Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

Specialization Project 1

Powerstack: Implementation of an energy monitoring environment in kubernetes

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
January 22, 2025

Study program:
Computer Science, M.Sc.

Imprint

Project: Specialization Project 1
Title: Powerstack: Implementation of an energy monitoring environment in kubernetes
Author: Caspar Wackerle
Date: January 22, 2025
Keywords: energy efficiency, cloud, kubernetes
Copyright: Zurich University of Applied Sciences

Study program:
Computer Science, M.Sc.
Zurich University of Applied Sciences

Supervisor 1:
Prof. Dr. Thomas Bohnert
Zurich University of Applied Sciences
Email: thomas.michael.bohnert@zhaw.ch
Web: Link

Supervisor 2:
Christof Marti
Zurich University of Applied Sciences
Email: christof.marti@zhaw.ch
Web: Link

Abstract

The abstract is like a miniature version of the entire manuscript. Structure it similarly: Begin with the context and motivation for the project, a brief description of the method and available data, your findings, and conclusions. Limit yourself to one page!

Contents

Chapter 1

Introduction and Context

1.1 Significance of Energy Efficiency in Cloud Computing

Cloud computing has revolutionized how computing resources are shared and utilized, offering increased operational efficiency through resource sharing. While economic benefits such as reduced costs and improved scalability are often highlighted, energy efficiency and environmental impact are equally important. By maximizing shared resource utilization, cloud computing inherently reduces energy waste, supporting global sustainability goals.

The rapid adoption of cloud computing has transformed it into a dominant segment of global IT infrastructure. Hyperscalers like Amazon Web Services, Google Cloud, and Microsoft Azure contribute significantly to global energy consumption, attracting attention from policymakers. While cloud providers have incorporated renewable energy sources into their operations, this alone does not address the efficiency of energy utilization for workloads.

Technological advancements have improved the energy efficiency of data centers, with modern facilities achieving Power Usage Effectiveness (PUE) values near 1. However, PUE measures facility-level efficiency, not workload-level efficiency. Even with a perfect PUE of 1, significant energy waste can occur if computational resources are underutilized. This highlights the need to focus on workload-level energy efficiency.

Containers, as lightweight virtualization technology, improve resource density and energy efficiency compared to traditional virtual machines (VMs). Despite these advantages, containers introduce additional complexity, especially in measuring energy consumption. Accurate container-level energy consumption requires granular monitoring tools, a challenge compounded by Kubernetes' dynamic resource allocation and scaling mechanisms.

Despite the importance of energy efficiency in cloud computing, research on this topic remains limited. While data center operations have been optimized and green coding practices promoted, Kubernetes' energy efficiency is underexplored. Addressing this gap is crucial for balancing economic and environmental goals.

1.2 The Need for Energy-Efficient Kubernetes Clusters

Kubernetes has become the standard for container orchestration, managing containerized workloads at scale. However, its success brings new challenges, particularly in energy efficiency. Kubernetes environments are complex, featuring dynamic scaling, resource allocation, and workload distribution across multiple nodes. These features, while essential for performance, complicate energy measurement and optimization.

Commonly, Kubernetes clusters are hosted on VMs to simplify infrastructure management, adding another abstraction layer and further complicating energy measurement. To improve energy efficiency, robust methods for measuring energy consumption in Kubernetes environments are necessary. These methods must translate measurements into valuable insights to that will lay the foundation for optimization efforts.

Given the growing energy footprint of cloud computing and the limited research focus on this area, energy-efficient Kubernetes clusters represent a pressing research topic. By addressing this gap, this work contributes to the broader goal of sustainable cloud computing.

1.3 Objectives and Scope of this Thesis

1.3.1 Context

This thesis is part of the Master's program in Computer Science at the Zurich University of Applied Sciences (ZHAW) and represents the first of two specialization projects. The current project (VT1) focuses on the practical implementation of a test environment for energy efficiency research in Kubernetes clusters. The second project (VT2) will explore theoretical aspects and methodologies for measuring and improving energy efficiency.

This thesis builds upon prior works focused on performance optimization and energy measurement. EVA1 covered topics such as operating system tools, statistics, and eBPF, while EVA2 explored energy measurement in computer systems, covering hardware, firmware, and software aspects. These foundational topics provide the basis for the current thesis but will not be revisited in detail.

1.3.2 Scope

This thesis focuses on the practical implementation of a test environment, excluding detailed theoretical analysis and literature reviews. The primary goal is to document the creation of a reliable and reproducible test environment that supports future research on energy efficiency in Kubernetes clusters.

1.3.3 Objectives

The main objective is to design and implement a test environment that facilitates:

- Analysis of key parameters affecting energy efficiency in Kubernetes clusters.
- Reliable and consistent experimentation.

- Reproducibility and automation in deployment and configuration.

The outcomes will provide the necessary infrastructure for subsequent research projects.

Parameters for Analysis

This project aims to reuse established tools and components where feasible. The parameters to be analyzed include:

- CPU utilization and energy consumption.
- Memory usage and its impact on power draw.
- Disk I/O and storage-related power consumption.

Additional parameters may be incorporated based on further evaluation.

Data Integrity and Persistence

Ensuring data integrity and persistence is critical for reliable analysis. Key requirements include:

- Persistent storage that survives system shutdown.
- A unified data store accessible by all nodes.
- Data retention across Kubernetes cluster reinstallations.
- The ability to power down unused worker nodes without data loss.

Reproducibility and Automation

Reproducibility and automation are bonus goals aimed at enhancing research efficiency. Benefits include:

- Simplified recovery from misconfiguration through rapid redeployment.
- Reduced troubleshooting time.
- Improved stack cleanliness by eliminating residual configurations.

Security

Security, while not a primary focus, will be addressed by implementing basic best practices. Key security measures include:

- Use of encrypted passwords.
- Adherence to basic Kubernetes security best practices.
- Minimization of potential vulnerabilities through careful configuration.

Chapter 2

Architecture and Design

2.1 Overview of Test Environment

The test environment consists of a Kubernetes cluster deployed on three bare-metal servers housed in a university datacenter. The three servers are identical in hardware specifications and connected through both a private network and the university network. The setup allows complete remote management and ensures direct communication between the servers for Kubernetes workloads. Below is a detailed description of the hardware and network topology. A diagram illustrating the architecture and network setup is provided in figure 2.1.

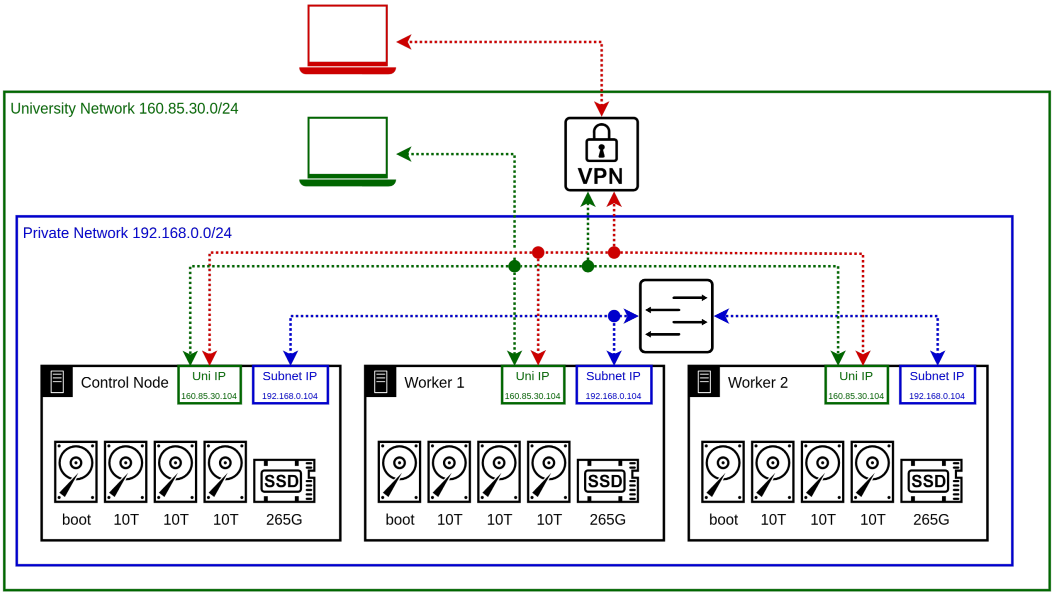

FIGURE 2.1: Physical Infrastructure Diagram

2.1.1 Hardware and Network

Bare-Metal Servers

The cluster is built using three identical Lenovo ThinkSystem SR530 servers, each equipped with the following hardware:

- CPU: 1x Intel(R) Xeon(R) Bronze 3104 @ 1.70GHz, 6 cores.

- Memory: 4x 16GB DDR4 DIMMs, totaling 64GB of RAM per server.
- Storage:
  - 2x 32GB M.2 SATA SSD for the operating system boot drive.
  - 1x 240GB 6Gbps SATA 2.5" SSD for persistent storage.
  - 3x 10TB 7.2K RPM 12Gbps SAS 3.5" HDDs for bulk storage.
- Power Supply: Dual redundant power supplies.
- Cooling: 4 out of 6 possible fans installed.
- Firmware:
  - BMC Version: 8.88 (Build ID: CDI3A4A)
  - UEFI Version: 3.42 (Build ID: TEE180J)
  - LXPM Version: 2.08 (Build ID: PDL142H)

The servers are equipped with Lenovo XClarity Controller (BMC) for remote management. Each server can be accessed via its BMC IP address for out-of-band management and monitoring.

Network Topology

The servers are connected using two distinct networks:

- **Private Network:** Each server has a private IP address (192.168.0.104–192.168.0.106), allowing direct, high-speed communication between nodes. This reduces the load on the university network and improves Kubernetes workload performance.
- **University Network:** Public-facing IP addresses (160.85.30.104–160.85.30.106) allow access within the university network, with external access enabled via VPN.

Note: Detailed switch and gateway configurations are managed by the university IT department and are beyond the scope of this document.

2.2 Key Technologies

2.2.1 Ubuntu

Ubuntu was chosen as the operating system for this project primarily due to the author's familiarity with it. Additionally, it was already installed on the servers when they were received, which saved time and reduced setup complexity. While there are other Linux distributions specifically designed for Kubernetes, using a familiar distribution ensured smoother initial configuration and operation.

2.2.2 Bare-Metal K3s

Installing Kubernetes directly on bare-metal servers (without using a hypervisor or virtual machines) was a fundamental decision to ensure direct access to hardware-level data. This approach allows Kubernetes to interact with the underlying hardware more effectively, which is critical for accurate energy consumption monitoring.

K3s was chosen for several reasons:

- It is lightweight, making it suitable even for weaker servers, while potentially also lowering energy consumption.
- Despite its lightweight nature, it remains fully compatible with stock Kubernetes, ensuring that standard Kubernetes resources and configurations can be used without modification.
- K3s is optimized for ARM architectures, making it ideal for deployment on devices like Raspberry Pis in a homelab environment.
- The author had prior experience with K3s and Rancher, which contributed to a faster and smoother deployment.

2.2.3 Ansible, Helm, Kubectl

For automation, Ansible and Helm were selected. Helm and Kubectl were an obvious choice due to their widespread use in Kubernetes for managing and deploying applications.

Ansible was chosen for its flexibility and ease of use in managing server configurations and automating repetitive tasks across multiple nodes. Additionally, Ansible's agentless architecture simplifies the management of bare-metal servers by requiring only SSH access and Python installed on the target machines.

2.2.4 Kube-Prometheus Stack

The Kube-Prometheus stack was chosen because it is the de-facto standard for monitoring in Kubernetes environments. This project has reached a high level of maturity, offering robust features and a wide range of integrations. Installation and configuration using Helm are straightforward, and the abundance of available resources makes troubleshooting easier.

Prometheus

Prometheus was selected as the primary monitoring tool because it is the standard in the Kubernetes ecosystem. Despite its advantages, Prometheus has some downsides: it can introduce significant overhead, and it is not suitable for monitoring low-second or sub-second intervals due to typical scrape intervals being longer. However, for container orchestration, where longer container lifetimes are expected, this limitation is acceptable.

Grafana

Grafana was chosen for its ability to provide excellent, customizable visualizations of metrics collected by Prometheus. It enables easy interpretation of complex data through dashboards and visual aids, making it a valuable addition to the monitoring stack.

AlertManager

AlertManager is included in the Kube-Prometheus stack and is used to handle alerts generated by Prometheus. While it was not utilized in this project, its inclusion is welcomed for potential future use in managing alerts and notifications in a production environment.

2.2.5 KEPLER

Purpose of KEPLER

KEPLER, or *Kubernetes-based Efficient Power Level Exporter*, is a promising project focused on measuring energy consumption in Kubernetes environments. It provides detailed power consumption metrics at the process, container, and pod levels, addressing the growing need for energy-efficient cloud computing.

With cloud providers and enterprises under increasing pressure to improve energy efficiency, KEPLER offers a practical solution. By enabling detailed real-time measurement of power usage, it bridges the gap between high-level infrastructure metrics and workload-specific energy consumption data. This capability makes KEPLER a valuable tool in advancing energy-efficient Kubernetes clusters.

Limitations of KEPLER

Despite its potential, KEPLER has some limitations in the context of this project:

- Active Development: KEPLER is still in active development, meaning its features and APIs may change over time. Additionally, the documentation is currently limited, and there are few community resources available for troubleshooting.

- Complexity: As a large and complex project, adapting KEPLER beyond basic configuration requires a deep understanding of its architecture. Implementing custom changes or enhancements can be challenging without significant expertise.

While KEPLER may not be perfect, it is currently the most promising approach to addressing the challenge of measuring energy consumption in Kubernetes environments. Consequently, a large focus of this thesis will be on evaluating KEPLER's capabilities and identifying areas for improvement.

2.3 Architecture and Design

2.3.1 Kubernetes Cluster Design

The Kubernetes cluster is deployed on three bare-metal servers running Ubuntu. One server is designated as the control plane, while the other two serve as worker nodes. This setup avoids high availability (HA) for simplicity, given the scope of this project. The servers are connected via their internal IP addresses, enabling direct communication without routing through external networks. All Kubernetes components, including the API server, controller manager, and scheduler, run exclusively on the control plane node, while workloads are distributed across all nodes by Kubernetes. Figure 2.1 provides an overview of the system architecture, including components and data flow.

2.3.2 Persistent Storage

Persistent storage is provided using the spare SSD disk on the control node. A partition on the disk is created, formatted with the BTRFS file system, and mounted. The NFS server is installed on the control node, and NFS clients are installed on the worker nodes, enabling them to access the shared storage. This centralized approach was chosen because the control node is the only server guaranteed to remain powered on throughout the experiment, ruling out the need for a distributed storage solution like CEPH.

Within the NFS share, separate directories are created for Prometheus and Grafana data. Persistent volumes (PVs) are defined in Kubernetes, and persistent volume claims (PVCs) are created for each service. The size of these PVs can be configured during installation, allowing flexibility for future storage needs.

2.3.3 Monitoring Architecture

The monitoring stack is deployed using the kube-prometheus-stack Helm chart. This stack includes Prometheus, Grafana, and AlertManager, providing a complete solution for monitoring, visualizing, and managing alerts in Kubernetes. Prometheus is configured to scrape metrics from KEPLER and Kubernetes endpoints (such as the kubelet API) at regular intervals. Grafana connects to Prometheus, enabling real-time visualization of metrics through customizable dashboards.

2.3.4 Metrics Collection and Storage

KEPLER generates metrics by collecting data from various sources:

- **Hardware-level metrics:** Using eBPF and kernel tracepoints to gather low-level data such as CPU cycles and cache misses.
- **Power-related metrics:** Collected via RAPL (Running Average Power Limit) and IPMI (Intelligent Platform Management Interface) to monitor CPU and platform energy consumption.
- **Container-level metrics:** Retrieved from the Kubernetes kubelet API, which provides cgroup resource usage data for running containers and pods.

KEPLER aggregates this data, calculates power consumption metrics, and exposes them in a Prometheus-friendly format. Prometheus scrapes these metrics at a configurable interval, storing them as time series data on the persistent volume. The time series format allows Prometheus to track changes over time, enabling detailed analysis of resource usage patterns. In chapter 2.4, the KEPLER architecture is briefly explained with a focus on metrics collection and generation. The information flow is pictured in diagram 2.2.

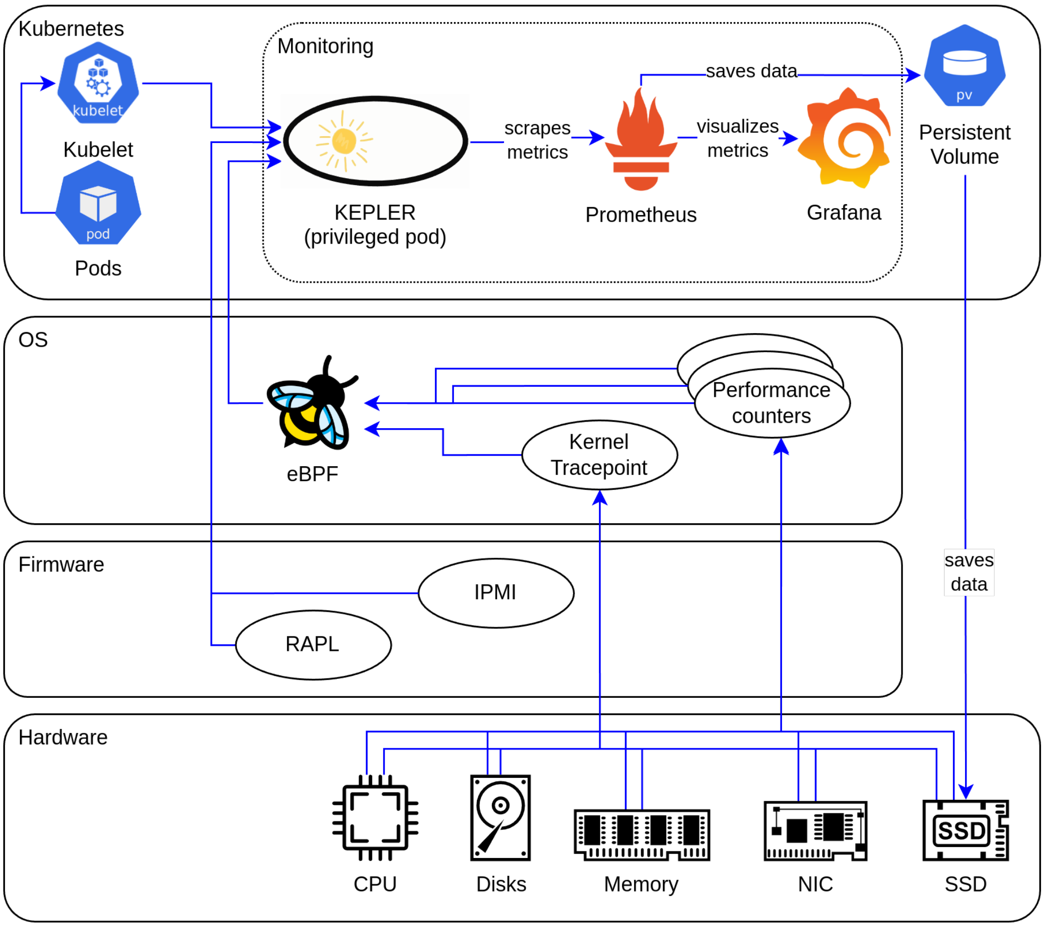

FIGURE 2.2: Monitoring data flow diagram of the entire stack

Since the KEPLER project has already done a significant amount of research, as well as practical implementation, the decision was made to utilize KEPLER as a core component in this project.

2.3.5 Repository Structure

The repository for this project is designed to include all aspects of the Kubernetes-based energy efficiency test environment, from deployment automation to documentation. Given the reliance on various external projects, a hybrid approach was adopted for managing dependencies:

Submodules for External Repositories

Several external projects with frequent updates were forked and included as submodules in the repository. This approach allows easy configuration and customization while maintaining the ability to sync changes from upstream repositories. Additionally, by freezing submodules at specific commits, the project is protected from unexpected upstream changes that could introduce instability.

Direct Deployment from External Repositories

For other external projects that require minimal customization, direct deployment from their original repositories was chosen. This reduces the complexity of repository maintenance and ensures that stable, tested versions are always used.

Structure Overview

The repository is organized to maintain clarity and separation of concerns:

- **ansible/:** Ansible playbooks and roles for automated deployments.
- **helm/:** Custom or external Helm charts managed through Ansible.
- **scripts/:** Bash scripts for executing Ansible playbooks.
- **config/:** Centralized configuration file and Ansible vault.
- **docs/:** Documentation files containing setup and usage of the project.
- **thesis/:** Contains all files related to the thesis, written in LaTeX.

2.3.6 Automation Architecture

Automation was a key focus in this project to ensure reproducibility, consistency, and ease of deployment. The automation architecture is primarily based on Ansible, with Helm nested into Ansible playbooks for Kubernetes-specific deployments.

Ansible and Helm Integration

Ansible was used for automating the setup of the base environment, including system-level configurations and Kubernetes deployments. All Helm installations, such as the kube-prometheus-stack, were wrapped in Ansible playbooks. This approach provided a unified automation framework where both system configurations and Kubernetes resources could be managed together. This also allowed for clear version control and logging of every deployment step.

Execution Scripts

Custom Bash scripts were written to handle the execution of Ansible playbooks. Apart from convenience, these scripts ensured:

- Correct execution context and configuration for playbook execution.
- Automatic log creation aiding in troubleshooting and auditing.

Centralized Configuration

All configuration values, such as IP addresses, storage paths, and deployment options, were centralized in a single configuration file. This design simplifies re-deployment on different hardware by only requiring changes in one location. When necessary, Jinja templates were used in Ansible to dynamically adapt configurations based on this central file.

Security

Sensitive information, such as passwords and API keys, was encrypted using an Ansible Vault. This ensured that confidential data could be securely managed within the repository without compromising security during deployment.

2.4 KEPLER Architecture and Metrics Collection

Since KEPLER plays a central role in this project, it is important to understand its architecture and how it collects metrics. This section provides a brief overview of KEPLER's components and its data collection methods. For more detailed information, the official KEPLER documentation[1] should be consulted.

2.4.1 KEPLER Components

KEPLER Exporter

The core component of KEPLER is the Exporter, which runs as a privileged daemonset pod on each node in the Kubernetes cluster. This exporter directly interacts with the hardware and kernel, collecting energy consumption and resource utilization metrics. It estimates power usage at the process, container, and pod levels, exposing the collected metrics in a Prometheus-friendly format.

A service monitor is also deployed, allowing Prometheus to scrape metrics from the KEPLER exporter endpoints.

KEPLER Model Server

Although the KEPLER Model Server is not used in this project, it is worth noting its purpose. The model server provides power estimation models based on available metrics, supporting different granularities such as node, pod, or processor component levels. It can also include an online trainer to update models dynamically during runtime.

2.4.2 KEPLER Data Collection

Process and Container Data

KEPLER employs eBPF to collect detailed CPU event data. eBPF programs run in a privileged context, enabling efficient, low-overhead monitoring of kernel-level events. Specifically, KEPLER hooks into the `finish_task_switch` kernel function, which handles task context switching, to collect process-level metrics, specifically the following Perf counters:

- `PERF_COUNT_HW_CPU_CYCLES`

- `PERF_COUNT_HW_REF_CPU_CYCLES`,
- `PERF_COUNT_HW_INSTRUCTIONS`
- `PERF_COUNT_HW_CACHE_MISSES`

By maintaining a BPF hash of process IDs, CPU IDs, and context switch timestamps, KEPLER correlates resource usage to individual processes and containers. This data is essential for deriving energy consumption estimates. The hash is shown in table 2.1.

TABLE 2.1: Hardware CPU events monitored by KEPLER

Key	Value	Description
pid	pid	Process ID
	cgroupid	Process CGroupID
	process_run_time	Total time a process occupies CPU (calculated each time process leaves CPU on context switch)
	cpu_cycles	Total CPU cycles consumed by process
	cpu_instr	Total CPU instructions consumed by process
	cache_miss	Total Cache miss by process
	page_cache_hit	Total hit of the page cache
	vec_nr	Total number of soft irq handles by process (max 10)
	comm	Process name (max length 16)

CPU Power Data

KEPLER leverages Intel RAPL (Running Average Power Limit) to monitor energy consumption across various CPU domains, including cores, DRAM, and integrated GPUs. RAPL provides real-time power consumption data with fine granularity and high sampling rates, allowing KEPLER to measure energy usage accurately. The supported power domains include:

- **Package (PKG):** Total energy consumption of the CPU socket, including cores, caches, and memory controllers.
- **Power Plane 0 (PP0):** Energy consumption of CPU cores.
- **Power Plane 1 (PP1):** Energy consumption of integrated GPUs (if present).
- **DRAM:** Energy consumption of memory attached to the CPU.

KEPLER uses the following methods to access RAPL data (in order of preference):

1. **RAPL Sysfs:** Direct access to energy counters via the Linux power capping framework located in `/sysfs`. This requires root access to the powercap driver and is the method used in this project.
2. **RAPL MSR:** Direct access through Model-Specific Registers (MSRs), providing detailed energy readings.
3. **Kernel Driver xgene-hwmon:** Used in specific ARM architectures.

Platform Power Information

KEPLER can also collect platform-level power consumption data, representing the total power usage of the node. This is achieved through:

- **ACPI (Advanced Configuration and Power Interface):** Used to access system-level power information.
- **IPMI (Intelligent Platform Management Interface):** Provides remote access to power data via the Baseboard Management Controller (BMC).

2.4.3 KEPLER Power Model

KEPLER uses a combination of two power modeling approaches, choosing a suitable approach based on available data: If total power is known, KEPLER uses a power ration modelling to compute finer-grained power figures for individual components at the node and container level. If detailed hardware-level power metrics are unavailable, such as in virtualized environments, KEPLER estimates power consumption based on system utilization metrics using a pretrained model (currently based on a Intel Xeon E5-2667 v3-processor). Since this modeling is inherintly flawed for any other processor, a goal of the project is to offer a higher number of models with other architectures.

In previous experiments conducted by the author, KEPLER was deployed on a Kubernetes cluster with virtualized nodes in an Openstack environment. Since no hardware-level power information was available, KEPLER attempted to estimate power consumption solely based on system metrics. The results were inconsistent and unreliable, highlighting the importance of accurate hardware data for meaningful power consumption analysis.

2.4.4 Metrics produced by KEPLER

KEPLER collects and exports a wide range of metrics related to energy consumption and resource utilization. The full list and description of metrics is provided in Appendix A, but they are summarized here.

Container-level metrics

On a container level, KEPLER estimates the total energy consumption in joules. The consumption is broken down to the components Core, DRAM, 'Uncore' (such as fast-level cache and memory controllers), the entire CPU package, GPU and other. Additionally, several resource utilization metrics are calculated, namely the total CPU time, cycles, instructions and cache misses. Some IRQ metrics are provided, namely the total number of transmitted and received network packets, and the number of block I/O operations.

Node-level metrics

On a node level, total energy consumption is once again estimated in joules. Estimations are provided for the entire node, as well as the Core, DRAM, uncore, CPU package, GPU, platform and other. Additionally, Node-specific metadata (such as the CPU archtecure), aggregated metrics (used by the KEPLER model server), and Intel QAT utilization are provided.

Chapter 3

Implementation

This chapter describes the implementation and configuration of the various components used in this project. All automation scripts are designed to be idempotent. All scripts can be executed with shell scripts in the `Powerstack/scripts`-directory. Generally, all configuration is to be done in the central configuration file (`/Powerstack/configs/inventory.yml`) unless otherwise stated. Sensitive information is to be defined in the ansible-vault file
(`/Powerstack/configs/vault.yml`). To keep this chapter brief, instructions for verification are written in Appendix B

3.1 K3s Installation

This section describes the steps involved in setting up a Kubernetes cluster using K3s on bare-metal servers. The installation was automated using an Ansible playbook forked from the official k3s-io/k3s-ansible [2] repository, with necessary customizations for internal IP-based communication.

3.1.1 Preparing the Nodes

Before running the Ansible playbook, the following prerequisites need to be in place on all servers:

- **Operating System:** Ubuntu 22.04 (used kernel version 5.15.0)
- **Passwordless SSH:** Passwordless SSH access must be configured for a user with sudo privileges on all servers.
- **Networking:** Each server should have both an internal IP (for cluster communication) and an external IP (for access via VPN or external management).
- Local Ansible Control Node Setup:
  - **Ansible-community** 9.2.0 (must be 8.0+).
  - **Python** 3.12.3 and Jinja 3.1.2 installed as dependencies.
  - **kubectl** 1.31.3

3.1.2 K3s Installation with Ansible

The playbook supports x64, arm64, and armhf architectures. For this project, it was tested on x64 architecture only.

Configuration Details

- Internal and external IP addresses of all servers must be specified.
- One server must be designated as the control node.
- Default configurations such as ansible-port, ansible-user, and k3s-version can be changed if needed.

Kubectl Configuration

- The playbook automatically sets up kubectl for the user on the Ansible control node by copying the Kubernetes config file from the control node to the local machine.
- The user must rename the config file from 'config-new' to 'config' and set the context to powerstack using the following command:
  `kubectl config use-context powerstack`

3.2 NFS Installation and Setup

3.2.1 NFS Installation with Ansible

Setting up the NFS server and clients was fully automated using an Ansible playbook. Before beginning the automated setup, the following manual step must be completed:

- **Disk Selection:** A suitable disk must be chosen on the control node to act as persistent storage. It is important to note that this disk will be reformatted, and all existing data will be lost.

The Ansible playbook performs the following actions:

- **Disk Preparation:** The selected disk is partitioned (if necessary) and formatted with a single Btrfs partition. The entire disk space is allocated to this partition. The partition is then mounted to `/mnt/data`, and an entry is added to `/etc/fstab` to ensure persistence across reboots.
- **NFS Server Setup:** The `nfs-kernel-server` package is installed and configured on the control node. The directory `/mnt/data` is exported as an NFS share, accessible to the worker nodes.
- **NFS Client Setup:** On each worker node, the `nfs-common` package is installed. The NFS share is mounted, and an `/etc/fstab` entry is created to ensure persistence across reboots.

Configuration Details

- The nfs network must be specified, and the control and worker nodes must be in that network
- The export path must be specified

3.3 Rancher Installation and Setup

3.3.1 Rancher Installation with Ansible and Helm

Although not strictly necessary for the project, Rancher was deployed in the `cattle-system` namespace to assist with debugging and system analysis. The installation was automated using an Ansible playbook, which integrates Helm for deploying Rancher and its dependencies. The key steps are as follows:

- **Helm Installation:** Helm was installed on the control node to facilitate the deployment of Rancher and its dependencies.
- **Namespace Creation:** The `cattle-system` namespace was created to host the Rancher deployment.
- **Cert-Manager Deployment:** Cert-Manager, a prerequisite for Rancher, was installed to manage TLS certificates.
- **Rancher Deployment:** Rancher was installed using the official Helm chart. During installation, the following parameters were configured:
  - **Hostname:** A Rancher hostname was defined to enable access.
  - The Helm chart was configured with the `-set tls=external` option to enable external access to Rancher.
  - **Bootstrap Password:** A secure bootstrap password was set for the default Rancher administrator account.
- **Ingress Configuration:** An ingress resource was configured to route traffic to Rancher, allowing access through the defined hostname.

3.4 Monitoring Stack Installation and Setup with Ansible

The monitoring stack, comprising Prometheus, Grafana, and AlertManager, was deployed using the kube-prometheus-stack[3]-Helm chart from the `prometheus-community/helm-charts` repository. While the repository was forked for convenience, no changes were made to the upstream chart, ensuring compatibility with future updates.

3.4.1 Prometheus and Grafana Installation with Ansible and Helm

The installation process was automated using Ansible roles, ensuring idempotency and centralization of configurations. The following key steps were executed:

- **Persistent Storage Configuration:**

  - Directories for Prometheus, Grafana, and AlertManager were created on the NFS-mounted disk.
  - A custom `StorageClass` was defined for the NFS storage. The default storage `StorageClass` local-path was overridden to be non-default.
  - PersistentVolumes (PVs) were created for Prometheus, Grafana, and AlertManager. A PersistentVolumeClaim (PVC) was explicitly created for Grafana, while PVCs for Prometheus and AlertManager were managed by the Helm chart.

- **Helm Chart Installation:**
  - A Helm values file was generated dynamically using a Jinja template. This template incorporated variables from the central Ansible configuration file to ensure consistency. Sensitive information, such as the Grafana admin password, was included in the values file. To mitigate potential security risks, the values file was removed from the control node after installation.
  - The Helm chart was installed using an Ansible playbook. The following customizations were applied via the generated values file:
    * PVC sizes for Prometheus and AlertManager were set based on the central configuration.
    * A Grafana admin password was defined.
    * Prometheus scrape configurations were adjusted to include the KEPLER endpoints.
    * Changes to the `securityContext` were made to allow Prometheus to scrape KEPLER metrics.

- **Service Port Forwarding:**
  - Prometheus, Grafana, and AlertManager services were exposed using static `NodePort`s defined in the central configuration file, enabling external access.

- **Cleanup:**
  - A cleanup playbook was executed to remove sensitive configuration files from both the control node and the Ansible control node.

3.4.2 Removal Playbook

An Ansible playbook was created to handle the complete uninstallation of the monitoring stack. This was necessary to ensure that PVs and PVCs were explicitly removed to avoid residual artifacts in the Kubernetes cluster.

3.5 KEPLER Installation and Setup with Ansible and Helm

3.5.1 Preparing the Environment

The KEPLER deployment uses the official KEPLER Helm chart repository. Before deploying KEPLER, several prerequisites must be addressed to ensure proper functionality.

Redfish Interface

The Redfish Scalable Platforms Management API is a RESTful API specification for out-of-band systems management. On the Lenovo servers used in this project, Redfish exposes IPMI power metrics, which KEPLER accesses through its Redfish interface. To verify Redfish functionality, navigate to the Lenovo XClarity Controller and ensure the following setting is enabled:

- **IPMI over LAN:** This option can be found under `Network -> Service Enablement and Port Assignment`.

The Redfish API can be tested by visiting the following endpoints in a web browser:

- General Redfish information: `https://<BMC-IP>/redfish/v1`
- Power metrics: `https://<BMC-IP>/redfish/v1/Chassis/1/Power#\PowerControl`

Kernel Configuration

KEPLER requires kernel-level access for eBPF tracing, which involves setting a tracepoint using the `syscall perf_event_open`. By default, this syscall is restricted. To allow KEPLER to function properly, an Ansible role is used to modify the kernel parameter `perf_event_paranoid` via `sysctl` without requiring a reboot.

The restriction level can be verified by checking the value of `/proc/sys/kernel/perf_event_par` For this project, all restrictions were removed by setting the value to `-1`.

3.5.2 KEPLER Deployment with Ansible and Helm

KEPLER is deployed using the KEPLER Helm chart[4] from the `sustainable-computing-io/kepler-helm-chart` repository, with Ansible automating the configuration and deployment process. The deployment configuration is centralized in a Jinja template, which is rendered locally and copied to the control node before applying it.

Key configurations in the Helm values file include:

- **Enabled metrics:** Various metric sources are enabled for detailed energy monitoring.
- **Service port:** The KEPLER service port is defined for Prometheus to scrape metrics.
- **Service interval:** The KEPLER service interval is set to 10 seconds.

- **Redfish metrics:** Redfish/IPMI metrics are enabled, and Redfish API credentials are provided. The Redfish credentials are the same as those used for the Lenovo XClarity Controller interface. Note that the BMC IP address differs from the node IP address.

3.5.3 Verifying KEPLER Metrics

After deployment, it is essential to verify that KEPLER is correctly collecting and exposing metrics. Verification involves the following steps:

3.5.4 Prometheus scraping

Use the Prometheus web interface to check that KEPLER endpoints are being scraped successfully.

3.5.5 Metric availability

Ensure that key metrics are available in Prometheus, and non-zero values are written. Each metric discloses exactly one `source`.

3.5.6 Kepler logs

Check the KEPLER logs to gain insight into the used sources:

LISTING 3.1: External file: `kepler.log`

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:103] Kepler running on version: v0.7.12-dirty
1 config.go:293] using gCgroup ID in the BPF program: true
1 config.go:295] kernel version: 5.15
1 config.go:322] The Idle power will be exposed. Are you running on Baremetal or using single VM per node?
1 power.go:59] use sysfs to obtain power
1 node_cred.go:46] use csv file to obtain node credential
1 power.go:79] using redfish to obtain power
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:84] Number of CPUs: 6
1 watcher.go:83] Using in cluster k8s config
1 reflector.go:351] Caches populated for *v1.Pod from pkg/kubernetes/watcher.go:211
1 watcher.go:229] k8s APIserver watcher was started
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_TOTAL Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms]
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_COMPONENTS Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms bpf_cpu_time_ms bpf_cpu_time_ms   gpu_compute_util]
1 node_component_energy.go:62] Skipping creation of Node Component Power Model since the system collection is
     supported
1 prometheus_collector.go:90] Registered Process Prometheus metrics
1 prometheus_collector.go:95] Registered Container Prometheus metrics
1 prometheus_collector.go:100] Registered VM Prometheus metrics
1 prometheus_collector.go:104] Registered Node Prometheus metrics
1 exporter.go:194] starting to listen on 0.0.0.0:9102
1 exporter.go:208] Started Kepler in 2.2651724s
```

Power from ACPI / IMPI

In the event that both ACPI and IPMI were configured to measure platform power, KEPLER prefered to use IPMI as its primary data source. This is to be expected, since IPMI allows to measure power at a higher level than ACPI, and can get detailed power data for the entire plaform. In the absence of IPMI data, KEPLER uses ACPI as the only power data source.

Redfish issues

KEPLER was sporadically unable to correcly handle individual redfish data values. These incedents were sparse for different data values. Unfortunately, the source of this issue could not be eliminated in the context of this thesis. The log below shows an instance of this error

```
Failed to get power: json: cannot unmarshal number 3.07 into Go struct
    field Voltages.Voltages.ReadingVolts of type int
```

Error Message cpu0/online

The following error message is notable:

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/
    online: no such file or directory
```

The missing file warning can be attributed to the fact that the Intel Xeon used in this project does not support Core Offlining, i.e. the dynamic disableling of individual CPU Cores at runtime. While Core Offlining is an interesting feature for energy-efficiency ananlysis, this can be acccepted as a hardware limitation of this project.

Chapter 4

Analysis

Appendix A

KEPLER-provided power metrics

A.1 Summary of KEPLER-Produced Metrics

KEPLER collects metrics at both the extbfcontainer and extbfnode levels, focusing on energy consumption, resource utilization, and platform-specific data.

A.1.1 Container-Level Metrics

Energy Consumption

- **kepler_container_joules_total**: Total energy consumed by a container, aggregated from various components.
- **kepler_container_core_joules_total**: Energy consumed by CPU cores.
- **kepler_container_dram_joules_total**: Energy consumed by DRAM.
- **kepler_container_uncore_joules_total**: Energy consumed by uncore components, such as last-level cache and memory controllers.
- **kepler_container_package_joules_total**: Energy consumed by the entire CPU package, including all cores and uncore components.
- **kepler_container_other_joules_total**: Energy consumed by other host components, derived from subtracting CPU and DRAM energy from total system power.
- **kepler_container_gpu_joules_total**: Energy consumed by GPUs (currently supports NVIDIA GPUs).

Resource Utilization

- **Base Metric**
  - **kepler_container_bpf_cpu_time_us_total**: Total CPU time used by a container, measured using eBPF.
- **Hardware Counter Metrics**
  - **kepler_container_cpu_cycles_total**: Total CPU cycles.

  - **kepler_container_cpu_instructions_total**: Total instructions executed.
  - **kepler_container_cache_miss_total**: Total last-level cache misses.
- **IRQ Metrics**
  - **kepler_container_bpf_net_tx_irq_total**: Transmitted network packets.
  - **kepler_container_bpf_net_rx_irq_total**: Received network packets.
  - **kepler_container_bpf_block_irq_total**: Block I/O operations.

A.1.2 Node-Level Metrics

Energy Consumption

- **kepler_node_core_joules_total**: Total energy consumed by all CPU cores on the node.
- **kepler_node_dram_joules_total**: Total energy consumed by DRAM on the node.
- **kepler_node_uncore_joules_total**: Total energy consumed by uncore components.
- **kepler_node_package_joules_total**: Total energy consumed by the CPU package.
- **kepler_node_other_joules_total**: Total energy consumed by other host components.
- **kepler_node_gpu_joules_total**: Total energy consumed by GPUs.
- **kepler_node_platform_joules_total**: Total energy consumption of the entire node (measured via ACPI/IPMI).

Node Metadata

- **kepler_node_info**: Node-specific information, including CPU architecture.
- **kepler_node_energy_stat**: Aggregated metrics used by the model server for further predictions.

Accelerator Metrics

- **kepler_node_accelerator_intel_qat**: Utilization of Intel QAT accelerators, if available.

Appendix B

System Verification Instructions

B.1 K3s

- **Verify Node Status:** Run `kubectl get nodes -o wide` on the local machine to check that all nodes are registered and ready. Ensure all nodes appear as `Ready` and are using their internal IP addresses for communication.
- **Verify K3s Services:** On each control node, verify that the K3s service is running using `systemctl status k3s`. For worker nodes, use `systemctl status k3s-agent`.
- **Verify Pod Status:** Use `kubectl get pods -A` to ensure all Kubernetes system pods are running without errors. Confirm that critical pods in the `kube-system` namespace are running.
- **Verify Cluster Access:** Test basic Kubernetes commands from the local machine:
  - `kubectl get namespaces`
  - `kubectl get pods -all-namespaces`
- **Network Connectivity:** Ping the internal IPs of other nodes from each server to verify internal connectivity. Ensure external IP connectivity is functional through the VPN.
- **Re-run Ansible Playbook:** Execute `sh scripts/deploy_k3s.sh` to confirm idempotency of the playbook. Verify that no errors occur and all tasks complete without unnecessary changes.

B.2 NFS Server

- **Disk Verification:** On the control node, verify the disk is mounted correctly using `/etc/fstab` and `df -h`. Confirm that `/mnt/data` is listed with the expected disk size and mount point.
- **NFS Server Export:** Run `exportfs -v` on the control node to verify that `/mnt/data` is exported as an NFS share.

- **NFS Client Mount:** On each worker node, verify the NFS share is mounted using `df -h` or `mount | grep nfs`. Ensure the share is listed with the correct mount point and server information.
- **Read/Write Test:** On a worker node, navigate to the mounted NFS directory and perform the following:
  - Create a file: `touch /mnt/data/testfile`.
  - Verify the file is visible on the control node.
  - Delete the file to confirm write permissions are functional.
- **Service Verification:** On the control node, verify the `nfs-kernel-server` service is running using `systemctl status nfs-kernel-server`.
- **Log Inspection:** Check `/var/log/syslog` on the control node for any NFS-related errors or warnings.

B.3 Rancher

- **Namespace Verification:** Confirm the `cattle-system` namespace exists using `kubectl get namespaces`.
- **Cert-Manager Deployment:** Verify that Cert-Manager pods are running using `kubectl get pods -n cert-manager`.
- **Rancher Pods:** Check the status of Rancher pods using `kubectl get pods -n cattle-system`.
- **Ingress Verification:** Use `kubectl get ingress -n cattle-system` to ensure an ingress resource exists with the expected hostname.
- **Web Interface Access:** Access Rancher via the hostname in a web browser. Log in using the bootstrap password to confirm access.
- **Cluster Registration:** Register the local Kubernetes cluster in Rancher and verify that communication is established.
- **Log Inspection:** Inspect Rancher pod logs using `kubectl logs <pod-name> -n cattle-system` to ensure no errors or warnings are present.

B.4 Kube-monitoring-stack

- **Namespace Verification:** Run `kubectl get namespaces` to confirm that the `monitoring` namespace exists.
- **Pod Status:** Verify that all pods in the `monitoring` namespace are running using `kubectl get pods -n monitoring`.
- **Service Status:** Confirm that services for Prometheus, Grafana, and AlertManager are running and accessible using `kubectl get svc -n monitoring`.

- **Web Access:** Access Grafana through the defined NodePort and log in using the configured admin credentials.
- **Scrape Configuration:** Check Prometheus targets to ensure that KEPLER endpoints are being scraped correctly.
- **Data Persistence:** Restart the cluster and verify that monitoring data is retained by checking PVs and PVCs.

Bibliography

[1] Sustainable Computing IO. *KEPLER Documentation: Kubernetes-based Efficient Power Level Exporter*. Accessed: 2025-01-05. 2025. URL: `https://sustainable-computing.io`.

[2] k3s-io. *k3s-ansible: Ansible playbook for deploying K3s clusters*. `https://github.com/k3s-io/k3s-ansible`. Accessed: 2025-01-05. 2025. URL: `https://github.com/k3s-io/k3s-ansible`.

[3] Prometheus Community. *Prometheus Helm Charts*. Accessed: 2025-01-05. 2025. URL: `https://github.com/prometheus-community/helm-charts`.

[4] Sustainable Computing IO. *KEPLER Helm Chart Repository*. `https://github.com/sustainable-computing-io/kepler-helm-chart`. Accessed: 2025-01-05.