* [15], making it natively compatible with HPC. The *ParaView* client is a desktop application, responsible for interacting with the user and displaying the results from the

[1]https://docs.paraview.org/en/latest/Tutorials/SelfDirectedTutorial/introduction.html

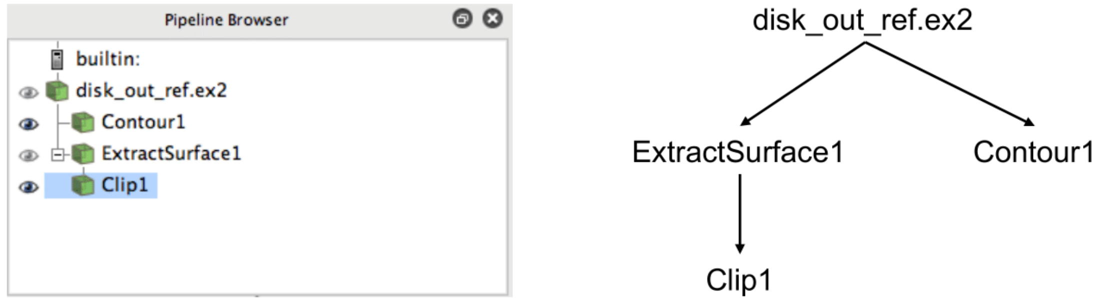


Figure 3.2: A *ParaView* pipeline, representation in the client and as a tree [2]

server. It can detach from its default server and connect to another given *ParaView Server*. *ParaView* also comes with a *Python* API called *pvpython*, which can be used to control *ParaView* from the command-line.

The visualization process of *ParaView* relies on a pipeline to produce output. Pipelines consist of sources, filters and writers. A source, also called reader, serves as the basis of the pipeline. It loads the input data from a file, a series of files, or other origins. Its output then serves as the input to one or more filters, which further process the data. The output of filters can then be again the input for other filters, creating a tree-like pipeline of filters, with the reader at the root node. With this concept, *ParaView* can be used to construct complex pipelines, enabling complex visualizations and analyses. Optionally, writers can be used to write the output of the pipeline, e.g., as an image, to disk. Sources, filters and writers provide properties, which can be used to configure it. An example pipeline is shown in Figure 3.2.

3.1.1 Catalyst

Catalyst [18] is the *in-situ* processing library of *ParaView*. It can be used to enable in-situ processing with *ParaView*. This is a vital feature for *JuViz* to be able to access large data from simulations.

Catalyst is both an API specification and complementary library for *ParaView*. It is executed using a *pvpython* script, which receives the input data from the simulation and generates images and videos.

While the first version of *Catalyst* only consisted of the software library, a more general approach was chosen with the release of the second version. Instead, *Catalyst* now refers to the API specification, while *ParaView Catalyst* is used to denote the default implementation from *Kitware*. The *ParaView Catalyst* implementation only allows for

[2]https://docs.paraview.org/en/latest/Tutorials/SelfDirectedTutorial/basicUsage.html

visualization concurrently running with the simulation, it is not capable of receiving and processing data generated from simulations on other nodes (in-transit).

To enable in-transit processing, *Catalyst* can be combined with in-transit libraries. At the JSC, *ADIOS2* [6] is being established, which allows the transmission of simulation data between nodes or systems through the *Sustainable Staging Transport (SST)* engine. The new *adios-catalyst* [3] implementation leverages *ADIOS2* to natively support in-transit processing with *ParaView*.

3.1.2 trame

trame [24] is a web framework, which is part of the official *ParaView* ecosystem and developed by *Kitware*. In *JuViz*, it is used to enable users creating a custom visualization app, that can then be accessed in the UI. This is a crucial step, as this transforms *JuViz* into a flexible, adaptive solution, where users are free to create visualization workflows in arbitrary formats.

trame allows the development of full-stack web applications in Python. It is designed with a heavy focus on visualization, providing extensions for many popular frameworks besides *ParaView* and *VTK*, e.g., *matplotlib* [10]. It also allows the usage of other web-based technologies, like the *VSCode text editor* [26].

trame is available as a set of Python packages, that implement different aspects of its functionality. Its frontend is based on *Vue.js* [4], an open-source front-end web framework, designed for creating single-page applications. *Vue.js* achieves reactivity (automatically updating the UI when values of elements change) through a central *state*, that is used to store all UI-relevant variables and update elements in the UI if related values in the state change. *trame* also provides a web server, based on the library *wslink* [5] from *Kitware*. This *server* is not just used to host web-pages and libraries for the frontend, but also allows the client to make remote procedure calls (RPC) to the backend. The communication in *wslink* is primarily implemented over a *websocket*.

While *trame* only serves as a framework for creating custom apps, *Kitware* also provides a default app, the *ParaView Visualizer* [20]. It is designed as a simplified *ParaView* UI for the browser, providing all the basic functionality of the *ParaView Client*. It allows users to load data, apply basic filters to them and modify the properties of all filters.

[3]https://gitlab.kitware.com/paraview/adioscatalyst
[4]https://vuejs.org/
[5]https://github.com/Kitware/wslink

3.2 JupyterLab

JupyterLab [23] is the underlying software for the *JupyterJSC* cloud platform. It serves as a browser-based development environment, running on the systems of the JSC. By running inside *JupyterLab*, *JuViz* becomes detached from local installations and is instead rendered in the browser. Thus, it becomes independent of the operating system.

JupyterLab is a powerful, extensible, interactive web-based integrated development environment (IDE), which is designed for data science, scientific computing, and research. It serves as the successor to *Jupyter Notebook*, providing a new environment with extended interactions. *JupyterLab* extensions can be added to extend the environment with new functionalities.

Jupyter Notebook is a file format containing a mix of code, text, interactive elements and other forms of media. It is derived from the *IPython* package — an improved interactive Python command-line shell — which can display more complex outputs, e.g., interactive media, plots, etc., and provide more advanced features compared to the basic Python shell. While originally developed for Python, *Jupyter Notebook* can nowadays be used with many different languages.

3.2.1 JupyterJSC

JupyterJSC is a cloud-platform developed at the JSC and is based on *JupyterLab*. It allows starting a *JupyterLab* on the HPC systems of the JSC and gives users access to a development environment, which runs on the cluster itself. It handles the automatic forwarding and encryption of the spawned instance to the user and checks for authentication with the identity management system of the JSC.

It is based on *JupyterHub* [6], a server for spawning, managing, and proxying *JupyterLab* instances for multiple users. It is based on the ideas of *authenticators* and *spawners*. *Authenticators* are used by *JupyterHub* to ensure that a connecting user is authenticated. *Spawners* handle the launching of *JupyterLab* on different systems. *JupyterJSC* provides multiple spawners, depending on the HPC system that was selected by the user.

In recent years, *JupyterJSC* has become increasingly popular for users of the HPC systems and has become established as a central hub for all HPC-related services. This brings a welcoming opportunity to establish a web-based service upon.

[6] https://jupyterhub.readthedocs.io/en/stable/

3.2.2 jupyter-server-proxy

Jupyter Server Proxy (JSP) [12] is required by *JuViz* to proxy the launched trame app to the user. It allows accessing the app in the browser.

JSP is an extension for *JupyterLab*, that allows executing arbitrary programs alongside *Jupyter* and forwarding resulting ports to the browser. In addition to launching and forwarding given processes, it allows users to access arbitrary ports on the machine where *JupyterLab* is running on.

3.3 jupyter-trame-proxy

As part of a previous work, the *jupyter-trame-proxy*[7] Python package has been developed. It laid the foundations for *JuViz* by allowing a user to launch the *ParaView Visualizer* in *JupyterJSC* and accessing it in the browser.

It brings *trame* to *JupyterJSC* / *JupyterLab* by utilizing *JSP*. It works by adding a new button to the *Launcher* window of the *JupyterLab*, that opens a *ParaView Visualizer* a new browser tab. This instance will automatically be opened on an available open port on the machine. It allows accessing *trame* from the browser by just a single click.

JuViz further builds on *jupyter-trame-proxy*. While *jupyter-trame-proxy* is limited to a single *trame* app (the *ParaView Visualizer*) and a single instance of it, *JuViz* expands this concept. It provides a complete interface, capable of managing multiple different *trame* apps and their respective instances. *JuViz* further allows connecting these *trame* apps with a *ParaView Server*, enriching the *trame* apps with the ability to render on a remote *ParaView Server*.

[7] https://github.com/jwindgassen/jupyter-trame-proxy

4 Development

The development process of this work encompassed two distinct components, the combination of which we call **JuViz (Jupyter Visualization)**. It consists of a *JupyterLab* extension called **JuViz Extension**, supplemented with a default *trame* app **JuViz App**. Additionally, a second, novel *trame* app, the **trame ESM-Slicer**, has been developed, to study and assess strengths and challenges of the development process. The three components were merged at the end of the development process and installed on the *JUWELS Booster* [13] HPC system at the *JSC*.

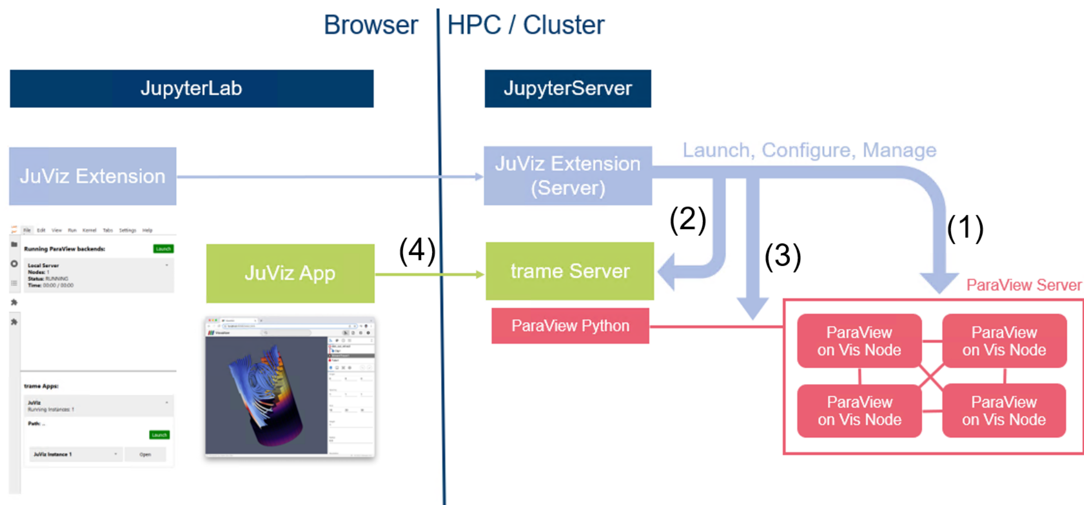


Figure 4.1: Figure 1.1 with steps outlining the *JuViz* functionality

The extension for *JupyterLab*, called *JuViz Extension* (see Figure 4.1, left, light blue), was the first target of the development process. Its primary function is to serve as a control interface, enabling users to start, configure and connect both *ParaView Servers* (backend, Figure 4.1, step 1) and *trame* instances (frontend, Figure 4.1, step 2). When launching both *ParaView Server* (Figure 4.1, bottom right, red) and *trame*, a user can opt to pair and establish connection between them (Figure 4.1, step 3), enabling remote rendering of the visualizations. *trame* can be accessed on the users' machine in the browser, where we use *Jupyter Server Proxy* to forward the *trame* server launched on the cluster to the user (Figure 4.1, step 4). Notably, both instances are designed to be

launched independently, as both of them can be used separately, and they do not strictly depend on one-another.

JuViz supports a simplified setup for smaller jobs and testing. Here, *trame* can be used without the need for an HPC-capable rendering server, and its required resource demand and startup time, attached to it. Standalone *ParaView Servers* can be useful if working with the desktop *ParaView Client* (either in remote desktop or through SSH) is preferred. Starting *ParaView Server* through *JuViz Extension* allows a centralized, unified server configuration on the whole system, resulting in an optimized configuration. Anyway, the essential concept of customizability comes into play if the ability to create custom apps is used. The customizability of *trame* is incorporated into the extension, allowing a user to add, and subsequently use, their own app to the extension, analogous to the default one.

JuViz App (see Figure 4.1, center, green) is derived from the *ParaView Visualizer* and tailored to the extension. It is provided as the default implementation for the extension. This also serves the purpose of providing an example for developers of new apps, as well as providing the basic structure for apps, which has been extracted into an empty template as a starting point for developers.

The *ESM-Slicer* is a specialized *trame* app, developed for the Earth-System-Modelling community, to visualize and analyze atmospheric data. In this context, the data was generated by the *Icosahedral Nonhydrostatic (ICON) Modelling framework* [29], designed for global numerical weather prediction, as part of the *ECMWF Reanalysis v5 (ERA5)* [9]. The development process was not restricted to the mere creation of the app, but also aimed to assess the strengths and weaknesses of *JuViz* in the visualization context. This encompasses the evaluation of development speed and flexibility when programming, but also highlights potential complications and their solution that a developer might encounter.

4.1 JuViz Extension

JupyterLab extensions, in general, consist of two sub-packages that separate the code into frontend – backend parts:

The frontend part is obligatory and contains code, encompassing JavaScript/TypeScript and CSS files. Its role is the construction of the user interface in its integration into *JupyterLab*. It is executed within clients' web browser. It is responsible for interaction with the user and communication with the backend.

The backend package is optional but necessary for this application. It is implemented

as a *JupyterLab* server extension in Python, which is loaded by the server upon its initialization. It is primarily responsible for configuring the API endpoints required by the frontend and delivering all the required functionality. In this context, it not only supplies the API routes, but also discovers, configures and launches *trame* apps, orchestrates the starting of *ParaView Servers* and initializes pairing *trame* and *ParaView Server*, as directed by the user in the *JupyterLab* extension.

The following sections show the implementation of the *JuViz Extension*. Its source code can be found on GitHub at https://github.com/jwindgassen/jupyter_viz_extension.

4.1.1 Implementation of the JupyterLab Extension

The frontend code is implemented in TypeScript and creates a sidebar panel in the left bar of *JupyterLab*. The entry point is an object inheriting from `JupyterFrontEndPlugin`, where the `activate` function is used to register the panel to the Lab. The panel encompasses two segments, splitting the UI into backend and frontend-related input.

Code 4.1: *JupyterLab* extension enty point

```
1  const plugin: JupyterFrontEndPlugin<void> = {
2    id: 'juviz-extension',
3    autoStart: true,
4    activate: (app: JupyterFrontEnd) => {
5      const panel = new SplitPanel();
6      // ... Panel Configuration
7
8      // ParaView Segment
9      const paraViewSegment = ReactWidget.create(<ParaViewSidepanelSegment />);
10     panel.addWidget(paraViewSegment);
11
12     // trame Segment
13     const trameSegment = ReactWidget.create(<TrameSidepanelSegment />);
14     panel.addWidget(trameSegment);
15
16     app.shell.add(panel, 'left');
17   },
18 };
```

While most of the official *JupyterLab* frontends use the native component library *Lumino*[1], most of the written code for this extension is written in React, a popular frontend JavaScript Library for UIs. It relies on the custom syntax extension to *JavaScript*, the **JavaScript Syntax Extension (JSX)**, to declare and manage reactive data and dynamic UI components. This significantly alleviates the requirement for manual DOM

[1]https://lumino.readthedocs.io/en/latest/

manipulation in JavaScript to create and modify the page. The resulting Interface can be seen in Figure 4.2

4.1.2 Implementation of the JupyterLab Server Extension

A central data management class named `Model` was created, responsible for managing the relevant data. It maintains the lists of the available *trame* apps, that can be launched with the extension, and their running instances, as well as *ParaView Server*s that have been dispatched. It provides the API for configuration and initialization of new instances. Given that a significant portion of this behavior is platform-specific, e.g., launching a new *trame* instance or listing running backends, it relies on an instance of the `Configuration` class for many of the functions. The instantiated `Configuration` class can be selected before the start of *JupyterLab* by the `JUVIZ_CONFIGURATION` environment variable. Available configurations are located in the `jupyter-viz-extension.configurations` submodule. To accommodate new platforms, custom configurations can be added by inheriting from `Configuration` and overwriting the relevant methods. The development process encompassed a configuration for desktop used in local development (with `JUVIZ_CONFIGURATION ="desktop"`) and a configuration for the *JUWELS Booster* system [13] at the *JSC* (with `export JUVIZ_CONFIGURATION="jsc.desktop"`).

The class diagram for the configuration base class can be seen in Figure 4.3. Inheriting configurations can override any number of the provided methods. The configuration provides the methods called from the *Model*, i.e., `launch_trame`, `launch_paraview`, etc., so the complete launching process can be overwritten. It also extracts some functionality to separate methods, so small aspects of the procedures can be updated, e.g., the generation of the environment *trame* is launched with. An extended overview with comments outlining the methods' purpose can be found in Code A.1

(a) *ParaView* panel

(b) *ParaView* launch dialog

(c) *trame* panel

(d) *trame* launch dialog

(e) Dialog for connecting *trame* and *ParaView*

(f) *trame* panel after a connection with a *ParaView Serve* has been established

Figure 4.2: Screenshots of *JuViz* Extension showing from a - f the different UI components for launching and pairing *ParaView Servers* and *trame* apps

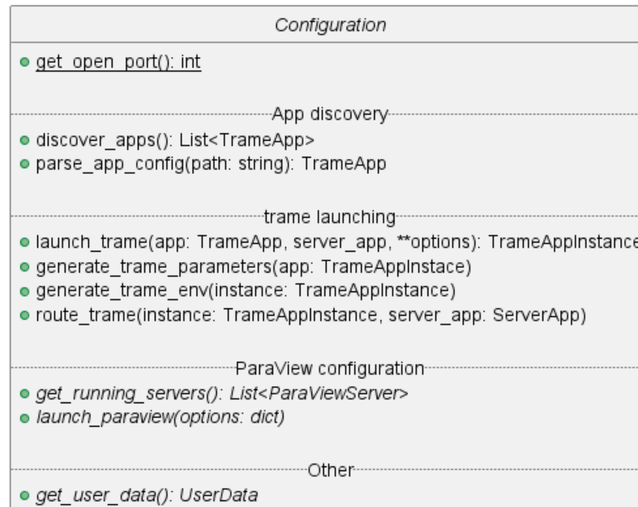


Figure 4.3: Class diagram for the Configuration class

The model is designed as a singleton and only instantiated once.

Central to the backend architecture are three API routes:

- The **user** route, accessible via `/user`, offers a single GET method, that provides the username and path to the home directory, as well as available partitions and accounts needed for submitting *ParaView* to the job manager.
- Regarding **trame**, two routes exist:
  - The first one, accessible under `/trame`, provides information about available apps and their active instances. It also handles the initiation of new app instances through a POST method.
  - The second route is parameterized, responding to URLs under `/trame/<action>`. The parameterization `action` is a string outlining what action should be performed on an instance; for instance, a POST to `/trame/connect` establishes a connection between the *trame* instance and server that are specified in the request body.
- The **ParaView** route, hosted at `/paraview`, is analogous to the *trame* route and can be called to receive a list of all currently running *ParaView* backends on GET. It, as well, launches new *ParaView* instances via a POST request.

Each extension's routes are organized within its namespace. For *JuViz*, this namespace

is `jupyter-viz-extension`. For instance, the user route is reachable under `https://localhost:8000/jupyter-viz-extension/user` (supposing the *JupyterLab* server runs on `https://localhost:8000`).

Within *JupyterLab* server, each route consists of a class inheriting from `APIHandler`. These route classes are registered to the server with a *Regular Expression* matches incoming requests. A reference to the previously constructed `Model` is also provided to enable the Handler to access it. When an incoming Request is matched by the given expression, the provided class will be instantiated and the `initialize` method will be invoked with the `Model` as a parameter.

The implementation of the `User APIHandler` is shown in Code 4.2.

Code 4.2: User API Handler

```
1  class UserHandler(APIHandler):
2      _model: Model
3
4      def initialize(self, model):
5          self._model = model
6
7      @authenticated
8      async def get(self):
9          user_data = await self._model.get_user_data()
10         self.log.debug(f"{user_data}")
11
12         await self.finish(user_data._asdict())
```

To launch a *ParaView Server*, the job scheduler, responsible for distributing the jobs of all users on the system, is used. On *JUWELS Booster*, this job scheduler is *SLURM* [28]. A job script with placeholders (see Code 4.3) will be filled with the parameters sent to the server in the request. The *Jinja2 Template Engine*[2] is available as a dependency of *JupyterLab* server and used for generating the job script. The temporary directory for the filled-in job script and the files containing the console output of the job will also be automatically generated by *JuViz Extension*.

[2] https://jinja.palletsprojects.com/en/latest/

Code 4.3: *ParaView* template script (except)

```
1  #!/bin/bash
2  #SBATCH --account={{ account }}
3  #SBATCH --job-name="{{ name }}"
4  #SBATCH --output={{ stdout }}
5  #SBATCH --error={{ stderr }}
6  #SBATCH --nodes={{ nodes }}
7  #SBATCH --time={{ timeLimit }}
8
9  # prepare environment
10 ...
11
12 # Start ParaView Server
13 srun --cpu_bind=verbose,rank pvserver --mpi --force-offscreen-rendering --
       multi-clients --displays='0,1,2,3'
```

4.1.3 Loading user trame Apps

A *trame* app usually consists of one or more Python files, often accompanied by data files like images or models. While one central file usually serves as the core of the application and constructs the app, various components can be extracted into separate files to maintain code cleanliness and facilitate better maintenance. Moreover, developers of an app might want to customize the launching behavior, like the setup of the environment or adding additional arguments to the program. While some of these modifications could be done in the main script, certain values need to be known before launching the app and must be stored in a format possible to be parsed before the app is executed. Considerations must also account for both system administrators providing certain apps, and users installing their own apps. In summary, a flexible approach to discovering available apps and their configuration, supporting users adding their *trame* app, as well as the addition of system-wide apps is required.

Various methods were evaluated:

- Passing a list of paths to configuration files as an environment variable, similar to `PATH`. This approach is fairly flexible and allows adding apps by software modules of the system, but requires the modification of the environment variable for each new app added by the user themselves, which can become laborious in the long term.

- Adding all apps to a predefined directory for each user, e.g., in `~/.local/trame/`, also allowing soft links for shared apps. Opposed to the method above, this approach can easily be utilized for adding user apps. But adding apps from software modules is difficult and required modifying a users' home directory.

- Allowing a user to add apps to *JuViz* in the UI. This approach also has only limited support for apps from software modules. Also, a persistent storage is required to store the added apps.

The chosen approach is a combination of the first two methods and works similarly as *Jupyter*s' process of discovering new kernels. This not only integrates smoothly into the existing *JupyterLab* ecosystem but also aligns with the installation of system-wide kernels, where the approach is already used for installations. With this approach, new apps can be added via the `JUPYTER_PATH` environment variable. It contains a list of paths that are examined by *Jupyter* for new kernels and, by extension, *trame* apps. Users can freely modify this path prior to starting *JupyterLab*, adding the location of a new app to it. `~/.local/share/jupyter` is the first entry on `JUPYTER_PATH`, serving as a default location for kernels and *trame* apps.

Within the specified paths, each *trame* app is placed in a separate subdirectory inside the `trame` directory, as seen in Figure 4.4, while kernels will similarly be placed in the `kernels` directory. Within this directory, a `app.yml` file is used for configuring the app. It contains essential details, i.e., the display name of the app (shown in the UI) and the launch command executed when starting this app. Optionally, it can contain a working directory from where the specified command will be launched. The reliance on a user-defined launch command empowers developers of apps with full control over the launching behavior, as they are free to prepare the environment or set the final Python execution command as required. Server-generated information, like the destined port and the path to the file containing the authentication key, are added to the Python execution command via the `JUVIZ_ARGS` environment variable.

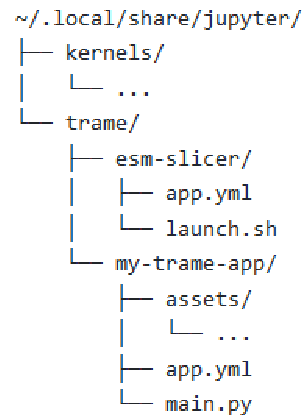


Figure 4.4: Directory tree for adding *trame* apps to *JuViz*

Code 4.4 shows the config file for the *JuViz App*, installed with *JuViz Extension* (see 4.3). Furthermore, Code 4.5 shows the launch script, responsible for loading the required modules and launching *JuViz App*. Since `app.yml` contains a `working_directory`, the launch script will be executed from this specified location.

Code 4.4: *JuViz* app.yml

```
name: JuViz
working_directory: ".../JupyterExtension-JuViz/0.3.1-GCCcore-11.3.0-2023.3.6/
    lib/python3.10/site-packages/juviz"
command: "source .../JupyterExtension-JuViz/0.3.1-GCCcore-11.3.0-2023.3.6/
    share/jupyter/trame/juviz/launch.sh"
```

Code 4.5: *JuViz* launch.sh

```
#! /bin/bash

# Load required modules
module purge
module load Stages/2023
module load GCC/11.3.0
module load ParaStationMPI/5.7.1-1
module load ParaView/5.11.0-EGL
module load trame/2.5.2

python __init__.py $JUVIZ_ARGS
```

4.2 JuViz App

JuViz App is the default *trame* app for *JuViz*. It features a basic set of functionality and is based on the *ParaView Visualizer*. This app is extended by a new *trame* widget, the **JuViz Widget**, which is required for integrating a *trame* app into *JuViz*. This widget serves a dual purpose: Firstly, it constructs the RESTful API endpoint for *JuViz Extension* to connect to. Secondly, it constructs a UI that allows the user to manually connect to a running *ParaView Server*. All *trame* apps used with the *JuViz Extension* must initialize this widget to construct the API endpoint and enable controlling the app from inside the extension. Lastly, Some minor adjustments were added, exchanging the Logo to the official FZJ Logo in the asset manager and refactoring the repo to contain a single main file that creates the extension.

For optimal development simplicity, the app introduces a single submodule `juviz.widgets`. This submodule serves as a single repository for all accessible widgets sourced from both the *ParaView Visualizer* and the custom *JuViz* panel. Any forthcoming widgets will also be integrated into this submodule, ensuring a unified infrastructure for accessing these widgets.

The source code of *JuViz App* can be found on GitHub at https://github.com/jwindgassen/juviz-app.

4.3 Installation and Distribution

The parts of the *Jupyter Extension* – frontend and backend – are separately distributed via the official *Python Package Index (PyPI)* and the *Node Package Manager (npm)* registry. Publishing is automated by GitHub Workflows, provided by the template of the *Jupyter* extension repository [3]. After appropriate configuration was performed, both *JuViz Extension* and *JuViz App* are automatically build and uploaded to *PyPI* and the *npm* registry. *JuViz Extension* can thus be simply installed via the respective command-line interfaces, *pip* and *npm*.

At the *JSC*, *EasyBuild* [5] and *LMod* [14] are responsible for installing and managing software, respectively, on the system. As part of this work, the *JuViz Extension* and *JuViz App* were installed onto the *JUWELS Booster* system at the JSC.

JuViz is installed using an easybuild script, downloading and configuring both *JuViz Extension* and the *JuViz App* from *PyPI*, as well as creating the necessary folder structure and files to make the *JuViz App* discoverable by the extension when starting. To keep the environment during the startup of *JupyterLab* as small as possible, loading the software modules required for launching *trame* (ParaView, matplotlib, etc.) is delayed as long as possible. Loading unnecessary modules that are not required for the extension itself will lead to longer loading times and a bloated default environment for all processes spawned by the extension. Any dependencies required by the apps, i.e., the *trame* packages or *ParaView*, are loaded when the app is launched as part of the launch command. To facilitate easier overview of the module, the launching is delegated to a `launch.sh` file alongside the `app.yml` and will be sourced by the launch command.

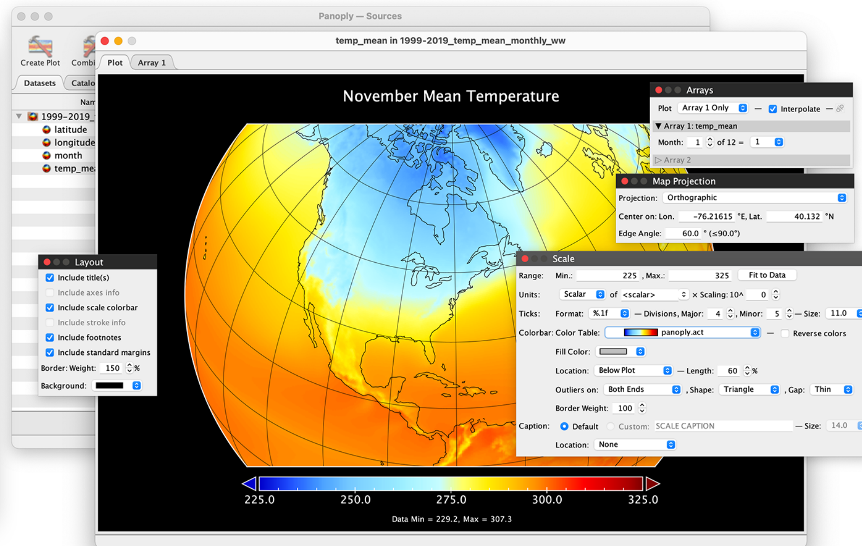
The *JuViz App* is additionally installed to the *trame* module, so creators of new apps can use the widgets from `juviz.widgets`. The *trame* module also contains all available *trame* extensions available from the Kitware Repositories and the *ParaView Visualizer*.

4.4 trame ESM-Slicer

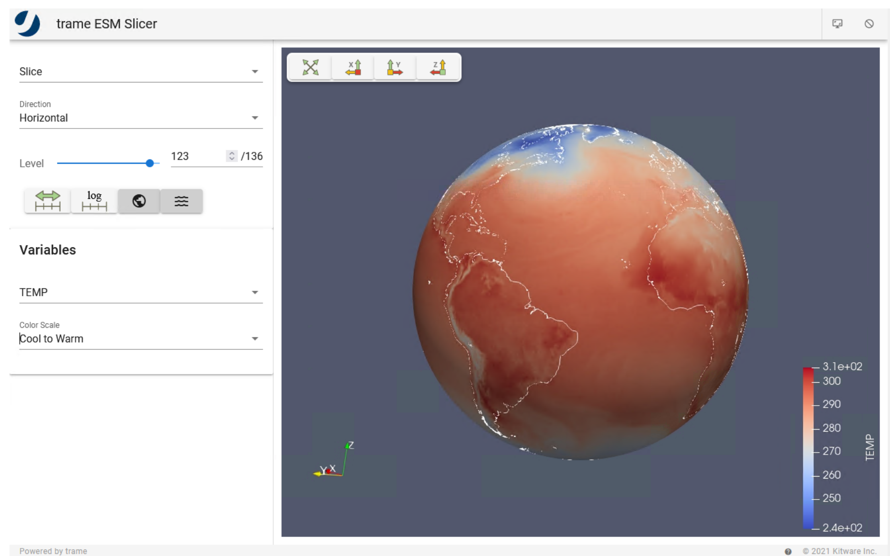
The **Earth-System-Modelling (ESM)** community is concerned with the analysis and simulations of the earths' atmosphere. They are represented at the JSC through the **Simulation and Data Laboratory Climate Science**. Previously, the *Panoply Data Viewer* [17], a closed-source, cross-platform data viewer for common ESM Formats, was relied upon for visualization. With the exponentially growing size of the simulations, the program began struggling with large data and could often only be used with preemptively shrunk data to maintain usable. In this context, a second *trame* app, the **trame**

[3]https://github.com/jupyterlab/extension-cookiecutter-ts

ESM-Slicer, was developed to address these needs, the development process of which was used to document and evaluate the capabilities of the *trame* framework.



(a) Panoply [17]



(b) trame ESM-Slicer

Figure 4.5: Visual comparison of the *Panoply* and *trame ESM-Slicer* User Interface

4.4.1 Requirements and Desired Features

During an initial discussion with colleagues from the community, a list of crucial features, that software for visualization and analysis of ESM data requires, were identified:

- The program should be able to handle large data and run on HPC systems, to allow visualization of large datasets.
- The input data usually comes in a *netCDF (Network Common Data Form)* [25] or *GRIB (GRIdded Binary)* [16] format, that the program must be able to read from. *netCDF* is a self-describing, machine-independent data format specialized for large arrays. *GRIB* is a similar format, predominantly used in meteorology, climate, and weather forecasts.
- For visualization, the program should be able to slice through the given dataset horizontally and vertically, where the latitude, longitude, and height level of the slice should be selectable by the user. For the initial design, a vertical slice along latitude or longitude was sufficient.
- Along with the slices, the user should be able to select one of the available variables from the dataset, by which the slice should be colored.
- The slice generated should be both displayable as a 2D plane and a 3D sphere.
- In addition to the slice mode, a mode for showing plots along a latitude, longitude, or vertical line was requested. Here, the variable selection could select the value shown in the plot.
- A mode for analyzing points along the time series over multiple time steps was requested. Multiple time steps can be split over multiple files or loaded as a single large file. Additionally, this should also include a selection for the time span examined.
- After the slices/plots are generated, a way to automatically screenshot and export the generated images should be available. For the plots, the export process should also produce the plot data, e.g., in CSV format.

4.4.2 Development

The development of the *trame* app was primarily located on a local desktop machine. While the default *ParaView* distribution contains a basic reader for *netCDF* files, the

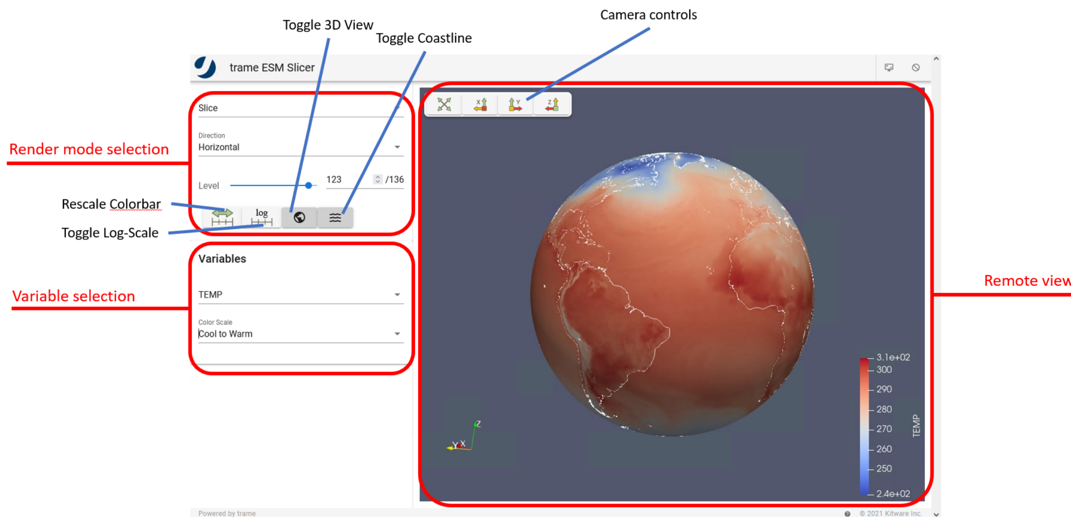


Figure 4.6: Schematic overview of the *trame ESM-Slicer* UI

installation on the *JSC* systems additionally provides the *CDI Reader* [2] [19], a more in-depth parser for *netCDF* files, able to also handle the associated metadata. For the initial development, the native *netCDF* reader was used, and the switch to the *CDI Reader* was added when the finished program was installed on the HPC system.

The *trame* layout (see Figure 4.6) usually contains a header and footer at the top and bottom. The footer is limited and mostly used for miscellaneous information and status indicators. The header is used for displaying the time step controls, as well as the controls to open and load input file(s) and the button to export the screenshot. Next to the main area, used for displaying the remote view, a side panel is shown, that can be opened and closed.

The side panel contains two sections: The first one is used to select and modify the current render mode and its parameters. It is used to switch between slice and plot mode and renders the appropriate controls for selecting the parameters, i.e., direction, latitude, and longitude of the slice or plot. Below is a collection of buttons, that can be used for switching other features of the visualization, e.g., the view between 2D and 3D or the logarithmic color scale for slices. In the second section, available variables of the dataset can be selected. While this selection is used in both slice and plot mode, a color scale can additionally be chosen in slice mode.

The main area of the page is occupied by the remote view, displaying the output of the remote rendering. By default, the *ParaView* render view contains a gizmo on the bottom left, showing the current orientation, and the color bar on the bottom right, for the currently selected variable. While the remote view allows default 3D movement (rotate,

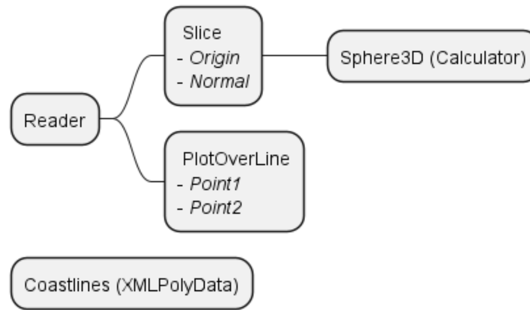


Figure 4.7: Pipeline structure of the *trame ESM-Slicer*

zoom) around the space, during slice mode the camera uses 2D controls, replacing the rotation action with panning, for comfortable control of the slice. On top of the viewport are camera controls, allowing the camera to snap to the three major axes and recenter the dataset.

After an input file is loaded, the full pipeline (shown in Figure 4.7) is constructed. Changes to the parameters (italic values) will result in appropriate changes to the pipeline filters, but the pipeline structure will remain static. The reader constructs the loaded data as a uniform rectilinear grid (image data format), to facilitate easier handling of the slice and plot filter. The *ParaView Calculator* is used to transform the coordinates and show the slice as a 3-dimensional sphere.

When viewing the sphere in 3D, coastlines can be shown on the globe to provide orientation. The coastlines are loaded by a separate reader as spherical data [4].

During the development process, another approach with multiple, resizable, dynamically addable, and removable remote views was experimented with. The structure of the app is designed with this feature in mind, so it can be added in upcoming work. Due to time constraints, however, the *ESM-Slicer* shown as part of this work, utilizes a single remote view.

4.4.3 Challenges and their Solution

- Handling the *ParaView* pipeline became error-prone, as it was accessed in multiple different files for different function calls. All handling of the pipeline was extracted to a single file and provides functions that change the variable, render mode, and configuration.

[4]https://www.earthmodels.org/date-and-tools/coastlines/world-coastlines-and-lakes

- Initially, filters were constructed dynamically on the first usage, resulting in overly complicated function calls. Instead, an approach is used, where the complete pipeline is constructed after the file is loaded. After the pipeline is constructed, switching between slice and plot or 2D/3D view, is reduced to hiding and showing datasets.

- Instead of manually constructing a pipeline with the *ParaView* Python API, a *ParaView* state file can be used. These files can be exported from the *ParaView* client and contain the state constructed there. This way, less manual setup work is required, and all properties can be tested and configured in the client.

- To separate backend and frontend related functions, all functions that use *ParaView* are put in a separate file with the pipeline, while all callbacks affecting the UI were kept in the file, where the connected UI was constructed.

- When constructing the UI, especially when experimenting with multiple views, it became very hard to differentiate between code run on construction on the server and code executed by *Vue.js* on the client. Special care was taken with loops constructing elements and the parameters they provide.

4.4.4 Summary

For the development of the *trame ESM-Slicer*, including the initial and follow-up meeting, less than two weeks of development time was sufficient. This time period was adequate for creating a small *trame* app, replicating the core features of *Panoply*, and also adding some supplementary functionality. Notable, the amount of boilerplate code used was trivial, and the app could be constructed without implementing any explicit communication between the backend and frontend.

Very positively stood the fact, that no intermediary build steps were necessary between implementation and access of the app, and starting the program is enough to take care of all prerequisites usually dedicated to building. The flexibility when making changes to the app remained high throughout the complete development process. Changes to the UI could be implemented in a few lines of Python code. While some latter features required more in-depth restructuring of the app, some features were trivially implementable and required minimal change.

Although the required knowledge for using *trame* is mostly Python, a general understanding of HTML and CSS is required to properly configure and design the website. Moreover, knowledge of *ParaView*, understanding of the *ParaView* Python API, the software structure of the pipeline, and experience with data processing in *ParaView* was required. While this was primarily governed by the fact *ParaView* was used to visualize

the data, some knowledge will certainly be required, even for applications not relying on *ParaView*.

In conclusion, the development of a visualization application can become significantly simplified when relying on *trame*. Opposed to traditional development techniques, novel visualizations or reimplementing existing visualization apps require less work and are fast and more flexible to develop. With the usage of *ParaView* also comes an optimized framework for lifting the heavy calculations, enabling apps that are capable of executing on HPC and large datasets.

5 Discussion

In this chapter, the core design decisions taken during the development and design of the architecture are discussed. For major design decisions taken during the development, advantages and disadvantages will be outlined. Potential solutions for the disadvantages, both already implemented and not implemented yet, will be reviewed where applicable. Oftentimes, alternative frameworks and approaches exist but have not been chosen for good reasons. These reasons will also be discussed.

These points are oriented by the aspects constituted in Visualizing Large Data for HPC.

5.1 JSC Concerns

JSC concerns are primarily from the viewpoint of a supercomputing centre. While they can, most definitely, be relevant for some users, most of them will only be secondary concerns to the average user. Three of the most significant aspects of the decisions made during development were chosen for this discussion:

HPC Compatibility of *JuViz* with HPC. This encompasses the performance and scalability for large datasets, as well as support for in-situ and in-transit processing.

Maintainability How well *JuViz* can be maintained by the software maintainers and the system administrators, and how easy software updates and patches can be enrolled to all users.

Security Security aspects of *JuViz*, especially for multi-user systems, and how they are addressed.

5.1.1 HPC

One central design decision for the setup is the reliance on *ParaView*.

ParaView is a well-established software in scientific visualization and is widely used. It has a large community and support for numerous application areas and file formats. *ParaView* is extensible and new functionality and formats can be added. Many communities have developed plugins and extensions for *ParaView*, allowing it to work with unfamiliar formats. As example can be found in the *trame ESM-Slicer*, which relies on the *CDI-Reader*. While the reader itself is part of the official *ParaView* source repository, it is not part of the officially shipped versions and must be explicitly enabled during the building process, given that the required libraries are available on the system.

Crucially, *ParaView* is designed to be used on HPC, due to the separation of client and server and the subsequent usage of *MPI*. *ParaView* is also able to process in-situ when using *Catalyst* (see chapter 3).

With the far-reaching support for formats and processes of *ParaView* also comes some costs. Notably, *ParaView* comes with excessive overhead, both regarding loading times and software size, which is often not required for small projects. While this holds true for all *ParaView* bound applications, *trame* apps do not necessarily rely on *ParaView* and can work with other visualization frameworks, some of them being more lightweight. *trame* apps can, e.g., rely on *matplotlib* [10] for plotting, or use the official *deck.gl* [27] extension, to allow data visualization with *WebGL*.

ParaView is also often associated with a steep learning curve with the complex structure it comes with. For new developers of *ParaView* working with more than the basic filters, developing *trame* apps using *ParaView* can be challenging. With the abstraction layer *trame* provides, however, the users of apps are not required to have previous knowledge of *ParaView*, as these aspects are abstracted by the developer of the app.

Another choice was the usage of *JupyterJSC* and subsequently *JupyterLab*.

JupyterJSC is developed in-house, which allows close communication with the developers for any problems occurring. It has been explicitly tailored to the systems of the JSC and runs on the cluster. It is a well-established interface for the users of the HPC systems at the JSC and provides a feature-rich, interactive UI for the browser. *JuViz* integrates smoothly into this central hub for HPC-related services, running as a *JupyterLab* extension, spawned by *JupyterJSC* during startup.

One disadvantage of collecting HPC-related services in a single point, like *JupyterJSC*, is the creation of a single point of failure. Since *JuViz* relies on *JupyterJSC* as platform, in case of a *JupyterJSC* failure, *JuViz*, and all related services running within *JupyterJSC*, cannot be accessed anymore.

5.1.2 Maintainability

One key concepts of the *JuViz* approach is to rely and build on top of other software frameworks, instead of reinventing the wheel with another visualization framework. *JuViz* serves as a connection between *ParaView* and *Jupyter*, gluing them together to allow interactive visualization, while preserving the possibilities both frameworks provide.

Relying on well-established framework, like *ParaView* and *Jupyter*, tremendously reduces the development effort otherwise required for implementing custom rendering software. Both ecosystems are widely popular, with a large company (e.g., kitware) and/or community behind them, providing stable, steady interfaces, and responsible for implementing new features and maintaining the existing codebase. This allows for simplified work by software maintainers, reduced to configuring and installing the software to the systems.

But relying on other software leads to dependencies for the created software, endangering the project in case they are discontinued. With the popularity of both *ParaView* and *Jupyter*, abandonment is unlikely.

Another pitfall for 3rd-party software is the distance to the developers: While self-maintained software can easily integrate new features or bug-fixes, for other software integrating updates can be difficult, relying on the software maintainers to accept or sometimes even develop requested changes into their software product. This is alleviated by the fact, that both software products are open-source (on GitHub and GitLab), allowing access to the code and requesting and integrating updates through *issues* a *merge requests*, repectively. With *EasyBuild*, small patches can be natively applied to any installed software on the system, allowing for quick changes without requesting permanent fixes to the program.

JuViz not only automates configuring and starting *trame* apps, but is also designed to dispatch *ParaView*. Since a central template script (see Code A.2) for *ParaView* is used, software maintainers are responsible for the configuration of *ParaView*. Some options for *ParaView*, e.g., switching between hardware and software rendering, are dependent on the system and out of reach for the average user of the system. Also, updates to software on the system, that lead to changes to the configurations being required, can be easily applied to the template, and subsequently, to all *ParaView* instances started with the *JuViz Extension*. However, this setup comes at the cost of allowing less customizability for the advanced user, needing to rely on the provided input option to express their requirements. To circumvent this, further configuration options will be added in future work and might also contain a way to directly modify the job script for the *ParaView Server* if desired.

5.1.3 Security

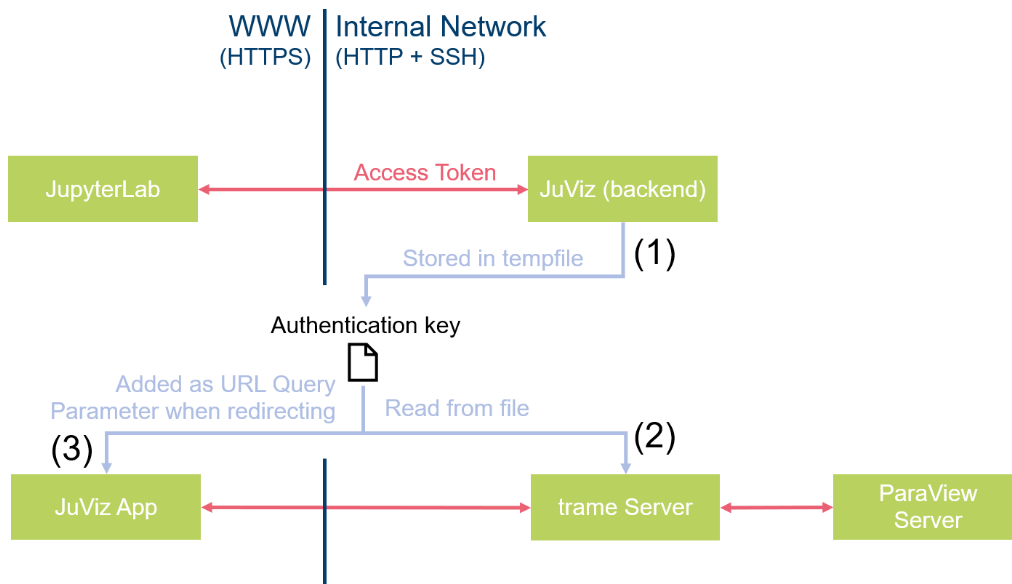


Figure 5.1: Diagram outlining the encryption and access restrictions for JuViz

For evaluating the security steps taken by *JuViz*, we differentiate between encryption of the transmission and authentication of the access.

For encryption, the major part of the *JuViz* communication runs in the internal network of the FZJ. These transmissions rely on plain, unencrypted HTTP. But access to the internal network is restricted by session keys and encryption with SSH between cloud and the HPC systems is the default. This simplifies the communication between the different components – *ParaView*, *trame* and *Jupyter*. As all data transmitted by *JuViz* runs through *JupyterJSC*, encryption will be added to all outgoing data, that leaves the FZJ and enters the World Wide Web. This is ensured by the HTTP-Proxy as gateway between internal and external networks.

While arguments can be made to ensure even the internal communication, this level of security was decided to be not vital. The access to the internal network is restricted, and *JupyterJSC* does not consider unauthorized access to the systems a risk to be considered on this software layer, but underneath.

For security, *JuViz* focuses on restricting access to the resources of the different components and their associated data. For *JuViz* apps, this is accomplished via an authentication key. It is generated with the start of each instance of an app and stored in a secure temporary file (Figure 5.1, step 1), only readable by the user themselves. It

is made available to the *trame* server through the `authKeyFile` command-line argument and will be read by the server during the startup procedure (Figure 5.1, step 2). *trame* servers are often run on the multi-user nodes of the systems (i.e., the login nodes), so the authentication key cannot be passed as plain text with the `authKey` argument, as all users of the system can inspect the call arguments for all processes running on the system through the `ps` command. The addition of this `authKeyFile` argument was already implemented in a previous work [1]. All clients connecting to this *trame* server must provide the authentication key as a URL query parameter to successfully establish a connection. This parameter will be automatically added by *JuViz* when opening the app through the interface, by relying on *JSP* (Figure 5.1, step 3).

For the *ParaView Server*, ensuring restricted access to the server is not possible with the currently provided command-line interface. While the *ParaView Server* allows adding a `connect-id`, this ID must be known to the client before startup and be passed as a command-line argument to the Python interpreter of the *trame* server. Since *JuViz* allows dynamically connection and switching between different servers, the ID cannot be known this early, rendering it infeasible for access restriction. Anyway, the risk of potential damage is considered small, as the attacker would need to have direct access to the compute node where *ParaView* is running.

5.2 User Concerns

Users' concern are, opposed to the JSC concerns, much more oriented towards usability and flexibility of the workflow. The following discussion presents advantages and disadvantages of the *JuViz* framework from the point of view of the users. For this, three aspects will be discussed:

Usability Interactivity, responsiveness, and accessibility on different platforms when working with *JuViz*, as well as simplicity of the installation and usage.

Communities Support for working with *JuViz* in communities. This encompasses the sharing of code, workflows and visualization pipelines for collaborating.

Flexibility and Compatibility How well can the program adapt to changing requirements or when being integrated into already existing workflows.

[1] https://github.com/Kitware/trame-server/pull/6

5.2.1 Usability

First, let's take a look at why *JuViz* uses *trame* and *JupyterLab*, which are both apps running in the browser.

When using remote desktop solutions, one major disadvantage is the high latency of interactive UI elements. This is due to the fact, that the UI is also rendered remotely, with all interaction needing to be transmitted twice, to and from the server, before they are shown to the user. This causes higher latency and a software, that feels less comfortable to use. With *JuViz*, however, the UI is rendered locally in the browser, which enables high-fidelity interaction. It should be explicitly pointed out, that the data is still rendered remotely, with the computing resources of the HPC system(s), due to the ability to connect to a *ParaView Server*. As a convenient byproduct, rendering the UI locally also reduces the size of the transmitted data, as only the rendered output from the visualization are sent and only a negligible amount of the data stream is retained for the UI. Rendering the interactive elements locally results in an improved user experience, as the app remains responsive and interactive.

By rendering the UI in the browser, developers must have experience with internet technologies, i.e., HTML, CSS and *Vue.js*, which are fundamentally required when creating new *trame* apps. This required knowledge comes as an addition to the required knowledge of the visualization framework, e.g., *ParaView*. These requirements, however, only pertains to the developers of apps. Users of *JuViz* are not required to have this knowledge.

While the usage of the existing *JupyterJSC* platform has also been discussed above, it yields further advantages related to the usability.

Since *JuViz* is build as an extension to *JupyterJSC* (or *JupyterLab* in general), a web-based interface has been created. Relying on the web browser eliminates the requirement of other local installations by the user. Web browsers are also available for the majority of platforms and operating systems, with developers not required to produce multiple executables for them.

5.2.2 Communities

A core idea of *JuViz* is to allow users to develop and use their own app with the extension, instead of providing a single app that is designed to work with all use-cases and requirements.

This enables scientific communities to develop their own workflows and visualization

programs, tailored to their specific needs and desires. It allows designing a UI, where essential functionalities can be placed easily accessible on the top level of the UI, where they can be accessed straight away. In the case of the *trame ESM-Slicer*, the selection of the variables of the dataset was deemed detrimental, and users can quickly switch between the variables when inspecting the data. Thus, a panel for this selection with a drop-down menu was position as a central part of the sidebar in the UI, where it can be directly accessed. In comparison, the corresponding actions in the *ParaView* client requires multiple steps: The filter responsible must be selected first, only after which users can select a variable in the properties of this filter. In more complicated cases, it might even be required to change the properties of multiple filter to achieve a single request. With the abstractions provided by the *trame* framework, this can be automated to only require a single press of a button.

While these custom apps allow for a large flexibility for a visualization workflow, the responsibility of implementing apps is moved to the communities themselves. While they might previously have been able to simply rely on an already written program, they are now required to invest development time into the creation of an app if they desire to use this advantage. The development effort required, however, is quite low, as was shown by the *trame ESM-Slicer*, which was completed from the ground up in under two weeks.

Created apps can easily be shared inside and between communities, making the distribution of workflows effortless. To share a *trame* app, it is sufficient to place them in a location where it can be read by all recipients and ensure they are found by *JuViz* during the startup process. This can either be done by extending the `JUPYTER_PATH` environment variable with the path to the app or by creating a directory soft-link in the `~/.local/share/juypter/trame` directory.

5.2.3 Flexibility and Compatibility

The ability to add user apps to *JuViz* also provides benefits to flexibility and integration of existing workflows.

trame apps are written in Python, a prominent language for scientific computing that most users are familiar with. As many workflows, that are utilizing other visualization frameworks, are already written in Python, they can often be integrated into *trame* without large adaptations. Already existing workflows in *ParaView* can be exported as a Python script from the menu of the *ParaView* client. Moving such a workflow to a *JuViz* app consists of importing this exported script to the app and possibly change the suitable values, so they adapt to input from the user. While the current implementation of the *trame ESM-Slicer* does not utilize this method, it will likely be added with a future update of the app. This incorporation also works with other Python workflow scripts, provided that *trame* can work with the utilized framework and display the data

generated.

As shown in 4.4.2, changes to the UI can be quickly added to an app, requiring only a few lines to be changed in the Python code. The same goes for new features in *JuViz* apps, which can often be implemented with only a handful of lines of code.

Due to the generic approach of building web-apps with HTML, *JuViz* could also be used with other types of visualizations, like dashboards or data explorers and viewers.

That being said, the existing extensions of the *trame* framework are skewed towards visualization applications, with other, more exotic usages requiring more development work for the implementation.

6 Conclusion and Outlook

This work was written with the goal of implementing a browser-based interface that combines *ParaView* and the cloud platform *JupyterJSC*. *JuViz* enables interactive visualization of large data sets in the browser without the need for complex setups and tedious manual work.

At the core of *JuViz* is *JuViz Extension*, an extension for *JupyterLab* integrated into *JupyterJSC*. Leveraging a browser-based interface, *JuViz Extension* eradicates the need for local installations, offering easy browser access. Through its interface, users can conveniently launch, configure, and manage both *ParaView Server* and *trame* applications. Since *JuViz* allows adding custom *trame* apps, this setup becomes very flexible and can be adapted to diverse use cases and requirements. The *JuViz app* also provided a default app that can be used in *JuViz*.

These customized apps enable both the recreation of existing visualization programs, as demonstrated by the *trame ESM slicer*, as well as novel visualization workflows. The customizability of a *trame* app further unlocks the potential for other forms of visualization, such as dashboards, editors, and viewers. *trame* also enables rapid development and quick changes to existing apps. Because *JuViz* is installed as a software module on JSC's HPC systems, software maintainers and system administrators can deploy patches and updates to all users, an important factor in the maintainability of a workflow.

Further work on *JuViz* will serve to improve the current software while also extending its functionality and providing new options for users. A critical aspect for enabling the handling of large data sets is the addition of in-situ and in-transit processing. This addition should provide both a suitable UI to enable in-situ and in-transit processing, as well as the necessary software structure on the *ParaView server* side. Possible configurations could allow the user to select among multiple methods for connecting to the simulation, with a user selecting appropriate software for the structure of their data. To allow in-transit processing, an *ADIOS2* endpoint could be run alongside the *ParaView Server*, which would enable in-transit processing of the data. The new *adios-catalyst* could be installed and tested to make this more straightforward.

Last but not least, *JuViz* needs to be extended with a way to share data and workflows more easily with other users. For example, this might allow other users to access both *ParaView Server*s and *trame* apps. To enable this, *trame* apps must not be started as

a sub-process of the *JupyterLab* server, but instead run as a *UNICORE* job. This has been skipped as it is beyond the scope of this paper, but it is an essential aspect for installing *JuViz* as *Visualization as a Service*.

Bibliography

[1] Janine C. Bennett et al. 'Combining in-situ and in-transit processing to enable extreme-scale scientific analysis'. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[2] *CDI - Climate Data Interface*. URL: https://code.mpimet.mpg.de/projects/cdi.

[3] Jerry Chen, Ilmi Yoon and Wes Bethel. 'Interactive, Internet Delivery of Visualization via Structured Prerendered Multiresolution Imagery'. In: *IEEE Transactions on Visualization and Computer Graphics* 14.2 (2008), pp. 302–312. DOI: 10.1109/TVCG.2007.70428.

[4] Hank Childs et al. 'A terminology for in situ visualization and analysis systems'. In: *The International Journal of High Performance Computing Applications* 34.6 (2020), pp. 676–691. DOI: 10.1177/1094342020935991.

[5] *EasyBuild - building software with ease*. URL: https://easybuild.io/.

[6] William F. Godoy et al. 'ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management'. In: *SoftwareX* 12 (2020), p. 100561. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100561.

[7] Aryaman Gupta et al. 'Efficient Raycasting of Volumetric Depth Images for Remote Visualization of Large Volumes at High Frame Rates'. In: *2023 IEEE 16th Pacific Visualization Symposium (PacificVis)*. 2023, pp. 61–70. DOI: 10.1109/PacificVis56936.2023.00014.

[8] A Henderson. 'Paraview guide, a parallel visualization application. Kitware Inc.(2007)'. In: *URL http://www.paraview.org* ().

[9] Hans Hersbach et al. 'The ERA5 global reanalysis'. In: *Quarterly journal of the Royal Meteorological Society* 146 (2020), pp. 1999–2049. ISSN: 0035-9009.

[10] J. D. Hunter. 'Matplotlib: A 2D graphics environment'. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[11] *JUPITER, The Arrival of Exascale in Europe*. URL: https://www.fz-juelich.de/en/ias/jsc/jupiter.

[12] *Jupyter Server Proxy Documentation*. URL: https://jupyter-server-proxy.readthedocs.io/.

[13] Stefan Kesselheim et al. *JUWELS Booster – A Supercomputer for Large-Scale AI Research*. 2021. arXiv: 2108.11976 [cs.DC].

[14] *Lmod: A New Environment Module System*. URL: https://lmod.readthedocs.io/.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[16] World Meteorological Organization. *Manual on Codes - International Codes, Volume I.2, Annex II to the WMO Technical Regulations: Part B – Binary Codes, Part C – Common Features to Binary and Alphanumeric Codes*. 2018.

[17] *Panoply netCDF, HDF and GRIB Data Viewer*. URL: https://www.giss.nasa.gov/tools/panoply/.

[18] *ParaView Catalyst*. URL: https://www.paraview.org/hpc-insitu/.

[19] *ParaView CDI-Reader (Source Code)*. URL: https://gitlab.kitware.com/paraview/paraview/-/tree/master/Plugins/CDIReader.

[20] *paraview-visualizer on GitHub*. URL: https://github.com/Kitware/paraview-visualizer.

[21] Dave Pugmire et al. 'Visualization as a Service for Scientific Data'. In: (Aug. 2020). URL: https://www.osti.gov/biblio/1659572.

[22] W Schroeder, K Martin and B Lorensen. 'The visualization toolkit, 4th edn. Kitware'. In: *New York* (2006).

[23] *The JupyterLab computational environment*. URL: https://jupyterlab.readthedocs.io/en/latest/.

[24] *trame Website*. URL: https://kitware.github.io/trame/.

[25] Unidata. 'Network common data form (netcdf) version 4 [software]'. In: (2019). DOI: http://doi.org/10.5065/D6H70CW6.

[26] *Visual Studio Code - Code editing. Redefined.* URL: https://code.visualstudio.com/.

[27] Yang Wang. *Deck.gl: Large-scale Web-based Visual Analytics Made Easy*. 2019. arXiv: 1910.08865 [cs.HC].

[28] A. B. Yoo et al. 'SLURM: Simple Linux Utility for Resource Management'. eng. In: *Journal on data semantics.* (2003), pp. 44–60. ISSN: 0302-9743.

[29] Günther Zängl et al. 'The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core'. In: *Quarterly Journal of the Royal Meteorological Society* 141.687 (2015), pp. 563–579. DOI: https://doi.org/10.1002/qj.2378.

Appendix

Code A.1: Configuration class Python slug with comments

```
class Configuration(ABC):
    @staticmethod
    def get_open_port() -> int:
        """ Query an open socket on the machine """
        ...

    def discover_apps(self) -> list[TrameApp]:
        """
        Search the system for trame apps that can be launched with JuViz.
        The default behavious will rely on the `JUPYTER_PATH`environment
         variable, where trame apps will be found in
        `trame/<app_name> directories`.

        @return: A list wich all the found trame apps
        """
        ...

    def parse_app_config(self, path) -> TrameApp:
        """
        Parse the app.yml file of a trame app. A default config file has the
         following values:

        - name: The display name of the app to be shown in the UI
        - command: The shell command that will be executed to lauch an
         instance for this trame app
            This command must append the $JUVIZ_ARGS environment variable to
             the python script, which provides
            some information for trame. See L{Configuration.
             generate_trame_env} for the generation of the variable.
        - working_directore: Optional, location here I{command} will be
         executed.

        @param path: The path to the app folder, i.e., `share/jupyter/trame/
         my-app/`
        @return: The parsed app information for the app
        """
        ...

    def generate_trame_parameters(self, app: TrameApp) -> dict:
        """
```

```
34          Some of the parameters for a launched trame app is generated on the
             server by this configuration. The parameters
35          generated by this function are directly passed to the constructor of
             L{TrameAppInstance}.
36
37          By default, this function generated a UUID, the port to run on, a
             logger where the outputs of the app are logged
38          to, and a tempfile where the authentication key is stored.
39
40          @param app: A reference to the trame app that should be launched
41          @return: A dict with the generated parameters
42          """
43          ...
44
45      def generate_trame_env(self, instance: TrameAppInstance) -> dict:
46          """
47          Generated the environment used by the trame instance. This
             environment contains the $JUVIZ_ARGS variable, that
48          passes information, e.g., the port, to trame.
49
50          @param instance: A reference to the trame instance that will be
             launched
51          @return: The generated environment
52          """
53          ...
54
55      def route_trame(self, instance: TrameAppInstance, server_app: ServerApp)
         -> str:
56          """
57          After trame has been lauched, it must be routed to the user and made
             accessible by the browser. This
58          implementation relies on L{jupyter_server_proxy.
             NamedLocalProxyHandler}, that will be registered to
59          `/trame/<UUID>/` on the server.
60
61          @param instance: The launched trame instance
62          @param server_app: A reference to the server of this JupyterLab
63          @return: The base_url of the trame instance that will be opened when
             the user click on this instance in the lab
64          """
65          ...
66
67      async def launch_trame(self, app: TrameApp, server_app, **options) ->
         TrameAppInstance:
68          """
69          Launch a new instance of the given trame app.
70
71          @param app: The trame app that should be launched
72          @param server_app: A reference to the server of JupyterLab, might be
             required to route trame
73          @param options: The options for this instance that were entered by
             the user in the launch dialog. Currently, the
74              name of the instance and the data directory
75          @return: The launched trame instance
```

```
76          """
77          ...
78
79      @abstractmethod
80      async def get_running_servers(self) -> list[ParaViewServer]:
81          """
82          Get the list of currently running ParaView Servers that were launched
              by JuViz. This information can, e.g., be
83          fetched from the job scheduler.
84
85          @return: A list of all ParaView Servers
86          """
87          pass
88
89      @abstractmethod
90      async def launch_paraview(self, options: dict) -> tuple[int, str]:
91          """
92          Lauch a new ParaView Server. This server should be lauched such that
             L{Configuration.get_running_servers} is
93          able to retrieve the information persistently, even after JupyterLab
             has been restarted.
94
95          @param options: The options for this instance that were entered by
             the user in the ParavIEW launch dialog.
96              Currently, this includes name, account, partition, nodes and
                 timeLimit.
97          @return: The status of the launch command. This include the return
             code and an error message. The message will
98              be displayed in the UI, as an error when the return code is != 0
99          """
100         pass
101
102     @abstractmethod
103     async def get_user_data(self) -> UserData:
104         """
105         Query information about the user. This includes:
106         - The name of the user
107         - The available accounts and partitions available when launching a
             ParaView Server
108         - The path to the home directory, served as the default data
             directory for trame apps
109
110         @return: The retrieved information about the user
111         """
112         pass
```

Code A.2: Complete *ParaView* template script for the *JUWELS Booster* system

```
#!/bin/bash
#SBATCH --account={{ account }}
#SBATCH --job-name="{{ name }}"
#SBATCH --output={{ stdout }}
#SBATCH --error={{ stderr }}
#SBATCH --nodes={{ nodes }}
#SBATCH --time={{ timeLimit }}

# change this only with caution and with respect to "--displays="
#SBATCH --partition={{ partition }}
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=12
#SBATCH --gres=gpu:4

# load modules
module purge
module load Stages/2023
module load GCC/11.3.0 ParaStationMPI/5.7.1-1 ParaView/5.11.0-EGL
module load ParaViewPlugin-Nek5000/20230208-EGL
module list

# Choose MESA if we have no GPU, else SMI
USE_NV=0
echo "checking for GPU"
lspci -k | grep -A 2 -i "NVIDIA" | grep "Kernel driver in use:" | grep "
    nvidia"
if [ $? -ne 0 ]; then
    echo "Using MESA"
    export __EGL_VENDOR_LIBRARY_FILENAMES=$EBROOTOPENGL/share/glvnd/
        egl_vendor.d/50_mesa.json
else
   USE_NV=1
    echo "Using NVIDIA"
    export __EGL_VENDOR_LIBRARY_FILENAMES=$EBROOTOPENGL/share/glvnd/
        egl_vendor.d/10_nvidia.json
    nvidia-smi
fi

# Some infos
which pvserver
eglinfo
srun bash -c 'echo CUDA_VISIBLE_DEVICES: $CUDA_VISIBLE_DEVICES'

export OMP_NUM_THREADS=12
export KNOB_MAX_WORKER_THREADS=12
export PV_PLUGIN_PATH=$PV_PLUGIN_PATH/lib64/paraview-5.10/plugins/
    pvNek5000Reader

# Start ParaView Server
srun --cpu_bind=verbose,rank pvserver --mpi --force-offscreen-rendering --
    multi-clients --displays='0,1,2,3' &
```

```
47  SRUN_PID=$!
48
49  if [ $USE_NV -eq 1 ]; then
50      sleep 10
51      nvidia-smi
52  fi
53  wait $SRUN_PID
```