Energy Consumption Reports Using Jupyter Notebooks

Julian Forster, Kilian Holzinger*, Sebastian Gallenmüller*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: j.forster@tum.de, holzinger@net.in.tum.de

***Abstract*—Reproducibility is a key aspect of scientific research, ensuring that experimental results can be independently verified. However, achieving reproducibility remains a challenge due to the additional effort required for documentation and automation. The plain orchestration service (pos) was developed to address these issues by enforcing reproducibility in network experiments. This paper presents an approach to enhance pos with an automated energy consumption reporting system based on Jupyter Notebooks. By integrating energy data collection through Prometheus and Grafana, we enable detailed insights into power usage during experiments. The proposed solution focuses on flexibility and adaptability, allowing it to be used across different environments. We discuss the architecture, implementation, and a case study demonstrating the practical application of the system.**

***Index Terms*—plain orchestration service (pos), Network Experiment, Energy consumption, automated reports**

1. Introduction

Having independent reproducible experimental results is crucial for any scientific research [1]. It makes results more trustworthy because it allows other researchers to recreate and verify the results. However, it is not that widespread to create reproducible experiments in the science community. The main problem is the increased effort researchers have to put into their experiments to make them reproducible, and it comes with some technical limitations [2] like privacy reasons or custom-built lab equipment resources.

To achieve that, the whole experiment process has to be automated (to prevent human errors) and documented. The documentation includes the hardware and environment, scripts and parameters used, and the actual results from the experiment. Depending on the experiment, more insights into the actual behavior of the hardware can be beneficial. This includes, for example, the energy consumption of a node during the experiment.

To encourage the creation of reproducible experiments, the ACM, which is the world's largest scientific and educational computing society, dedicated to advancing computing as a science and profession, introduced badges, which are awards for papers that make their experiments reproducible [3].

pos was created to assist researchers in achieving reproducibility with almost no additional effort, it is a methodology and a testbed where network experiments can be tested on [4]. The main advantage of pos is, that reproducibility was a main design consideration and, therefore, is enforced to run the experiments. At the end of an experiment, all the scripts, parameters, and results are saved and optionally published.

To gain deeper insights into the experiments, we want to create a dashboard that can collect the information from an experiment and give valuable insights into the energy consumption during the different runs. Although the dashboard is already created and works with pos [5], the goal is to make it flexible and adaptable to work in different environments and backends.

There are already some projects and tools supporting the creation of reproducible experiments. Some of them are OMF [6], which is a testbed controller, or SNDZoo [7] or WalT [8]. However, all of them only allow the creation of reproducible experiments but do not enforce them, which means that an experiment can be run without the proper documentation for other researchers. In the project [9], they tested the capability to do reproducible network experiments using container-based emulation. Adding a layer of abstraction always interferes with the results of an experiment, especially on low latency network experiments. These are the reasons why we want to focus on a solution, which works especially for pos, which is built for direct hardware access, but makes it adaptable for other projects and tools as well.

The remainder of this paper is structured as follows: Sec. 2 gives more background on the project, Sec. 3 talks about the approach, Sec. 4 describes a short case study, Sec. 5 talks about the future work, and Sec. 6 concludes the paper. Appendix 1 shows the output of the jupyter energy evaluation dashboard (all the required dependencies and code can be found in the repository [10]).

2. Background

This section focuses on the architecture and technology stack the testbed uses to run and evaluate reproducible experiments. First, we discuss pos and how it is built, and then the other technologies used to retrieve the energy data required for the jupyter-based experiment energy assessment dashboard.

2.1. Architecture of pos

pos was created with the following requirements for reproducible experiments in mind [4]:

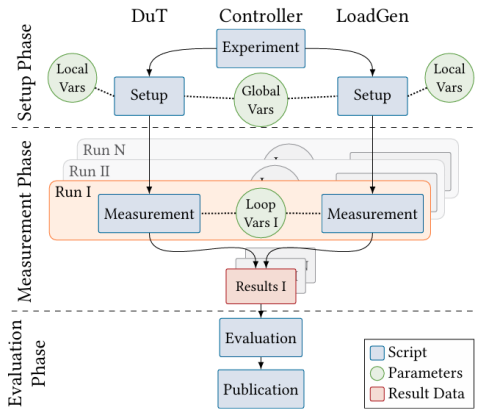


Figure 1: Architecture of pos [4]

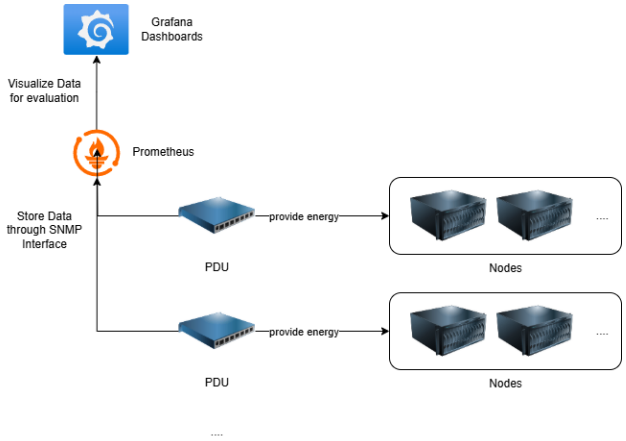


Figure 2: Architecture and Data Collection of the Testbed (diagram selfmade)

- *Heterogeneity (R1)* support for wide range of different devices
- *Isolation (R2)* the experiment nodes from non-experiment related devices
- *Recoverability (R3)* reverts devices into working mode even after a crash
- *Automation (R4)* avoids errors of misconfiguration
- *Publishability (R5)* document everything automatically

With these requirements in mind, pos architecture is built to enforce these with as little effort as possible. To achieve this, pos offers a flexible and well-defined workflow as shown in Fig. 1.

In the setup phase of the experiment, one or more nodes are allocated and booted with a predefined live image to ensure that, per each run, the experiment starts from the same point. After the boot, a basic setup script runs to make some configuration like installing network measurment software etc. After the setup phase, the actual measurement phase begins, where the nodes are monitored, and the results are stored. One experiment can involve multiple different runs with different parameters. pos separates the scripts and the according parameters into two files to ensure easy adaptation to different environments without needing to change the script. After all runs, the evaluation phase starts. Custom scripts can aggregate and further process the results and optionally publish them. It is currently used, for example, for TSN-based network experiments [11].

2.2. Energy Data Collection & Visualization

In the following, we discuss how the energy data is collected from the testbed and later evaluated and visualized in the current form.

As shown in Fig. 2, each node of the testbed gets its power through a power distribution unit (PDU), which can log the power consumption of each power outlet. In the current testbed of the chair, we have 4 PDUs connected to the nodes. A local instance of Prometheus, an open-source monitoring and alerting toolkit [12], collects and stores power consumption data via the SNMP interface. The stored metrics are then visualized using Grafana, a widely-used open-source platform for monitoring and observability [13]. This setup enables detailed visualization

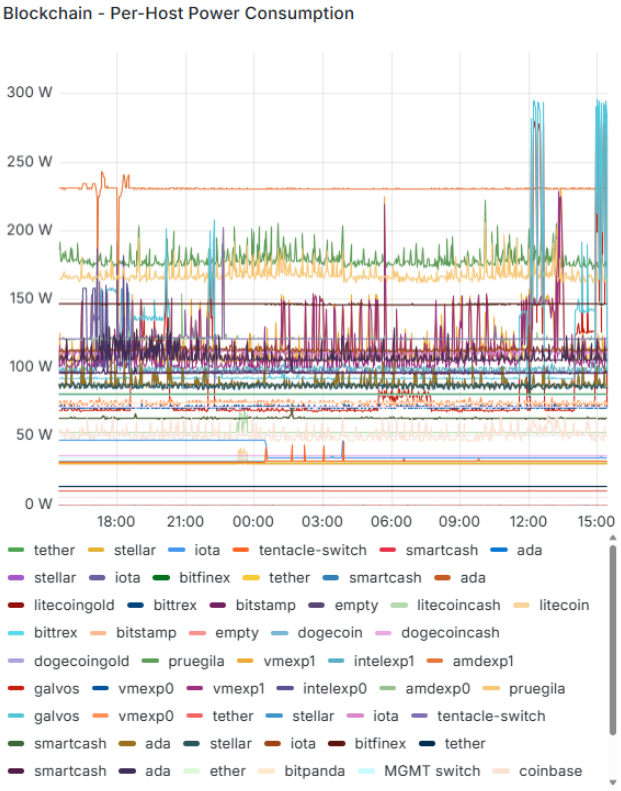


Figure 3: Per-Host Power Consumption in Grafana [14]

of per-node power consumption statistics, as illustrated in Fig. 3.

During each run, pos also save the energy consumption data and all the other experiment-related data in an RO-Crate format. So it is possible to retrieve the data through experiment results (only the used nodes and only for the specific time range) and through Grafana.

2.3. Jupyer Based Experiment Energy Assessment

In the current jupyter notebook [5], the output folder, which is in RO-Crate format, displays important data about the experiment and plots energy data for different

runs. These include current, voltage, power consumption, and aggregated power consumption. In specific, the notebook shows the following information:

- *Creator information* extracted from the RO-Crate metadata
- *Node information & topology visualization* extracted from the RO-Crate as well.
- *Power consumption over time* shows the power consumption each node has per run
- *Cumulative energy consumption*
- *Current and voltage trends*
- *Energy Consumption Rate Over Time* the rate at which energy is consumed over time (mW/s)

The dashboard currently uses two data sources to create the tables and plots: the *RO-Crate metadata file* and the *energy data folder* in which the data is stored in CSV format per each node and run with the following structure: current_mA, voltage_V, power_active_W, and energy_counter_Wh.

3. Approach

With our approach we want to make the dashboard more universally adaptable, as currently the dashboard can only work with the RO-Crate format. To do so, we made it possible for the dashboard to work with the Grafana API to get the required information.

3.1. Design

Grafana has no information about the experiments and related runs or used nodes. This is why in the first prototype, we used the pos python API to retrieve the reserved nodes as well as the date and time slots for each experiment. However, pos currently does not support querying entries from the past, which is why this is not a feasible solution. So for the calendar data, we used an exported CSV dump from the pos database. This is just a temporary solution until the pos API is further developed to get the data through the API directly.

We had multiple choices on how to implement the flexibility for the dashboard. One option was to utilize an object-oriented approach to make the dashboard extensible and open for multiple different interfaces and data sources. An alternative approach was to create different scripts that work as preprocessors, which collect the data from the sources and store them in a similar CSV energy format which then can be read by the dashboard.

The Grafana API currently only supports power consumption data. As already written, the pos API is currently too limited for practical use, which is why we had to switch to the CSV dump of the pos database. This shows that every testbed environment is unique, which makes it quite difficult with an object-oriented approach as this requires some similar processes to work correctly. So we decided to go with the preprocessor approach.

3.2. Implementation

For that, we created a new script, which queries the 4 PDUs of the pos testbed via the Grafana API, aggregates

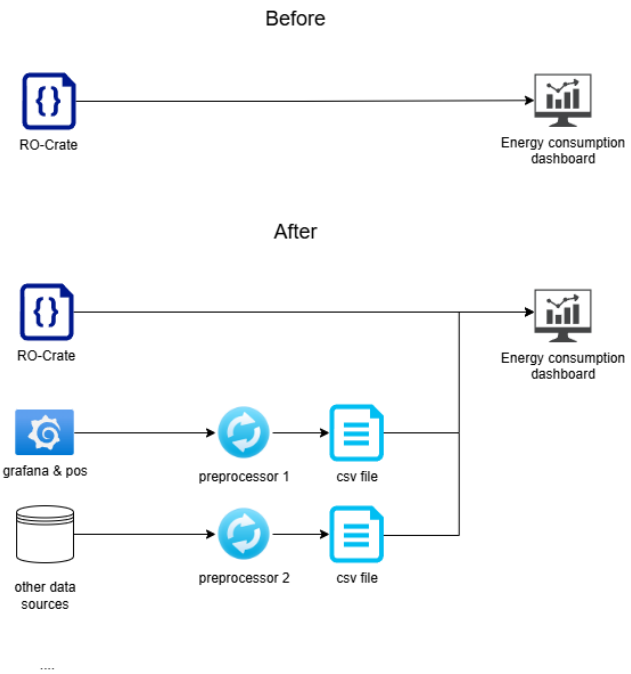


Figure 4: Energy Consumption Dashboard Architecture Before & After

them together, and cleans up the data. Next, we take the information from the pos calendar CSV database dump and choose one experiment. The data is then exported to CSV file(s), which then can be read by the main energy consumption dashboard notebook as shown in Fig. 4. The "other data sources" in Fig. 4 show, that we can add any other data sources later, which work with the dashboard as well.

Because only the RO-Crate Metadata file contains creator and node information, we can not show this data, when we use grafana and pos as the datasource. So we changed the dashboard script to be able to visualize the information with and without the RO-Crate Metadata file and with the CSV energy files.

Because we only get the power consumption data from Grafana, we also had to adjust the dashboard to be able to work with more or less data, depending on what is provided in the CSV file.

4. Case Study

In this section, we do a short example go through the project to show how the process works. First, we have to choose one experiment run. Because the web portal does not show any IDs for one experiment, we have to look that up in the CSV. For this example, we chose the entry with the id 7807, which uses in total three nodes (bitcoin, bitcoincash, bitcoingold) and it ran on 07.03.2025 between 12 am and 6 pm. Next, we query the data from Grafana in this time range. When we plot that data, we get the energy consumption of all the nodes. After some cleanup, we get the following diagram as shown in Fig. 5.

Now, we only have to filter for the nodes we are interested in and export that data to a CSV file. After we run the process, we can check the results by running

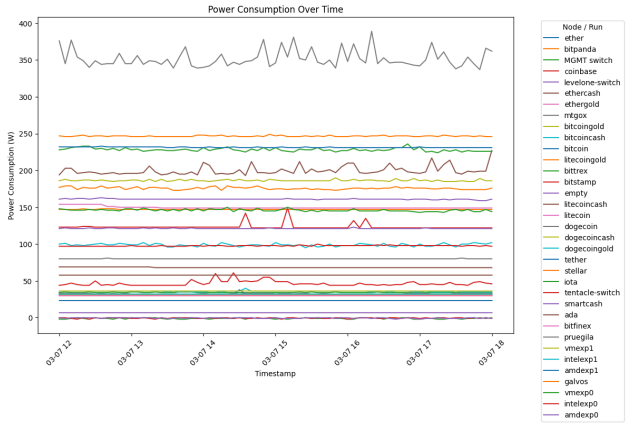


Figure 5: Power Consumption over Time for all Nodes

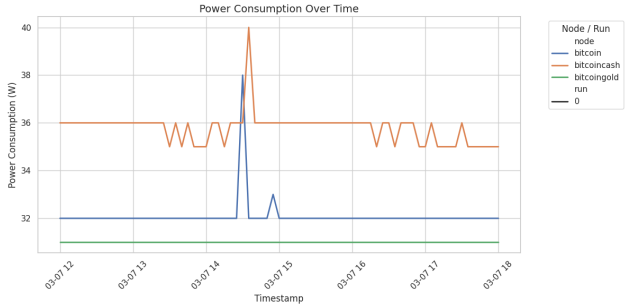


Figure 6: Example Power Consumption over Time for Three Nodes [5]

the dashboard script. We then get the energy consumption over time for the nodes as shown in Fig. 6. A full output of the energy consumption dashboard can be found in Appendix 1.

Because Grafana currently only supports power consumption, we had to set all the other columns to 0. Therefore the other diagrams are currently not working. As soon as we have the option to query more data, we can use all the other diagrams in the dashboard.

5. Conclusion

In this paper, we introduced an approach to improve the monitoring of energy consumption for network experiments conducted with pos [4]. Using Prometheus, Grafana, and Jupyter Notebooks, we developed a system that enables automated and reproducible power usage reporting. The implemented solution provides valuable information on trends in energy consumption and can be extended to other testbeds by adapting data sources.

References

[1] C. S. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Communications of the ACM*, vol. 59, no. 3, pp. 62–69, 2016.

[2] N. Zilberman, "An artifact evaluation of ndp," *Computer Communication Review*, vol. 50, no. 2, pp. 32–36, 2020.

[3] ACM. (2020) Artifact review and badging version 1.1. Last accessed: 2022-05-06. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current

[4] S. Gallenmüller*, D. Scholz*, H. Stubbe, and G. Carle, "The pos Framework: A Methodology and Toolchain for Reproducible Network Experiments," in *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21)*, Munich, Germany (Virtual Event), Dec. 2021.

[5] K. Warmuth. (2025) Greendigit evaluation repository. Chair of Network Architectures and Services, School of Computation, Information and Technology, Technical University of Munich, Germany. GitLab repository, last accessed: November 17, 2025. [Online]. Available: https://gitlab.lrz.de/GreenDIGIT/evaluation

[6] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "Omf: A control and management framework for networking testbeds," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, 2010.

[7] M. Peuster, S. Schneider, and H. Karl, "The softwarised network data zoo," in *15th International Conference on Network and Service Management, CNSM 2019*. Halifax, NS, Canada: IEEE, Oct. 21-25 2019, pp. 1–5.

[8] P. Brunisholz, E. Dublé, F. Rousseau, and A. Duda, "Walt: A reproducible testbed for reproducible network experiments," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2016, pp. 146–151.

[9] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012, pp. 253–264.

[10] J. Forster. (2025) Seminar: Energy consumption reports using jupyter notebook. Chair of Network Architectures and Services, School of Computation, Information and Technology, Technical University of Munich, Germany. GitLab repository, last accessed: November 17, 2025. [Online]. Available: https://gitlab.lrz.de/netintum/teaching/iitm/repos/2025ss-bs/u112

[11] M. Bosk, F. Rezabek, K. Holzinger, A. G. Marino, A. A. Kane, F. Fons, J. Ott, and G. Carle, "Methodology and infrastructure for tsn-based reproducible network experiments," *IEEE Access*, vol. 10, pp. 109 203–109 239, 2022.

[12] Prometheus Authors, "Prometheus: Monitoring system & time series database," https://prometheus.io, 2024, accessed: 2025-05-11.

[13] Grafana Labs, "Grafana: The open observability platform," https://grafana.com, 2024, accessed: 2025-05-11.

[14] Gude Dashboard, "Grafana dashboard of energy consumption per node," https://catalepsy.net.in.tum.de/d/8cJTjb1nk/gude?orgId=1&from=now-24h&to=now&timezone=browser, 2025, accessed: 2025-04-05.

Appendix 1: Output of the Jupyter Energy Evaluation Dashboard From The Run

evaluate

April 6, 2025

1 Energy Measurement Evaluation

This notebook evaluates power consumption and energy trends from experiment results.
The data is collected from multiple nodes and analyzed for insights into power usage, voltage, and energy consumption.

1.1 Specify the Result Folder

Before loading data, enter the path to your experiment result folder.
By default, the last used path is shown, but you can change it to any valid directory.

[7]:
```
import os
from IPython.display import display, HTML
import json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

base_result_folder = "/srv/testbed/results/warmuth/default/"
default_result_folder = os.path.join(base_result_folder,␣
 ↪"2025-03-18_13-03-16_395230")

user_input = input(f"Enter result folder path - if leaving empty it uses the␣
 ↪default folder:[{default_result_folder}]: ").strip()

if not user_input:
    RESULT_FOLDER = default_result_folder
elif "/" in user_input:
    RESULT_FOLDER = user_input
else:
    RESULT_FOLDER = os.path.join(base_result_folder, user_input)

if not os.path.exists(RESULT_FOLDER):
    raise FileNotFoundError(f"Result folder not found: {RESULT_FOLDER}")

display(f"Using result folder: {RESULT_FOLDER}")
```

```
'Using result folder: /home/forster/dashboard/.transfer'
```

1.1.1 Creator Information

The following table presents details about the experiment's creator, extracted from the **RO-Crate metadata**.

- **Name:** The name of the creator.
- **ORCID:** A unique researcher identifier, linked to the official ORCID profile.
- **Affiliation:** The institution the creator is affiliated with.
- **Affiliation ROR:** A **Research Organization Registry (ROR) ID**, used for standard identification of research institutions.
- **Affiliation URL:** A direct link to the institution's website.

[8]:
```python
def load_creator_info():
    """
    Extracts all creators from the RO-Crate metadata JSON file.
    Retrieves each creator's name, ORCID, and affiliation details.
    """
    rocrate_path = os.path.join(RESULT_FOLDER, "ro-crate-metadata.json")
    if not os.path.exists(rocrate_path):
        raise FileNotFoundError(f"RO-Crate metadata file not found: {rocrate_path}")

    with open(rocrate_path, "r") as f:
        metadata = json.load(f)

    creators = []

    for item in metadata.get("@graph", []):
        if item.get("@type") == "Person" and "creator" in item.get("keywords", []):
            creator_info = {
                "Creator Name": item.get("name", "Unknown"),
                "ORCID": item.get("@id", "Unknown"),
                "Affiliation Name": "Unknown",
                "Affiliation ROR": "Unknown",
                "Affiliation URL": "Unknown"
            }

            # Find affiliation
            affiliation_id = item.get("affiliation", {}).get("@id", None)
            if affiliation_id:
                for org in metadata.get("@graph", []):
                    if org.get("@id") == affiliation_id:
                        creator_info["Affiliation Name"] = org.get("name", "Unknown")
```

```
                        creator_info["Affiliation ROR"] = org.get("@id",␣
↪"Unknown")
                        creator_info["Affiliation URL"] = org.get("url",␣
↪"Unknown")
                        break

            creators.append(creator_info)

    return creators

try:
    creator_data = load_creator_info()

    creator_df = pd.DataFrame(creator_data)

    def make_link(text, url):
        return f'<a href="{url}" target="_blank">{text}</a>' if url !=␣
↪"Unknown" else "Unknown"

    creator_df["ORCID"] = creator_df["ORCID"].apply(lambda x: make_link("ORCID␣
↪Profile", x))
    creator_df["Affiliation ROR"] = creator_df["Affiliation ROR"].apply(lambda␣
↪x: make_link("ROR ID", x))
    creator_df["Affiliation URL"] = creator_df["Affiliation URL"].apply(lambda␣
↪x: make_link("University Website", x))

    html_table = creator_df.to_html(escape=False, index=False)
    styled_table = f"""
    <style>
        table {{ width: 80%; border-collapse: collapse; margin: 20px 0; }}
        th, td {{ padding: 8px 12px; border: 1px solid #ddd; text-align: left;␣
↪}}
        th {{ background-color: #f4f4f4; font-weight: bold; }}
    </style>
    {html_table}
    """

    display(HTML(styled_table))

except FileNotFoundError:
    display("RO-Crate file not available, will skip")
```

```
'RO-Crate file not available, will skip'
```

1.2 Node Information & Topology Visualization

Each experiment setup includes metadata about the participating nodes.
This section extracts details such as: - Node names - Links to the Testbed - Fully Qualified Domain Names (FQDN) - Topology information (if available).

If a **topology visualization** is provided in the RO-Crate metadata, it is displayed below.

[9]:
```
def load_rocrate_metadata():
    """
    Load and parse the RO-Crate metadata JSON file.
    Extract node information and locate paths for hardware details and topology␣
↪PDFs.
    """
    rocrate_path = os.path.join(RESULT_FOLDER, "ro-crate-metadata.json")
    if not os.path.exists(rocrate_path):
        raise FileNotFoundError(f"RO-Crate metadata file not found:␣
↪{rocrate_path}")

    with open(rocrate_path, "r") as f:
        metadata = json.load(f)

    nodes_info = []

    for item in metadata.get("@graph", []):
        if "keywords" in item and "node" in item["keywords"]:
            node_name = item.get("name", "Unknown")
            fqdn = item.get("fqdn", "Unknown")

            topology_pdf_path = None
            if isinstance(item.get("visualizedTopology", {}), dict) and "@id"␣
↪in item["visualizedTopology"]:
                topology_pdf_path = os.path.join(RESULT_FOLDER,␣
↪item["visualizedTopology"]["@id"])
                if not os.path.exists(topology_pdf_path):
                    topology_pdf_path = None

            hardware_json_path = None
            if isinstance(item.get("hardware", {}), dict) and "@id" in␣
↪item["hardware"]:
                hardware_json_path = os.path.join(RESULT_FOLDER,␣
↪item["hardware"]["@id"])
                if not os.path.exists(hardware_json_path):
                    hardware_json_path = None

            nodes_info.append({
                "name": node_name if isinstance(node_name, str) else "Unknown",
                "fqdn": fqdn if isinstance(fqdn, str) else "Unknown",
```

```
                "topology_pdf": topology_pdf_path if topology_pdf_path else␣
↪"None",
                "hardware_json": hardware_json_path if hardware_json_path else␣
↪"None"
            })

    return nodes_info

def extract_hardware_info(hardware_json_path):
    """
    Extract processor, network, and memory information from the hardware.json␣
↪file.
    Returns a dictionary with processor details, NIC models, and installed␣
↪memory.
    """
    if not hardware_json_path or not os.path.exists(hardware_json_path):
        return {
            "cpu_model": "Unknown", "cpu_cores": "Unknown", "cpu_threads":␣
↪"Unknown",
            "memory": "Unknown", "nic_models": "Unknown"
        }

    try:
        with open(hardware_json_path, "r") as f:
            hardware_data = json.load(f)

        cpu_data = hardware_data.get("processor", [{}])[0]
        cpu_model = cpu_data.get("model", "Unknown")
        cpu_cores = cpu_data.get("cores", "Unknown")
        cpu_threads = cpu_data.get("threads", "Unknown")

        nic_models = []
        if isinstance(hardware_data.get("network"), list):
            for nic in hardware_data["network"]:
                if isinstance(nic, dict) and "model" in nic:
                    nic_models.append(nic["model"])

        memory_val = hardware_data.get("memory", {}).
↪get("installed_capacity_human_val", "Unknown")
        memory_unit = hardware_data.get("memory", {}).
↪get("installed_capacity_human_unit", "")
        memory_str = f"{memory_val} {memory_unit}" if isinstance(memory_val,␣
↪(int, float, str)) else "Unknown"

        return {
            "cpu_model": cpu_model if isinstance(cpu_model, str) else "Unknown",
```

```
            "cpu_cores": cpu_cores if isinstance(cpu_cores, int) else "Unknown",
            "cpu_threads": cpu_threads if isinstance(cpu_threads, int) else␣
 ↪"Unknown",
            "memory": f"RAM: {memory_str}" if memory_str != "Unknown" else␣
 ↪"Unknown",
            "nic_models": "<br>".join(nic_models) if nic_models else "No NICs␣
 ↪detected",
        }

    except (json.JSONDecodeError, KeyError, TypeError):
        return {
            "cpu_model": "Unknown", "cpu_cores": "Unknown", "cpu_threads":␣
 ↪"Unknown",
            "memory": "Unknown", "nic_models": "Unknown"
        }

# Load node metadata and hardware details
try:
    nodes_info = load_rocrate_metadata()
    nodes_df = pd.DataFrame(nodes_info)
    hardware_details = [extract_hardware_info(node["hardware_json"]) for node␣
 ↪in nodes_info]
    hardware_df = pd.DataFrame(hardware_details)
    nodes_df = pd.concat([nodes_df, hardware_df], axis=1)
    nodes_df.drop(columns=["hardware_json"], inplace=True)

    def extract_testbed(fqdn):
        parts = fqdn.split(".")
        if len(parts) > 1:
            return parts[1].capitalize()

    nodes_df["Testbed"] = nodes_df["fqdn"].apply(extract_testbed)

    testbed_urls = {
        "Baltikum": "https://kaunas.net.cit.tum.de/",
        "Blockchain": "https://coinbase.net.cit.tum.de/"
    }

    def make_testbed_link(testbed):
        url = testbed_urls.get(testbed, "Unknown")
        return f'<a href="{url}" target="_blank">{testbed}</a>' if url !=␣
 ↪"Unknown" else "Unknown"

    nodes_df["Testbed"] = nodes_df["Testbed"].apply(make_testbed_link)

    def make_clickable(path):
```

```
        return f'<a href="{path}" target="_blank">Open PDF</a>' if path !=␣
↪"None" else "No topology available"

    nodes_df["topology_pdf"] = nodes_df["topology_pdf"].apply(make_clickable)

    # Rename columns for better readability
    nodes_df.rename(columns={
        "name": "Name",
        "fqdn": "FQDN",
        "topology_pdf": "Topology",
        "cpu_model": "CPU",
        "cpu_cores": "Cores",
        "cpu_threads": "Threads",
        "memory": "Memory",
        "nic_models": "NICs",
        "Testbed": "Testbed"
    }, inplace=True)

    nodes_df = nodes_df[["Name", "FQDN", "Testbed", "Topology", "CPU", "Cores",␣
↪"Threads", "Memory", "NICs"]]

    html_table = nodes_df.to_html(escape=False)
    styled_table = f"""
    <style>
        table {{ width: 90%; border-collapse: collapse; margin: 20px 0; }}
        th, td {{ padding: 8px 12px; border: 1px solid #ddd; text-align: left;␣
↪}}
        th {{ background-color: #f4f4f4; font-weight: bold; }}
    </style>
    {html_table}
    """

    display(HTML(styled_table))
except FileNotFoundError:
    display("RO-Crate file not available, will skip")
```

```
'RO-Crate file not available, will skip'
```

1.3 Loading and Previewing Data

The energy measurement data is stored in CSV format, with each node having its own folder inside the `energy` directory.

The dataset includes: - **Timestamp** (`timestamp`): Time when the measurement was recorded. - **Current** (`current_mA`): Measured current in milliamps (mA). - **Voltage** (`voltage_V`): Measured voltage in volts (V). - **Power Consumption** (`power_active_W`): Active power in watts (W). - **Energy Counter** (`energy_counter_Wh`): Cumulative energy usage in watt-hours (Wh).

Below, we load the data and display a preview.

[10]:
```
sns.set_theme(style="whitegrid")
plt.rcParams.update({"axes.titlesize": 14, "axes.labelsize": 12})

def load_energy_data():
    """
    Load all CSV files from the energy folder inside the result folder.
    Each node has its own subfolder containing multiple _runXX.csv files.
    """
    energy_folder = os.path.join(RESULT_FOLDER, "energy")
    if not os.path.exists(energy_folder):
        raise FileNotFoundError(f"Energy folder not found: {energy_folder}")

    all_data = []

    for node in os.listdir(energy_folder):
        node_path = os.path.join(energy_folder, node)
        if os.path.isdir(node_path):
            for file in os.listdir(node_path):
                if file.endswith(".csv") and "_run" in file:
                    file_path = os.path.join(node_path, file)
                    print(file_path)
                    df = pd.read_csv(file_path)

                    df["timestamp"] = pd.to_datetime(df["timestamp"],␣
 ↪format="%Y%m%d%H%M%S%f")
                    df["node"] = node
                    df["run"] = file.split("_run")[-1].split(".")[0]  # Extract␣
 ↪run number
                    all_data.append(df)

    if not all_data:
        raise ValueError("No valid CSV files found in the energy folder.")

    return pd.concat(all_data, ignore_index=True)

df = load_energy_data()

def remove_outliers(df):
    """
    Removes extreme outliers from all numeric columns using the IQR method.
    """
    numeric_cols = df.select_dtypes(include=[np.number]).columns

    for col in numeric_cols:
        Q1 = df[col].quantile(0.25)
```

```
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1

        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]

    return df

df = remove_outliers(df)
display(df.head())
display(df.tail())
```

```
/home/forster/dashboard/.transfer/energy/bitcoin/measurement_run0.csv
/home/forster/dashboard/.transfer/energy/bitcoincash/measurement_run0.csv
/home/forster/dashboard/.transfer/energy/bitcoingold/measurement_run0.csv
```

	timestamp	current_mA	voltage_V	power_active_W	\
0	2025-03-07 12:00:00	-1	-1	32	
1	2025-03-07 12:05:00	-1	-1	32	
2	2025-03-07 12:10:00	-1	-1	32	
3	2025-03-07 12:15:00	-1	-1	32	
4	2025-03-07 12:20:00	-1	-1	32	

	energy_counter_Wh	node	run
0	-1	bitcoin	0
1	-1	bitcoin	0
2	-1	bitcoin	0
3	-1	bitcoin	0
4	-1	bitcoin	0

	timestamp	current_mA	voltage_V	power_active_W	\
214	2025-03-07 17:40:00	-1	-1	31	
215	2025-03-07 17:45:00	-1	-1	31	
216	2025-03-07 17:50:00	-1	-1	31	
217	2025-03-07 17:55:00	-1	-1	31	
218	2025-03-07 18:00:00	-1	-1	31	

	energy_counter_Wh	node	run
214	-1	bitcoingold	0
215	-1	bitcoingold	0
216	-1	bitcoingold	0
217	-1	bitcoingold	0
218	-1	bitcoingold	0

1.4 Data Overview

After loading the data, we analyze its structure using summary statistics.
This helps in identifying potential issues such as missing values, anomalies, or trends.

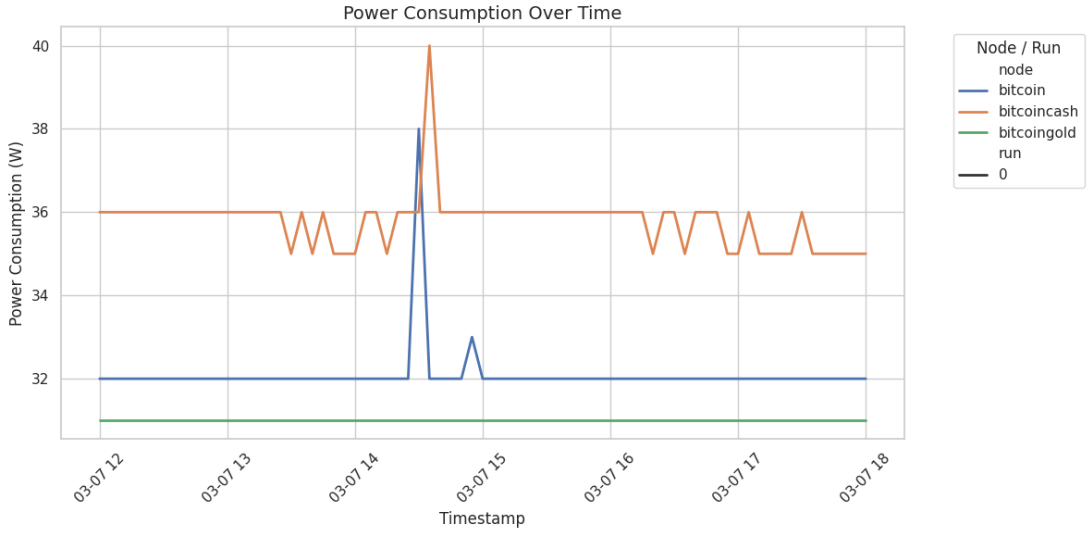
[11]:
```
temp_df = df.drop(columns=['node', 'run'])
display(temp_df.describe(exclude=[np.datetime64]))
```

	current_mA	voltage_V	power_active_W	energy_counter_Wh
count	219.0	219.0	219.000000	219.0
mean	-1.0	-1.0	32.958904	-1.0
std	0.0	0.0	2.125242	0.0
min	-1.0	-1.0	31.000000	-1.0
25%	-1.0	-1.0	31.000000	-1.0
50%	-1.0	-1.0	32.000000	-1.0
75%	-1.0	-1.0	35.000000	-1.0
max	-1.0	-1.0	40.000000	-1.0

1.5 Power Consumption Over Time

The following plot shows the power consumption trends over time for different nodes.
This helps us observe variations in power usage and detect potential anomalies.

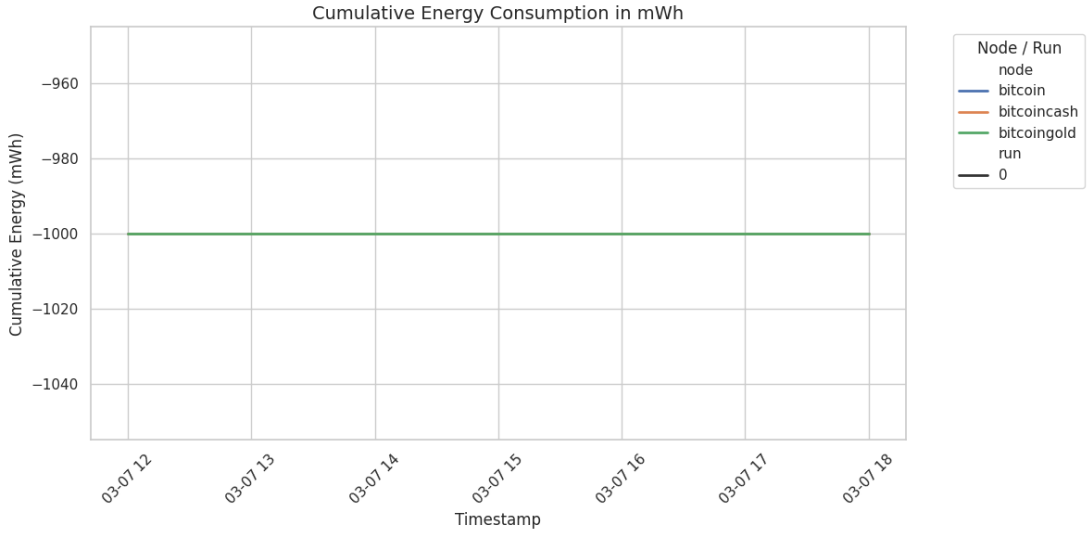
[12]:
```
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x="timestamp", y="power_active_W", hue="node",␣
 ↪style="run", linewidth=2)
plt.xlabel("Timestamp")
plt.ylabel("Power Consumption (W)")
plt.title("Power Consumption Over Time")
plt.xticks(rotation=45)
plt.legend(title="Node / Run", bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```



1.6 Cumulative Energy Consumption

The energy counter represents the cumulative energy consumed over time.
This plot provides insights into the total energy usage per node and how it changes over the experiment duration.

[13]:
```
df["energy_counter_mWh"] = df["energy_counter_Wh"] * 1000

plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x="timestamp", y="energy_counter_mWh", hue="node",␣
 ↪style="run", linewidth=2)

plt.xlabel("Timestamp")
plt.ylabel("Cumulative Energy (mWh)")
plt.title("Cumulative Energy Consumption in mWh")
plt.xticks(rotation=45)
plt.legend(title="Node / Run", bbox_to_anchor=(1.05, 1), loc='upper left')

plt.tight_layout()
plt.show()
```

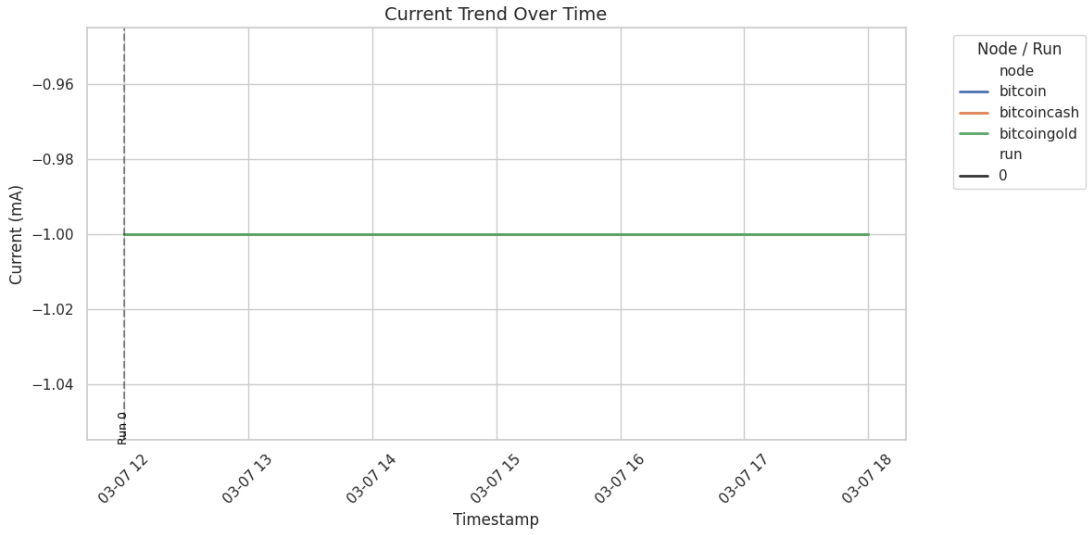


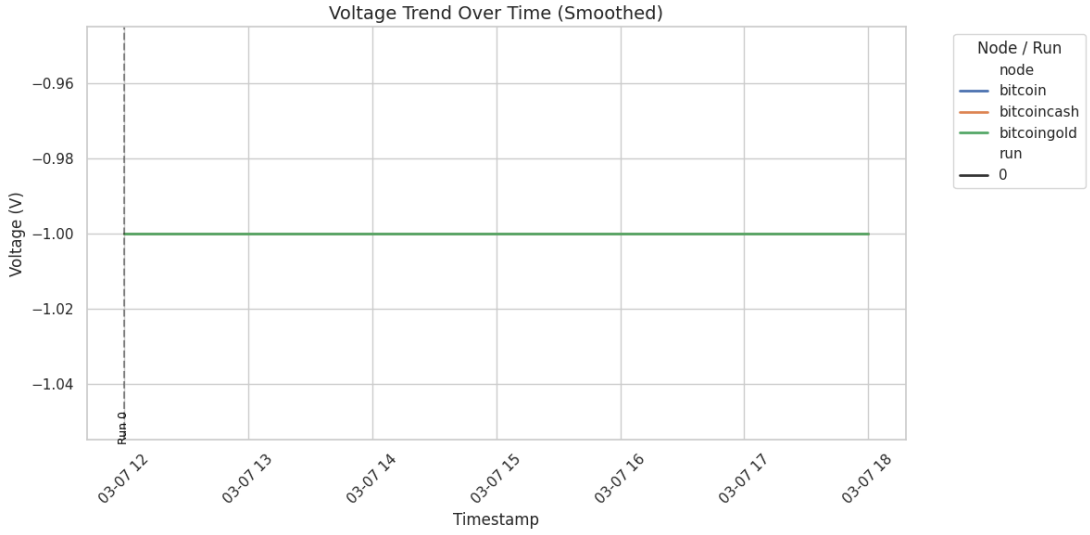
1.7 Current and Voltage Trends

To better understand the electrical characteristics, we visualize: - **Current (mA) over time** to see how power draw fluctuates. - **Voltage (V) over time** to ensure stability across measurements.

[14]:
```
df["voltage_V_smoothed"] = df["voltage_V"].rolling(window=5, min_periods=1).
 ↪mean()
first_timestamps = df.groupby("run")["timestamp"].min()

# --- Plot Current Trend ---
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x="timestamp", y="current_mA", hue="node", style="run",␣
 ↪linewidth=2)
plt.xlabel("Timestamp")
plt.ylabel("Current (mA)")
plt.title("Current Trend Over Time")
plt.xticks(rotation=45)
plt.legend(title="Node / Run", bbox_to_anchor=(1.05, 1), loc='upper left')

# Position run markers at the bottom of the plot
ylim = plt.ylim()
text_ypos = ylim[0] - (ylim[1] - ylim[0]) * 0.01  # 5% below the lowest value

for run, ts in first_timestamps.items():
    plt.axvline(x=ts, color="black", linestyle="dashed", alpha=0.4)
    plt.text(ts, text_ypos, f"Run {run}", rotation=90, fontsize=9,␣
 ↪color="black",
             verticalalignment="bottom", horizontalalignment="center")

plt.tight_layout()
plt.show()

# --- Plot Voltage Trend (Smoothed) ---
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x="timestamp", y="voltage_V_smoothed", hue="node",␣
 ↪style="run", linewidth=2)
plt.xlabel("Timestamp")
plt.ylabel("Voltage (V)")
plt.title("Voltage Trend Over Time (Smoothed)")
plt.xticks(rotation=45)
plt.legend(title="Node / Run", bbox_to_anchor=(1.05, 1), loc='upper left')

# Position run markers at the **bottom** of the plot
ylim = plt.ylim()
text_ypos = ylim[0] - (ylim[1] - ylim[0]) * 0.01  # 5% below the lowest value

for run, ts in first_timestamps.items():
    plt.axvline(x=ts, color="black", linestyle="dashed", alpha=0.4)
    plt.text(ts, text_ypos, f"Run {run}", rotation=90, fontsize=9,␣
 ↪color="black",
             verticalalignment="bottom", horizontalalignment="center")
```

```
plt.tight_layout()
plt.show()
```





1.7.1 Energy Consumption Rate Over Time

This plot shows the **rate at which energy is consumed over time (mW/s)**.
Instead of cumulative energy, this visualization helps identify **periods of high workload**.
A higher energy rate means that the system was **actively consuming more power**,
which may indicate high CPU load or network traffic.