EMOI: CSCS Extensible Monitoring and Observability Infrastructure

Massimo Benini*, Jeff Hanson†, Mathilde Gianolli*, Jean-Guillaume Piccinali*,
Michele Brambilla*, Gianna Marano*, Gianni Mario Ricciardi*, Monica Frisoni *, Dino Conciatore*
* Swiss National Supercomputing Center, ETH Zurich, 6900 Lugano, Switzerland
† HPE, USA

***Abstract*—- The Swiss National Supercomputing Centre (CSCS) is enhancing its computational capabilities through the expansion of the Alps architecture, a Cray HPE EX system equipped with approximately 10000 Grace-Hopper GH200, in addition to the pre-existing 1000 nodes of diverse combination of CPUs and GPUs. This scale-up introduces significant challenges in monitoring and observability, particularly due to the increased hardware heterogeneity encompassing AMD Rome CPUs, AMD Mi250x and Mi300 GPUs, Nvidia A100 GPUs, and the Arm-based superchip GH200. Effectively managing the vast amounts of data produced in modern supercomputing, from hardware to software to facilities, poses a significant challenge. This paper introduces EMOI, a scalable architecture that simplifies observability for large clusters. EMOI offers a flexible and reliable solution, seamlessly integrating with existing HPE monitoring applications while adhering to HPE guidelines. We demonstrate EMOI's effectiveness by analyzing energy consumption across different architectures within the Alps supercomputer, providing insightful analysis.**

I. Introduction

Managing large clusters presents a significant challenge: efficiently collecting, ingesting, and analyzing vast amounts of data from diverse sources. Operators require a platform capable of handling massive data loads while remaining adaptable to various data types. This includes logs, metrics, and traces originating from a wide range of sources, encompassing everything from hardware sensors to user activity data.

This paper proposes the EMOI platform, a flexible and scalable observability and monitoring solution designed specifically for large clusters. Crafted to integrate seamlessly with the HPE-CRAY native monitoring application and adhering to HPE guidelines, EMOI offers a robust solution. While the paper details specific software components, these can be readily replaced with equivalent tools that better suit the specific needs of a different facility.

The EMOI platform addresses the challenges presented by next-generation supercomputers like ALPS, the new CSCS machine. ALPS retrieves a wide variety of data, including logs, metrics, and sensor data, for both analytics and storage. Its runtime partitioning allows for multiple virtual clusters [12] to run concurrently. EMOI tackles the complexities of collecting, segmenting, and presenting this data to different stakeholders. Flexibility, scalability based on workload, and minimal maintenance effort are core design principles for the EMOI platform. This ensures it can effectively handle the ever-growing data demands of modern supercomputing.

This work was supported by the Swiss National Supercomputing Centre (CSCS).

In the second part of this paper, II EMOI Infrastructure, we will show how can we leverage the EMOI infrastructure to analyze a critical business case - monitoring the energy consumption of our production machines. In modern supercomputers, electricity is a major operational cost. The top500.org list has been tracking power usage alongside peak performance (FLOPs/sec) since 1993 and 2013, respectively. Even though the current top systems in the Green500 list are highly efficient, there still remains a considerable power consumption.

Our case aims to understand and report energy consumption across different architectures, at both the node and cabinet level. Accurate and timely power data monitoring is essential for controlling system energy usage. Additionally, this information can raise user awareness about their own consumption, fostering a culture of energy responsibility in High-Performance Computing (HPC).

The paper is structured as follows:

- in Section II we present EMOI's architecture, focusing on its key components. We highlight how the integration of GitOps and Kubernetes enables dynamic deployment and efficient service management;
- Section III details the seamless integration of Cray System Management (CSM) and Cray System Monitoring Application (SMA) within EMOI. We discuss how the Kafka-centric approach fosters enhanced system interoperability;
- in Section IV we delve into the structure and quality of the energy datasets collected, with a specific focus on power consumption data and its relevance in today's rising energy cost environment. We share our practical experience in crafting an energy dataset that facilitates effortless correlation between telemetry and job-level data. We showcase results obtained by comparing power consumption across various CPU and GPU node architectures within the ALPS system.

In closing, this paper explores best practices for data sharing between internal and external stakeholders. This collaborative approach fosters not only improved operational efficiency but also the creation of valuable datasets for developing machine learning models dedicated to infrastructure analysis. We then

conclude by discussing the achieved results, key takeaways from our development journey, and outlining our future plans for EMOI's continued development.

II. EMOI Infrastructure

One of our main objective is to minimize the burden on human operators while ensuring the deployment, monitoring, maintenance, and upgrading of the observability infrastructure. To achieve this, we have adopted common software engineering best practices, such as DevOps and GitOps, and integrated a set of technologies together, in a cloud-like environment.

Terraform [5] is used to instantiate Kubernetes clusters, which form the core of our infrastructure. Kubernetes, the industry standard for container orchestrations, allows for a declarative description of necessary services, provides scalability and high availability, and reduces downtime of services during rolling upgrades.

Although Kubernetes is an excellent tool for orchestration, it still requires human supervision and coordination efforts when multiple operators are responsible for deploying and upgrading different services. To further automate the process, we use Argo CD [1], a declarative continuous delivery tool for Kubernetes that follows the GitOps pattern and uses Git repositories as the source of truth to define the desired application state. It also automates the deployment of the desired application, monitors running applications, and compares the current, live state against the desired target state.

The primary component of the monitoring and observability solution is Elastic Stack, a toolset that combines metrics, logs, and traces, providing unified visibility into system behaviour. Elastic Stack includes agents for data collection (Beats), transformation (Logstash), a database for storage and search (Elasticsearch), and a visualization tool (Kibana). Elastic Cloud on Kubernetes (ECK) extends the basic Kubernetes orchestration capabilities to deploy, secure, and upgrade the components of the stack natively on Kubernetes.

Apache Kafka serves as a decoupled, fault-tolerant, and asynchronous intermediary for communication between microservices, agents, and Elasticsearch. Kafka offers a buffer for log storage and data replication for fault tolerance, ensuring a smooth flow of metrics, logs, and traces for centralized analysis and system monitoring.

When working with collected data, it may be necessary to enrich it further in order to extract more information. This process may involve querying an API for additional resources or information. It is important to keep in mind the limitations of the API, as making too many requests in a short period of time can lead to slowdowns or temporary blocks from accessing the API. The use of Memcached [7], a high-performance key-value store for small chunks of arbitrary data, can help overcome this issue. By regularly querying the API, the Memcached database can be updated with up-to-date information, and Memcached can handle high-frequency queries from agents responsible for data enrichment.

The rest of this section describes in detail the components that constitute EMOI.

A. Infrastructure for massive data analytics

In order to cope with the massive amount of telemetry data to be analyzed and to ensure configuration and deployment flexibility, we leverage the hyper-converged cloud infrastructure already adopted at CSCS to fulfill several use cases and focus on the integration of Kubernetes (specifically the RKE2 distribution), ArgoCD, Rancher [10], Harvester [4], and Terraform. The integration of these technologies collectively forms the backbone of a scalable and adaptable private cloud infrastructure.

Rancher plays a crucial role in our infrastructure by serving as a central tool for managing and deploying RKE2 clusters across our environment. We use Ubuntu MAAS (Metal as a Service), to manage commodity hardware and Rancher allows for direct deployment on MAAS-managed nodes as well as specialized HPC nodes designed for high-intensity workloads. This flexibility allows for efficient resource allocation and management of diverse workloads. Harvester, is an orchestrator for Kubernetes distributions deployed on virtual clusters. Rancher integrates seamlessly with Harvester which enhances our platform's flexibility and simplifies orchestration. This integration empowers us to manage diverse workloads efficiently while ensuring optimal resource utilization.

Through the use of Terraform, an *Infrastructure as Code* (IaC) approach has been implemented by providing a declarative approach to infrastructure management. Using Terraform, we can easily create and manage new clusters, facilitating rapid deployment and scaling of our cloud infrastructure. This deep integration allows us to leverage the full capabilities of ArgoCD, Rancher, and Harvester into our infrastructure setup. By interfacing directly with Rancher and Harvester, Terraform ensures smooth coordination throughout the entire lifecycle of cluster creation, management, and deletion. The use of Infrastructure as Code principles not only improves reproducibility and consistency but also streamlines collaboration between teams accelerating the pace of the design/implementation/deployment cycle.

We adopted a *GitOps* Configuration Management approach featuring ArgoCD as a key component of our configuration and deployment strategy, automating processes and ensuring consistency across different environments. We achieve this by creating repositories on Git with cluster definitions, streamlining workflows and enabling continuous delivery. This automated approach accelerates the deployment of updates and new features, while minimising the risk of configuration drift, thereby increasing operational stability and ensuring that cluster configuration remains consistent and reproducible across the environment.

In our GitOps workflow, all configurations for each cluster and associated components are stored in a Git repository. This repository serves as a single source of truth for the desired state of the cluster. Any changes to the configurations such as updates, settings or deployment of new applications are made via Git commits.

Argo CD continuously monitors the Git repository and

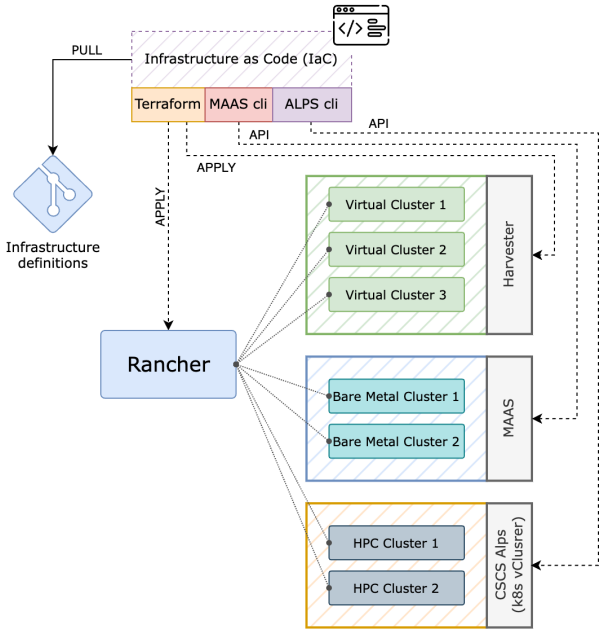


Fig. 1: Deployment Infrastructure

automatically synchronises the state of the cluster with the desired state defined in the repository. In addition, ArgoCD provides a centralised dashboard for visualising cluster health and managing deployments making it easier for operators to track changes and troubleshoot problems.

The combination of Kubernetes, ArgoCD, Rancher, Harvester, and Terraform creates a hyper-converged infrastructure that is stable, scalable, adaptable, and operationally efficient. By using these tools together we achieve several operational benefits:

- Agility: Rapid provisioning and scaling allow us to respond quickly to changing workload demands, making our operations more agile.
- Stability: Automated deployment processes and consistent configuration management reduce the risk of errors and improve operational stability.
- Efficiency: Streamlined workflows and centralized management tools optimize resource utilization and increase operational efficiency.
- Innovation: Infrastructure as Code principles and continuous delivery practices enable rapid iteration and innovation, allowing for seamless delivery of new features and updates.

This hyper-converged cloud-like infrastructure at CSCS demonstrates the synergy achieved through the integration of Kubernetes, ArgoCD, Rancher, Harvester, and Terraform.

B. Elastic Cluster on Kubernetes

Although Kubernetes offers a powerful framework for orchestrating containerized workloads, managing complex applications like Elasticsearch, Kibana, Logstash, APM Server, and Beats on Kubernetes can be difficult. Elastic Cloud on Kubernetes (ECK) simplifies the deployment and management of the Elastic Stack on Kubernetes. ECK is an official distribution of Elasticsearch, Kibana, APM Server, and Beats that runs on Kubernetes. It automates deployment, scaling, and management of the Elastic Stack allowing users to focus on their applications instead of infrastructure management. Our deployment of ECK utilises a GitOps workflow based on ArgoCD as described in the previous section. The key component of ECK is the *Elastic Operator*. A k8s Operator is a software extensions to Kubernetes that make use of custom resources to manage applications and their components. We use k8s Operators to manage our deployment as it simplifies the provisioning and management of Elasticsearch clusters, Kibana instances, and other Elastic stack components on Kubernetes. The Elastic Operator manages the lifecycle of all Elastic application instances running on Kubernetes ensuring that provisioning, scaling, and configuration management are properly addressed.

Through the Elastic Operator, we deploy our setup including the following Elastic Stack applications:

- *Elasticsearch*: a distributed search and analytics engine designed for storing and analyzing large volumes of data
- *Kibana*: a tool for visualizing and exploring data in Elasticsearch
- *Logstash*: a data processing pipeline that ingests, transforms, and sends data to Elasticsearch
- *Beats*: Lightweight agents that gather various types of operational data for shipping to Logstash or Kafka

Besides Elastic Stack components, we also deploy other applications not provided by ECK:

- *Strimzi Operator*: a tool for managing Apache Kafka clusters on Kubernetes. It simplifies the deployment and management of Kafka brokers, topics, and other related resources
- *Helm Chart for Grafana*: we use Helm, a package manager for Kubernetes, to deploy *Grafana*, a popular open source analytics and monitoring platform used to visualise data from Elasticsearch. This tool is useful to better integrate with external party data

Elastic Cloud on Kubernetes, when coupled with a GitOps workflow powered by Argo CD, offers a robust solution for deploying and managing the Elastic Stack on Kubernetes. By utilizing the native capabilities of Kubernetes and the automation provided by ECK and Argo CD, CSCS can streamline its operations processes and focus on delivering value through their applications, while ensuring reliability, scalability, and maintainability.

C. Apache Kafka on Kubernetes

Collecting metrics is an essential aspect of infrastructure observability. Traditionally, some tools poll each device for the relevant metrics on a regular interval using a GET operation, with each device replying with the metrics requested. Although effective, this method has inherent drawbacks. Firstly, only a subset of the devices can be polled at the same time. In large environments, this reduces the granularity of the data

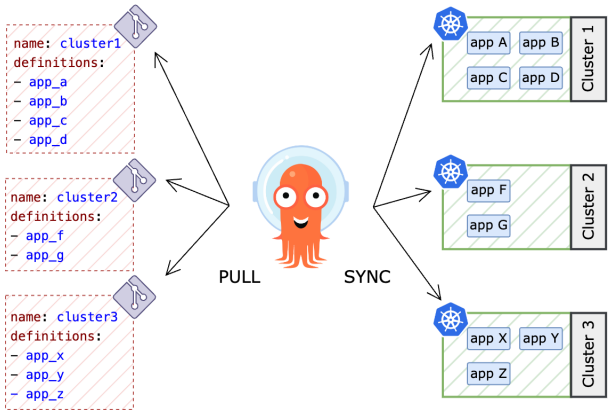


Fig. 2: GitOps Schema

since there might be large intervals between consecutive polls of a device. Secondly, to avoid overloading the network, the communication might be based on an unreliable protocol such as UDP. Finally, each poll returns all the data even when nothing has changed since the last poll.

The introduction of streaming telemetry has been a shift in the gathering paradigm and can mitigate these issues. Unlike traditional polling, streaming telemetry relies on a push model, where the entity being observed pushes the metrics to a subscriber. An effective telemetry architecture relies on a robust data streaming backbone that ensures a reliable data transport mechanism, allows data enrichment or transformation, and supports a microservice architecture.

Nowadays, the *de facto* standard for data streaming is Apache Kafka [14]. Kafka is a distributed event streaming platform extensively used to build real-time data pipelines and streaming applications. Kafka offers a distributed, highly scalable, and fault-tolerant publish-subscribe messaging system. A Kafka cluster comprises one or more servers, known as *brokers*, that run Kafka. The processes that push events into Kafka are called *producers*, while the events, which are organized in categories known as *topics*, are stored in the brokers and eventually replicated among brokers for reliability. The processes that read messages from one or more topics are called *consumers*. Topics are divided into several *partitions* that contain a subset of the records and are located on different Kafka brokers. This distributed placement of data is critical for scalability because it enables client applications to read and write the data from/to many brokers simultaneously.

Different Kubernetes operators are implementing Kafka, among which are Confluent, Bitnami, BanzaiCloud and Strimzi. Strimzi [11] was developed by Red Hat in 2017 and entered sandbox stage of the Cloud Native Computing Foundation (CNCF) in August 2019. As of February 2024, Strimzi reached the stage of a CNCF incubating project. Strimzi follows the GitOps model, where all changes to the system can be introduced via declarative Kubernetes resource YAML files. In this sense, its integration with a GitOps workflow such as the one ArgoCD provides is straightforward.

III. CSM and SMA integration with EMOI

A. CSM and SMA

HPE *Cray System Management Software* (CSM) is the foundation upon which other software product streams for the HPE Cray EX system depend. The CSM installation sets up and launches a distributed system utilizing a cluster of management nodes organized within a Kubernetes framework and Ceph for utility storage.

System services on the management nodes are provided as containerized micro-services packaged for deployment via Helm charts. Kubernetes controls the orchestration of these services, determining their placement on Kubernetes worker nodes and enabling horizontal scaling. This scaling adjusts the number of service instances based on demand fluctuations, such as during the initialization of numerous compute nodes or application nodes.

HPE *System Monitoring Application* (SMA) is a bundle of software that runs on top of CSM with the aim to provide a comprehensive set of tools enabling Cray Shasta administrators to analyze logs and metrics from multiple subsystems using both GUI and CLI tools. SMA utilizes the HPE *System Monitoring Framework* (SMF) to provide a cohesive solution to collect and present monitoring information. SMA provides multiple monitoring and analysis tools including custom graphs and tables, a CLI for non-GUI work flows, as well as notifications and alarms.

SMF boasts a tightly-integrated monitoring system that extracts detailed telemetry data from various subsystems. This includes information from the fabric, environmental conditions, network, storage, and operating systems, encompassing metrics from vmstat and iostat. The collected data is channeled into a centralized message bus, specifically Kafka, where it is not only persisted but also made accessible through the UI infrastructure.

The data integration and infrastructure layer of the SMA [4] uses a distributed streaming platform to publish and subscribe to streams of records. Kafka's architecture allows horizontal scaling, supports multiple subscribers, and effectively balances consumers during failures. Additionally, it persists messages on disk and provides multiple client-side APIs for both consumers and producers, commonly referred to as the "Kafka Bus".

In its essence, SMA is a comprehensive solution that collects and presents monitoring information and makes it available in Kafka topics. With EMOI, we select some of the Kafka topics of interest and redirect the information to our data analytics platform, minimizing the effort on the SMA side.

However, there are two challenges with the bundled Kafka server. Firstly, it does not expose any external listener, which means external applications can't connect to the Kafka cluster. Modifying the SMA configuration can become tedious work in terms of maintenance, hence we decided to use an alternative solution. Secondly, each message bundles the value for a set of different sensors in the node. While this is a convenient

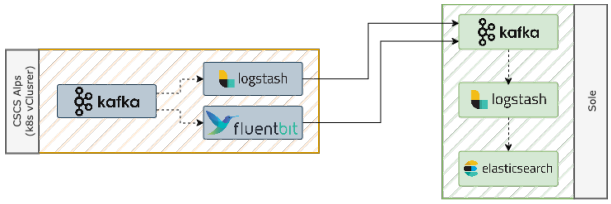


Fig. 3: Flow of data used to replicate the telemetry data of Alps

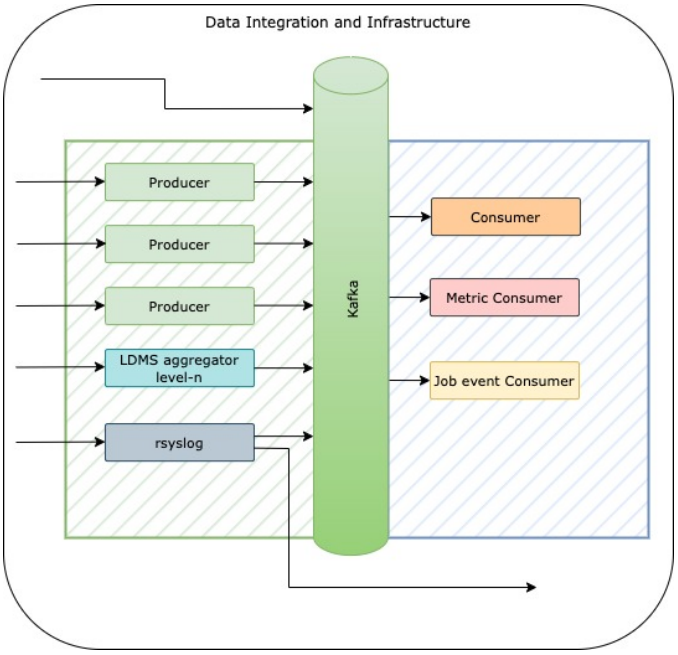


Fig. 4: Kafka Bus

strategy for reducing message size by removing redundant information, it is not suitable for ingestion into ElasticSearch where we want to be able to inspect the value of every single sensor.

To address these challenges, we developed a pipeline 3 for extracting data that consists of a combination of Logstashes [2] (alternatively Fluentbit [3]) and a KafkaStream [14]:

- a Logstash running in SMA and forwarding messages to the Kafka server in EMOI. As an alternative, we explored the usage of *Fluentbit*, a lightweight logging and metrics processor and forwarder. Unfortunately, we couldn't make it stable for large throughputs;
- a KafkaStream, part of the Kafka instance running in EMOI, that takes care of splitting the message bundle in a message per sensor;
- a Logstash running in EMOI which further manipulates and enriches the message. Finally, it ingests the message into ElasticSearch. It is worth noting that in principle this component could be replaced with a KafkaConnector.

B. CSM API and Memcached

As streams of telemetry data flow into Elasticsearch, there arises the need to enrich the collected metrics with additional information. Given the high frequency of telemetry data arriving, enrichment needs to occur rapidly. Memcached [13], available as a Logstash plugin has been identified as a suitable tool for storing additional information due to its fast access capabilities. Memcached is a powerful distributed memory caching system initially designed to accelerate dynamic web applications by alleviating database load. It stores data in key-value pairs, offering flexibility for various applications.

A specific use case for Memcached emerges when we seek to assess the energy consumption of individual jobs and nodes. To accomplish this, we need to know which job is running on which node, at which moment in time, and for how long. This information about the job must then be correlated with the energy consumption data gathered from each node. Since nodes are identified by their **nid** in the former case and by their **xname** in the latter, establishing a mapping between **nid** and **xname** is needed to merge these data sources. Furthermore, knowledge of a node's vCluster affiliation is required for calculating the energy consumption of an entire vCluster. Mappings between **xname**, **nid** and **vCluster** can be stored in Memcached and can be used to enrich telemetry data flowing through Logstash. Consequently, each metric received from Alps and processed through Logstash will contain this supplementary information upon arrival in Elasticsearch.

Information regarding a node's **xname** and **nid**, as well as its vCluster affiliation, is obtainable through the HPE *CSM RESTful API*. Serving as an alternative to directly running commands from the management plane, the API offers flexible access to the status and configuration of running components. Details about component inventory, status, and hardware specifications can be accessed through a series of exposed endpoints.

We developed a Python application, Kafka2Memcached App [6] with the aim to gather information about the relationship between **xname**, **nid**, and **vCluster** for each node in Alps through the exposed API endpoints and transmitting this data to a dedicated Kafka topic. As the affiliation of a node in a vCluster can change dynamically, this process is triggered periodically. Messages from the Kafka topic are continuously consumed by a process spawned within Kafka2Memcached, which then writes them to Memcached. Subsequently, Logstash, can access this cached information to enrich consumed telemetry data.

IV. Energy Dataset

CSCS is expanding its computational capabilities by upgrading the Alps architecture with a Cray HPE EX system equipped with approximately 10,000 Grace-Hopper GH200s, in addition to the pre-existing 1,000 nodes with a diverse combination of CPUs and GPUs.

Alps system infrastructure will replace CSCS's existing Piz Daint supercomputer and serve as a general-purpose system open to the broad community of researchers in Switzerland and the rest of the world. It will enable breakthrough research on a wide range of fields, including climate and weather,

materials sciences, astrophysics, computational fluid dynamics, life sciences, molecular dynamics, quantum chemistry and particle physics, as well as domains like economics and social sciences.

The expansion presents significant challenges in monitoring and observability, notably attributable to the augmented hardware diversity, including AMD Rome CPUs, AMD Mi250x and Mi300 GPUs, Nvidia A100 GPUs, and the Arm-based superchip GH200.

A. Introduction and Motivation

With the goal to raise awareness among users and management about the job's energy consumption, we began to construct an energy dataset, where we will merge the informations about the SLURM jobs and the telemetry data of the node at the time of the jobs. To advise users is very important to better grasp the energy impact linked to computational operations, to optimize the energy-to-solution ratio. The strategic planning and sustainable development of computational resources is on the other hand equally important: The insights gained from user awareness regarding the energy impact of their computational tasks is needed as input for the management at CSCS (Swiss National Supercomputing Centre) to proactively plan and design future HPC systems. With this aim in the background, we did in this section a node-level correlation analysis to compare SLURM energy data with pm-counters-file energy data and telemetry energy data, where we apply our EMOI Infrastructure to collect telemetry data and verify if these can be considered a source of truth.

B. Cray PM Energy data

The Cray EX platform incorporates Out-Of-Band (OOB) data collection capabilities for monitoring power and energy at the node level, as well as at system level. Collected node level power and energy data are published via special */sys/cray/pm_counters/* sysfs files, as described by Martin in [15] and [16]. The pm_counters files are read-only and available to (unprivileged) users for monitoring purposes and to report energy usage. The energy usage of node, CPU and memory of a compute node can be read from the **energy**, **cpu_energy** and **memory_energy** pm files. The energy usage of accelerators can be read from the **accel[0,1,2,3]_energy** pm files. On the GH200 nodes, the cpu_energy pm file is split into four **cpu[0,1,2,3]_energy** pm files, one for each Grace CPU. Other useful information including power cap settings are not being discussed here. The **raw_scan_hz** counter informs the user of the rate at which all counters should be updating. Collection of aggregate power and energy telemetry data is enabled by default with a default collection rate of 10 Hz.

1) Slurm Energy data: Slurm[17] is an open-source resource manager designed to schedule user jobs, monitor system and job status and launch user applications. Slurm can be configured to track energy used by jobs and tasks by adding the *energy* label to the **AccountingStorageTRES** list of Trackable RESources (TRES). Additional *AccountingStorage* parameters allow to store the data in an accounting database. Energy accounting data is included in accounting records, and can be accessed through the *sacct* and *sreport* commands.

Slurm can also store energy data using the job completion plugin. A major advantage of the plugin is that it integrates natively with our ElasticSearch and Kafka interfaces. The *Account Gather* plugin can be set to collect per job energy usage data by adding the *acct_gather_energy/pm_counters* label to the **AcctGatherEnergyType** parameter. Prior to SLURM versions 23.11.1, the job completion plugin lacked the ability to correctly report energy data. We proposed and successfully merged a patch to rectify this issue.

C. Cray Telemetry Data

TABLE I: Kafka domains and topics

Domains	**Topics**
System nodes	cray-node
Fabric Telemetry	cray-fabric-telemetry
Power, Energy and Voltage	cray-telemetry-energy cray-telemetry-voltage cray-telemetry-power
Environmental Telemetry	cray-telemetry-temperature cray-telemetry-fan cray-telemetry-pressure
System Hardware	cray-dmtf-resource-event cray-hsmstatechange-notifications
Kubernetes	cray-logs-containers

SMA offers access to a wide range of telemetry metrics encompassing power, energy, voltage, system environment, and more. Table I lists the available Kafka topics and their domains.

For energy consumption analysis, we can gather metrics from the *cray-telemetry-energy*, *cray-telemetry-voltage*, and *cray-telemetry-power* Kafka topics, and store them in our EMOI infrastructure. Each stored message furnishes a comprehensive description of the sensor-measured value corresponding to a specific component.

Appendix tables VII and VIII list the sensors available for the topic message ids Power and Energy for the type of nodes shown in table III.

D. Validation of Cray PM and Cray Telemetry Energy Data

One of our goals is to validate the accuracy of telemetry-based energy measurements. To achieve this, we compare the energy usage reported via Slurm data with the energy usage reported via Telemetry data for several jobs executed over multiple days, considering the four type of nodes listed in Table III. While we extract the Cray PM energy data with the Slurm *sacct* command, using the fields shown in Table II, we extract the Cray Telemetry Energy data by querying our Elasticsearch database. Considering the streaming nature of the telemetry data, processing is necessary. For each request, we get the energy in Joules at the beginning and end time of the jobs, by adjusting the time range of the request to align with the duration of the jobs. As the default timezone in our Elasticsearch is set to UTC, we adjust our queries to ensure synchronization with the timezone used in Slurm. As

TABLE II: Slurm sacct formatting fields

Field name	Description
Cluster	Cluster name
JobID	Identification number of the Job/step
Start	Initiation time of the Job
End	Termination time of the Job
Elapsed	Job(s) Elapsed time
NodeList	List of nodes in Job/step
NNodes	Number of nodes in a Job or step
ConsumedEnergy	Total Energy of the Job (in Joules)

TABLE III: Alps Compute Node Specifications

Blade architecture	CPU(s) per node	GPU(s) per node
EX-425 Windom (MC)	2 AMD 64-core 7742	0
EX-325A Bard Peak (AG)	1 AMD 64-core 7A53	4 AMD MI200
EX-325N Grizzly Peak (NG)	1 AMD 64-core 7713	4 NVIDIA A100
EX Blanka Peak (GH)	1 ARM 288-core Grace	4 NVIDIA GH200

TABLE IV: Job 2665753

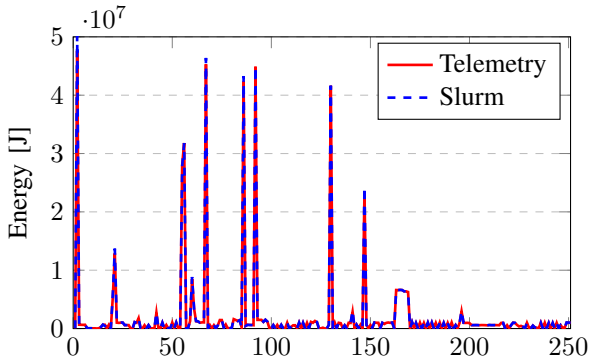
node id	Telemetry [J]	Slurm [J]
nid001001	1'281'466	-
nid001182	1'471'533	-
nid001183	1'467'066	-
nid001184	1'443'091	-
sum	5'663'156	5'662'307
average	1'415'789	1'415'576.75

TABLE V: node statistics

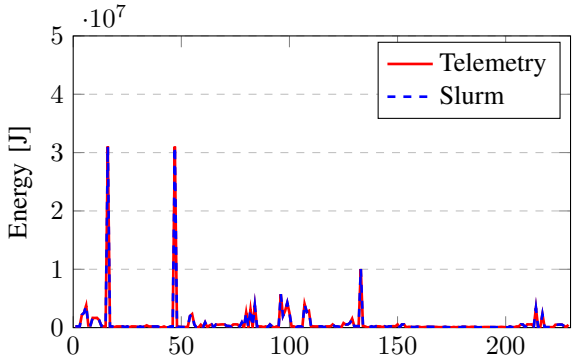
Statistics	job count	correlation
MC node	251	0.9997357
AG node	229	0.9999999
NG node	305	0.9999997
GH node	199	0.9999837

Elasticsearch reduces memory usage by restricting the default maximum number of searches per query, the Python script [9] is retrieving data in batches of 10^4 hits. We consider using the *index.max_result_window* parameter to raise that value. The energy of each job can then be calculated by getting the difference of the energy at the end and start times of the job: $E_{end} - E_{start}$.

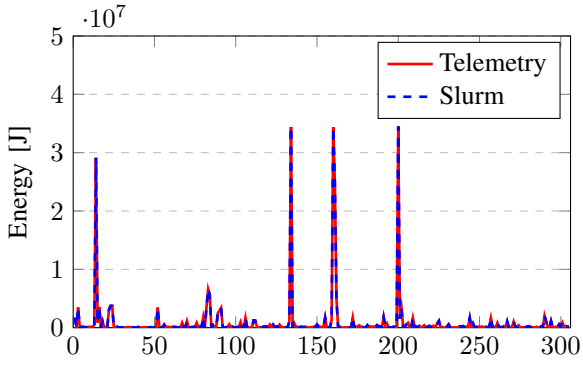
1) Energy data analysis: to find out whether Slurm and Telemetry energy data exhibit congruence, we report in Fig. 5a-d the measured energy of a number of jobs ran within a specified time frame, excluding the jobs with a runtime lower than 30 seconds. We occasionally encountered outliers in the Slurm energy data, with unrealistic high values probably due to buffer overflow or race condition, which were consequently excluded from the overall analysis. Further investigations on it will be done. In both energy datasets and across all (MC, AG, NG and GH) types of node, we observe a strong correlation between telemetry and Slurm energy values. Additionally, we correctly see peaks corresponding to energy-intensive jobs. Fig. 6 shows the difference between Slurm and Telemetry data, defined as the energy ratio $\frac{(E_{telemetry}-E_{Slurm})*100}{E_{telemetry}}$. Most of the delta values fluctuate around zero, confirming a minimal difference between the measured Slurm and Telemetry energy data. Occasionally, the energy values differ significantly. For instance, we observe jobs with a -10% trough and a 30% peak on the multicore (MC) node, a job with a -15% trough on



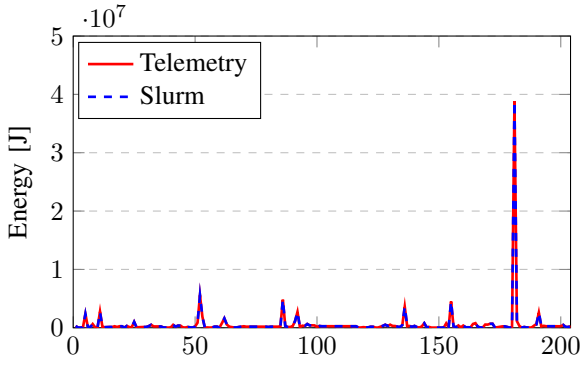
(a) nid001001: MC node (18-31/12/2023)



(b) nid002556: AG node (08-21/02/2024)



(c) nid002792: NG node (08-21/02/2024)



(d) nid002984: GH node (14/02/2024-12/03/2024)

Fig. 5: Energy Usage (in Joules) reported by Slurm and Telemetry Measurements (984 Jobs, 4 types of compute nodes)

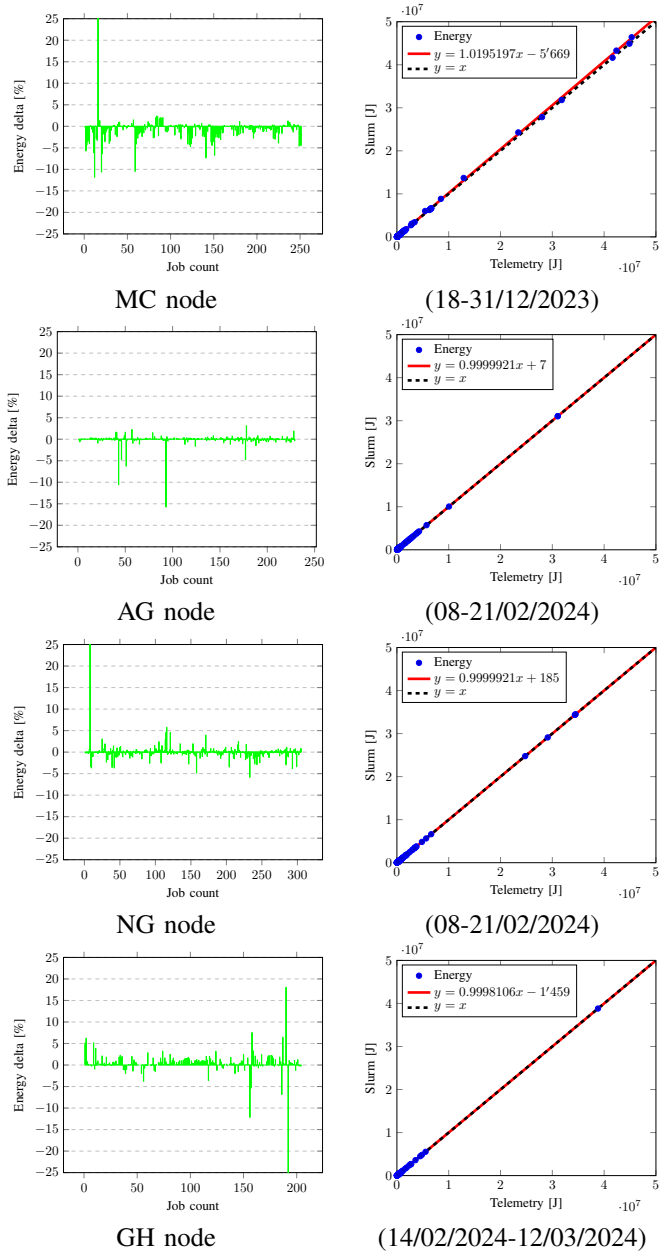


Fig. 6: Energy usage $\delta_{\%}$ on the left and correlation plots on the right of Telemetry Measurements relative to Slurm Measurements (984 Jobs) of 4 types of compute nodes: MC (nid001001), AG (nid002556), NG (nid002792), GH (nid002984)

the AMD MI200 (AG) node, a job with a 27% peak on the NVIDIA A100 (NG) node and a job with a -70% trough on the Grace-Hopper (GH) node.

To illustrate these deviations, we can look at job with jobid *2665753* of Table IV that ran on 4 nodes for 106 minutes: using the Telemetry value of nid001001 (*1,281,466 J*) and the average energy per node of the Slurm value (*5,662,307/4*), we compute a $\delta_{\%}$ of *-10.465*. However, relying only a single telemetry value overlooks the variance in energy consumption across nodes. By incorporating telemetry data from all nodes used by the job, as shown in Table IV, we find a more acceptable $\delta_{\%}$ of *0.015*.

2) Energy data statistics: Finally, we first checked by the means of Pearson correlation to see if there is a linear mathematical relationship between this two different data sources and next we tried to retrieve this function and represent it graphically.

In table V we observed a correlation value very close to 1 between telemetry and Slurm energy data: this means that the function between Telemetry energy data and Slurm energy data is linear of type:

$$E_{SLURM} = E_{Telemetry} * a + b$$

In the following part we will dig a little deeper into telemetry and Slurm relationship by the means of a linear regression. If the a-coefficient is very close to 1 for all four nodes and b is negligible, then our telemetry energy data are a good representative for energy consumed at node-level.

With this goal in mind, we applied linear regression to better define the relationship between telemetry energy data with Slurm energy data for our four different kind of nodes.

For the multicore node we got the function:

$$E_{SLURM} = E_{Telemetry} * 1.0195197 - 5'669.$$

We also noticed a higher fluctuation of the points around the regression line. This could be explained with the rough energy estimation done for multi-node jobs. This kind of node is also mainly used together with other nodes to perform tasks at the same time. The offset parameter is around 6kJ, but since it is mainly due to our rough average and the order of the number of energy data that we get is generally much higher, the function between telemetry and Slurm is still comparable to the identity function.

For the AMD GPU node we got:

$$E_{SLURM} = E_{Telemetry} * 0.999991 + 7.$$

This function is comparable to the identity function. b0-coefficient can be considered to be 0 in this context, because we are working with energy data with numbers of a much higher order 10^7. The data are well aligned because there is much less variance in it than for the multicore node.

For the NVIDIA GPU node we got:

$$E_{SLURM} = E_{Telemetry} * 0.9999921 + 185.$$

This is also comparable to the identity function for the same reason cited for AMD GPU node. This node behaves similarly like the amd GPU node.

For the NVIDIA GH200 node we got:

$$E_{SLURM} = E_{Telemetry} * 0.9998106 - 1'459.$$

This kind of node is also mainly used together with other nodes to perform tasks at the same time because it is still in testing phase. The offset parameter b is around -1.5 kJ but since it is mainly due to our rough average and the order of the number of energy data that we get is generally much higher, the relationship between telemetry and Slurm is still comparable to the identity function.

We can conclude that our telemetry seems to be reliable in matter of energy data at node level. We can see a strong similarity between Slurm and telemetry energy data in the correlation plots of Fig. 6. Thus, the energy differences that we sometimes observed for some jobs could be omitted and our telemetry energy data could be a valid replacement for Slurm energy data with a correlation value close to 1. We can then work with our telemetry data to better grasp the energy workload of the supercomputer, with the future goal to increase the staff-and user-awareness in terms of energetic footprint of the jobs and to see how and where we can spare energy. Slurm on the other hand, shows sometimes a weird behaviour and cannot therefore always be trusted. For this reason, we will dedicate the following section to read directly the pm files of the nodes and compare them with our telemetry components. The following experiment was also significant to understand which telemetry component represents the total energy of the node.

E. Classify Telemetry Data

In this section we compared the energy obtained running a test-job for the different node-components described by the pm files of the node with the energy measured by the telemetry. The test-job was node-burn [8], duration:1 second. When we refer to the energy of the node, we mean the energy measured by the pm counters of the node and saved in the pm file of the node. The various components in the node-telemetry energy are listed in the appendix of the energy table. For every node-architecture we ran the test-job first only on the CPU, distinguishing every run by a different number of CPU-cores charged with the job node-burn, and then on the GPU-package, running each time on a different GPU. Our first goal is to see how the energy of a node is divided between its component on the pm file and if there is an equivalence with some components of the telemetry data, to verify which component of the telemetry corresponds to the total energy of the node. Our second goal is to see if the source of Slurm, which is the pm file on the node, is trustworthy.

In all resulting comparison tables a high correlation between the pm file and the telemetry upon some small discrepancies has been highlighted. The small differences can be explained both with the different frequency of measurement used for the energy data in the pm counter file, which would be 10 Hz ($\equiv$ 10 measurements per second), and the one used for the telemetry data, which is around 1Hz ($\equiv$ 1 measurement per second), and the fact that pm file of the node is not scaled with the node-Temperature and Telemetry-data are.
The energy data of the node for the CPU, respectively the memory, correspond to the telemetry energy data given by "Sensor.ParentalContext:CPU" and Sensor.Index:0, and respectively "Sensor.ParentalContext:CPU" and Sensor.Index:1. In case of two CPUs per node, like on multicore processor, we get in the energy telemetry data two times the Sensor.Index 0 for the two different sockets, the CPU energy of the node would be given by their sum and analogously, the two values for Sensor.Index:1 for the two different sockets, sums up to give the energy used by the Memory of the node.
In addition, for the AMD GPU case, the energy measured for the four GPUs ($\equiv$ $accelerator$) in the node matches one-to-one the energy given by the four different GPUs of the node telemetry.

In conclusion, we demonstrated that the pm file is reliable and that the energy of the telemetry data with attributes:

$$\begin{aligned}\mathrm{Energy}_{\mathrm{node}} = \ & \textbf{Sensor.Location:}\mathrm{xname}_{\mathrm{node}} \\ & \& \ \textbf{MessageId:}\mathrm{CrayTelemetry.Energy} \\ & \& \ \textbf{Sensor.ParentalContext:}\mathrm{Chassis} \\ & \& \ \textbf{Sensor.PhysicalContext:}\mathrm{VoltageRegulator} \\ & \& \ \textbf{Sensor.PhysicalSubContext:}\mathrm{Input} \\ & \& \ \textbf{Sensor.Index:}0\end{aligned}$$

corresponds to the total energy of the node that we read in the pm file and which is then reported by Slurm.

V. Conclusion

In this paper, we have introduced the Extensible Monitoring and Observability Infrastructure (EMOI) used at CSCS.

We started with a general description of the infrastructure where we highlighted the advantages of adopting a GitOps approach. By leveraging GitOps principles, organizations can streamline the creation of versatile environments and facilitate the seamless sharing of data among internal and external stakeholders. This not only enhances collaboration and knowledge sharing but also fosters agility and adaptability in responding to evolving data requirements and analytical needs.

Then, we took the example of defining an energy dataset for the new HPE Cray EX to address all the different aspect of exploiting the EMOI infrastructure with such complex use case. We have elucidated the disparities in how different architectural handle exposure of hardware data. These discrepancies underscore the need for adaptable solutions that

can accommodate varying infrastructural nuances to ensure seamless integration and data accessibility across platforms. One prevailing theme throughout our exploration is the inherent difficulty in acquiring relevant data. Whether due to the dispersed nature of the information or the differences in how various architectures expose hardware data. HPE is working in this direction, and the next versions of SMA demonstrate a promising trajectory towards overcoming these challenges.

While not extensively covered in this paper, the critical aspect of performance tuning various components along data pipelines remains of greatest importance, particularly when dealing with massive datasets. Parameters such as *max_poll_records*, *fetch_min_bytes*, *fetch_max_wait_ms* in Kafka, and *pipeline.workers*, *pipeline.batch.size*, and *pipeline.batch.delay* in Logstash necessitate meticulous adjustment. Custom dashboards have been developed to aid in tuning and monitoring data pipeline ingestion. We also built custom dashboards to help us tune and monitor our data pipelines ingestion.

Now that we have a robust and flexible infrastructure in place, we can explore further avenues for research. One possible avenue for exploitation of these data sets is to optimise the code in accordance with energy usage and to model a more efficient power supply.

In conclusion, we advocate for the adoption of the EMOI's approach within the community to integrate observability and monitoring platforms effectively. The advantages of an easily extensible platform are significant, particularly in scenarios where observed systems can be partitioned or clustered, and data are of interest to diverse actors and stakeholders.

VI. Acknowledgment

The authors would like to thank HPE/Cray, and in particular Fabio Verzelloni and James Brunson, for their assistance.

References

[1] Argo cd - declarative gitops cd for kubernetes. https://argo-cd.readthedocs.io/en/stable/. Accessed: (2024-04-30).

[2] Centralize, transform & stash your data. https://www.elastic.co/logstash. Accessed: (2024-04-10).

[3] Fluentbit, high performance. https://fluentbit.io. Accessed: (2024-04-10).

[4] Harvester - the open-source hyperconverged infrastructure. https://harvesterhci.io. Accessed: (2024-04-30).

[5] Hashicorp terraform. https://www.terraform.io. Accessed: (2024-04-30).

[6] Kafka 2 memcached app. https://github.com/eth-cscs/dwdi/tree/main/emoi/kafka_memcached. Accessed: (2024-04-30).

[7] Memcached. https://memcached.org. Accessed: (2024-04-30).

[8] Node-burn test job developed by cscs. https://github.com/eth-cscs/node-burn. Accessed: (2024-04-30).

[9] Pyhon query on elasticsearch. https://github.com/eth-cscs/dwdi/tree/main/emoi/query_ES. Accessed: (2024-04-30).

[10] Rancher - enterprise kubernetes management. https://www.rancher.com. Accessed: (2024-04-30).

[11] Strimzi, kafka on kubernetes. https://strimzi.io. Accessed: (2024-04-10).

[12] S. R. Alam, M. Gila, M. Klein, M. Martinasso, and T. C. Schulthess. Versatile software-defined hpc and cloud clusters on alps supercomputer for diverse workflows. *The International Journal of High Performance Computing Applications*, 37(3-4):288–305, 2023.

[13] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004:5, 2004.

[14] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[15] S. Martin. Cray xc30 power monitoring and management. 2014.

[16] S. Martin and G. J. Koprowski. Cray ® xc tm advanced power management updates. 2018.

[17] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, 2003.

Appendix

TABLE VI: Sensors depth and name

Sensor Name	Abbreviated Name
Sensor.ParentalContext	S.ParCtx
Sensor.PhysicalContext	S.PhyCtx
Sensor.PhysicalSubContext	S.PhySCtx
Sensor.ParentalIndex	S.ParIdx
Sensor.Index	S.Idx
Voltage Regulator	VR
CPU Subsystem	CPU_S
GPU Subsystem	GPU_S
Memory Subsystem	MEM_S
GPU Module	GPU_M
System Board	Board_S

TABLE VII: Telemetry data structure (MessageId:**Energy**)

EX-425 Windom (AMD 7742 cpus)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	INVALID	-	0	0,1
			1	0,1
Chassis	CPU_S	Output	-	0
	Memory_S	Output	-	0
	VR	Input	-	0
EX-325A Bard Peak (AMD CPU+AMD MI200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	INVALID	-	0	0,1
			1	0,1
Chassis	CPU_S	Output	-	0
	Memory_S	Output	-	0
	VR	Input	-	0
Accelerator	VR	-	0-3	0,0,0,0
EX-325N Grizzly Peak (AMD CPU+NVIDIA A100 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	INVALID	-	0	0,1
			1	0,1
Chassis	CPU_S	Output	-	0
	Memory_S	Output	-	0
	VR	Input	-	0
EX Blanka Peak (ARM Grace+NVIDIA H200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
Chassis	VR	Input	-	0
	CPUSubsystem	Output	-	0

TABLE VIII: Telemetry data structure (MessageId:**Power**)

EX-425 Windom (AMD 7742 cpus)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	INVALID	-	0	0,1,2,3,4
			1	0,1,2,3,4
	VR	Input	0	4
			1	4
		Output	0	3,4
			1	3,4
Chassis	CPU_S	Output	-	0
	MEM_S	Output	-	0
	VR	Input	-	0
MEM_S	VR	Input	1	0
				1
				5
				6
CPU_S	VR	Input	-	3
				4
EX-325A Bard Peak (AMD CPU+AMD MI200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	INVALID	-	0	0,1,2,3,4
	VR	Input	0	6
		Output	0	0,1,5,6
Chassis	CPU_S	Output	-	0
	MEM_S	Output	-	0
	VR	Input	-	0,2
		Output	-	0
MEM_S	VR	Input	0	7,8
		Output	0	2,3,7,8
Accelerator	VR	Input	0,1,2,3	0,0,0,0
CPU_S	Accelerator	-	-	0,1,2,3
EX-325N Grizzly Peak (AMD CPU+NVIDIA A100 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	INVALID	-	0	0,1,2,3,4
	VR	Input	0	6
		Output	0	0,1,5,6
Chassis	CPU_S	Output	-	0
	MEM_S	Output	-	0
	VR	Input	-	0
		Output	-	0
MEM_S	VR	Input	0	7,8
		Output	0	2,3,7,8
GPU_S	Accelerator	-	-	0,1,2,3
EX Blanka Peak (ARM Grace+NVIDIA H200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
-	CPU	-	-	0,1,2,3
Chassis	CPU_S	Output	-	0
	VR	Input	-	0,1,2,3,4,5
		Output	-	0
-	-	-	-	-
-	GPU	-	-	0,1,2,3
-	GPU_M	-	-	0,1,2,3

TABLE IX: Telemetry data structure (MessageId:**Voltage**)

EX-425 Windom (AMD 7742 cpus)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	Input	0	2,3,4
			1	2,3,4
		Output	0	2,3,4
			1	2,3,4
Chassis	VR	Input	-	0
		Output	-	0
$Memory_S$	VR	Input	0	0,1,5,6
			1	0,1,5,6
		Output	0	0,1,5,6
			1	0,1,5,6
CPU_S	VR	Input	-	3,4
		Output	-	3,4
EX-325A Bard Peak (AMD CPU+AMD MI200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	Input	0	0,1,4,5,6
		Output	0	0,1,4,5,6
Chassis	VR	Input	-	0,2
		Output	-	0,2
$Memory_S$	VR	Input	0	2,3,7,8
		Output	0	2,3,7,8
CPU_S	VR	Input	0,1,2,3	0
		Output	0,1,2,3	0
EX-325N Grizzly Peak (AMD CPU+NVIDIA A100 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	Input	0	0,1,4,5,6
		Output	0	0,1,4,5,6
Chassis	VR	Input	-	0
		Output	-	0
$Memory_S$	VR	Input	0	2,3,7,8
		Output	0	2,3,7,8
EX Blanka Peak (ARM Grace+NVIDIA H200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
Chassis	VR	Input	-	0,1,2,3,4,5,6
		Output	-	0

TABLE XII: Telemetry data structure (MessageId:**Power**)

ALPS - CHASSIS				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
-	Rectifier	Input	-	0
				1
				2
		Output	-	0
				1
				2
PREALPS - CHASSIS				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
-	Rectifier	Input	-	0
				1
				2
				3
		Output	-	0
				1
				2
				3

TABLE X: Telemetry data structure (MessageId:**Current**)

EX-425 Windom (AMD 7742 cpus)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	Input	0	2,4
			1	2,4
		Output	0	2,3,4
			1	2,3,4
Chassis	VR	Input	-	1
		Output	-	0
Memory$_S$	VR	Input	0	5,6
			1	5,6
		Output	0	0,1,5,6
			1	0,1,5,6
CPU$_S$	VR	Output	-	3,4
EX-325A Bard Peak (AMD CPU+AMD MI200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	Input	0	4,6
		Output	0	0,1,4,5,6
Chassis	VR	Input	-	2
		Output	-	0
Memory$_S$	VR	Input	0	7,8
		Output	0	2,3,7,8
Accelerator	VR	Input	0	0
			1	0
			2	0
			3	0
EX-325N Grizzly Peak (AMD CPU+NVIDIA A100 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	Input	0	4,6
		Output	0	0,1,4,5,6
Chassis	VR	Output	-	0
Memory$_S$	VR	Input	0	7,8
		Output	0	2,3,7,8
EX Blanka Peak (ARM Grace+NVIDIA H200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
Chassis	VR	Output	-	0,1,2,3,4,5,6

TABLE XI: Telemetry data structure (MessageId:**Temperature**)

EX-425 Windom (AMD 7742 cpus)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	-	0	2,3,4
			1	2,3,4
Chassis	VR	-	-	0
				2
Memory$_S$	VR	-	0	0
				1
				5
				6
			1	0
				1
				5
				6
CPU$_S$	VR	-	-	3
				4
EX-325A Bard Peak (AMD CPU+AMD MI200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	-	0	0,1,4,5,6
Chassis	VR	-	-	0,1,2
	GPU$_S$	-	-	0-7
Accelerator	VR	-	0	0
			1	0
			2	0
			3	0
GPU$_S$	GPU	-	-	0-7
EX-325N Grizzly Peak (AMD CPU+NVIDIA A100 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
CPU	VR	-	0	0,1,4,5,6
Chassis	VR	-	-	0
				1
	GPU$_S$	-	-	0,1,2,3
GPU$_S$	GPU	-	-	0,1,2,3
	Board$_S$	-	-	0
Memory$_S$	VR	-	0	2,3,7,8
EX Blanka Peak (ARM Grace+NVIDIA H200 GPU)				
S.ParCtx	S.PhyCtx	S.PhySCtx	S.ParIdx	S.Idx
-	CPU	-	-	0,1,2,3
Chassis	VR	-	-	0,2,3,4,5,6
-	GPU	-	-	0,1,2,3
GPU	Memory	-	-	0,1,2,3