Adding data visualization to pos-testbeds

Daniel Tarassenko, Kilian Holzinger*, Sebastian Gallenmüller*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: daniel.tarassenko@tum.de, holzinger@net.in.tum.de, gallenmu@net.in.tum.de

***Abstract*—The pos-testbeds provide a controlled and modular environment focusing on reproducible network experiments. At their core, the plain orchestrating service (pos) manages the allocation, configuration, execution, monitoring, and data collection of the experiments. However, pos lacks a user-friendly interface for accessing and visualizing the collected results. This paper presents a solution for this problem: a web-based application that can easily be integrated into the existing testbed infrastructure. Using it, the users can explore the testbeds' result directories in the browser and visualize the collected data on an organized dashboard. This approach supports data accessibility and reproducibility and enhances the overall user experience on pos-based testbeds.**

***Index Terms*—pos-testbeds, reproducibility, plain orchestrating service (pos), results visualization, web application**

1. Introduction

Reproducibility is a fundamental principle of scientific research. As described by Gallenmüller et al. [1], reproducibility enables researchers to verify and validate the results of their own and others' experiments, making it a key source of trust for scientific work.

In the context of computational research, ACM emphasizes [2] that a result is only fully established when it can be reproduced independently. They also define three categories to describe different levels of result validation:

- **Repeatability:** Using the same experimental setup, the same team obtains the same results. (As the weakest requirement for an experiment.)
- **Reproducibility:** Using the same setup and tools, a different team can independently reproduce the original results.
- **Replicability:** Using their own implementation and environment, a different team can reach similar results.

Despite the importance of reproducibility, it remains a challenging task for computer science testbeds. The systems are often complex and consist of many different components, tools, and services, which can significantly impact the results of an experiment, as described by Nussbaum in [3].

To address these challenges and enable reproducible experimentation in networking research the pos-testbeds were developed at Technical University of Munich. These testbeds provide a controlled and modular infrastructure for automated network experiments, which are reproducible by design [4]. With access to such a testbed, researchers from different institutions can run their experiments or verify and replicate the results of other researchers, using the same environment.

With the specially developed plain orchestrating service, the pos-testbeds are able to automate resource allocation, configuration, experiment execution, monitoring, and result collection [1]. While storing the results data in a predefined and structured way, pos lacks a built-in user-friendly interface for accessing and visualizing the collected results. Currently, the users are required to navigate the testbed filesystem manually and search through the output files generated by pos or run external scripts to visualize the data. As accessibility is a key factor for reproducibility, the absence of a user-friendly interface may be a barrier for researchers to effectively access and verify the results. ACM also requires the results to be accessible at least for the reviewers for granting an "Artifacts Evaluated" badge [2].

In this paper, we present an approach to improve the built-in accessibility and visibility of the vast amount of data generated by pos, contributing to better reproducibility and enhancing the user experience of the pos-testbeds. The solution consists of two integrated parts:

- **A dashboard generator**, which generates a static HTML page to organize and visualize data such as the used hardware nodes, energy data, executed scripts, and logs.
- **A lightweight server application**, which allows the user to browse the results directory of the testbed and select an experiment. The dashboard generator is then called for the selected experiment and generates a HTML page, which is displayed in the browser.

The paper is structured as follows: Section 2 provides an overview of the pos-testbeds and the plain orchestrating service itself. Then, Section 3 presents the design goals and structure of the visualization system. Section 4 describes the implementation details. Finally, Section 5 summarizes the results and highlights some possibilities for future improvements.

2. Background

The pos-testbeds are a research infrastructure developed by the Chair of Network Architectures and Services at the Technical University of Munich. They were designed to provide an environment for reproducible and automated network experiments [1]. Now these testbeds are even a part of large-scale European projects like

 doi: 10.2313/NET-2025-11-3_08

SLICES-RI [5] or GreenDIGIT [6], and used by hundreds of students and researchers from various institutions.

Pos ensures exclusive hardware usage by allowing only one experiment per hardware node to run at the same time [1]. In contrast to other platforms like Planet-Lab [3], which rely on container-based virtualization, the pos-testbeds offer bare-metal machines for each experiment. This approach eliminates distortions of the results caused by varying hardware loads at different times, which would make the results unreproducible and unreliable.

Furthermore, while offering many different OS images to choose from, the testbed nodes are live-booted [1]. In other words, the operating systems are not installed on the disks of the test nodes, but are only loaded into the main memory from the management node for each experiment run. This guarantees a clean and consistent software environment for each experiment and avoids any side effects from previous ones, enforcing repeatability.

According to the official pos documentation [7], each testbed consists of multiple test nodes and one central management node. These management nodes are hosting the plain orchestrating service (pos), which is therby the core component of the testbed infrastructure.

Running as a daemon on the management node, pos is responsible for:

- Managing the access among the researchers.
- Configuring and booting the test nodes with the selected OS image.
- Loading the experiment scripts on the test nodes.
- Synchronizing the test nodes and executing the experiment scripts.
- Collecting the logs and results from the test nodes.

For each finished experiment pos stores the results in a defined directory of the following structure:

- `/config` directory containing metadata files like:
  - `allocation.json` with the experiment ID, user name, loop variables, boot parameters, etc.
  - `NODE.json` with the hardware specifications for each node.
- `/energy` directory containing a CSV file per node with the energy data. The CSV files may contain different data sets, but in general, they always include this data for the time period of the experiment execution:
  - Current, Voltage, Power and total consumed energy
- `/setup` directory containing a PDF file with the topology for each node.
- One `/NODE` directory for each node containing files, like:
  - python bootstrap scripts
  - Log files like `status`, `stdout`, and `stderr` for each experiment command, named with the timestamp of their occurrence.

As described by Gallenmüller et al. [4], the entire output of the experiment script is recorded, including utility tools output, executed scripts, vars, device hardware, and topology information. All this information is stored in the result folder of the experiment and guarantees publishability (R5).

The approach of pos aligns with the principles of Open Science and FAIR data management. According to the FAIR principles [8], scientific artifacts should be Findable, Accessible, Interoperable, and Reusable. pos enforces structured data collection and centralized artifact storage, making experimental results easier to evaluate, compare, and reproduce. This also aligns with reproducibility guidelines such as the ACM Artifact Review and Badging standard [2], which emphasize transparency, automation, and accessibility as key enablers for trustworthy computational research.

3. Design

The goal of this solution was to provide the experiment results generated by pos in a user-friendly and accessible way. To achieve this, two main components were developed: a static dashboard generator and a dynamic experiment browser. These two components are not separate solutions or independent approaches, but rather two integrated parts of a single system and are designed to work together.

3.1. Dashboard Generator

As a standalone Python script, the dashboard generator takes a path to the desired pos result directory as an argument and generates a static HTML page to visualize the data stored in it. For the dashboard to be well structured and easy to navigate, it was divided into the following sections: (Screenshots for each section are provided in the appendix.)

Information. Figure 1 shows the information section containing the data taken from `/config/allocation.json`. It displays all key information about the experiment, such as ID, user name, creation date, modification date, used nodes, their global- and loop-variables, as well as their boot parameters. By parsing this file, the dashboard generator can provide a comprehensive and consistent overview of the experiment's configuration.

Nodes. As shown in Figure 2, this section provides hardware specifications for each of the used nodes. This data is taken from `/config/NODE.json` and includes information about the CPUs, RAM, networking cards, and all their interfaces, but also the storage, motherboard, and operating system image the experiment was run on.

Topology. The topology section displays images for each node, showing the network interfaces and their interconnections with other nodes. These images are taken from the testbeds topology directory, which can be defined as a program argument. An example is shown in Figure 3.

Timeline. As probably the most important part of the dashboard, the timeline displays all events for each node in chronological order. By selecting an event, the users are presented with all associated files, such as executed scripts, sent commands, standard output, standard error,

 doi: 10.2313/NET-2025-11-3_08

and status files, which allows them to analyze the experiment execution in detail. The timeline section is displayed in Figure 4.

Energy. The final section of the dashboard is the energy section, shown in Figure 5. It displays the energy consumption data for each node, found in the `/energy` directory. Initially, the data is stored in CSV files with varying columns, but in general, they contain the following datasets: Voltage, Current, Power, and Total consumed energy. The data is measured periodically with a sample rate of about 1 Hz for the duration of an experiment. In this section, users can turn on and off each data row to display only the data they are interested in.

While for the most part, the dashboard generator fulfills the main requirements, as a result visualizing tool, it produces only static HTML pages. This implies that pos would need to call the dashboard generator after every finished experiment to create a page that only contains the results of this specific experiment. As a consequence, the server would need to permanently store all generated HTML pages for each experiment, even if they are never visited after some time. This approach leads to redundant storage usage, as the information would be saved twice: as the original experiment result directory and as a static HTML page. In particular, it also would repeatedly store the identical web page layout structure, such as the header, navigation sidebar, CSS and JavaScript as part of the HTML files for each experiment, despite the fact they are always the same. Another drawback resulting from the static approach is that after making some changes to the dashboard, for example, adding a new section or changing the design, by editing some CSS, all previously generated HTML pages would be outdated. This would require the testbed to regenerate all HTML pages after every change.

3.2. Result Browser

To address these issues, a small flask web application was implemented as the second part of the solution. It is designed to run directly on the management node of each testbed and provide a user-friendly web interface for accessing the results of the testbed. Once configured with the path to the result directory containing all experiment results, it would allow the users of the testbed to browse through all these results from their browser. Figure 6 shows a demo results directory of the experiment browser. Here, the users can find and select the experiment they are interested in. The flask application will then call the dashboard generator to create a dashboard for the selected experiment and display it to the user. The benefit of this approach is, that the dashboard is constructed only in the main memory of the server, without being stored on the disk, therefore solving the problem of storing redundant data. Also, since the dashboard HTML pages are only created on demand, there are no extra operations for creating a dashboard after each experiment. Moreover, this allows the dashboard generator to be updated at any time, as the result browser will just call the updated version of the generator. So the users will get displayed the latest version of the dashboard on every experiment visit.

By replacing manual file navigation with an interactive web interface, the experiment browser improves accessibility. The dashboard follows established design principles such as logical grouping, progressive disclosure, and hierarchical layout, which reduce cognitive load and contribute to a more intuitive user experience [9]. Together, these two components significantly improve the usability of the pos-based testbeds and support reproducibility [10].

4. Implementation

4.1. Project structure

This section gives some insights into the details of the proposed solution's implementation. The project is structured as follows:

```
dashboardGenerator/
├── static/
│   └── images/
│       ├── tum-logo.svg
│       ├── folder.svg
│       └── ...
├── templates/
│   ├── base.html
│   ├── dashboard.html
│   ├── section_energy.html
│   ├── section_information.html
│   └── ...
├── generator.py
├── results_browser.py
└── README.md
```

4.2. Dashboard Generator

The dashboard generator is a standalone Python script (`generator.py`), which is the core of the visualization system. While it is not recommended to run the script directly for creating HTML pages, it may be used for testing purposes. The script takes three optional arguments: (`-i, --input`), (`-t, --topologies`) and (`-o, --output`). They define the paths to the experiment result directory, the topology images directory, and the output directory, respectively. The argument parsing is implemented using the `argparse` library. When executed, the script reads all files from the experiment directory and renders a static HTML page using Jinja2 templates.

4.2.1. Data Parsing. First of all, the dashboard generator parses all relevant files from the experiment directory. JSON files such as `allocation.json` and `NODE.json` are read using the built-in `json` module, while the energy data is processed using the built-in `csv` module. To improve the performance of the dashboard in the browser and reduce the file size of the generated HTML file, the energy data from the CSVs is downsampled to a default value of 200 data points per node. Since downsampling was not the main focus of this paper, a simple algorithm is currently used, that selects every N-th row and discards the rows in-between. This approach is very easy to implement, fast, and sufficient for demonstration purposes. In the future, a more complex algorithm like LTTB (Largest-Triangle-Three-Buckets) could be integrated to improve the downsampling quality. Unlike the N-th row approach, which could omit important values like peaks, LTTB selects the most significant points of the dataset, thereby preserving the visual characteristics of the original data.

 doi: 10.2313/NET-2025-11-3_08

All the parsed data is then stored in nested Python data structures, like lists of dictionaries or lists of JSON objects, which are then passed to the Jinja2 template engine for rendering the HTML page.

4.2.2. Template Rendering. The core part of the dashboard generator is the Jinja2 templating engine, which was chosen for its simple integration as a Python library, the Python-like syntax, good documentation and support for modular templates. It enables the creation of HTML templates with not only predefined static data, such as the layout elements but also dynamic placeholders, which are then filled in the rendering process on execution, as described in the Jinja2 documentation [11]. Furthermore, Jinja2 templates can contain variables, loops and conditions for displaying dynamic data, like tables or lists. An example of a Jinja2 loop is shown below:

```
<tbody>
    {% for node in nodes %}
    <tr>
        <td>{{ node.hostname }}</td>
        <td>
            <ul>
                {% for cpu in node.processor %}
                <li>{{ cpu.model }}</li>
                {% endfor %}
            </ul>
        </td>
        ...
        <td>{{ node.image }}</td>
    </tr>
    {% endfor %}
</tbody>
```

This code snippet would generate a table row for each node in the list of nodes while listing all CPU models within a table cell.

Another feature of Jinja2 is the support of template inheritance. Thus, a base template (`base.html`) defines the overall layout structure, like the page header. This base template is then extended by `browse.html` and `dashboard.html`, which are also templates.

For better code structure and easier maintainability, each section of the dashboard - `Information`, `Nodes`, `Topology`, `Timeline`, and `Energy` - is implemented in its own template file. Each of these section templates is then included in the `dashboard.html` template.

4.2.3. Layout and Interactivity. To ensure a clean and intuitive user interface, the Bootstrap framework [12] was used. Bootstrap provides many ready-to-use components for web applications, such as accordions, cards, buttons, and more. In particular, the accorions as collapsible boxes, are used to group the information, structure the dashboard and visually hide as much information as possible. Thus, the users are not overwhelmed by too much information at once and giving them control over how much detail they want to see.

The required Bootstrap CSS and JavaScript are currently included with CDN links in the `base.html` template. In the future, only the used assets could be compiled from Bootstraps SASS source files and stored locally on the testbed. This would avoid loading unused assets, reduce external dependencies and obsolete internet access.

In addition to Bootstrap, some simple JavaScript is used to hide all the dashboard sections (`Information`, `Nodes`, `Topology`, `Timeline`, and `Energy`) and only show the section selected on the sidebar. In combination with Bootstrap elements, this contributes to a clean structure and an interactive user experience.

4.3. Result Browser

The second part of the visualization system is a small web application built using Flask, a lightweight web framework for Python [13]. As it is intended to run directly on the management node of each pos-testbed, the path to the testbed base directory must be provided as an argument `-d, --directory`. Furthermore, the arguments (`-H, --host`) and (`-p, --port`) are used to specify the host and port of the web server. The application defines the following endpoints:

- `/browse/<path>` – Allows navigation through the results directory and displays its contents.
- `/experiment/<path>` – Initiates a dashboard generation and returns an HTML for a selected experiment.
- `/topology/<filename>` – Serves topology images on demand, as they are not stored in the static directory but on the testbed.

When listing a directory while browsing, the application checks for each subfolder if it is an experiment result directory. This is done by checking if the `config/`, `energy/`, and `setup/` directories are present. If so, the experiment directory is displayed with a link to the `/experiment/<path>` endpoint. When an experiment directory is clicked, a dashboard is generated using a direct function call to `generate_page()` from `generator.py`. The rendered HTML is delivered to the browser using a `Response` object without storing it on the disk of the hosting management node.

This on-demand generation of the dashboard avoids redundant file storage and enables the dashboard generator to be updated at any time, as already discussed in Section 3. In its current state, the application is designed for internal use only, as no authentication or encryption is implemented. This is because no HTTPS certificate was available during the development.

4.4. Performance Considerations

As the proposed visualization solution is designed for occasional, human-triggered interactions, rather than continuous or high-frequency data processing, the performance is not a critical factor for the system. Therefore a formal performance evaluation was left out of the scope of this paper. Nevertheless, some considerations are worth mentioning:

- The execution time of the dashboard generator depends on the size of the given experiment result directory. In particular, the number of used nodes and amount of recorded events for each node plays a significant role, as all these files must be parsed, saved in a data structure, and then rendered to an HTML page.
- JinJa2 extensively uses caching and avoids repeated compiling of the templates, making the rendering process nearly as efficient as executing a Python function [14].

- Since the dashboard generator is only invoked on demand, the system avoids redundant processing and remains idle most of the time.

5. Conclusion and Future Work

This paper presented a lightweight and modular solution to improve the accessibility and visibility of experiment results generated by the plain orchestrating service (pos) in reproducible testbed environments. The system consists of two parts: a static dashboard generator and a dynamic experiment browser, both implemented in Python. After integrating these tools directly into the testbed infrastructure, users can access information like experiment metadata, execution logs, hardware information, and energy consumption through an interactive and structured interface. This improves the user experience and supports reproducibility by making data more accessible and easier to interpret.

Future Work: Responsiveness improvements could enable usage on mobile devices. While security was not a focus in this prototype, future versions should include HTTPS and user authentication using login, to support secure and used-specific access to experiment results. Filter and sorting functions in the experiment browser would improve navigation, especially for users with many experiments. Integration of live result data streaming while experiment execution would allow real-time monitoring of experiments. Finally, exporting results in RO-Crate format [15] would enable standardized, machine-readable packaging of experiments and simplify publishing and sharing via FAIR-compliant repositories.

References

[1] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, "The pos framework: A methodology and toolchain for reproducible network experiments," *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21)*, 2021.

[2] ACM, "Artifact review and badging version 1.1," https://www.acm.org/publications/policies/artifact-review-and-badging-current, 2020, online; Last access: 05.04.2025.

[3] L. Nussbaum, "Testbeds support for reproducible research," *Proceedings of Reproducibility '17*, 2017.

[4] S. Gallenmuller, D. Scholz, H. Stubbe, E. Hauser, and G. Carle, "Reproducible by design: Network experiments with pos," *Würzburg Workshop on Next-Generation Communication Networks (WueWoWas'22)*, 2022.

[5] S. R. Infrastructure, "Post-5g blueprint," https://doc.slices-ri.eu/BlueprintServices/beyond5G/beyond5G.html, 2024, online; Last access: 05.04.2025.

[6] C. of Network Architectures, I. Services, School of Computation, and T. U. o. M. Technology, "Greendigit," https://net.in.tum.de/projects/greendigit/, 2024, online; Last access: 05.04.2025.

[7] D. Scholz and S. Gallenmüller, "Welcome to the plain orchestrating service (pos)," https://i8-testbeds.pages.gitlab.lrz.de/pos/cli/#welcome-to-the-plain-orchestrating-service-pos, 2025, online; Last access: 04.04.2025; Note: Internal documentation, access restricted to registered users only.

[8] M. D. W. et al., "The fair guiding principles for scientific data management and stewardship," *Scientific Data*, 2016.

[9] UXPin, "Effective dashboard design principles for 2025," https://www.uxpin.com/studio/blog/dashboard-design-principles/, 2025, online; Last access: 10.05.2025.

[10] J. Leipzig, D. Nüst, C. T. Hoyt, K. Ram, and J. Greenberg, "The role of metadata in reproducible computational research," *Patterns*, 2021.

[11] Pallets, "Jinja — jinja documentation (3.1.x)," https://jinja.palletsprojects.com/en/stable/, 2025, online; Last access: 05.04.2025.

[12] T. B. Authors, "Get started with bootstrap," https://getbootstrap.com/docs/5.3/getting-started/introduction/, 2025, online; Last access: 05.04.2025.

[13] Pallets, "Welcome to flask — flask documentation (3.1.x)," https://flask.palletsprojects.com/en/stable/, 2025, online; Last access: 05.04.2025.

[14] Pallets, "Frequently asked questions," https://jinja.palletsprojects.com/en/stable/faq/, 2025, online; Last access: 10.05.2025.

[15] E. Hauser, S. Gallenmuller, and G. Carle, "Ro-crate for testbeds: Automated packaging of experimental results," *IFIP Networking Conference (IFIP Networking)*, 2024.

Appendix

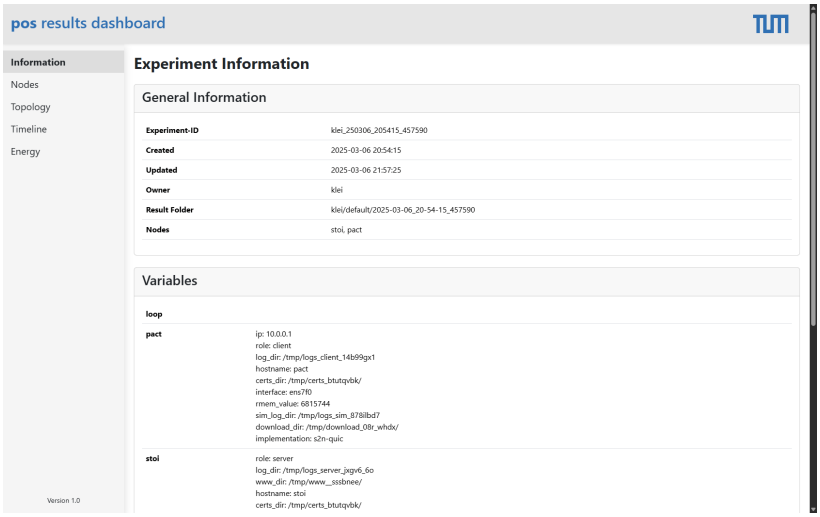

Figure 1: Informatin section

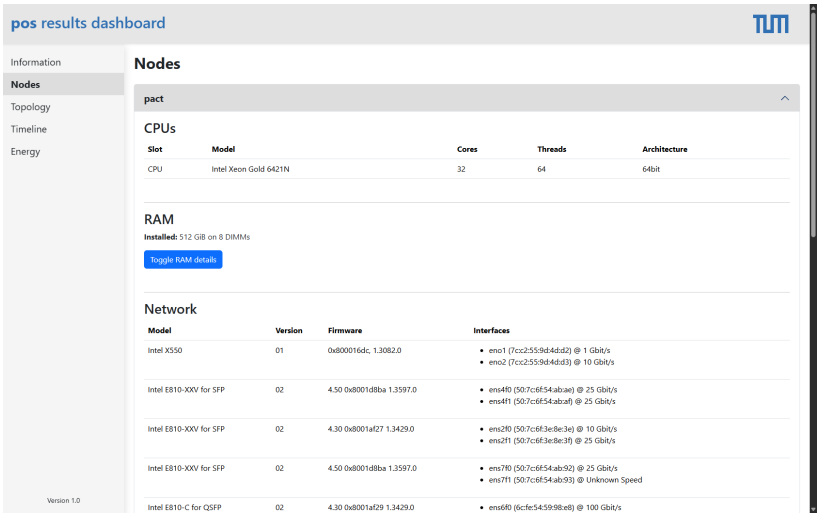

Figure 2: Nodes section

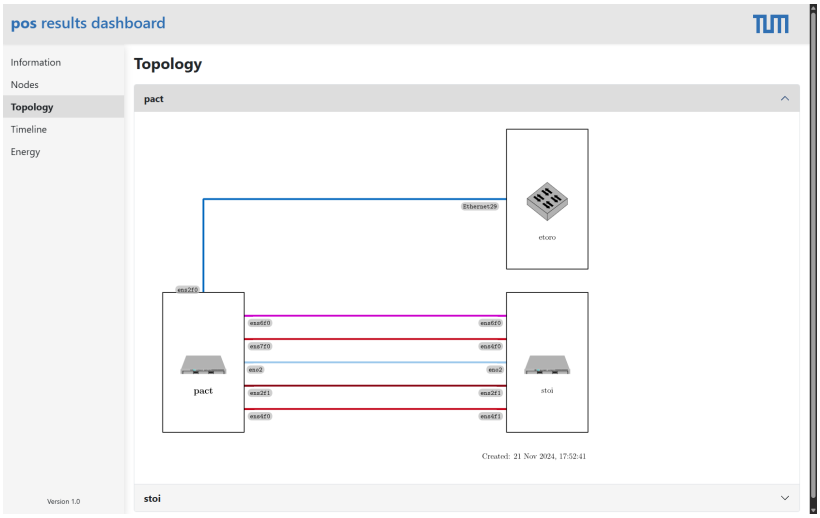

Figure 3: Topology section

 doi: 10.2313/NET-2025-11-3_08

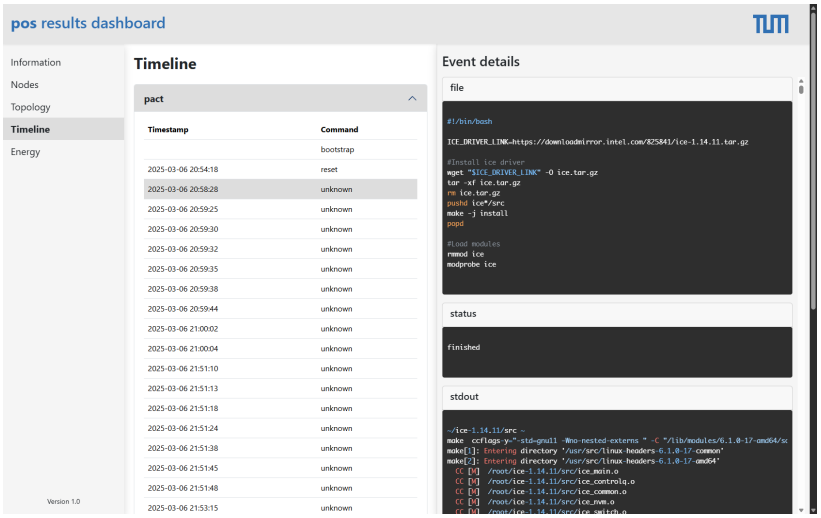

Figure 4: Timeline section

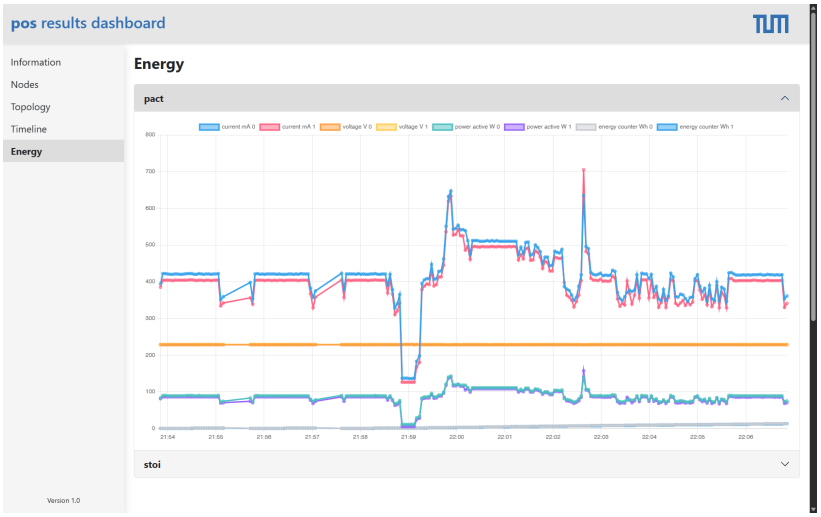

Figure 5: Energy section

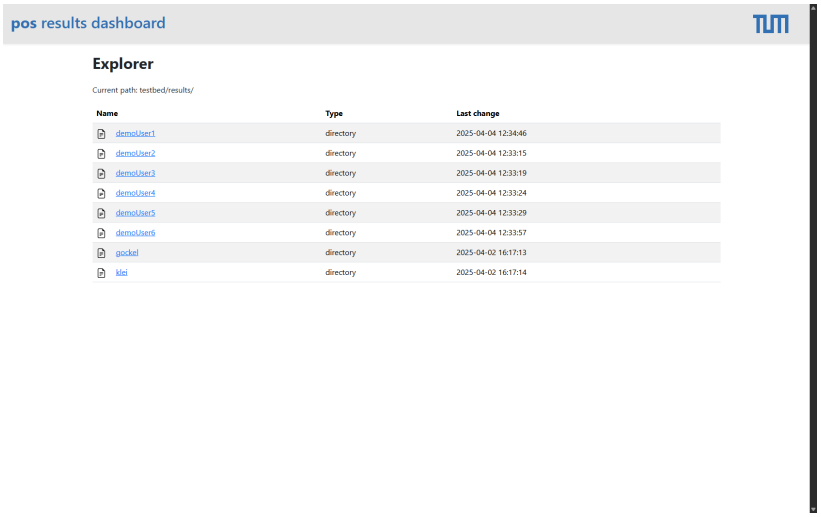

Figure 6: Experiment browser