Comparing Fault-tolerance in Kubernetes and Slurm in HPC Infrastructure

Mirac Aydin, Michael Bidollahkhani, Julian M. Kunkel
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG)
Universität Göttingen
Göttingen, Germany
e-mail: {mirac.aydin | julian.kunkel}@gwdg.de , michael.bkhani@uni-goettingen.de

***Abstract*—In this paper, we explore the fault-tolerance mechanisms in Kubernetes and Slurm within High-Performance Computing (HPC) infrastructures. As computational workloads and data requirements continue to expand, ensuring reliable and resilient HPC systems becomes increasingly critical. Our study examines the strategies employed by Kubernetes and Slurm to handle failures, maintain system stability, and provide continuous service. We present a comparative analysis, highlighting the strengths and limitations of each system in various failure scenarios. We review and synthesize findings from existing literature and case studies to infer the effectiveness of these fault-tolerance mechanisms. Through this comprehensive review, we provide insights into the current state of fault-tolerance in Kubernetes and Slurm and propose recommendations for enhancing resilience in HPC environments.**

Keywords-fault-tolerance; Kubernetes; Slurm; High-Performance Computing; HPC; resilience; comparative analysis; literature review; case studies.

I. Introduction

HPC systems are integral to a wide range of scientific and industrial applications, driving advancements in fields, such as climate modeling, bioinformatics, and complex simulations. As the demand for computational power and data processing continues to grow, maintaining the reliability and resilience of HPC infrastructures becomes increasingly crucial. Given that a single protocol failure can potentially disrupt the entire HPC system, fault-tolerance, which is the capacity of a system to maintain operational functionality in the presence of failures, is a critical factor in ensuring the robustness and reliability of such systems [1].

Kubernetes and Slurm are two prominent orchestration and workload management systems used in HPC environments. Kubernetes, originally designed for managing containerized applications [2], has gained traction for its flexibility and scalability [3] [4] [5]. Slurm, on the other hand, is a traditional HPC workload manager known for its efficiency in scheduling and resource management [6] [7]. Both systems offer mechanisms to handle failures, but their approaches and effectiveness can vary significantly.

This paper aims to explore and compare the fault-tolerance mechanisms of Kubernetes and Slurm in the context of the EU DECICE Project [8]. The DECICE Project, funded by the Horizon Europe program, aims to develop an AI-based, open, and portable cloud management framework for the automatic and adaptive optimization and deployment of applications across a federated infrastructure, encompassing HPC systems, cloud, edge, and IoT devices. A key objective of DECICE is to enhance the resilience and efficiency of this compute continuum through dynamic scheduling and fault-tolerance mechanisms.

By reviewing and synthesizing findings from existing literature and case studies, we seek to understand how Kubernetes and Slurm manage common failure conditions, such as node crashes, network partitions, and resource exhaustion, within the framework envisioned by DECICE. Our goal is to provide a comprehensive analysis that highlights the strengths and limitations of each system's fault-tolerance capabilities, contributing to the project's objectives of developing an intelligent management plane and achieving high levels of performance and energy efficiency.

The remainder of this paper is structured as follows: Section II provides a background on fault-tolerance in HPC systems and an overview of Kubernetes and Slurm. Section III discusses the methodology used for our literature review and comparative analysis. Section IV presents the results of our review, detailing the fault-tolerance mechanisms and their effectiveness. Finally, Section V offers conclusions and recommendations for enhancing fault-tolerance in HPC environments using Kubernetes and Slurm, aligned with the goals of the DECICE Project.

By understanding the current state of fault-tolerance in these systems, we aim to contribute valuable insights that can inform future developments and best practices in HPC infrastructure management, ultimately supporting the broader aims of the DECICE Project.

II. Background

In this section, we provide an overview of the key components and mechanisms relevant to fault-tolerance in HPC systems. We begin by outlining the common faults encountered in HPC environments, discussing their types and origins to set the context for the fault-tolerance strategies required. Following this, we explore the architectures and fault-tolerance mechanisms of Kubernetes and Slurm, two prominent orchestration and workload management systems used in HPC. Understanding these foundations is crucial, as Kubernetes and Slurm represent different approaches to managing workloads and ensuring system resilience. This background sets the stage for a detailed comparison of their fault-tolerance capabilities in subsequent sections. The historical evolution and importance of robust fault-tolerance mechanisms are underscored by the increasing demand for reliable and efficient HPC systems, which are critical for advancing scientific research and industrial applications. HPC systems, while immensely powerful, are not immune to faults and failures. Knowing the common types of faults in HPC environments and their origins is crucial for developing effective fault-tolerance mechanisms.

A. Types of Failure in HPC

1) Hardware Failures: Hardware failures are one of the most common and impactful faults in HPC systems. In the paper [9], the authors found out an increase in failure probability as high as 170X for environmental failures and nearly 10X for software failures. These can include:

- **Node Failures**: These occur when individual compute nodes malfunction due to issues, such as overheating, power supply problems, or component wear and tear.
- **Network Failures**: HPC systems rely on complex network infrastructures for inter-node communication. Failures in network hardware, such as switches or routers, can disrupt these communications.
- **Storage Failures**: HPC applications often require high-throughput and low-latency access to large datasets. Failures in storage devices or file systems can lead to significant performance degradation or data loss.

Figure 1 provides a histogram of the failure frequency across different components in an HPC system. The x-axis lists various components prone to failures, including applications (APPL), CPU cores (CORE), controllers (CTRL), disk storage (DISK), cooling fans (FAN), hypervisors (HSV), operating systems (OS), power supplies (PS), and scientific backplanes (SCI_BP). The y-axis represents the frequency of failures observed in each component. Each bar in the histogram corresponds to the failure frequency of a specific component, with the height of the bar indicating how often failures occurred for that component. The histogram shows that disk storage (DISK) has the highest frequency of failures among all components, followed by controllers (CTRL) and power supplies (PS). Other components, such as applications (APPL), CPU cores (CORE), and hypervisors (HSV) have relatively lower failure frequencies. This figure highlights the critical areas within an HPC system that are more prone to failures, emphasizing the need for robust fault-tolerance mechanisms, especially for components with high failure frequencies. Understanding these failure patterns can help in designing more resilient HPC systems and implementing effective predictive maintenance strategies.

2) Software Failures: Software-related faults can arise from bugs, configuration errors, or resource management issues. Common software faults include:

- **Application Crashes**: Applications running on HPC systems may crash due to bugs, unhandled exceptions, or invalid inputs.
- **Operating System Failures**: The operating system on compute nodes can experience kernel panics or other critical failures that disrupt node operations.
- **Middleware Failures**: HPC systems often utilize middleware for job scheduling, resource allocation, and data management. Failures in these middleware components can lead to job delays or failures.

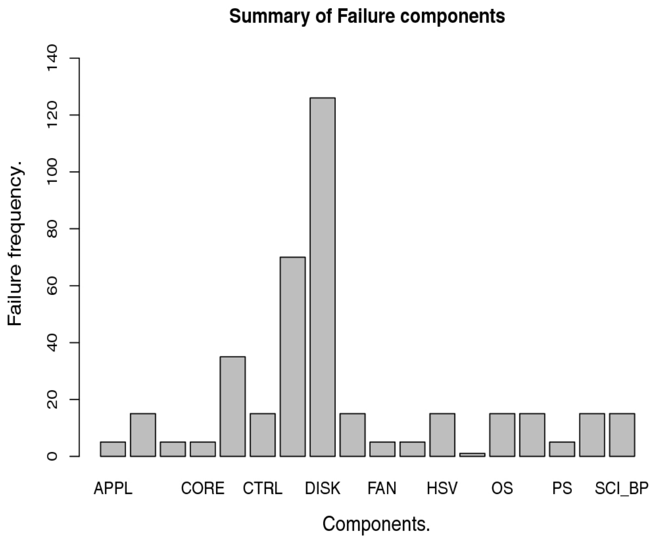

Figure 1. Summary of HPC hardware components' failure [10].

3) Human Errors: Human errors are another significant source of faults in HPC systems. These can include:

- **Configuration Errors**: Incorrect configuration of hardware, software, or network settings by administrators can lead to system instability or inefficiencies.
- **Operational Mistakes**: Mistakes made during system maintenance, updates, or job submissions can cause unintended downtimes or performance issues.

4) Environmental Factors: Environmental factors, although less common, can also affect HPC systems. These include:

- **Power Outages**: Unexpected power outages can lead to abrupt system shutdowns, data corruption, and hardware damage.
- **Cooling Failures**: HPC systems generate substantial heat and rely on efficient cooling mechanisms. Failures in cooling systems can cause overheating and subsequent hardware failures.

B. Origins of Faults in HPC

The origins of faults in HPC systems are varied and can be traced back to several underlying causes:

1) Complexity and Scale: HPC systems are inherently complex, comprising thousands of interconnected nodes, sophisticated networking, and vast storage solutions. This complexity increases the likelihood of faults due to the sheer number of components and interactions involved.

2) Technological Limits: As HPC systems push the boundaries of current technology to achieve greater performance, they also encounter the limitations of that technology. This can include issues like hardware failures due to higher operational stress and software bugs that surface under extreme conditions.

3) Evolution and Upgrades: HPC systems continuously evolve with new hardware and software upgrades. However, integrating new components with existing infrastructure can introduce compatibility issues, configuration errors, and unforeseen bugs.

4) Operational Environment: The operational environment of HPC systems, including physical location, power supply stability, and cooling efficiency, can significantly impact their reliability. Adverse conditions in these areas can exacerbate fault occurrences.

Understanding these common faults and their origins is essential for developing robust fault-tolerance strategies. The subsequent sections will explore how Kubernetes and Slurm address these challenges, providing insights into their fault-tolerance mechanisms within HPC infrastructures.

C. Kubernetes

Kubernetes is an open-source platform that helps to manage containerized workloads and services. It supports easy configuration and automation. With a quickly growing community, there are plenty of services, support options, and tools available [11].

1) Kubernetes Architecture: Figure 2 provides a high-level overview of Kubernetes architecture. A cluster is made up of compute machines, called nodes, and a control plane. The control plane manages the cluster, while the nodes run user applications. Typically, the control plane operates on a separate physical device, but it can also share a device with a compute node or be spread across multiple devices for redundancy. Pods, which are the scheduling units in Kubernetes, house the application containers. The following sections detail the Kubernetes components [12]:

- **API Server:** Exposes the Kubernetes API via HTTP.
- **Controller manager:** Manages all standard controllers for Kubernetes resources, like pods, services, and deployments. Each controller ensures that the resource's current state matches the desired state.
- **etcd:** A persistent distributed key-value store, which stores resources and cluster configuration information.
- **Scheduler:** Component responsible for determining the most suitable node for a given pod.
- **kubelet:** Agent that runs on every node within a Kubernetes cluster. It communicates via a REST API with the API server and handles interactions with the underlying container runtime.
- **Container runtime:** Component responsible for managing the containers' lifecycle. Examples of runtimes are CRI-O [13] and Containerd [14].
- **kube-proxy:** Facilitates communication with pods and implements the Service concept in Kubernetes, which abstracts the IP addresses of application pods to provide a unified method for exposure.

D. Fault Tolerance in Kubernetes

By default, Kubernetes offers two mechanisms for ensuring reliability: self-healing and replication. Self-healing involves Kubernetes restarting or replacing containers that fail, terminating unresponsive containers based on user-defined health checks, and delaying client access until containers are ready to serve [11]. Replication ensures a specified number of pod replicas are constantly operational, guaranteeing continuous availability of pods or homogeneous pod sets [16].

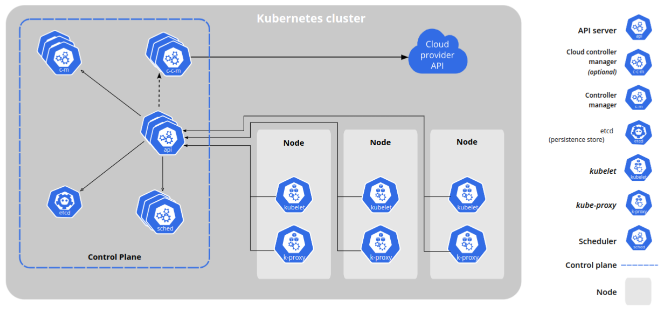

Figure 2. Kubernetes components [15].

Different fault-tolerance mechanisms can be also be implemented into Kubernetes. They are described below:

1) Horizontal Pod Autoscaler (HPA): Kubernetes boasts autoscaling as a crucial feature. This capability allows containerized applications to function reliably without constant manual intervention. A core autoscaling tool within Kubernetes is the HPA.

HPA guarantees high availability by dynamically adjusting the number of running pods, which are the execution units within the cluster. When triggered, HPA seamlessly adds new pods to distribute the workload, preventing negative impacts on existing pods.

To make autoscaling decisions, Kubernetes relies on metrics, which are statistics gathered from pods, applications, host machines, and the overall cluster. By default, Kubernetes uses Resource Metrics, which primarily monitor CPU and memory usage of pods and host machines. To enhance HPA's performance and flexibility, you can incorporate Customizable Metrics with the help of external software [17].

2) CRIU: CRIU (Checkpoint/Restore in Userspace) is a critical tool for managing process state in Linux environments. It empowers users to capture a running container or individual application at a specific point in time by performing a checkpoint. This checkpoint essentially freezes the process, recording its entire state, including memory, registers, and open file descriptors. This captured data can then be leveraged to restore the application and resume execution precisely from the checkpointed state. CRIU unlocks a multitude of powerful functionalities within the container orchestration landscape. These functionalities include live migration of containers or applications across machines without downtime, facilitating seamless application rollbacks through the creation of snapshots, and enabling remote debugging of running processes for efficient troubleshooting [18].

3) RAFT Protocol: Raft ensures that all replicas within the cluster maintain identical state machines, guaranteeing data integrity even in the face of node failures. It leverages a passive replication approach, where each node can assume one of three roles: leader, follower, or candidate [19]:

- **Leader:** The cluster elects a single leader node responsible for steering communication. The leader receives incoming requests, processes them, replicates state machine changes across follower nodes, and relays responses back to clients.
- **Follower:** While a leader is active, all other nodes transition to a follower state. Followers passively wait for the leader to transmit state machine updates, which they then apply to maintain consistency.
- **Candidate:** In the event of a leader failure, follower nodes transition to a candidate state and initiate a voting process to elect a new leader, ensuring rapid recovery and continued cluster operation.

E. SLURM

Slurm is a robust, open-source system designed to manage clusters and schedule jobs on Linux platforms, suitable for both large-scale and smaller deployments. It operates seamlessly without requiring kernel modifications and functions as a self-contained framework. As a cluster workload manager, Slurm performs three key roles: it allocates exclusive or non-exclusive access to compute nodes for users over defined time periods, facilitates the initiation, execution, and monitoring of jobs (especially parallel tasks) across allocated nodes, and manages resource contention by prioritizing pending work in a queue. These capabilities make Slurm essential for efficiently orchestrating cluster operations and optimizing job performance across diverse computing environments [20].

The main components of Slurm are detailed below, with a sample architecture depicted in Figure 3:

- **slurmctld:** This component monitors resource states, decides when and where to initiate jobs and job steps, and handles nearly all user commands, excluding database-related operations.
- **slurmd:** As Slurm's compute node daemon, slurmd is responsible for executing actual work on the node.
- **slurmdbd:** This component records accounting information and centrally manages certain configuration details, such as limits, fair share information, Quality of Service (QoS) settings, and licenses [21].

F. Fault Tolerance in SLURM

In the realm of HPC, fault-tolerant software is crucial in HPC environmeparamounts. Job resubmissions due to errors significantly impact resource utilization, leading to longer queue times for all users. While Slurm offers inherent fault tolerance, it primarily focuses on hardware issues rather than software errors. This ensures that malfunctioning jobs, exceeding resource limits or encountering failures, are isolated and prevented from disrupting other running jobs. However, the responsibility remains with users and developers to craft robust software, minimizing the need for job resubmissions caused by execution-time errors [1].

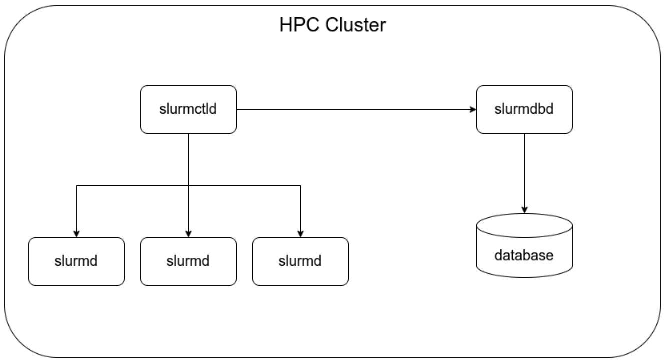

Figure 3. Typical Slurm architecture.

III. METHODOLOGY

In this section, we describe the methodology used for our literature review and comparative analysis of fault-tolerance mechanisms in Kubernetes and Slurm. We detail our approach to data collection, analysis techniques, and the criteria used for evaluating the effectiveness of fault-tolerance strategies.

A. Literature Review

To collect relevant academic papers, technical reports, and case studies on fault-tolerance in Kubernetes and Slurm, we searched various databases including IEEE Xplore, Google Scholar, and ACM Digital Library. We used search terms, such as "fault-tolerance in Kubernetes," "fault-tolerance in Slurm," "HPC fault-tolerance," and "resilience in HPC systems." We reviewed and summarized key findings and methodologies used in these studies to understand the current state of research and identify gaps for further exploration.

B. Comparative Analysis Framework

To collect logs from a Kubernetes cluster, including server logs and logs from objects, such as pods, deployments, and services, the EFK stack (Elasticsearch, Fluentd, and Kibana) has been installed. Elasticsearch [22] is a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. Fluentd [23], a fast, lightweight, and highly scalable logging and metrics processor, handles log forwarding. Kibana [24] provides a data visualization dashboard for Elasticsearch. Together, these tools enable the collection and analysis of Kubernetes-related system logs. Figure 4 illustrates the general architecture and workflow of the EFK stack.

In the context of Slurm workload management, agent logs generated by both the slurmctld control daemon and slurmd worker daemons on corresponding servers were collected and analyzed thoroughly. This analysis aimed to identify errors and malfunctions within HPC systems. Through a systematic examination of these logs, recurring patterns and underlying causes of system failures were identified.

We systematically evaluated each system against specific criteria, tracking essential aspects throughout the process. The analysis included metrics and benchmarks, such as Mean

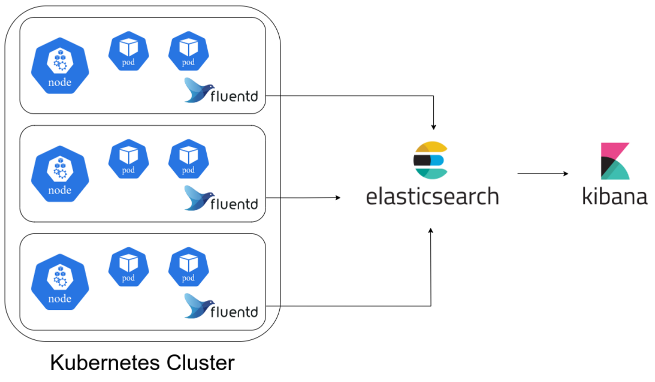

Figure 4. EFK stack on Kubernetes.

Time to Recovery (MTTR), detection accuracy, and resource utilization rates. Table I presents a comparison of these criteria for Kubernetes and Slurm.

As previously mentioned, various tools have been employed to collect logs and data from Slurm and Kubernetes. Table II provides a detailed list of these tools, outlining their functionalities and the specific data they collect.

C. Log Files Processing Steps

The processing of log files involved several intricate steps to ensure accurate labeling and analysis of the log entries, facilitating the identification and categorization of fault-tolerance mechanisms in HPC environments:

1) Data Cleaning and Preprocessing: Initially, raw log data from both Kubernetes and Slurm environments were cleaned to remove any extraneous information and ensure consistency in formatting. This involved:

- Removing duplicate entries to prevent data redundancy.
- Parsing timestamps and standardizing date-time formats.
- Filtering out non-informative log entries, such as routine status updates that do not indicate errors or significant events.

2) Labeling Using Language Models: To categorize the log entries, we employed a Large Language Model (LLM) for automated labeling. The steps included:

- Extracting messages from the cleaned log files.
- Using the LLM to generate unique labels for each log entry, indicating the type of failure or operational state.
- Implementing a similarity check using TF-IDF vectorization and cosine similarity to avoid redundant labels. If a new log entry was found to be 70
- Periodically saving the progress to avoid data loss during processing.

The following pseudocode summarizes this process:

```
for each log entry in cleaned_log_data:
    if is_similar(log_entry, existing_entries):
        assign existing label
        else:
            label = generate_label_with_LLM(log_entry,
             existing_labels)
        update existing_labels and existing_entries
    save progress periodically
```

3) Parallel Processing with GPU Acceleration: Given the large volume of log data, processing was accelerated using parallel computing techniques on GPUs. This approach significantly reduced the time required for vectorization and similarity computations:

- Utilizing CuPy for GPU-based array operations to handle large-scale vector computations efficiently.
- Employing parallel processing libraries, such as Dask to distribute the workload across multiple GPU cores.

4) Generating Summary Statistics and Visualizations: After labeling the log entries, summary statistics were computed to understand the distribution and frequency of different failure types. Visualization tools were employed to create charts and graphs depicting these distributions:

- Bar charts showing the number of log entries per label.
- Pie charts illustrating the percentage distribution of each label category.

The results were saved in both graphical and tabular formats to facilitate further analysis and reporting.

IV. Results

This section presents the findings from our literature review and comparative analysis. We detail the fault-tolerance mechanisms of Kubernetes and Slurm, highlighting their strengths and limitations in different failure scenarios.

A. Fault-Tolerance Mechanisms in Kubernetes

Kubernetes employs several fault-tolerance mechanisms to ensure system resilience:

- **Self-Healing**: Kubernetes restarts failed containers, replaces containers, and kills containers that do not respond to health checks.
- **Replication**: Ensures a specified number of pod replicas are running at any time.
- **HPA**: Adjusts the number of pods based on CPU/memory usage or custom metrics, improving availability.
- **CRIU**: Enables checkpointing and restoring of container states, supporting live migration and snapshots.
- **RAFT Protocol**: Ensures consistency among replicas, with leader election and state synchronization mechanisms.

Case studies have shown these mechanisms to be effective in maintaining high availability and quick recovery in various failure scenarios [12] [19].

B. Fault-Tolerance Mechanisms in Slurm

Slurm's fault-tolerance mechanisms focus primarily on hardware faults:

- **Node Failover**: Automatically reassigns jobs from failed nodes to healthy ones.
- **Job Checkpointing**: Allows jobs to be checkpointed and restarted from the last checkpoint in case of failure.
- **Health Checks**: Monitors node health and takes corrective actions, such as draining or rebooting unhealthy nodes.

TABLE I. DETAILED COMPARISON OF FAULT DETECTION, EVALUATION, AND TOLERANCE FACTORS IN KUBERNETES AND SLURM ENVIRONMENTS

Factor	Kubernetes	Slurm
MTTR	MTTR in Kubernetes is calculated by adding up all the downtime in a specific period and dividing it by the number of incidents.	MTTR in SLURM is calculated by taking the total repair time resulting from a particular failure and dividing it by the total number of repairs that are performed during a specific period.
Fault Detection Accuracy	Kubernetes has high fault detection accuracy due to liveness, readiness, and startup probes.	SLURM has moderate fault detection accuracy, primarily relying on node health checks and job statuses.
Resource Utilization Overhead	Kubernetes has a moderate overhead (10-15%) from extensive monitoring and replication.	SLURM has lower overhead (5-10%) due to simpler fault detection mechanisms.
Recovery Mechanisms	Kubernetes has built-in recovery mechanisms. For example, if a pod fails, Kubernetes automatically restarts it.	SLURM has mechanisms for recovery, but specific details are not readily available.
Failure Types and Frequencies	Specific data on failure types and frequencies for Kubernetes is not readily available.	Specific data on failure types and frequencies for SLURM is not readily available.

TABLE II. FAULT DETECTION TOOLS USED IN SLURM AND KUBERNETES ENVIRONMENTS

Environment	Tools	Functionality	Data Collected
SLURM	`sacct` [25]	Job accounting and fault detection.	Job accounting data.
	`sview` [26]	Graphical interface for fault detection and monitoring.	Slurm configuration, job, step, node and partitions state information.
	`Slurm simulator` [27]	Simulation for PdM and fault detection.	Effects of different Slurm parameters on HPC resource performance.
Kubernetes	`kubectl` [28]	Command line tool for fault detection and management.	Kubernetes cluster's control plane data.
	`Elasticsearch` [22]	Distributed search engine.	Different types of searches on collected data.
	`Fluentd` [23]	Data collector for fault detection.	System and component logs in the cluster.
	`Kibana` [24]	Visualization and monitoring tool for fault detection.	Visualization dashboards, cluster metrics.

- **Job Requeueing**: Jobs that fail due to node failures can be requeued and run on different nodes.

These mechanisms help mitigate the impact of hardware failures, but software faults and application-level failures require additional user intervention [1].

C. Error Distribution in Slurm and Kubernetes

The scenario analyzed in this subsection involves a critical examination of node failures during high-intensity computational tasks. To clarify, the analysis considers both the immediate impact on job execution and the subsequent recovery processes initiated by the fault-tolerance mechanisms in Kubernetes and Slurm.

Based on the data collected from a Slurm environment, we identified and categorized various errors. Figure 5 illustrates the distribution of different error types encountered. The most frequent error was the General Warning, comprising 64.95% of all errors. This was followed by PrologRunningError (21.13%), NodeError (13.61%), and OutOfMemoryError (0.14%). Other errors, such as JobCancelError, PartitionError, JobExitError, and TimeLimitExhaustedError each accounted for less than 0.1% of the total errors. The least frequent errors included LogrotateError, TopologyError, and NodeListError, each constituting approximately 0.0002% of the total errors.

In contrast, the Kubernetes environment exhibited a different error distribution 6. The most frequent error identified was the HTTP2StreamClosedError, which accounted for 20.08% of the total errors. This was followed by StreamClosedError (8.01%) and KubernetesRejectedConnection (4.25%). Additionally, errors such as KubernetesAPIEndpointConnectionFailure, ContextCanceledFailure, and ErrorSyncingHelmClusterRepo were also significant, each contributing to around 4.01% of the total errors. Less frequent issues included ConnectionRefusedError, DialTimeoutError, and etcdConnectionFailure, which occurred with a frequency of less than 1%. These findings highlight the critical areas where Kubernetes fault-tolerance mechanisms must focus, particularly on managing API server errors and stream-related failures.

The comparison of these distributions underscores the varying challenges faced by Slurm and Kubernetes environments, emphasizing the need for tailored fault-tolerance strategies in each system.

This distribution highlights the prevalence of certain errors over others, emphasizing the need for targeted fault-tolerance strategies that prioritize the most common and impactful issues.

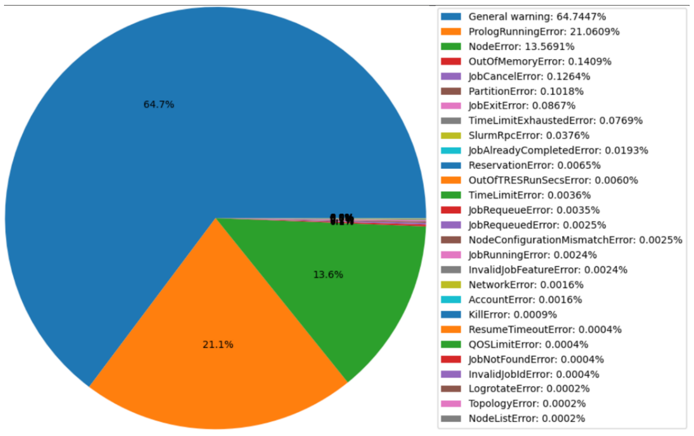

Figure 5. Distribution of error types in GWDG SCC Node Clusters Case study.

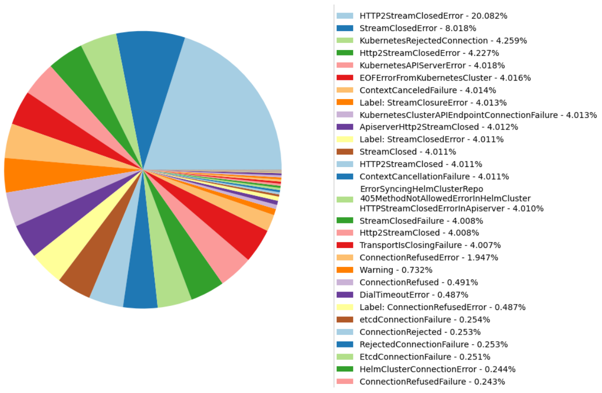

Figure 6. Distribution of error types in EU DECICE Project Kubernetes Clusters Case study.

D. Comparative Analysis

Our comparative analysis highlights the following differences:

- **Recovery Time**: Kubernetes generally achieves faster recovery times due to its self-healing and replication mechanisms. Slurm's recovery time depends on the effectiveness of node failover and job checkpointing.
- **Fault Detection**: Kubernetes has robust fault detection capabilities with its health checks and RAFT protocol. Slurm relies more on node health checks, which may not detect all types of faults promptly.
- **System Overhead**: Kubernetes introduces some overhead due to its extensive monitoring and replication processes. Slurm has lower overhead but may require more manual intervention for fault management.

Kubernetes excels in scenarios requiring high availability and rapid recovery, making it suitable for dynamic and scalable environments. Slurm is highly effective in traditional HPC setups with stringent resource management needs but may need enhancements for better software fault-tolerance.

V. Discussion

In this section, we interpret the results of our comparative analysis, discussing their implications for HPC infrastructure management. We also explore potential improvements and future directions for enhancing fault-tolerance in Kubernetes and Slurm.

A. Implications for HPC Management

Our findings suggest that:

- Kubernetes is well-suited for environments that require high scalability and rapid fault recovery. Its self-healing and autoscaling capabilities provide robust fault-tolerance with minimal manual intervention.
- Slurm remains a strong candidate for traditional HPC environments, particularly where job scheduling and resource management efficiency are highly valued.
- Combining elements of both systems could potentially yield a more comprehensive fault-tolerance strategy for HPC infrastructures, leveraging Kubernetes' dynamic fault management with Slurm's efficient resource scheduling.

In addition to the error distribution analysis in Slurm, a similar examination of error distribution in Kubernetes clusters reveals distinct patterns. Errors related to pod restarts, node unresponsiveness, and network partitions are prevalent. These findings underscore the importance of tailored fault-tolerance strategies in Kubernetes, comparable to those implemented in Slurm, to mitigate these common issues. Key considerations for choosing between Kubernetes and Slurm include the specific fault-tolerance needs, the complexity of workloads, and the desired level of automation in fault management. To address the limitations of both systems effectively, we recommend exploring practical solutions such as implementing dynamic fault-tolerance mechanisms tailored to specific failure scenarios. For instance, as discussed in [29], AI plays a crucial role in predictive maintenance strategies, integrating machine learning algorithms to predict potential failures and automatically adjusting resource allocations could enhance system resilience. Additionally, adopting hybrid models that combine the strengths of Kubernetes and Slurm may offer a balanced approach to fault tolerance in diverse HPC environments.

B. Future Directions

Future research and development could focus on the following areas:

- Enhancing Slurm's software fault-tolerance capabilities, possibly by integrating Kubernetes-like self-healing and replication mechanisms.
- Developing hybrid models that combine the strengths of Kubernetes and Slurm for improved fault-tolerance in diverse HPC environments.
- Exploring emerging technologies, such as machine learning for predictive fault detection and proactive fault management in HPC systems.

Potential enhancements to existing mechanisms in Kubernetes and Slurm could also involve better integration with AI-based monitoring tools and more granular control over fault-tolerance policies, enabling more effective and efficient fault management.

VI. Conclusion

The goal of this study was to compare the fault-tolerance mechanisms in Kubernetes and Slurm within HPC infrastructures. By examining the logs retrieved from GWDG SCC Node Clusters managed by Slurm, we identified and categorized various failures to understand their prevalence and impact. Our findings indicate that the most prevalent failure type is the General Warning, which accounts for 64.95% of all errors. This is followed by PrologRunningError (21.13%) and NodeError (13.61%), highlighting common issues related to job initialization and node malfunctions. Less frequent errors include OutOfMemoryError (0.14%), JobCancelError (0.13%), and PartitionError (0.10%). The rarest errors, such as LogrotateError, TopologyError, and NodeListError, each constitute approximately 0.0002% of the total errors, indicating specific system or configuration issues that occur infrequently.

Kubernetes can effectively handle these types of failures through its robust fault-tolerance mechanisms. Self-healing capabilities can automatically restart failed containers, addressing General Warnings and NodeErrors. HPA and resource limits can mitigate OutOfMemoryErrors by adjusting resources dynamically. Kubernetes' replication and RAFT protocol ensure high availability and data consistency, which can reduce the impact of network and configuration errors.

Slurm provides node failover and job checkpointing to handle node and job-related errors. Health checks and job requeueing mechanisms help maintain system stability by reallocating jobs from failed nodes to healthy ones, thereby addressing PrologRunningErrors and NodeErrors. For resource management issues like OutOfMemoryError and PartitionError, Slurm's resource scheduling and management features can be tuned to optimize usage and prevent such failures.

While our recommendations provide a strong foundation, concrete guidelines for achieving fault tolerance are necessary. These guidelines should include best practices for configuring and tuning fault-tolerance mechanisms, such as setting appropriate thresholds for autoscaling and designing robust health check probes.

By understanding these error distributions and leveraging the fault-tolerance mechanisms of Kubernetes and Slurm, HPC administrators can enhance system resilience and fault-tolerance. This targeted approach will improve the overall reliability and efficiency of HPC infrastructures. Future research should focus on integrating advanced fault detection and management technologies to further bolster the resilience of HPC systems.

Acknowledgment

The authors gratefully acknowledge the Göttingen Scientific Compute Cluster (SCC) for providing crucial cloud computing resources that greatly aided this research. We extend our special thanks to Felix Stein, Mojtaba Akbari, and Jonathan Decker for their exceptional support and contributions to the DECICE Project, including Felix's assistance with code optimization for analysis and their collective work on the project's infrastructure development.

This research was performed in the context of the DECICE project, funded by the European Union (grant agreement no. 101092582). Additionally, this project was supported by the Federal Ministry of Education and Research (BMBF), Germany, under grant no. 01|S22093A for the AI service center KISSKI.

References

[1] K. Klenk and R. J. Spiteri, "Improving resource utilization and fault tolerance in large simulations via actors," *Cluster Computing*, vol. 27, no. 6323–6340, 2024.

[2] G. Sayfan, *Mastering Kubernetes: Master the art of container management by using the power of Kubernetes*, 2nd ed. Packt Publishing Ltd., 2018.

[3] L. Dewi, A. Noertjahyana, H. Palit, and K. Yedutun, "Server scalability using kubernetes," in *2019 4th technology innovation management and engineering science international conference (TIMES-iCON)*, IEEE, 2019, pp. 1–4.

[4] G. Turin *et al.*, "A formal model of the kubernetes container framework," in *International Symposium on Leveraging Applications of Formal Methods*, Springer International Publishing, 2020, pp. 558–577.

[5] S. Muralidharan, G. Song, and H. Ko, "Monitoring and managing iot applications in smart cities using kubernetes," in *The Tenth International Conference on Cloud Computing, GRIDs, and Virtualization*, 2019, pp. 1–7.

[6] M. D'Amico, "Scheduling and resource management solutions for the scalable and efficient design of today's and tomorrow's hpc machines," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2021.

[7] N. A. Simakov *et al.*, "A slurm simulator: Implementation and parametric analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings 8*, Springer International Publishing, 2018, pp. 197–217.

[8] J. M. Kunkel *et al.*, "Decice: Device-edge-cloud intelligent collaboration framework," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, 2023, pp. 266–271.

[9] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how hpc systems fail," in *2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, IEEE, 2013, pp. 1–12.

[10] B. Mohammed, I. Awan, H. Ugail, and M. Younas, "Failure prediction using machine learning in a virtualised hpc system and application," *Cluster Computing*, vol. 22, pp. 471–485, 2019.

[11] *Kubernetes concepts - overview*, https://kubernetes.io/docs/concepts/overview/, Accessed: 2024-06-28.

[12] H. Schmidt, Z. Rejiba, R. Eidenbenz, and K.-T. Förster, "Transparent fault tolerance for stateful applications in kubernetes with checkpoint/restore," in *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*, IEEE, 2023, pp. 129–139.

[13] *Cri-o - lightweight container runtime for kubernetes*, https://cri-o.io/, Accessed: 2024-06-30.

[14] *Containerd - an industry-standard container runtime with an emphasis on simplicity, robustness and portability*, https://containerd.io/, Accessed: 2024-06-30.

[15] *Kubernetes components*, https://kubernetes.io/docs/concepts/overview/components/, Accessed: 2024-06-28.

[16] *Replication controller*, https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/, Accessed: 2024-06-28.

[17] T. T. Nguyen, Y. J. Yeom, T. Kim, D. H. Park, and S. J. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 462, 2020.

[18] *Criu: Checkpoint/restore in userspace*, https://criu.org/Main_Page, Accessed: 2024-06-28.

[19] G. M. Diouf, H. Elbiaze, and W. Jaafar, "On byzantine fault tolerance in multi-master kubernetes clusters," *Future Generation Computer Systems*, vol. 109, pp. 407–419, 2020.

[20] *Slurm overview*, https://slurm.schedmd.com/overview.html, Accessed: 2024-06-28.

[21] M. A. Jette and T. Wickberg, "Architecture of the slurm workload manager," in *Workshop on Job Scheduling Strategies for Parallel Processing*, Springer Nature Switzerland, 2023, pp. 3–23.

[22] *Elasticsearch - a distributed, restful search and analytics engine*, https://www.elastic.co/elasticsearch, Accessed: 2024-06-30.

[23] *Fluentd - an open source data collector*, https://www.fluentd.org/, Accessed: 2024-06-30.

[24] *Kibana - a data visualization dashboard*, https://www.elastic.co/kibana, Accessed: 2024-06-30.

[25] *Slurm workload manager - sacct*, https://slurm.schedmd.com/sacct.html, Accessed: 2024-06-30.

[26] *Slurm workload manager - sview*, https://slurm.schedmd.com/sview.html, Accessed: 2024-06-30.

[27] *Slurm simulator*, https://ubccr-slurm-simulator.github.io/, Accessed: 2024-06-30.

[28] *Command line tool (kubectl) | kubernetes*, https://kubernetes.io/docs/reference/kubectl/, Accessed: 2024-06-30.

[29] M. Bidollahkhani and J. M. Kunkel, "Revolutionizing system reliability: The role of ai in predictive maintenance strategies," in *CLOUD COMPUTING 2024: The Fifteenth International Conference on Cloud Computing, GRIDs, and Virtualization*, ISBN: 978-1-68558-156-5, 2024.