FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

LISA Pipeline Runner Dashboard

Marija Jakovleva

Masters of Software Engineering

Supervisor: Paulo Garcia

Co-supervisor: Ademar Aguiar

July 31, 2025

LISA Pipeline Runner Dashboard

Marija Jakovleva

Masters of Software Engineering

July 31, 2025

Abstract

Effective resource management and workflow execution monitoring are essential for ensuring performance, stability, and minimal downtime in high-performance computing (HPC) environments. This dissertation presents the design and implementation of a modular, extensible monitoring dashboard tailored to the LISA Pipeline Runner infrastructure, which is responsible for executing complex scientific workflows within the context of the LISA mission.

LISA (Laser Interferometer Space Antenna) is a space-based mission aimed at detecting gravitational waves generated by massive cosmic events such as merging black holes, supermassive black holes, and white dwarfs. To achieve this, a constellation of detectors will be deployed in space to capture these gravitational wave signals, which will then be transmitted back to Earth for analysis. This analysis will be carried out in dedicated Data Computing Centers (DCCs), each hosting a Pipeline Runner that orchestrates the data processing workflows. Given the computational demands and critical nature of the data, each Pipeline Runner must be supported by a dedicated monitoring dashboard to provide real-time visibility into system health and resource utilization during data analysis operations.

A comparative analysis of existing open-source and research-developed technologies was conducted, focusing on their capabilities in collecting, processing, and visualizing telemetry data from multiple sources. Key metrics such as CPU, memory consumption and workflow execution data were identified as critical for effective monitoring in the context of the LISA mission.

Based on this evaluation, a monitoring solution was implemented using Prometheus for metrics collection, Grafana for visualization, and Thanos for long-term storage of time-series data. Loki was also integrated to aggregate and query workflow logs. The system was deployed within a Kubernetes cluster, tightly coupled with the developed Pipeline Runner, with special attention to monitoring Argo Workflow executions and the overall health of the cluster. The resulting dashboard offers a centralized, intuitive interface for operators and developers, supporting observability, traceability, and future scalability within the LISA mission.

Acknowledgements

Firstly, I would like to express gratitude to my professor, Paulo Garcia, for his guidance, insightful feedback and support throughout the course of this dissertation, as well as for offering me the incredible opportunity to work on a project that aligns so closely with my passion for space.

I would also like to thank Tiago Gomes for his availability, advice, patience and practical help during the development of the project.

A special thank you to the LISA DDPC Team for their valuable guidance and insightful discussions throughout the development of this thesis.

I would like to thank my colleague, Manuel Pietrini, for developing the initial Pipeline Runner architecture, which served as a solid foundation for building the monitoring infrastructure presented in this dissertation. To my family and friends back home - thank you for your support and belief in me. Your encouragement kept me motivated throughout this journey.

To Iarina Majeri, sharing the highs and lows of this experience with you has made it all the more meaningful. Your optimism, support and late-night study sessions were an essential part of this accomplishment.

To my dear Ivan, thank you for being my biggest supporter and for always standing by my side.

Marija Jakovleva

"If you're offered a seat on a rocket ship, don't ask what seat!"

Sheryl Sandberg

Contents

List of Figures

List of Tables

Abbreviations and Symbols

API	Application Programming Interface
ARM	Advanced RISC Machine
CI/CD	Continuous Integration and Continuous Delivery
CNES	Centre National D'études Spatiales
CPU	Central Processing Unit
DB	Database
DCC	Data Computing Center
DDPC	Distributed Data Processing Center
ECSS	European Cooperation for Space Standardization
ESA	European Space Agency
GPU	Graphics Processing Unit
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
ICD	Interface Control Document
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
JSON	JavaScript Object Notation
LISA	Laser Interferometer Space Antenna
OOM	Out-Of-Memory
RAM	Random Access Memory
REST	Representational State Transfer
S3	Simple Storage Service
SLURM	Simple Linux Utility for Resource Management
SQL	Structured Query Language
TASC	Tuning Applications for SuperComputers
TSDB	Time Series Database
UI	User Interface
VAE	Variational Autoencoder
YAML	Yet Another Markup Language

Chapter 1

Introduction

1.1 Context

Data visualization is an important tool in software development, allowing businesses from different industries to quickly analyze composite data, gain insights, and optimize operations through data-driven strategies (Islam and Jin, 2019). By transforming complicated data into understandable visual components, engineers are able to interpret the results and act more efficiently. This is especially important when it comes to space missions, where the intricacy of systems and the vast amount of data obtained are more than what can be effectively analyzed through textual means.

The Laser Interferometer Space Antenna (LISA) is a flagship mission of the European Space Agency (ESA) under the Cosmic Vision program (European Space Agency, 2024). This project aims to detect gravitational waves in space time that are created by different cosmic events, such as merging black holes, supermassive black holes, and white dwarfs. These waves can be produced over various timescales, ranging from milliseconds to billions of years. Due to the limitations of ground-based measurements, such as interference from meteorological events and seismic activity, certain gravitational waves, particularly those in the low-frequency range, can only be detected in interplanetary space. These low-frequency waves have extremely large wavelengths, which cannot be effectively observed on Earth due to size limitations. In contrast, space-based detectors can be placed millions of kilometers apart, making it possible to capture these otherwise inaccessible signals. By capturing these waves, LISA will provide valuable insights into some of the universe's most violent and distant events, which have the potential to provide new understanding of galaxy formation and evolution, the presence and properties of dark matter, and other foundational questions in physics (Li et al., 2020; Danzmann and Rudiger, 2003).

The data collected by LISA is periodically transmitted via data streams to Earth, where it is received by the Ground Segment. At this stage, the data undergoes extensive analysis. A significant portion of this analysis is handled by a specialized system known as the GlobalFit Framework. The GlobalFit Framework is a sophisticated data processing system designed to combine and analyze various signals detected by LISA, identifying and extracting individual sources from noisy data. It is one of the frameworks executed within the Distributed Data Processing Center (DDPC)

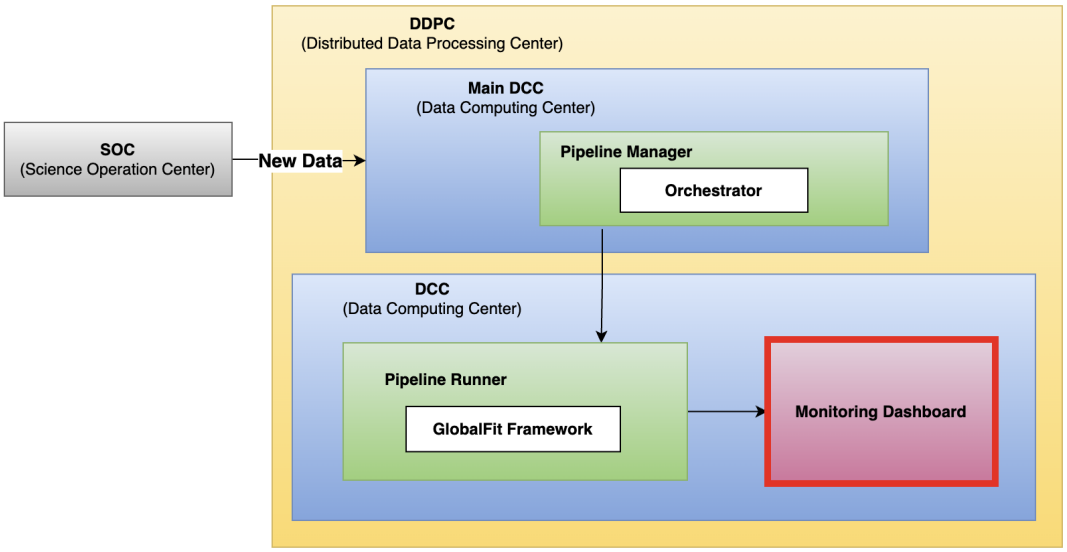


Figure 1.1: Simplified diagram of the LISA DDPC structure, with the objective of this thesis outlined in red

(See Figure 1.1), which is composed of multiple Data Computing Centers (DCCs). Each DCC contains a Pipeline Runner which is a framework-agnostic component responsible for managing the execution of scientific workflows. It is responsible for overseeing the execution of all LISA-related frameworks and orchestrating the flow of tasks by triggering, pausing, and stopping the different processing components. The Pipeline Runner can handle various frameworks, including GlobalFit, and is designed to support others in the future. (Colpi et al., 2024).

A visualization dashboard is a common solution for real-time monitoring and control of the system's performance, used by various organizations across different industries (Sarikaya et al., 2019). These dashboards are essential tools that allow users to interactively monitor, analyze, and interpret complex data through graphical representations. They organize vast amounts of information into visual formats, such as charts, graphs, maps, and tables, that are easier to understand and analyze at a glance. Dashboards allow decision-makers to quickly identify trends, patterns, and outliers, helping them make informed choices in real time, ensuring fast and accurate decision making (Yermalovich, 2020).

1.2 Motivation

Given the significance of the LISA mission, it is important to monitor the state of the system to ensure that processes are not obstructed. This is where the importance of a visualization dashboard comes into play. The success of such project relies not only on the scientific instruments but also on the infrastructure that supports data collection, processing, and analysis. Given the highly precise nature of LISA's instruments, the complexity of data processing, and the large volume of

data collected, effective monitoring and data visualization tools are essential to ensure smooth and reliable operations.

The LISA mission relies on a complex network of interconnected frameworks and data pipelines, requiring precise management to maintain data accuracy, integrity, and system performance. Advanced visualization and monitoring tools are essential for mission operators to track the overall health of the system, efficiently manage resources, and detect any bottlenecks or errors during execution.

A dashboard would offer an intuitive interface for users to monitor both internal and external system statuses. Internally, it would enable users to track the Pipeline Runner's state, ensuring that processes execute correctly, anomalies are detected, and intervention is possible when needed. Externally, it provides insights into system health, including critical metrics like CPU usage, RAM consumption, disk I/O, and disk utilization. Monitoring these parameters helps verify that resources are effectively used and prevents bottlenecks or faults during operation.

The motivation for this project lies in the operational efficiency a robust visualization dashboard can provide for a mission like LISA. Space missions operate under strict constraints on computing power, memory, and communication bandwidth, therefore inefficient resource usage could compromise mission success. A dashboard providing real-time insights into system status and resource utilization helps optimize resource use, ensuring that the system remains functional and responsive. This is especially crucial for long-duration missions, where undetected faults or inefficiencies can escalate over time and lead to significant setbacks.

By enabling the precise management of system resources and ensuring that data pipelines run smoothly, this project aims to contribute to the overall success of the LISA mission.

1.3 Objectives

The primary objective of this project is to design and develop a comprehensive dashboard that allows users to monitor the execution of LISA frameworks. This dashboard should provide real-time insights into the performance and status, helping users detect and address potential issues swiftly. The ultimate goal is to ensure that the frameworks operate efficiently and without unexpected interruptions.

Key Functionalities

- **Monitoring pipeline execution status**
  The dashboard should continuously track and display the status of the various pipelines within the LISA frameworks. This includes indicating whether a pipeline is currently running, paused, terminated, or completed. A clear visualization of pipeline statuses will allow users to monitor progress in real-time and quickly respond to any unusual conditions such as an unexpected pause or termination.

- **Profiling system resource consumption**
  System resource profiling is a critical aspect of monitoring framework performance. The dashboard should provide detailed metrics on the total utilization of key system resources, including CPU, RAM, disk I/O and network I/O. By displaying resource consumption in real time, users will be able to track overall system performance, ensuring that resources are being used efficiently and avoiding any obstacles.

- **Error diagnostics**
  The dashboard should provide clear error diagnostics. Error logs, stack traces, and metadata that helps pinpoint the root cause of a failure must all be included. In order to quickly identify patterns or reoccurring issues, the dashboard should allow users to filter and search for certain failure types.

- **Historical data retention**
  The dashboard should support historical data retention which is critical for teams to make informed decisions based on past events.

- **Correlation features**
  The dashboard should support correlating views by combining data from different sources in a single view. This solution should help analyze the related data together which can provide deeper visibility into how different elements of a system interact with each other.

- **Scalability insights**
  The dashboard should offer scalability insights. It would be possible to forecast where to add or reduce resources in response to demand based on past trends, and emphasizing this feature in the dashboard would greatly enhance that.

- **User interface requirements**
  An interface must be visually appealing and created with the user's experience as a priority. An effective user interface transcends mere functionality, presenting an intuitive design that makes seamless and pleasurable navigation. Thoughtful application of colors, typography and spacing improves readability and establishes a sense of order. Interactive elements must display responsiveness, and the layout should maintain consistency across all pages to minimize cognitive strain for the user.

Chapter 2

State of the Art

2.1 Literature review

This chapter starts with a detailed methodology outline which was used in conducting the literature review, including the search strategy, which outlines how the literature was filtered and selected using relevant keywords, with particular attention given to the relevance of the topic indicated in the abstracts.

Methodology

The research for this study was conducted through a structured and methodical approach to ensure that all relevant information was well-reviewed. A systematic search strategy was engaged to gather the most applicable literature on monitoring dashboards and data visualization techniques. For organizing and keeping track of the selected studies, a reference management tool, Zotero (Zotero, 2024), was used. This allowed for efficient collection, categorization, and annotation of sources, ensuring that the review process was well-documented and easily traceable.

Search strategy

The relevant literature for this study was identified and selected through a systematic search process. The academic databases - Web of Science and IEEE Xplore, were used to find peer-reviewed articles, conference papers, and books. The following keywords were used to find the literature: **LISA, Dashboards, Visualization, HPC, Monitoring**.

Preference was given to recent publications and after an initial review of abstracts and summaries, the most relevant sources were selected based on their alignment with the study's objectives.

2.2 Data visualization

Data visualization is an important aspect that allows to simplify complex information by converting data into a visual format, making it easier to identify patterns, trends, and relationships that might not be obvious. This helps users to quickly understand and process data of any size. (Islam and Jin, 2019).

Dashboards

The definition of a dashboard can be interpreted in different ways. According to Toasa et al. (2018), a dashboard is a tool that provides a visual representation of information, while Bach et al. (2023) defines dashboards as means of providing a quick overview of large and complex datasets. Meanwhile Sarikaya et al. (2019) describes a dashboard as a visual tool that presents important information organized on a single screen for quick and easy monitoring. Pappas and Whitman (2011) also states that a dashboard can consolidate key information in a single location, where it can typically be accessed through a web browser.

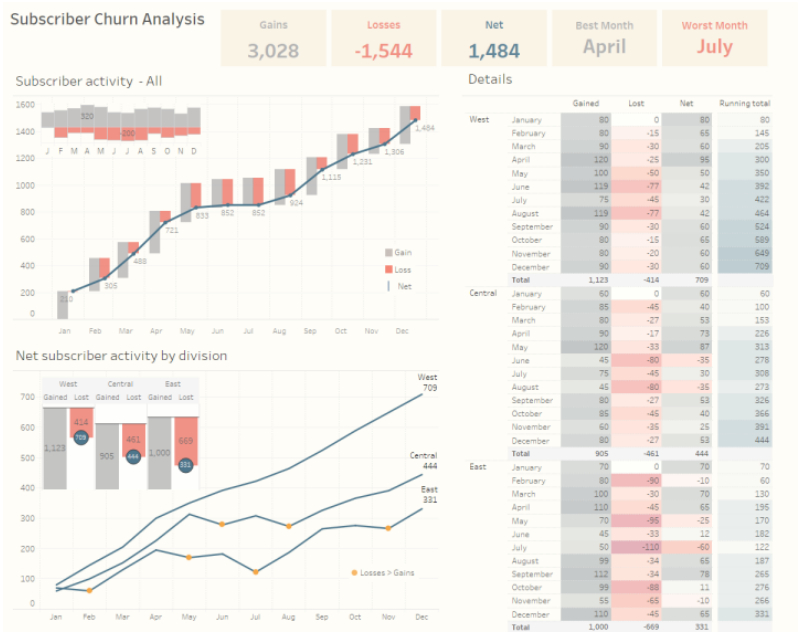
Dashboards play a critical role in supporting informed decision-making by consolidating and presenting complex information through intuitive visual elements. By introducing a layer of abstraction, dashboards can transform raw data into structured formats, such as charts, graphs, and tables that allow users to quickly grasp the information presented. This is especially important in environments that demand quick responses, as it allows users to view critical metrics at a glance and take immediate action when needed. Beyond simple visualization, dashboards are designed to display key insights, allow real-time data exploration, and offer interactive tools that support more detailed visualization. This interactivity allows users to customize their view according to their needs. Fundamentally, dashboards serve not only as monitoring interfaces but also as decision-support systems that improve operational efficiency (Sarikaya et al., 2019; Pappas and Whitman, 2011; Bach et al., 2023).

Categories of dashboards

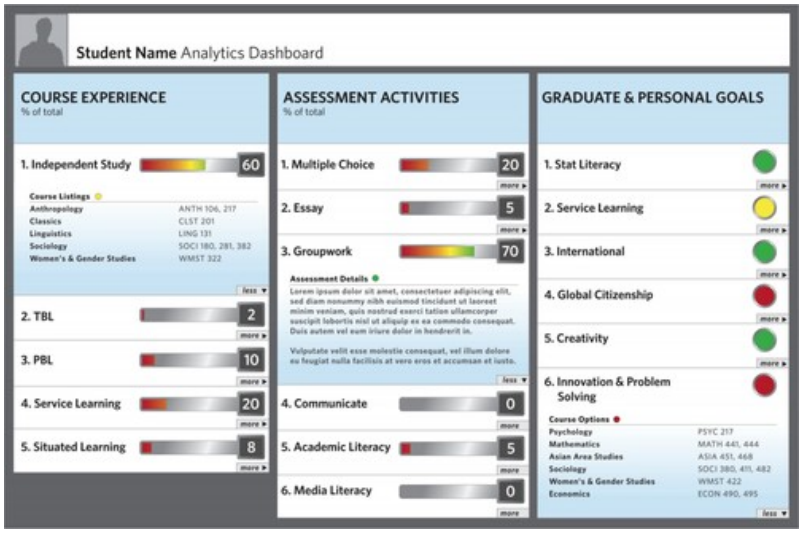
According to Sarikaya et al. (2019); Pappas and Whitman (2011); Bach et al. (2023), dashboards can be classified in four different ways based on their purpose and function:

1. Decision support dashboards can help guide decision making at various stages and include:
    - **Strategic Dashboards** - focus is on high-level metrics over a longer period of time to help evaluate a strategy.
    - **Tactical Dashboards** - provide insights for workers to help refine tactics
    - **Operational Dashboards** - deliver real-time metrics to front-line staff for process management.
2. Monitoring Dashboards are designed for real-time tracking of current conditions and providing timely feedback if an action is needed.

3. Analytical Dashboards offer a more interactive experience by combining tables and charts into a single screen view, which can provide an in-depth analysis of the data.



(a) Strategic dashboard



(b) Tactical dashboard

(c) Operational dashboard

Figure 2.1: Examples of decision support dashboards (Sarikaya et al., 2019).

Figure 2.1 presents examples of three distinct types of dashboards, each designed to support different levels of decision-making. The strategic dashboard provides a high-level overview of long-term trends, focusing on key performance indicators such as subscriber growth. The tactical dashboard is intended to assist with day-to-day management and making informed decisions at an intermediate level of the organization. In this example, it collects educational metrics to provide an overview of a student's academic performance. Lastly, the operational dashboard focuses on real-time, task-specific information without combining it into higher-level summaries. It highlights performance metrics, making it well-suited for daily monitoring and decision-making. However, it may lack the broader context provided by strategic or tactical dashboards (Sarikaya et al., 2019).

Key characteristics of dashboards

According to Bach et al. (2023), dashboards should be designed to prioritize clarity and simplicity, consequently avoiding overwhelming users or introducing unnecessary visual clutter. They should combine practical functionality with a thoughtful visual design to present information effectively.

Consistency, user interaction support, and complex data handling are essential traits of a well-designed dashboard. Charts should be neatly organized, grouped by related attributes, clearly separated, and arranged in a logical order to make the information easy to understand and navigate.

Dashboard design patterns

Bach et al. (2023) describes the key patterns a dashboard could have:

1. Content design patterns are important to the presentation of data and include the following patterns:
    - **Data information** - defines how data is abstracted and presented within a dashboard. Uses detailed datasets, aggregations, filters, derived values, thresholds, and single values.
    - **Meta information** - contains additional information about the data, such as data sources, disclaimers, descriptions, update details, and annotations.
    - **Visual representations** - refer to the various formats used to display data within a dashboard, like tables, lists, detailed visualizations, and miniature charts.
2. Composition design patterns define how the individual content elements are arranged and displayed and include the following patterns:
    - **Page layout** - define how widgets are arranged and visually grouped within a dashboard. Common layouts include open, stratified, table-based, grouped, and schematic designs.
    - **Screenspace layout** - focuses on optimizing content to fit within a single screen. Contains several approaches to managing the screenspace, such as screenfit, overflow, detail on demand, parameterization and multiple pages.

- **Structural** - when applicable, define the relationships between multiple pages of a dashboard. Multi-page dashboards can have the following structures: parallel, hierarchical, or open.
- **Interaction** - outlines typical interaction methods in dashboards. Allow for exploration, drill-down, navigation, and personalization.
- **Color** - a crucial component in visualization. They can enhance the visual appeal and elicit an emotional response. Commonly used in dashboards containing infographics.

Design trade-offs

Designing a dashboard involves carefully considering different trade-offs to achieve a practical balance. The **level of abstraction** determines how much the data should be simplified for clarity, while the **use of screen space** focuses on the extent of visual space allocated. The **number of pages** plays an important role in effectively presenting information, and the **degree of interactivity** emphasizes incorporating user engagement to improve overall usability.

Visual graphs and charts

Elements such as lines, bars, points, and shapes can be used to illustrate trends, patterns, and relationships within a dataset. They simplify complex information by formatting it to become more accessible and intuitive to enhance understanding. They are an essential component of dashboards, presenting data in a clear and interpretable manner (Islam and Jin, 2019; Toasa et al., 2018). The type of visual graph chosen should depend on the type of data being presented and the message that needs to be conveyed (Pappas and Whitman, 2011). Some common chart types can be described as follows:

- **Line charts** are well-suited to visualize data changes over a continuous period of time (Toasa et al., 2018) which helps to recognize patterns and changes, as well as performance issues (Aggarwal et al., 2024)
- **Bar charts** are useful for comparing values because the length of the bar is easily comparable to the eye. They can also highlight categories that may require attention or further investigation (Pappas and Whitman, 2011)
- **Pie charts** are effective for illustrating shares or proportions of different items (Toasa et al., 2018) but are less efficient than bar charts because people often find it challenging to accurately compare angles (Pappas and Whitman, 2011).
- **Scatter plots** use individual data points (Pappas and Whitman, 2011) and can be used to display a relationship between two variables (Aggarwal et al., 2024; Sedrakyan et al., 2019)
- **Area charts**, like line charts, highlight the extent of change over time and draw focus to trends (Sedrakyan et al., 2019).

- **Radar charts** arrange data points evenly around a circle, using their distance from the center to represent their values (Toasa et al., 2018).
- **Tree diagrams** show information clusters and hierarchical structures, particularly when coming from a single point of origin (Islam and Jin, 2019).
- **Bullet bars** function well for visualizing data comparisons and are considered to be sparklines, which can be utilized to quickly express messages (Pappas and Whitman, 2011).
- **Sankey diagrams** show a series of connected nodes with the width of each link representing frequency, clarifying the dynamics of transaction flow within a system (Toasa et al., 2018).
- **Network diagrams** are able to represent relationships as nodes and connections (Toasa et al., 2018).
- **Correlation matrices** present a table that enumerates the correlation coefficients of the columns and rows which can be used to rapidly identify connected variables (Toasa et al., 2018).

Visual graph selection

Graph selection is crucial because it plays a key role in conveying information effectively. Graphs should not only encourage viewers to analyze and compare data but also focus their attention on understanding the content itself, rather than focusing on the way it is presented (Tufte, 2001).

When selecting a certain visualization graph for data, it is crucial to ensure that the visualization corresponds with the data type, the intended message, and the target audience. Certain visualizations show greater efficiency than others, especially in dashboards featuring multiple graphics concurrently (Pappas and Whitman, 2011).

Line graphs, bar charts, and bullet bars are the ideal choices in cases when data has to be compared (Pappas and Whitman, 2011; Sedrakyan et al., 2019). Meanwhile, pie charts or speedometers are considered to be less efficient choices since it is difficult to compare angles, however, in some situations they might be utilized to highlight anything that could call for quick response (Pappas and Whitman, 2011).

Bar charts, boxplots, and column charts are quite effective when it comes to analyzing data distributions. In case of time-series data, line charts and area charts can show trends and patterns over a period of time. For displaying composition of data or percentual shares of a whole, pie charts and donut charts are great options to consider (Sedrakyan et al., 2019).

An important aspect to consider when creating visualizations include keeping them simple for easier interpretation (Bach et al., 2023). Efficient use of space is essential, with bar and line graphs being preferable in constrained layouts, while avoiding overly spacious visuals like pie charts. Eliminate background visuals, pointless animations, and pictures to minimize distractions (Pappas and Whitman, 2011).

Colors play an important role in display of information. Tufte (1990) states that "tying color to information is as elementary and straightforward as color technique in art". However, applying color effectively is more complex than it appears, requiring thoughtful placement and strategy. Color plays an important role in visual communication and should be applied with care. Excessive brightness or the use of too many colors can overwhelm the viewer, while accessibility considerations, such as combining color with intensity or borders, help ensure that information is clear to individuals with colorblindness. In performance measurement, colors often carry specific meanings: red typically signals underperformance, green indicates satisfactory performance, and yellow suggests no immediate action is needed (Pappas and Whitman, 2011). An example of correct color coordination can be seen in Figure 2.2

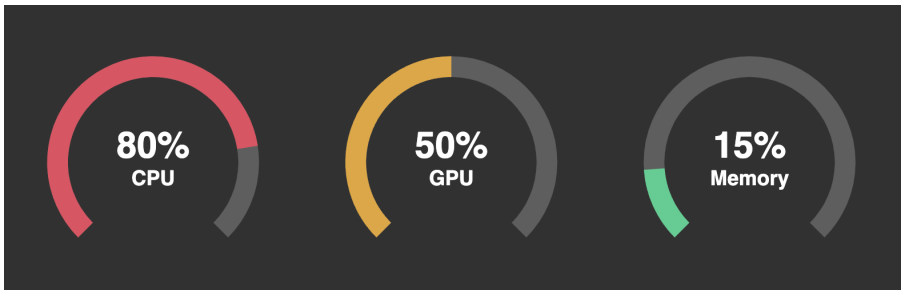


Figure 2.2: Example of using a speedometer to display consumption metrics with appropriate color coordination.

Overall, the design of dashboards should be an iterative process, with feedback being an essential component for making improvements. By including additional users in the review process, a variety of different perspectives can be brought to light, which can bring significant insights that can be used to modify and improve the dashboard (Yermalovich, 2020).

2.3 HPC monitoring

After a detailed presentation of data visualization and its importance in the previous section, we will now focus on High Performance Computing (HPC) monitoring and specific applications for it. HPC monitoring is data collection and analysis aimed at comprehending the state and performance of HPC systems and their supporting infrastructure (Allcock et al., 2011; Ott et al., 2020). When working with supercomputers, it is essential to have an understanding of the state of the system as a whole. It is important to keep a close eye on various components of the system, including the central processing unit (CPU) and graphics processing unit (GPU) resources, while performing tasks that need a significant amount of these resources (Litzinger et al., 2023).

Key aspects of HPC monitoring

HPC monitoring facilitates important key aspects:

- **System understanding** - HPC monitoring tools provide information about how HPC systems are being used and whether the resources that have been allocated are being used efficiently. It assists users in confirming that they are utilizing the computational resources they have purchased and allocated (Litzinger et al., 2023).
- **Performance analysis** - evaluation of work performance, locating inefficiencies and bottlenecks. Examining CPU and GPU use, memory usage, disk read/write times, and network traffic are all part of performance analysis (Litzinger et al., 2023; Allcock et al., 2011; Netti et al., 2022).
- **Anomaly detection** - monitoring helps with detection of irregularities, flaws, and failures. It makes it possible to spot odd trends or deviations from typical behavior, which may point to possible issues (López-Peña et al., 2020; Borghesi et al., 2022) .
- **Proactive issue resolution** - early identification of possible issues allows to support proactive maintenance and corrective action. Since failures may destroy work and waste time, this helps to prevent failures and reduce downtime, which is vital (Allcock et al., 2011; López-Peña et al., 2020; Borghesi et al., 2022).
- **Resource optimization** - monitoring can be utilized to optimize resource allocation and scheduling. This may result in enhanced utilization of computing resources and improved performance (Ott et al., 2020; Netti et al., 2022; Isakov et al., 2020; Yang et al., 2023).
- **Capacity planning** - Monitoring provides a full view of system utilization, which is useful for capacity planning and future system expansions (Yang et al., 2023).
- **Data-driven decision making** - monitoring allows to make well-informed choices about system maintenance, upgrades, and operation (López-Peña et al., 2020) .
- **Continuous feedback** - Monitoring makes it possible for operations to provide development with quick and ongoing feedback (López-Peña et al., 2020).
- **Cost efficiency** - Monitoring can lower operating expenses by improving performance and resource usage (Ott et al., 2020; Netti et al., 2022) .

Preferences in technology

Ott et al. (2020) outlines the preferred technologies used for monitoring in HPC environments, specifically the popularity of different databases and visualization tools. Relational databases like MySQL, MariaDB, and PostgreSQL are clearly giving way to NoSQL solutions, such as document stores like Elasticsearch, which is commonly used as a time series database, and time series databases like InfluxDB and Cassandra. Grafana is clearly preferred when it comes to viewing the data that has been gathered. Many sites use Grafana for visualization, either in addition to or instead of Kibana, despite Kibana's close integration with the Elastic Stack (Elasticsearch). This is

due to the fact that Grafana is able to be connected to a multitude of different data sources because of its plugin architecture. In order to ensure scalability and fault tolerance, a cluster deployment of modern message busses is common. They typically follow one of two main architectures: the smart broker/dumb consumer model used by systems like RabbitMQ and MQTT, or the dumb broker/smart consumer model employed by Kafka. The difference between these approaches is that in the smart broker/dumb consumer scenario, the consumers acknowledge the receipt of messages and the broker then deletes them from the queue, whereas in a dumb broker/smart consumer scenario, the consumer is accountable for sustaining their state.

Data retrieval

Information about resources can be obtained from agents in two distinct ways: through *push* or *pull* mode. The term "pull mode" refers to the situation in which agents are queried by the server for data, whereas the term "push mode" represents the situation in which agents submit data about resources to the server on their own schedule (Stefanov et al.).

2.4 Related work

Open source software overview

There are many open-source tools available on the market to be used for visualization. Verginadis (2023) provides a comprehensive overview of the 20 most well-known monitoring tools. The review includes:

- Vendor-specific Cloud Monitoring Services
  - Amazon CloudWatch (Amazon Web Services)
  - Azure Cloud Monitoring (Microsoft)
  - Google Cloud Monitoring (Cloud)
- Open-Source Monitoring Tools
  - Collectd (collectd)
  - Cacti (Cacti)
  - Icinga (Icinga)
  - Munin (Munin)
  - Nagios (Nagios)
  - Netdata (Netdata)
  - Prometheus (Prometheus)
  - Shinken (Shinken)
  - Zabbix (Zabbix)

  - Openstack Telemetry Service (OpenStack)

- Commercial Monitoring Solutions
  - Datadog (Datadog)
  - Dynatrace (Dynatrace)
  - LogicMonitor (LogicMonitor)
  - New Relic (Relic)
  - Opsview (Opsview)

- Other
  - Apache Chukwa (Apache Software Foundation)
  - Hadoop Performance Monitoring (Hadoop)
  - SequenceIQ (SequenceIQ)

Out of the 20 tools described in the paper, the focus will be on the open-source monitoring tools which will be thoroughly reviewed and used as inspiration for developing the LISA Pipeline Runner dashboard.

In Table 2.1 the comparison of the mentioned tools is presented, where the evaluation was based on the requirements and functionalities necessary for the LISA Pipeline Runner dashboard, where the column names denote the following properties:

- Pull mode - receive resource information by query.
- Logs - log collection and display.
- Faults - error handling.
- Jobs - job or pipeline execution monitoring.
- Resources - resource metric collection, such as CPU/GPU, memory, etc.
- Historical - historical data retention.
- Real-time - real-time data display.
- Visuals - integration of visualization methods to display data.

Table 2.1: Comparison of open-source tools

Technology	Pull mode	Logs	Faults	Jobs	Resources	Historical	Real-time	Visuals
Collectd (collectd)	no	yes	yes	maybe	yes	yes	yes	no
Cacti (Cacti)	yes	yes	yes	no	yes	yes	yes	no
Icinga (Icinga)	no	yes	yes	yes	yes	yes	yes	yes
Munin (Munin)	yes	yes	yes	yes	yes	yes	yes	yes
Nagios (Nagios)	yes	yes	yes	yes	yes	no	yes	yes
Netdata (Netdata)	yes	yes	yes	yes	yes	yes	yes	yes
Prometheus (Prometheus)	yes	yes	yes	yes	yes	yes	yes	yes**
Shinken (Shinken)	yes	maybe	yes	no	yes	maybe	maybe	yes***
Zabbix (Zabbix)	yes	yes	yes	yes	yes	yes	yes	yes
Openstack Telemetry (OpenStack)	no	maybe	yes	maybe	yes	yes	maybe	no

*The "maybe" option in this table denotes that the parameter is not specifically stated in the reference.
** Possible with the integration of Grafana (Labs).
*** Possible with integration of visualization tools.

2.5 Custom frameworks and tools

In addition to widely-used open-source software tools for monitoring, other distinct frameworks and tools have been developed, some expressly for high-performance computing (HPC) monitoring. Nine tools were evaluated against the objectives of the LISA Pipeline Runner dashboard described in the third section to determine their compatibility with the required features.

1. **STREAM** - (Streaming Telemetry for Resource Events, Analytics, and Monitoring) is a scalable platform built to help analyze real-time telemetry data coming from many high-performance computers, making it easier to track and respond to events happening in composite systems. While it does not employ any complex visualization methods within the framework itself and focuses on real-time data primarily, it does contain Apache Kafka integration for collecting data (Adamson et al.).

2. **Unnamed** - a browser based HPC monitoring dashboard which tracks resource and job performance metrics. This tool has access to both real-time and historical data, but lacks complex correlating views, where different sets of data would be presented in a single view (Aggarwal et al., 2024).

3. **Prodigy** - a framework developed to identify performance anomalies HPC systems. It is based on a variational autoencoder (VAE) which is a category of a machine learning model. Prodigy has job tracking with resource metrics, as well as historical and real-time data, but it lacks complex visualizations and does not clearly specify if it contains any error handling for job failures (Aksar et al., 2023).

4. **LLload** - a tool that was developed for monitoring HPC jobs and provides job usage resource metrics in real-time. While this tool does have great advantage in correlating resource usage to job executions and displaying that in a clear human-readable format, it does lack a lot of the other required features, such as log collection, visualizations, error handling and historical data retention (Byun et al., 2024).

5. **LIMITLESS** - a lightweight monitoring tool designed for monitoring HPC clusters. This tool mainly focuses on monitoring system resources in real-time with variable monitoring period, allowing to monitor changes down to each sub-second. Despite these features, the paper does not mention explicit log collection or monitoring job status and execution (Cascajo et al., 2022).

6. **JobViewer** - web-based tool used to monitor HPC systems by using graph-based visualization to display the structure of the clusters, their metrics, and task scheduling information. This tool has the useful functionality of comparing the job information with the health metrics and includes both historical and real-time data display, but does not clearly specify collecting of logs or implementation of any fault monitoring protocols (Dang et al., 2022).

7. **MonSTer** - a novel monitoring tool for HPC systems engineered to correlate applications with resource use. By using a time-series database, it is able to show both real-time and historical data and it uses a tool called HiperJobViz (Nguyen et al., 2019) to visualize the data. It does not explicitly state about log collection (Li et al., 2020).

8. **NetGraf** - a network monitoring service that combines data from several monitoring sources into a single dashboard to display network status in real time. NetGraf also supports historical data analysis, since it has anomaly detection and uses historical data to train on. Since the focus of this tool is on network monitoring and correlated resource consumption, it does not have pipeline or job execution monitoring (Mohammed et al., 2021) .

9. **PowerJoular and JoularJX** - a software tool for power monitoring that assists developers in understanding and assessing the power consumption of their apps. It works for different architectures (eg. x86_64, ARM) and single-board devices like Raspberry Pi and has both support for historical data and real-time data. The visualization is done through the command-line interface which enables it to be used in a remote server environments but removes the option of having complex visualization diagrams to help analyze data. Even though JoularJX does have visualization graphs, they are only limited to programs written in Java (Noureddine, 2022) .

10. **TASC** - also called Tuning Applications for SuperComputers. focuses on optimizing the performance of modern supercomputers by detecting inefficiencies in HPC applications and analyzing how effectively system resources are utilized. It uses SLURM (SchedMD) to collect data about job execution and stores data in a MongoDB (MongoDB, Inc.) database for easy retrieval. Authors state that the data is never deleted from the database, which allows them to work with historical data. TASC also has a service which displays resource information in configurable dashboards for users (Voevodin et al., 2024) .

11. **JuMonC** - a user-controlled monitoring tool for tracking resources and jobs. Unlike administrator-managed tools, it operates with a dedicated instance for each job. It offers a REST API that delivers data in JSON format, which allows users to query information as frequently as needed (JuM, 2025) .

The comparison of the aforementioned tools is presented in Table 2.2, where the evaluation was based on the same requirements as in Table 2.1

Table 2.2: Comparison of custom tools

Technology	Pull mode	Logs	Faults	Jobs	Resources	Historical	Real-time	Visuals
STREAM (Adamson et al.)	no	yes	yes	no	maybe	no	yes	no
Unnamed (Aggarwal et al., 2024)	yes	yes	maybe	yes	yes	yes	yes	no
Prodigy (Aksar et al., 2023)	yes	yes	no	yes	yes	yes	yes	no
Lload (Byun et al., 2024)	yes	no	no	yes	yes	no	yes	no
LIMITLESS (Cascajo et al., 2022)	yes	maybe	yes	no	yes	yes	yes	yes
JobViewer (Dang et al., 2022)	yes	maybe	maybe	yes	yes	yes	yes	yes
MonSTer (Li et al., 2020)	no	maybe	yes	yes	yes	yes	yes	yes
NetGraf (Mohammed et al., 2021)	no	no	yes	no	yes	yes	yes	yes
PowerJoular and JoularJX (Noureddine, 2022)	yes	no	no	no	yes	yes	yes	maybe
TASC (Voevodin et al., 2024)	no	no	yes	maybe	yes	yes	maybe	yes
JuMonC (JuM, 2025)	yes	yes	yes	yes	yes	yes	yes	yes**

*The "maybe" option in this table denotes that the parameter is not specifically stated in the reference.
**Possible only with integration with LLview (Jülich).

Sanchez et al. (2016) also proposed a design for an HPC monitoring system which will be an extension on top of their existing system. The drive for this expansion stems from the desire to correlate data and store that data historically. This design is considered to be scalable and will use RabbitMQ for data distribution. The main focal points of the design include data correlation, historical data analysis and analysis of meaningful data.

Research Gaps and Proposed Contribution

The primary objective of this thesis is to design and implement a robust, extensible monitoring dashboard tailored to the LISA Pipeline Runner, where performance and system health must be monitored during workflow execution.

While there are many existing monitoring solutions ranging from open-source tools like Netdata, Prometheus, and Munin to custom research-developed platforms such as JuMonC, the choice of Prometheus and Grafana for this project was driven by a combination of technical suitability and alignment with the preferences of the LISA DDPC system team. Both tools are open-source and can be self-hosted, offering considerable flexibility in terms of deployment and customization. Prometheus supports dynamic service discovery and can collect metrics from any service that aligns to its naming conventions, making it highly adaptable to evolving infrastructure. Grafana complements this by supporting a wide variety of data sources and graphs, enabling not only real-time visualization of resource consumption but also the future integration of domain-specific data to be visible in a single environment as system requirements evolve. Additionally, the system team's prior experience with both tools resulted in a strong preference for their adoption which will minimize the onboarding time and reduce the learning curve, promoting long-term maintainability beyond the scope of this thesis.

Chapter 3

Systems Engineering

This Chapter presents the methodologies applied throughout the development of the monitoring dashboard. It begins with an overview of the requirements elicitation process, which led to the creation of an official requirements document structured in accordance with European Cooperation for Space Standardization (ECSS) standards: ECSS-E-ST-40C (2009) and ECSS-D-00-01C (2020). Next, this Chapter provides an Interface Control Document (ICD) overview, followed by a system architecture Section presenting a high-level breakdown of the core components. Continuing on to data acquisition, it explains how relevant information is collected and integrated into the system. This is followed by an explanation of data processing, covering how Argo (Argo Project) exposes metrics through dedicated endpoints, which are then automatically discovered and scraped by Prometheus in a time-series format.

3.1 Requirements elicitation

Before developing an application, it is important to elicit requirements that will describe all necessary functionalities. When specifying software requirements for the European Space Agency, certain guidelines must be followed. These guidelines are outlined on the ECSS (2025) website.

In the scope of this thesis, a comprehensive requirements document was developed to guide the design and implementation phases of the dashboard, where the functional and non-functional requirements are outlined. The initial set of requirements was verbally provided by the CNES (Centre national d'études spatiales) DDPC system team, who act as key stakeholders in this project. Throughout the development process, requirements were further refined and validated through regular interactions with the team, primarily through biweekly review meetings and bilateral discussions. This ongoing collaboration ensured that the dashboard design remained aligned with the operational needs and expectations of the CNES DDPC infrastructure and offered practical, domain-specific input that helped guide the technical choices and determine the priorities of the features implemented. The process shown in the Figure 3.1 outlines a clear and iterative approach to requirements development, consisting of elicitation, analysis, validation, and documentation

and was followed throughout this project to ensure a structured development of system requirements.

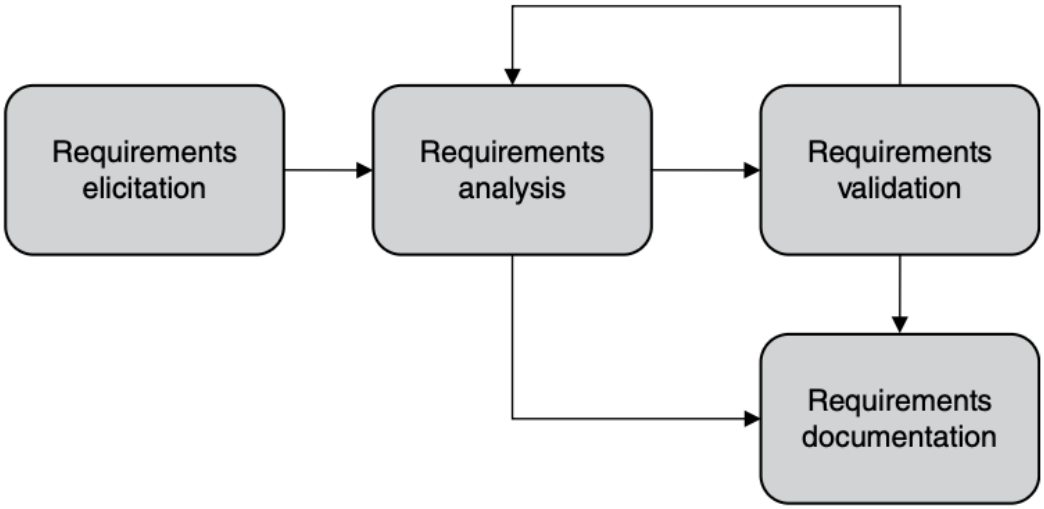


Figure 3.1: Simple requirements development process (Kossiakoff et al., 2020)

Significant effort was devoted to ensuring that these requirements and the resulting system design were in full compliance with the ECSS standards. It should serve as a base reference during the development process to ensure alignment between the dashboard's objectives and its execution. It was developed using the ECSS-D-00-01C (2020) document which describes in detail how to draft requirements and the ECSS-E-ST-40C (2009) document which contains general guidelines for developing software in the space sector.

General requirements. Outlined below are the general requirements established to guide the development of the monitoring dashboard. These requirements build upon a core set of general expectations verbally provided by the LISA DDPC system team, which served as the foundation for further requirements specification. A complete list of all elicited requirements can be found in the official requirements document in Appendix A.

a) The dashboard shall support workflow monitoring, providing visibility into the status, progress, and logs of all monitored workflows.

b) The dashboard shall provide error diagnostics to assist users in identifying, analyzing, and understanding errors occurring in monitored workflows.

c) The dashboard shall ensure that all monitored data, including workflow states and metrics are stored in a retrievable manner for both real-time and historical analysis.

d) The dashboard shall provide a user interface that supports intuitive navigation and visualization suitable for both high-level overviews and detailed analysis.

e) The dashboard shall associate all monitored resource metrics (e.g., CPU, memory, disk usage) with the corresponding workflows, services, or processes to provide insights into resource consumption.

f) The dashboard should support correlation features that allow users to cross-reference workflow events, errors, and resource usage within a shared timeline or analysis view.

g) The dashboard may support scalability insights, providing visibility into system performance under varying loads.

Validation. To ensure the monitoring dashboard meets both functional and non-functional requirements, a structured validation process was established. The selected validation methods, such as analysis, demonstration, and inspection shall be applied according to the nature of each requirement (see Table 3.1). Analytical validation involves reviewing Prometheus queries, dashboard panels, and data export mechanisms to confirm data accuracy and completeness. Demonstrations shall be performed by executing Argo workflows and verifying their visibility and traceability within the Grafana dashboards. Configuration files, including YAML definitions and retention policies, shall be inspected to assess compliance with system design and maintainability objectives. Together, these methods provide comprehensive coverage in the reliability of the implemented solution. For a full validation matrix, please refer to the full requirements document attached in Appendix A.

Table 3.1: Overview of validation methods

Validation Type	Description
Analysis	Reviewing Prometheus queries, dashboard panels, and data export mechanisms to ensure correctness and completeness.
Demo	Executing Argo workflows and verifying their correct visualization and behavior in the Grafana dashboard.
Inspection	Examining configuration files, Kubernetes manifests, and data retention policies to ensure compliance and maintainability.

Due to the strict structural and formatting guidelines imposed by the ECSS standards, the full developed requirements document could not be embedded directly in the main body of this thesis. As a result, it has been included in Appendix A. This document represents a substantial portion of the work conducted in the scope of this dissertation, therefore the readers are strongly encouraged to review it carefully for a full understanding of the system requirements.

3.2 Interface Control Document

An Interface Control Document (ICD) defines the internal interfaces of the monitoring dashboard within the LISA Pipeline Runner architecture. It describes how Prometheus, Grafana,

Thanos (Thanos), Loki (Loki) and related Kubernetes (Kubernetes) components interact to enable reliable metric and log collection, long-term metric storage and observability of workflow execution. The ICD ensures that all interface configurations, naming conventions, data flows and responsibilities are documented and maintained under version control according to ECSS-E-ST-10-24C (2024) and ECSS-E-ST-40C (2009).

By formally specifying these interfaces, the ICD provides a foundation for consistent development, improvement and verification of the Pipeline Runner monitoring dashboard. It reduces ambiguity between development teams and supports traceability of metric data throughout the system lifecycle.

As shown in Figure 3.2, the product tree offers a hierarchical view of the LISA Pipeline Runner system, with a focus on the monitoring subsystem. Besides the main tools used in the development of the monitoring subsystem (Prometheus, Grafana, Thanos, Loki), it includes several supporting components (node_exporter, kube-state-metrics, promtail, etc.) of these tools that enable the monitoring stack's core functions.

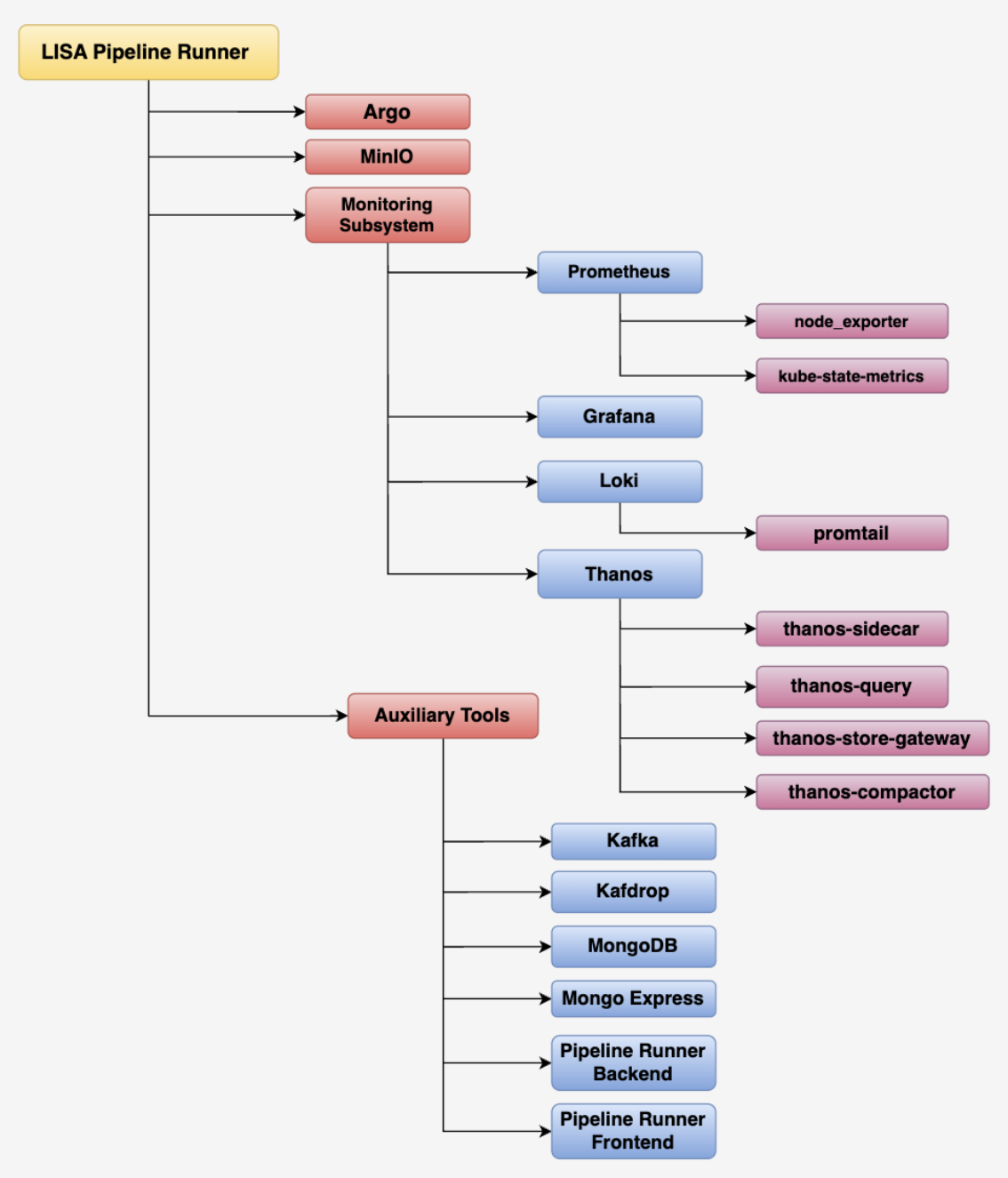


Figure 3.2: LISA Pipeline Runner monitoring dashboard product tree

Due to the strict structural and formatting constraints imposed by the ECSS standards, this document could not be incorporated directly into the main body of the thesis. Instead, it is provided in Appendix B. This document captures a substantial portion of the research and development performed in this thesis and since it defines the structure and interface responsibilities, readers are strongly encouraged to read it for a comprehensive understanding of this thesis.

3.3 System architecture

The monitoring dashboard system is a part of the Pipeline Runner Kubernetes cluster, the architecture of which can be seen in Figure 3.3. At its core, the monitoring dashboard includes four key components: Prometheus, Grafana, Thanos, and Loki. These tools work together to provide a comprehensive metric overview and system health for both infrastructure services and the execution of Argo Workflows.

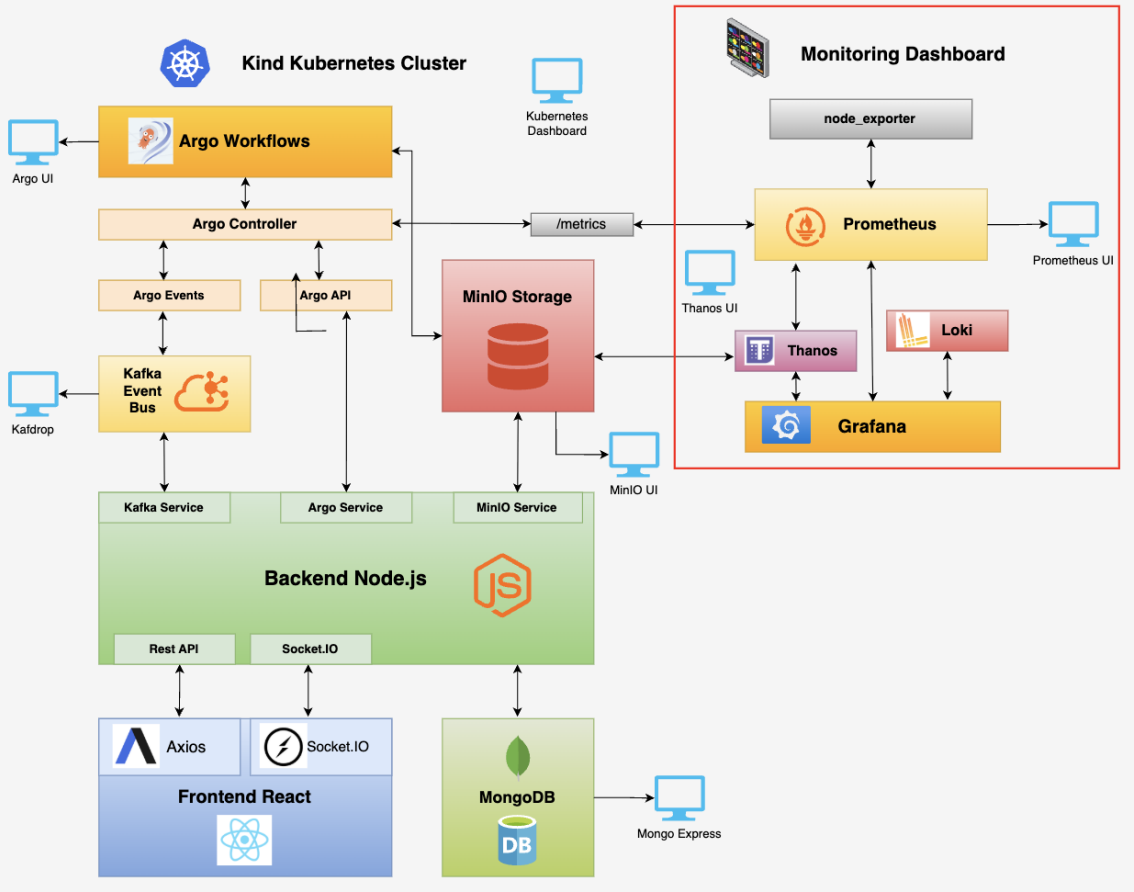


Figure 3.3: Diagram of the LISA Pipeline Runner architecture, with the architecture of the monitoring dashboard outlined in red

Rationale for technology choices. The chosen monitoring stack, composed of Prometheus, Grafana, Thanos, and Loki was selected for its alignment with the operational needs of the LISA DDPC system. These technologies are widely adopted, open-source, and support self-hosted deployments, which allow full configurability and long-term maintainability. Most importantly, the CNES system team's prior experience with Prometheus and Grafana was a key factor, as it supports faster onboarding, shorter learning curve and maintainability beyond the scope of this thesis.

Prometheus acts as the backbone of the monitoring system, handling the collection of time-series metrics from multiple services across the Pipeline Runner infrastructure. Its built-in support for dynamic service discovery makes it especially effective in Kubernetes environments.

Prometheus gathers metrics from infrastructure components as well as from Argo workflows through its metrics endpoint. One of its key strengths is flexibility, it can easily be extended in the future to capture custom metrics, including domain-specific data that may become important for LISA's scientific goals. Prometheus stores data locally and only for a limited period of time, typically around 15 days. To support longer-term storage and allow access to historical trends, Thanos was integrated into the stack. Thanos allows Prometheus metrics to be exported to an external storage bucket and queried, even across multiple Prometheus instances. Together, Prometheus and Thanos make it feasible to monitor the system both in real time and over extended periods, with respective web interfaces that allow easy exploration of all collected metrics.

Grafana complements Prometheus by offering a powerful and intuitive interface for visualizing collected metrics. It supports a wide variety of visualization options, such as graphs, gauges, and tables, and integrates seamlessly not only with Prometheus, but also with multiple other data sources. This extensibility makes Grafana well-suited for building dashboards that can evolve as system requirements grow. Whether it's adding new metrics or incorporating domain-specific views relevant to the LISA mission, Grafana allows these changes to be made with minimal disruption to the existing system. Its web-based interface enables users, even without prior programming experience, to create and adjust dashboards, while still offering advanced customization options for technical users. This combination of usability and flexibility makes Grafana an ideal long-term solution for monitoring complex systems like the Pipeline Runner.

To complete the stack, Loki was integrated to provide centralized log aggregation across all infrastructure components. Developed by Grafana Labs, Loki offers native integration with Grafana, allowing for seamless correlation between logs and metrics within a unified interface. This integration enhances system visibility, supports effective debugging, and improves traceability by allowing users to explore logs in context with related performance metrics, which is crucial for identifying and resolving issues quickly.

Together, this monitoring stack fulfills all the mandatory functional and non-functional requirements outlined in Appendix A, providing a scalable, maintainable, and extensible infrastructure for monitoring the health and performance of the Pipeline Runner system.

3.4 Data acquisition

Prometheus collects metrics using a pull-based model over HTTP, meaning it actively sends GET requests to defined target endpoints at regular intervals to retrieve data. Prometheus serves as the primary metrics collector. It continuously scrapes time-series data from a wide range of targets, including infrastructure services (such as node exporters and Kubernetes components) and Argo Workflows, which expose their metrics on a standardized metrics HTTP endpoint. These scrapes occur at regular intervals, typically every 15 seconds, which can be configured otherwise, and the collected data is stored in Prometheus's local time-series database.

Due to Prometheus's limitations in retaining data over extended periods of time, Thanos was configured to export historical metrics to a dedicated MinIO bucket while still making them accessible for querying. Thanos retrieves historical metric data from a MinIO bucket through its `thanos-store-gateway` component. When Prometheus finishes writing a block of time-series data, the `thanos-sidecar` uploads it to MinIO in the Prometheus TSDB block format. The `thanos-store-gateway` connects to MinIO using S3-compatible credentials and continuously scans the bucket for available blocks. It loads metadata about each block into memory, allowing it to efficiently index and return relevant data. When a query is received that targets a time range or label set covered by stored blocks, the `thanos-store-gateway` streams the appropriate chunks directly from MinIO and returns them to the `thanos-query` node which returns it to Grafana.

Chapter 4

Monitoring Dashboard

This Chapter presents the results obtained from implementing and evaluating the Pipeline Runner monitoring dashboard infrastructure described in the previous Chapters. It demonstrates how the system performs in a real-world Kubernetes environment, highlighting its ability to collect, store and visualize metrics and logs across multiple components. The results validate the integration of Prometheus, Thanos, Loki, and Grafana, and illustrate how each tool contributes to resource monitoring. Additionally, this section examines workflow behavior under different error scenarios, analyzes resource usage, and confirms that the defined goals were effectively achieved.

4.1 Tool results

When the infrastructure cluster is started and the user logs into Grafana for the first time, they are redirected to the Dashboards page, where the four primary dashboards are visible (See Figure 4.1). When a user executes workflows through the Pipeline Runner UI, the corresponding metric data is automatically scraped by Prometheus and subsequently visualized in Grafana dashboards, providing real-time insights into workflow execution and system performance.

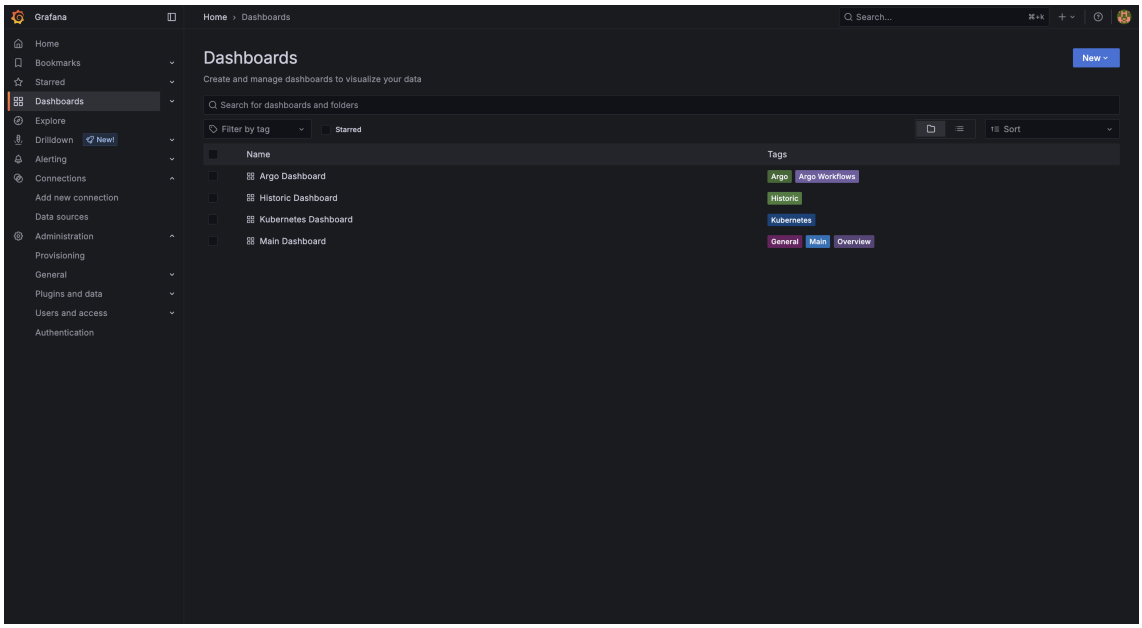


Figure 4.1: Initial dashboards page in Grafana

When users navigate to the “Main Dashboard,” they are presented with a high-level overview of the system’s operational status. This dashboard serves as the primary entry point for an immediate visibility into system health while minimizing unnecessary complexity. It includes an organized set of panels, such as tables listing completed workflows, visualizations of pod counts, node counts, currently running workflows, a table of workflow authors and a filtered log summary that shows only critical messages with levels "error" or "fatal" (see Figures 4.2 and 4.3).

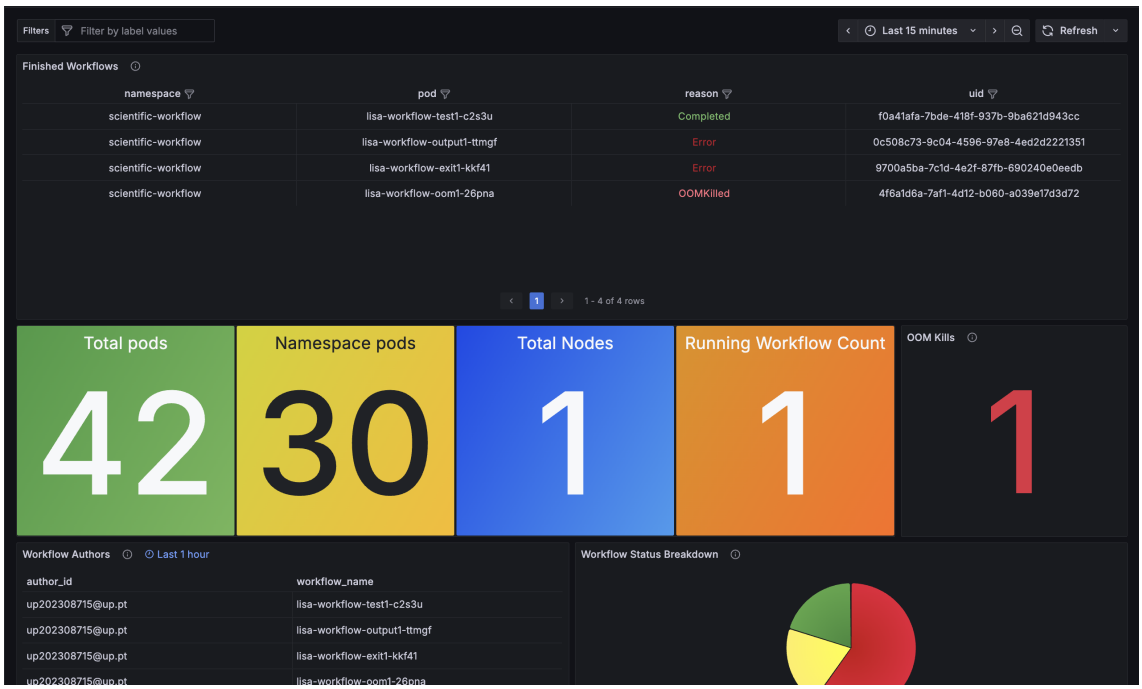


Figure 4.2: Main Dashboard initial view

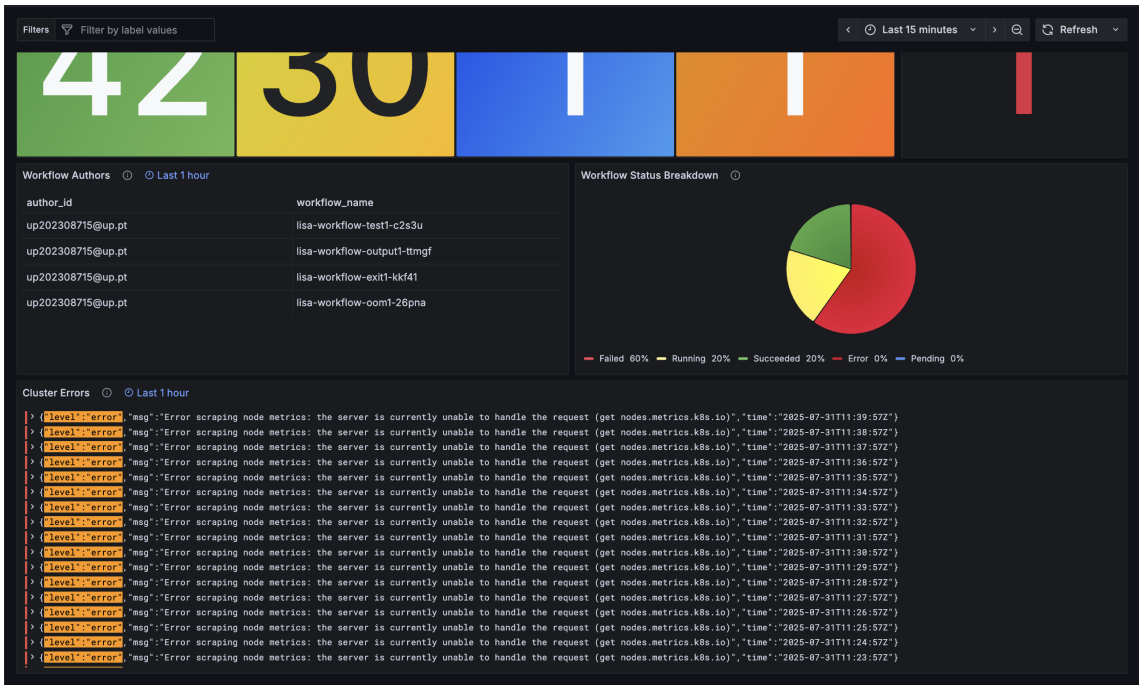


Figure 4.3: Main Dashboard initial view scrolled down

The design prioritizes the display of essential real-time information to support quick troubleshooting and analysis. For example, the workflow status pie chart offers a clear display of the success-to-failure ratio of recent workflows, allowing to detect abnormal system behavior early. Real-time counters provide additional context, such as the total number of pods, the scale of the infrastructure, and the current number of running workflows, which allows the user to assess resource utilization.

The dashboard was deliberately structured to avoid visual clutter, instead focusing on key metrics that provide a precise view across multiple operational domains, including workflow statuses, system capacity and failure visibility. This approach enhances the users' awareness and reduces the cognitive load. The Main Dashboard is a central component in maintaining operational stability and responding quickly to issues as they arise.

For users who need in-depth insights into workflow execution, the "Argo Dashboard" offers a more focused and detailed view customized specifically to Argo workflows. While the Main Dashboard provides a general overview of the system health, the Argo Dashboard is designed to help users monitor, analyze and troubleshoot workflows at a more deeper level.

At the top of the dashboard (see Figure 4.4), a high-level summary displays key information such as the number of workflows currently running, the distribution of workflow statuses, workflow authors, and the version of the Argo build instance currently in use. This setup allows users to quickly understand what's happening in the system regarding workflows, who initiated the activity and whether the correct version of Argo is deployed which is an important detail when ensuring consistency and compatibility. Overall, the Argo Dashboard supports more informed debugging and operational decisions by displaying the workflow-specific metrics that matter the most.

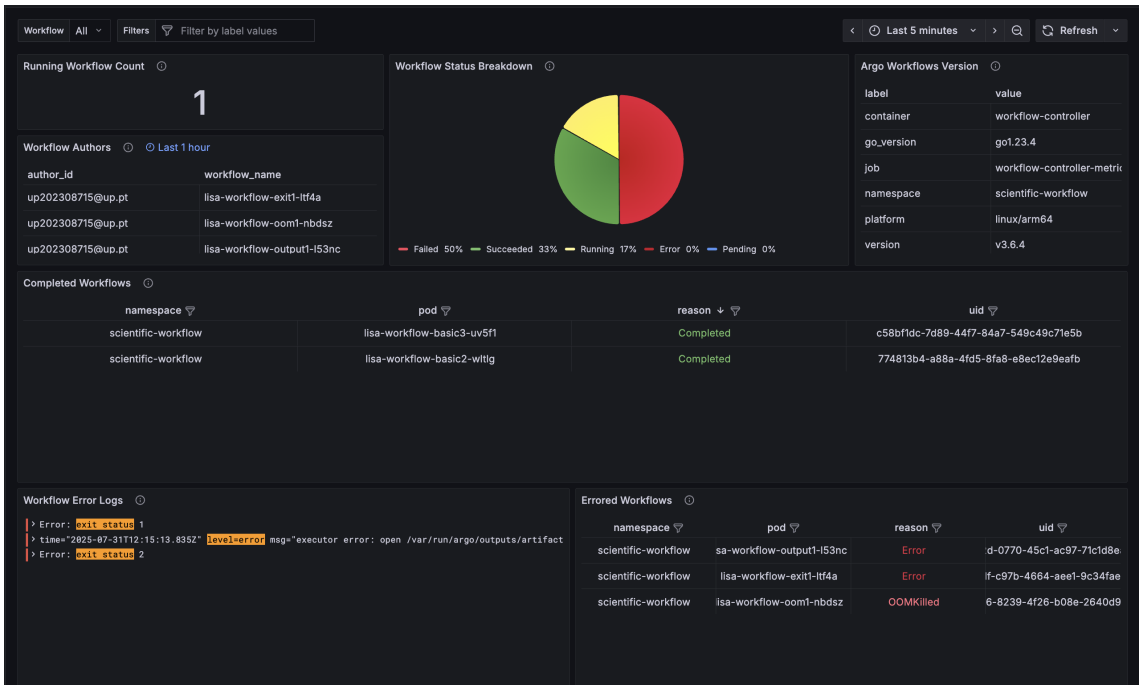


Figure 4.4: Argo Dashboard initial view

Below this overview, the dashboard presents other detailed metrics, such as pod uptime and execution logs for each individual pod involved in a workflow. These details are essential for diagnosing performance issues, understanding execution timing, and tracking pod-level behavior across the system. Many of the visualizations are replicated per workflow pod, ensuring that each workflow instance has its own dedicated set of metric panels. This approach isolates data for each pod and provides individual clarity, making it easier to quickly pinpoint resource consumption of individual workflows. As new workflows are launched, the dashboard dynamically expands to include corresponding graphs and panels for the newly created pods. This behavior allows users to maintain continuous, real-time visibility into workflow activity as it progresses, without the need to manually update the dashboard layout (see Figure 4.5). It allows operators to monitor workflow execution in a flexible and scalable way, which is especially valuable in environments where workflows are frequently initiated.

Figure 4.5: Argo Dashboard panels for workflow pod uptime and logs

As the user scrolls further down the Argo Dashboard, detailed resource consumption metrics become visible, including graphs for CPU and RAM usage, network I/O, and disk I/O activity (see Figures 4.6, 4.7, and 4.8). These metrics are critical for identifying performance restrictions, analyzing workload efficiency, and guaranteeing the optimal use of system resources during workflow execution. To make this data more immediately interpretable, CPU and RAM usage are also visualized using percentage-based gauges, which are repeated individually for each running workflow. This per-workflow breakdown allows users to quickly assess whether specific workflows are operating within acceptable resource boundaries or exceeding the predefined thresholds. Displaying usage as a percentage makes it easier to compare consumption relative to allocated limits, rather than relying only on raw consumption values.

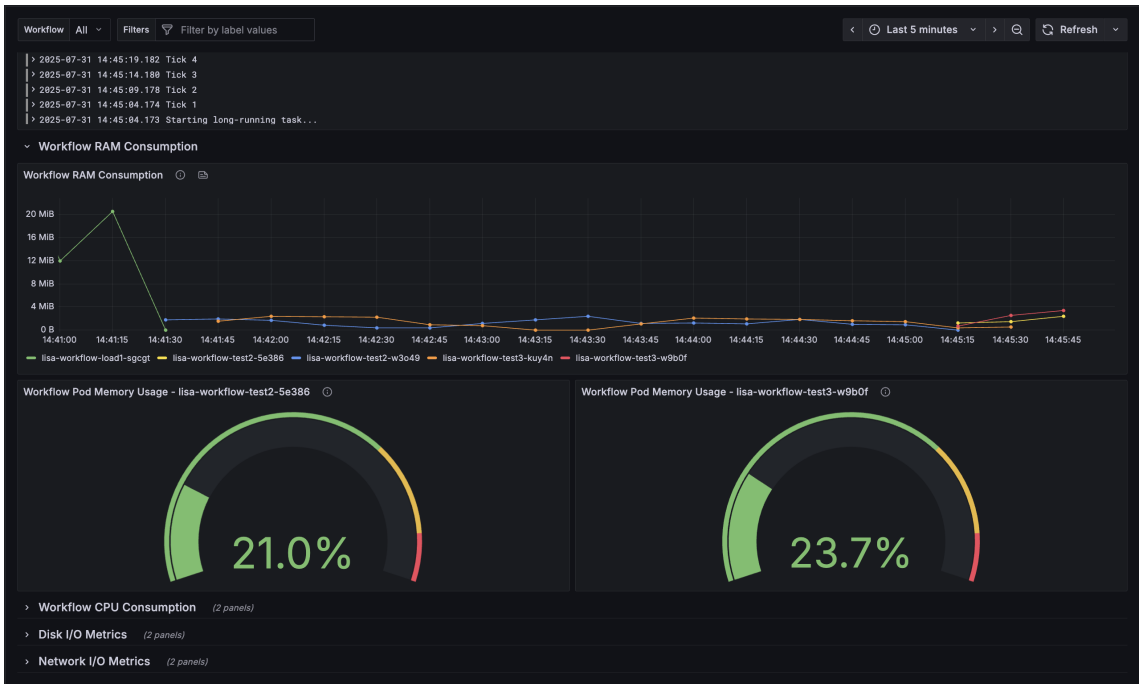


Figure 4.6: Argo Dashboard RAM consumption metrics

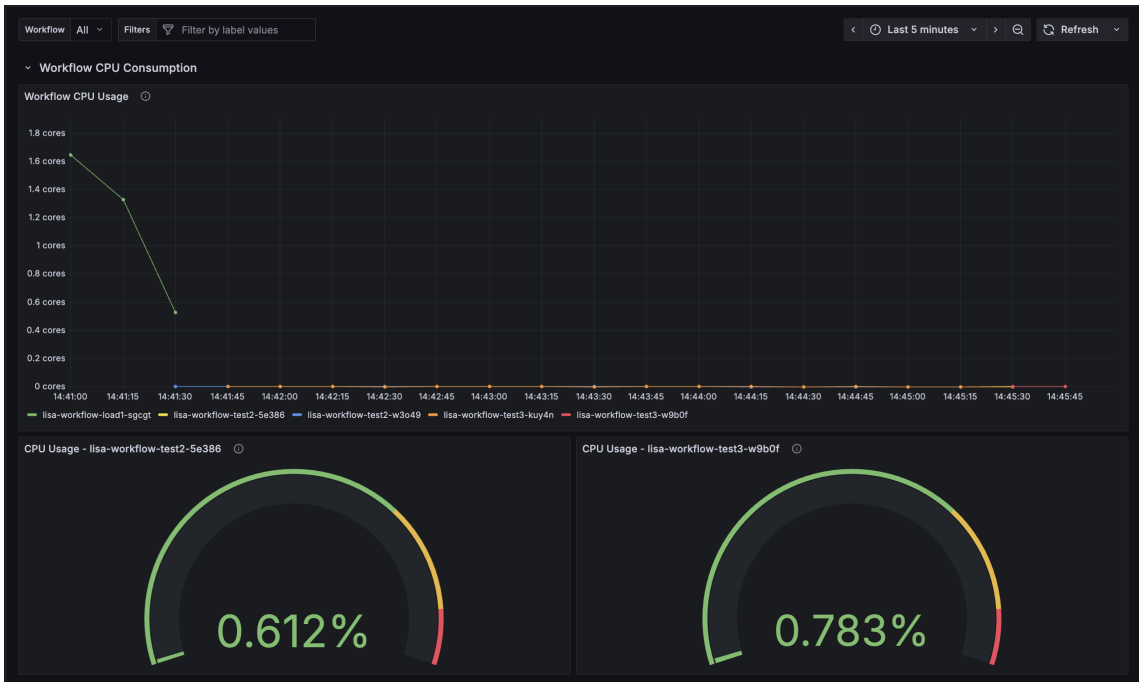


Figure 4.7: Argo Dashboard CPU consumption metrics

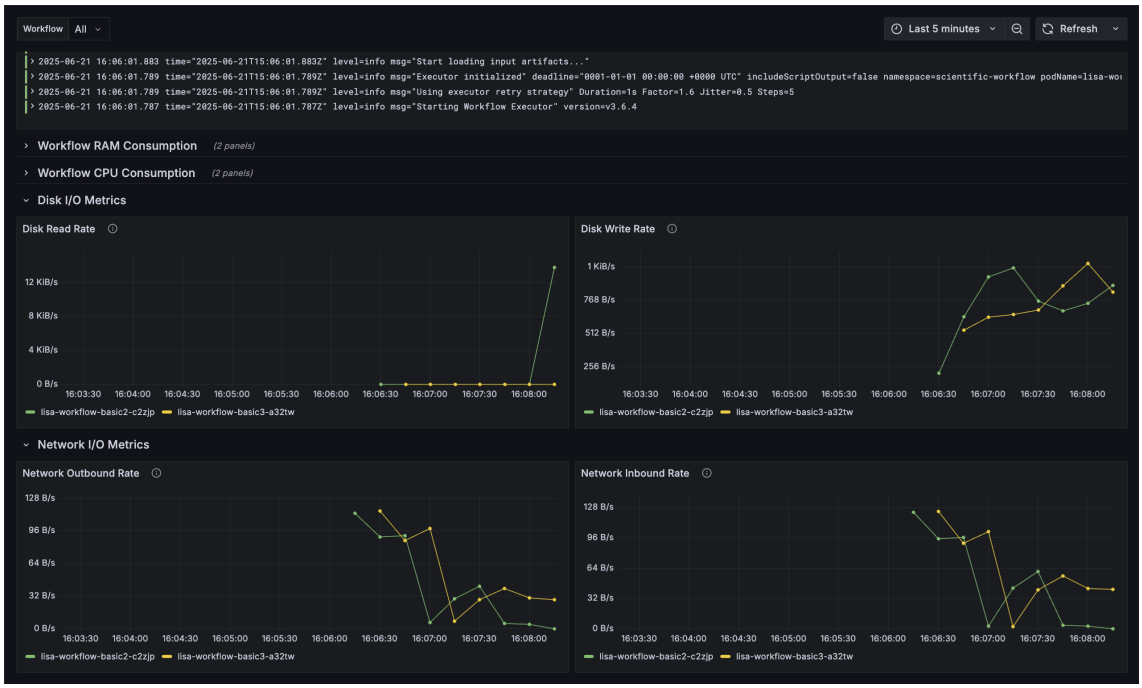


Figure 4.8: Argo Dashboard disk I/O and network I/O consumption metrics

For this percentage calculation to be relevant, the resource limits for each workflow must be either explicitly defined in the workflow's YAML template, by setting a predefined value to the Argo deployment configuration, or by specifying it in the Pipeline Runner source code during workflow creation. Defining these limits not only permits accurate gauge visualizations but also enforces resource boundaries, helping to prevent an overload of the infrastructure. By combining visual feedback with proper configuration, the dashboard helps ensure that resources are used efficiently and that the system remains stable.

In addition to the Argo Dashboard, the "Kubernetes Dashboard" serves as a key resource for system engineers responsible for maintaining the infrastructure cluster. Upon navigating to this dashboard, users are presented with a system-wide overview of resource consumption. It includes graphs for CPU, RAM, and disk usage, as well as information about total allocated resources and system uptime. These metrics provide a clear understanding of the cluster's current load and operational stability; they include not only Argo-related metrics, but also the entire cluster's resource consumption, which allows engineers to proactively detect performance or capacity issues.

Further down the dashboard, node-specific metrics are available and can be filtered using a node selector at the top. In this particular deployment, a single node named `scientific-workflow-control-plane` is available. These node-level insights are essential for identifying areas of high resource usage and ensuring that workloads are efficiently distributed across the available infrastructure.

Additionally, the dashboard includes statistical panels that summarize pod counts, total node count, and the number of active workflows, which are indicators of workload distribution and system activity. Just below that, bar charts illustrate the connection between requested resources,

predefined limits and actual resource usage. This level of visibility helps infrastructure engineers adjust resource definitions accordingly. By aligning infrastructure monitoring with real-time usage data, the Kubernetes Dashboard supports informed scaling decisions and contributes to maintaining overall cluster efficiency (see Figure 4.9).

Figure 4.9: Kubernetes Dashboard initial view

As the user continues scrolling, time series graphs display resource utilization trends over time, providing a clear view of cluster-level behavior (see Figures 4.10 and 4.11). These short historical trends are especially valuable for identifying patterns, such as periodic spikes in usage or gradual increases in recent loads, which may not be obvious from real-time metrics alone. By analyzing these trends, engineers can better anticipate future resource demands and make more informed decisions about scaling and optimization in the cluster's compute performance.

Figure 4.10: Kubernetes Dashboard node-level resource consumption and allocation

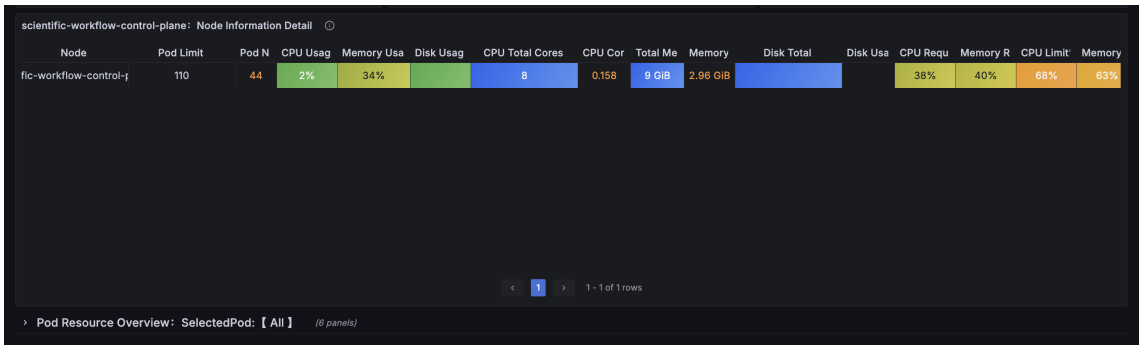


Figure 4.11: Kubernetes Dashboard node-level metrics panel

Finally, at the bottom of the Kubernetes Dashboard, detailed pod-level metrics are available. These metrics cover both workflow-related pods and core infrastructure pods, providing a combined view of activity across the entire cluster. By default, the dashboard displays data for all pods, giving users a comparitive view on overall resource consumption. However, to support more targeted analysis, users can filter the view by selecting a specific pod using the selector at the top of the dashboard. Once a pod is selected, the displayed metrics dynamically update to reflect only that pod's performance and resource usage. This level of detail is especially useful for debugging individual workflows, monitoring the behavior of critical services, or validating whether specific pods are staying within their defined resource limits. It enables infrastructure engineers to isolate problems quickly, correlate pod behavior with system events, and make adjustments based on actual usage patterns (see Figure 4.12).

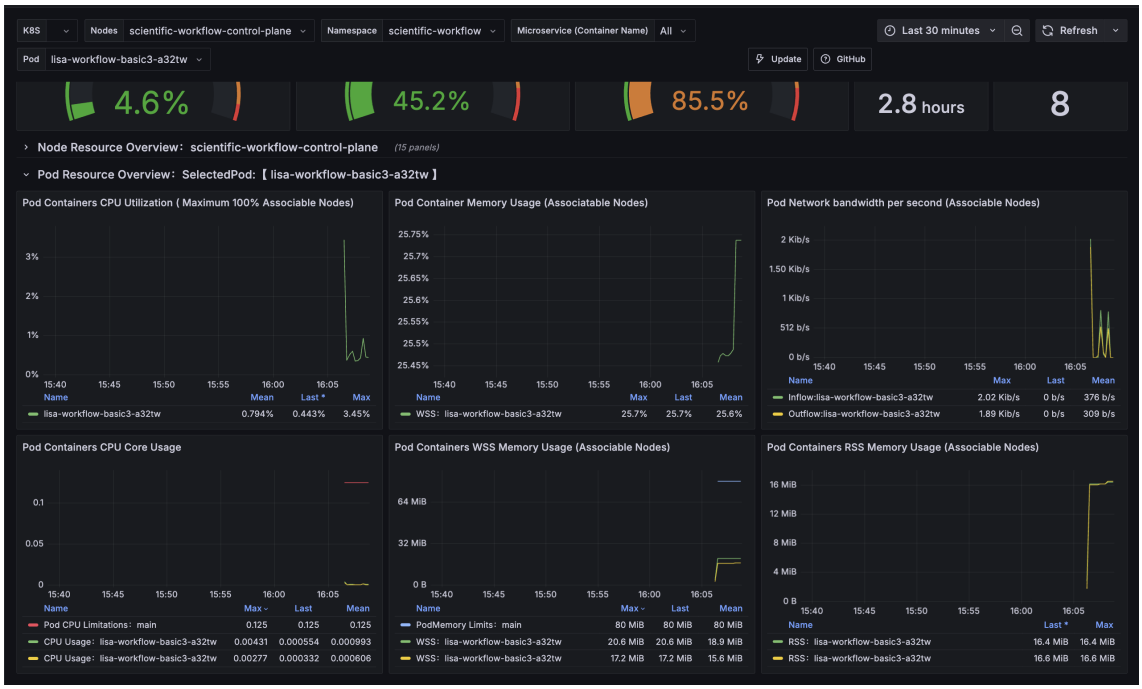


Figure 4.12: Kubernetes Dashboard pod metrics

Many of the dashboards described above are designed for real-time monitoring or short-term historical views, which are important for a quick response to incidents. However, when it comes to identifying recurring issues or understanding long-term behavior, access to historical data becomes necessary. This is where the “Historical Dashboard” plays a critical role.

The Historical Dashboard is specifically designed to provide long-term visibility into resource consumption metrics across the system. It uses Thanos as its data source, which enables storage and querying of Prometheus metrics well beyond the default retention period. This allows users to explore historical trends in CPU, RAM, network, and disk I/O usage over extended periods of time. Such visibility is crucial for detecting slow-developing performance regressions, correlating incidents with past workload spikes and analyzing areas where additional resources may be needed. By default, the dashboard displays data from the past 12 hours, but users can adjust the time range using a selector in the top-right corner to view data spanning days, weeks, or even months (see Figure 4.13). This flexibility supports a wide range of analytical use cases, accompanying the real-time dashboards and contributing to a fully integrated observability solution. An example of the Historical Dashboard with expanded panels for CPU and RAM consumption can be seen in Figure 4.14.

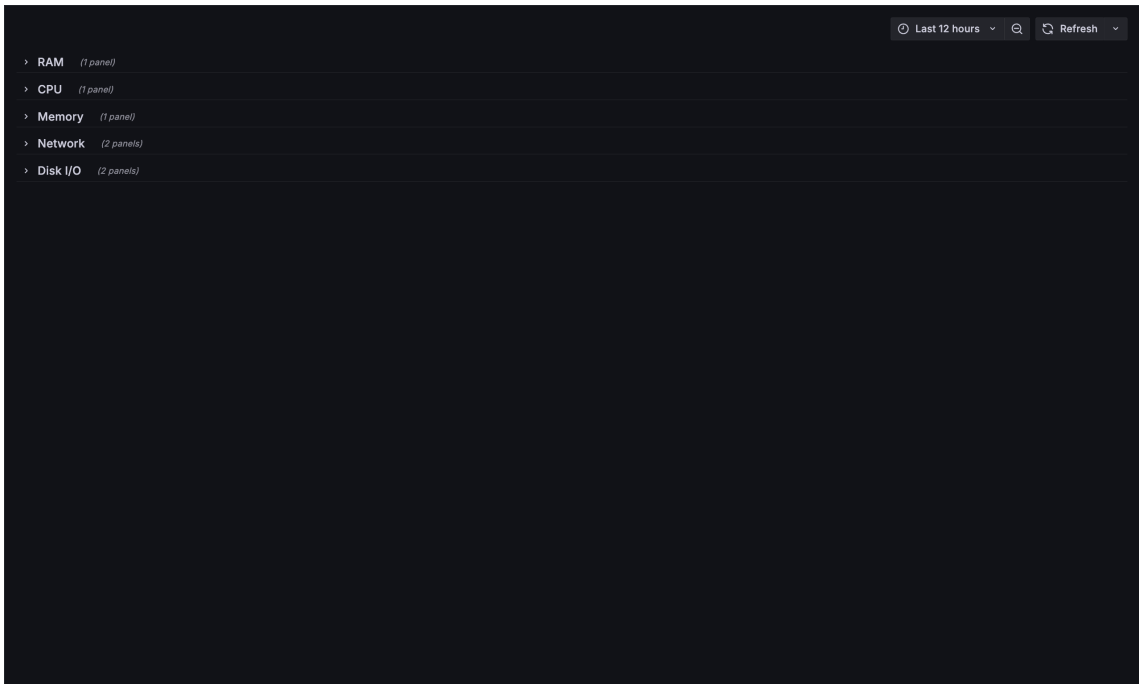


Figure 4.13: Historical Dashboard initial view with collapsed panels

Figure 4.14: Historical Dashboard with expanded RAM and CPU panels

The resulting monitoring dashboard is highly extensible and well-suited for future expansion due to Grafana's ability to integrate with a wide variety of data sources. This flexibility allows the dashboard to evolve alongside the system as new requirements are developed, without needing major reconfigurations. Grafana supports connections to numerous data sources which makes

it straightforward to incorporate additional metrics or monitoring data over time. Furthermore, the Pipeline Runner codebase can be modified to expose custom metrics or system-specific data, which can then be visualized directly within the Grafana interface. This design ensures that the monitoring solution is not only effective in its current form, but also capable of adapting to the growing complexity of the system in a scalable and maintainable way.

All dashboards were developed through a combination of individual effort and community-driven resources. Each dashboard was built by carefully reviewing the documentation for the relevant metrics to ensure accurate and meaningful representation. In addition, inspiration was drawn from existing dashboards available on Grafana's official dashboard repository (Grafana Labs), with particular inspiration from the Node Exporter and Kubernetes dashboards, which were especially helpful for understanding effective layout choices and selecting the most relevant resource consumption metrics to display.

4.2 Evaluation

Dashboard Evaluation

In order to effectively evaluate the monitoring dashboard, different types of workflows were executed, each designed to represent different scenarios and configurations. In Argo, workflows are defined using YAML templates, which allow for dynamic customization through the use of arguments, resource limits, and embedded metrics.

For evaluation of this dashboard, five distinct YAML workflow templates were used, each designed to simulate a specific scenario:

1. **OOMKill Workflow** – A template intentionally configured to exceed its memory limit, causing Kubernetes to terminate the pod due to an Out-Of-Memory (OOM) condition.

2. **Missing Output Path Workflow** – A template that omits the required output path parameter, leading to execution failure.

3. **Error Exit Workflow** – A template in which the container is explicitly set to exit with a non-zero status code (`exit 1`), simulating a failed task.

4. **Long-Running Workflow** – A standard template representing normal execution, which runs for an extended period of time while logging an incrementing integer every second for a duration of approximately 5 minutes.

5. **Higher CPU and RAM Load** - A template that tries to execute logic which accounts for approximately 80%-90% of allocated CPU and RAM of the workflow pod.

These workflows were chosen to ensure the dashboard could accurately reflect a range of real-world scenarios, including both successful executions and common failures that can be encountered. Executing a variety of workflow templates allows to verify how different errors were

visualized on the dashboard, ensuring that they were clearly highlighted and easily noticeable to draw the user's attention. The workflow templates are attached as Appendix C of this thesis.

In Figure 4.15, the amount of OOMKilled pods is clearly displayed on the "Main Dashboard". They are counted and marked in red to draw the user's attention. Additionally, the panel above displays the pod name and namespace, with the OOMKilled reason colored in light red to visually distinguish it from other pod termination reasons categorized as "Error" but still attracting the attention of the user, signalizing that the workflow failed.

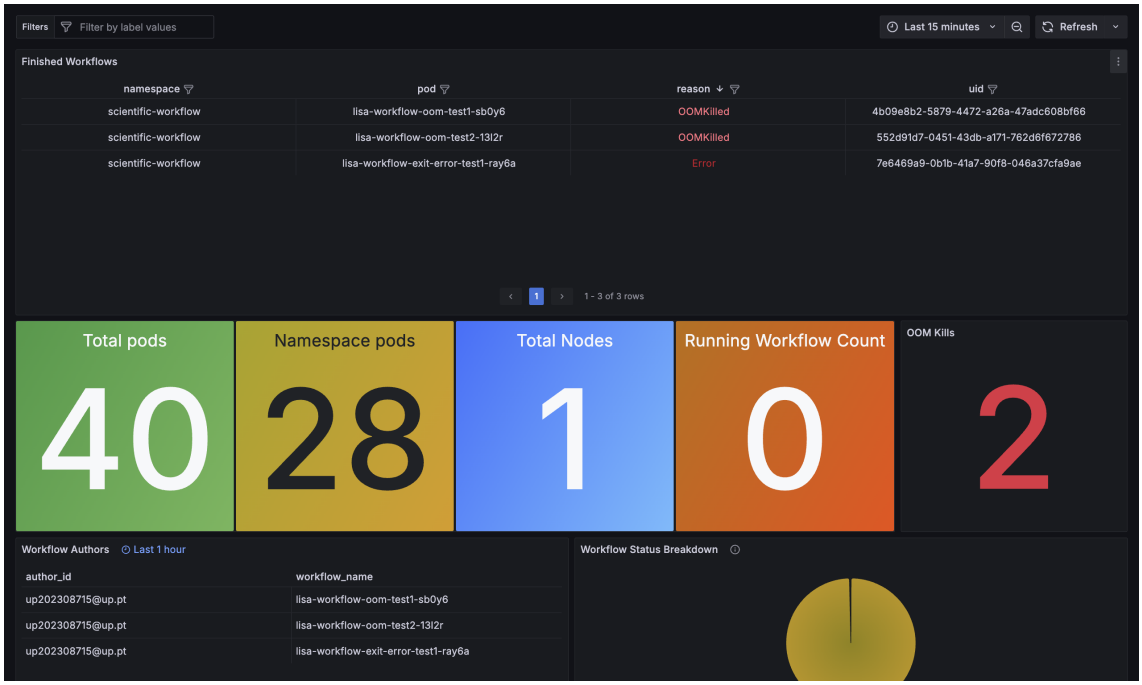


Figure 4.15: Main Dashboard showing OOMKilled pods

When executing the output-missing workflow, the container exit reason is reported simply as "Error". However, by examining the logs in the panel on the left inside the "Argo Dashboard", it becomes evident that the error was caused by a failure to open a specific directory, with the pod terminating with exit status = 2 (see Figure 4.16). Within the dashboard, these error panels support filtering, making it easier to correlate specific error messages with the pods that exited with an error status.

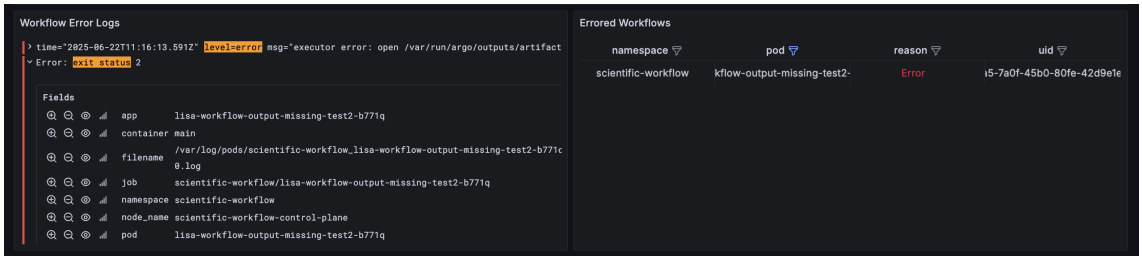


Figure 4.16: Argo Dashboard workflow error logs and errored workflows table filtered to show data for a specific workflow pod

In addition to analyzing workflow errors, it is also important to assess how the dashboard displays workflows under medium resource load. For this purpose, the medium-load workflow was used, which allocates 1 CPU core and approximately 572 megabytes of memory to a pod. The pod executes logic that consumes roughly 80–90% of the allocated resources. As shown in Figures 4.17 and 4.18, the gauge graphs are colored yellow to draw the user's attention to workflows approaching their resource limits. These gauges use threshold-based color coding: when resource usage exceeds 70%, the gauges turn yellow to draw the attention of the user, meanwhile when resource usage exceeds 90%, the gauges turn red, indicating that the workflow has entered a critical state and may require immediate action.

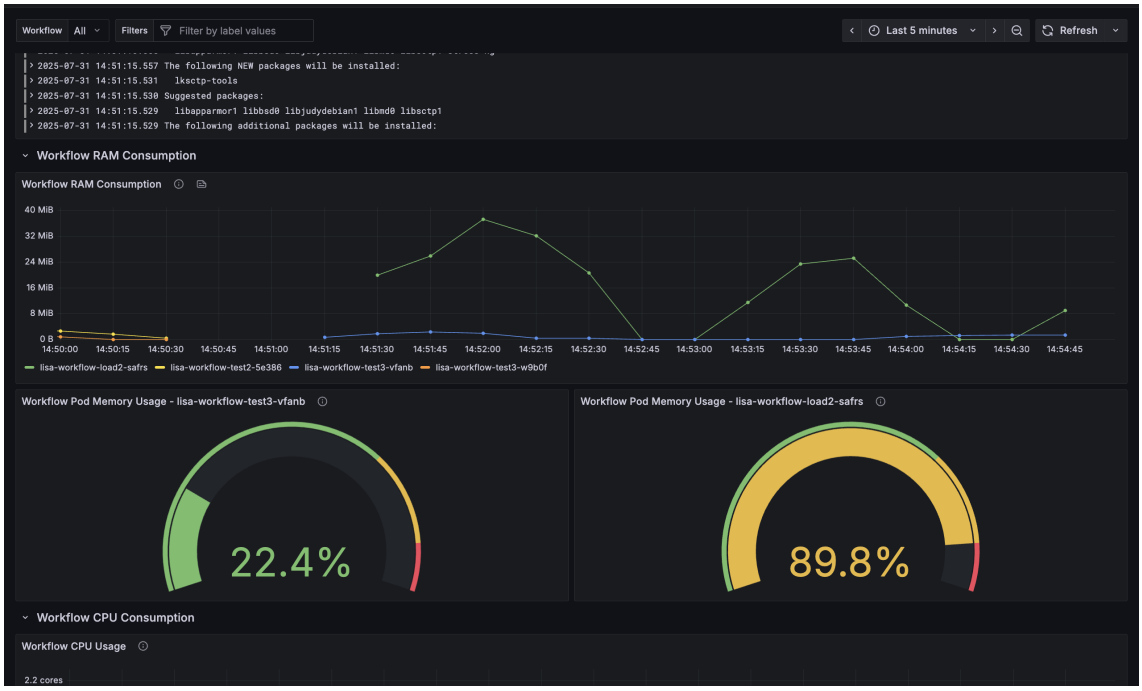


Figure 4.17: Gauge graph displaying RAM consumption of almost 90%

Figure 4.18: Gauge graph displaying CPU consumption of over 90%

In addition to evaluating negative scenarios such as high resource consumption and execution errors, it is equally important to assess how successful workflows are displayed. For this purpose, a regular workflow was selected that simply logs a value every second. These types of workflows are easily identifiable in both the Argo Dashboard and the Main Dashboard, as shown in Figures 4.19 and 4.20. The green-colored "Completed" status clearly indicates successful execution and is easy to interpret when overviewing the dashboards.

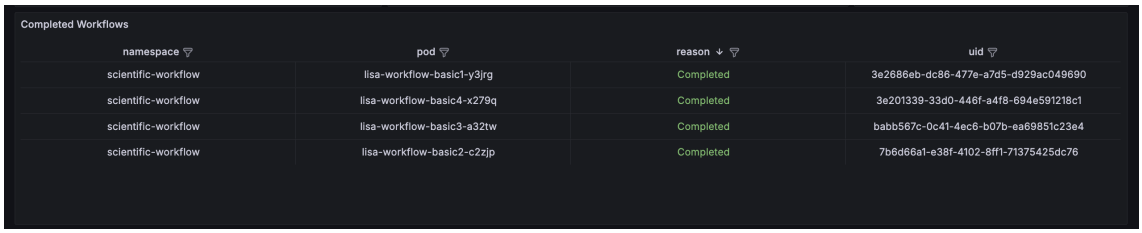


Figure 4.19: Completed workflows table in the "Argo Dashboard" which displays workflows that finished with the status "Completed", signifying success

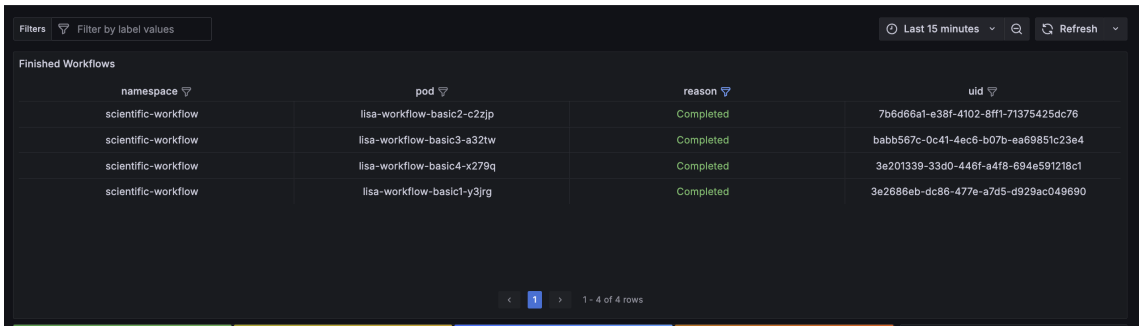


Figure 4.20: Finished workflows table located in the "Main Dashboard" which has the option to filter workflows based on the exit reason

Thanos Evaluation

To verify that Thanos can successfully retrieve historical data that is no longer available in Prometheus, a test was conducted on the historical dashboard. For demonstration purposes, the RAM consumption panel was duplicated, with its data source changed from Thanos to Prometheus and the time range set to the last 24 hours. As shown in Figure 4.21, the Prometheus panel does not display any data, whereas the Thanos panel successfully retrieves and displays data from the past 24 hours, confirming that historical queries are functioning as expected.

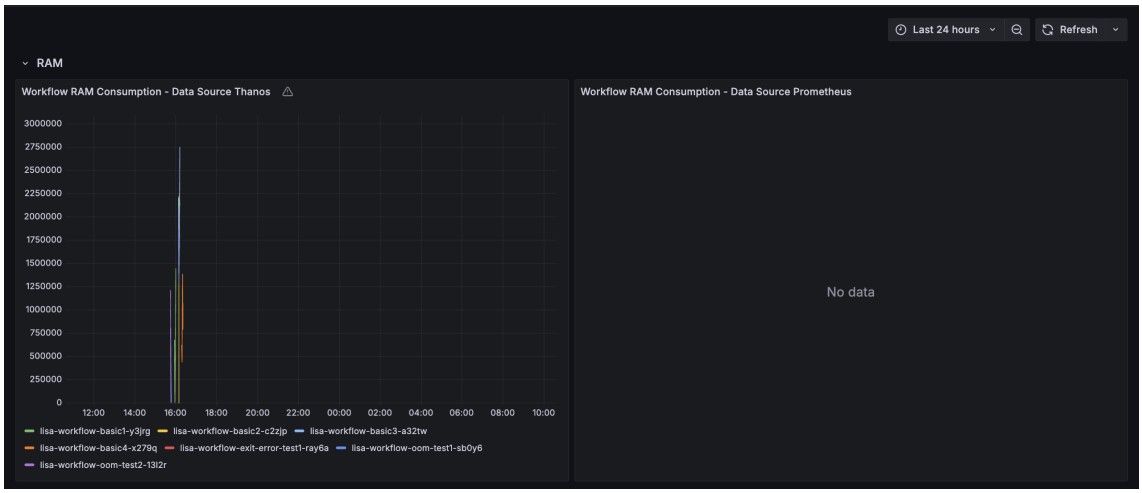


Figure 4.21: Historical Dashboard RAM panels showing the difference between using Thanos and Prometheus when querying data from the past 24 hours, proving that Thanos is successfully able to retrieve historical data

Chapter 5

Conclusions and future work

The Laser Interferometer Space Antenna (LISA) is a space mission led by the European Space Agency (ESA), aiming to detect and analyze gravitational waves originating from cosmic events. Given the complexity and precision of this mission, a distributed network of data computing centers is being developed to execute the scientific workflows and ensure accurate analysis of the collected data.

To support this infrastructure, a structured systems engineering approach was used to make sure the monitoring solution would be robust, easy to maintain, and in line with the CNES DDPC teams' expectations. The process started with gathering and refining requirements through regular biweekly and bilateral meetings with the CNES DDPC system team. These meetings ensured that the operational needs turned into clear functional and non-functional requirements. The outcome was a detailed Requirements Document that covers essential aspects like real-time and historical resource consumption monitoring, intuitive UI design and flexibility for future updates, all developed in line with ECSS standards. Additionally, an Interface Control Document (ICD) was created to clearly specify how the monitoring components interact with the Pipeline Runner system, including both the integration constraints and the functional features introduced by the monitoring dashboard. Together, these documents laid a solid foundation for a solution that fits both the technical needs and the long-term vision of the LISA data processing infrastructure.

Building upon these documents, the final monitoring solution was implemented using Prometheus and Grafana. These technologies were selected not only for their technical capabilities, such as support for dynamic service discovery and data visualizations, but also to meet with the CNES DDPC team's existing needs. This choice ensures a smoother onboarding process for the team and simplified future maintenance. The dashboards were designed to be modular and adaptable, capable of evolving with changing mission needs. Each one focuses on key operational areas while maintaining a clean and uncluttered visual layout that highlights essential metrics for different requirements. The architecture also allows for future integration of domain-specific data, making the system scalable as the LISA mission evolves. The configured monitoring infrastructure integrates Prometheus, Grafana, Thanos, and Loki into the Pipeline Runner Kubernetes cluster to deliver comprehensive visibility into system performance and workflow execution. Dedicated dashboards

were developed for monitoring Argo workflows, Kubernetes infrastructure, and historical consumption trends. Thanos ensures long-term storage and query capability for historical metrics, while Loki adds centralized log aggregation for error tracing. Together, these components create a monitoring system adapted to the operational and analytical demands of the LISA data processing pipeline.

The design and implementation of the dashboard required careful planning, structured requirements elicitation, and familiarity with established dashboard design patterns to ensure that the solution met both functional goals and the documentation standards defined by ECSS for space-related software systems. At this time, no other dashboard is as specifically tailored to the LISA Pipeline Runner, offering an intuitive, UI-driven configuration interface that allows users to create and update visualizations without requiring prior programming knowledge.

This work establishes a solid foundation for future extension and adaptation of the dashboard as requirements evolve. Its close integration with the Pipeline Runner system allows for flexible customization, enabling the display of any relevant data within a single, unified interface, eliminating the need for multiple dashboards or UIs to serve specific use cases.

Future Work

While the current version of the monitoring dashboard offers a solid foundation for observability in the LISA Pipeline Runner, there are several areas where the system can be extended and improved to better support real-world demands and future mission needs. A key enhancement would be the addition of an alerting system using Prometheus. Currently, the users can monitor metrics in real-time, but there are no automatic notifications configured in case of failures or errors. Setting up alerts on workflow errors or when resource usage exceeds predefined thresholds would allow for quicker responses from the operators and would support uninterrupted workflow execution, especially in production environments.

Another crucial improvement involves improving the infrastructure configuration for deployment on an actual HPC cluster. The dashboard was developed in a Kubernetes-based test environment, but moving it to a production-grade HPC environment requires ensuring that the system is correctly configured and successfully deployed within it. These optimizations are crucial for making sure the monitoring dashboard runs efficiently alongside other applications and is configured to collect metrics successfully.

To better understand how the monitoring setup performs under stress, load testing should be introduced. This would help identify any weak points or performance bottlenecks in the system before they become issues in a production environment.

Improving deployment through a CI/CD pipeline is another important enhancement. This would simplify the deployment process and help maintain consistent environments across development, testing, and production stages. Additionally, it would accelerate future updates and support zero-downtime deployments, allowing updates to be rolled out without interrupting ongoing monitoring operations. This would increase the reliability and maintainability of the system.

Finally, there's room to expand the dashboard's capabilities by exposing more custom metrics from the Pipeline Runner backend. For instance, domain-specific information or workflow-specific details could be integrated as new Grafana panels by exposing custom metrics to Prometheus through a custom service discovery. This would allow teams to go beyond basic infrastructure monitoring, making the dashboard not just a technical tool for system health visualization, but one that consolidates information to support deeper analysis of the system state.

Together, these features would help ensure that the dashboard remains scalable, maintainable, and aligned with the evolving needs of the mission as it moves from development to deployment.

References

Ecss-e-st-10-24c rev.1: System engineering - interface management. Technical Report ECSS-E-ST-10-24C Rev.1, European Cooperation for Space Standardization (ECSS), November 2024. Available at: https://ecss.nl/wp-content/uploads/2024/12/ECSS-E-ST-10-24C-Rev.1(15November2024).pdf.

Future Generation Computer Systems, 164:107541, March 2025. ISSN 0167-739X. doi: 10.1016/j.future.2024.107541.

Ryan Adamson, Tim Osborne, Corwin Lester, and Rachel Palumbo. Stream: A scalable federated hpc telemetry platform.

Deepak Aggarwal, Hemant Joshi, and Someswar Dutta. Advancements in high performance computing cluster resource utilization through a comprehensive monitoring dashboard. In *2024 11th International Conference on Computing for Sustainable Global Development (INDIACom)*, page 158–165, February 2024. doi: 10.23919/INDIACom61295.2024.10498826. URL https://ieeexplore.ieee.org/document/10498826/?arnumber=10498826.

Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J. Leung, Jim Brandt, Brian Kulis, Manuel Egele, and Ayse K. Coskun. Prodigy: Towards unsupervised anomaly detection in production hpc systems. In *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 01–14, November 2023. doi: 10.1145/3581784.3607076. URL https://ieeexplore.ieee.org/document/10485048.

William (Bill) Allcock, Evan Felix, Mike Lowe, Randal Rheinheimer, and Joshi Fullop. Challenges of hpc monitoring. In *State of the Practice Reports*, SC '11, page 1–6, New York, NY, USA, November 2011. Association for Computing Machinery. ISBN 978-1-4503-1139-7. doi: 10.1145/2063348.2063378. URL https://dl.acm.org/doi/10.1145/2063348.2063378.

Amazon Web Services. Amazon cloudwatch. URL https://aws.amazon.com/cloudwatch/. Accessed: 2025-01-10.

Apache Software Foundation. Apache chukwa. URL https://chukwa.apache.org/. Accessed: 2025-01-10.

Argo Project. Argo workflows documentation. URL https://argoproj.github.io/. Accessed: 2025-06-15.

Benjamin Bach, Euan Freeman, Alfie Abdul-Rahman, Cagatay Turkay, Saiful Khan, Yulei Fan, and Min Chen. Dashboard design patterns. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):342–352, January 2023. ISSN 1941-0506. doi: 10.1109/TVCG.2022.3209448.

Andrea Borghesi, Martin Molan, Michela Milano, and Andrea Bartolini. Anomaly detection and anticipation in high performance computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):739–750, April 2022. ISSN 1558-2183. doi: 10.1109/TPDS.2021.3082802.

Chansup Byun, Julia Mullen, Albert Iwersen Reuther, William Arcand, William Bergeron, David Bestor, Daniel Burrill, Vijay Gadepally, Michael Houle, Matthew Hubbell, Hayden Jananthan, Michael Jones, Peter Michaleas, Guillermo Morales, Andrew Prout, Antonio Rosa, Charles Yee, Jeremy Kepner, and Lauren Milechin. Llload: Simplifying real-time job monitoring for hpc users. In *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, PEARC '24, page 1–4, New York, NY, USA, July 2024. Association for Computing Machinery. ISBN 9798400704192. doi: 10.1145/3626203.3670565. URL https://dl.acm.org/doi/10.1145/3626203.3670565.

Cacti. Cacti. URL https://www.cacti.net/. Accessed: 2025-01-10.

Alberto Cascajo, David E. Singh, and Jesus Carretero. Limitless — light-weight monitoring tool for large scale systems. *Microprocessors and Microsystems*, 93:104586, September 2022. ISSN 01419331. doi: 10.1016/j.micpro.2022.104586.

Google Cloud. Google cloud monitoring. URL https://cloud.google.com/monitoring. Accessed: 2025-01-10.

collectd. collectd. URL https://www.collectd.org/. Accessed: 2025-01-10.

Monica Colpi, Karsten Danzmann, Martin Hewitson, Kelly Holley-Bockelmann, Philippe Jetzer, Gijs Nelemans, Antoine Petiteau, David Shoemaker, Carlos Sopuerta, Robin Stebbins, Nial Tanvir, Henry Ward, William Joseph Weber, Ira Thorpe, Anna Daurskikh, Atul Deep, Ignacio Fernández Núñez, César García Marirrodriga, Martin Gehler, Jean-Philippe Halain, Oliver Jennrich, Uwe Lammers, Jonan Larrañaga, Maike Lieser, Nora Lützgendorf, Waldemar Martens, Linda Mondin, Ana Piris Niño, Pau Amaro-Seoane, Manuel Arca Sedda, Pierre Auclair, Stanislav Babak, Quentin Baghi, Vishal Baibhav, Tessa Baker, Jean-Baptiste Bayle, Christopher Berry, Emanuele Berti, Guillaume Boileau, Matteo Bonetti, Richard Brito, Riccardo Buscicchio, Gianluca Calcagni, Pedro R. Capelo, Chiara Caprini, Andrea Caputo, Eleonora Castelli, Hsin-Yu Chen, Xian Chen, Alvin Chua, Gareth Davies, Andrea Derdzinski, Valerie Fiona Domcke, Daniela Doneva, Irna Dvorkin, Jose María Ezquiaga, Jonathan Gair, Zoltan Haiman, Ian Harry, Olaf Hartwig, Aurelien Hees, Anna Heffernan, Sascha Husa, David Izquierdo, Nikolaos Karnesis, Antoine Klein, Valeriya Korol, Natalia Korsakova, Thomas Kupfer, Danny Laghi, Astrid Lamberts, Shane Larson, Maude Le Jeune, Marek Lewicki, Tyson Littenberg, Eric Madge, Alberto Mangiagli, Sylvain Marsat, Ivan Martin Vilchez, Andrea Maselli, Josh Mathews, Maarten van de Meent, Martina Muratore, Germano Nardini, Paolo Pani, Marco Peloso, Mauro Pieroni, Adam Pound, Hippolyte Quelquejay-Leclere, Angelo Ricciardone, Elena Maria Rossi, Andrea Sartirana, Etienne Savalle, Laura Sberna, Alberto Sesana, Deirdre Shoemaker, Jacob Slutsky, Thomas Sotiriou, Lorenzo Speri, Martin Staab, Danièle Steer, Nicola Tamanini, Gianmassimo Tasinato, Jesus Torrado, Alejandro Torres-Orjuela, Alexandre Toubiana, Michele Vallisneri, Alberto Vecchio, Marta Volonteri, Kent Yagi, and Lorenz Zwick. Lisa definition study report. (arXiv:2402.07571), February 2024. doi: 10.48550/arXiv.2402.07571. URL http://arxiv.org/abs/2402.07571. arXiv:2402.07571 [astro-ph].

Tommy Dang, Ngan V.T. Nguyen, Jie Li, Alan Sill, Jon Hass, and Yong Chen. Jobviewer: Graph-based visualization for monitoring high-performance computing system. In *2022 IEEE/ACM*

International Conference on Big Data Computing, Applications and Technologies (BDCAT), page 110–119, December 2022. doi: 10.1109/BDCAT56447.2022.00021. URL https://ieeexplore.ieee.org/abstract/document/10062112.

K. Danzmann and A. Rudiger. Lisa technology - concept, status, prospects. *Class. Quant. Grav.*, 20:S1–S9, 2003. doi: 10.1088/0264-9381/20/10/301.

Datadog. Datadog. URL https://www.datadoghq.com/. Accessed: 2025-01-10.

Dynatrace. Dynatrace. URL https://www.dynatrace.com/. Accessed: 2025-01-10.

ECSS. European Cooperation for Space Standardization (ECSS) Website, 2025. URL https://ecss.nl/. Accessed: 2025-02-18.

European Cooperation for Space Standardization (ECSS). Ecss-e-st-40c – space engineering. software, March 2009. URL https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/. Accessed: 2025-02-18.

European Cooperation for Space Standardization (ECSS). Ecss-d-00-01c rev.1 – drafting rules and template for ecss standards, July 2020. URL https://ecss.nl/home/ecss-d-00-01c-rev-1-drafting-rules-and-template-for-ecss-standards-1-july-202 Accessed: 2025-02-18.

European Space Agency. Lisa - laser interferometer space antenna, 2024. URL https://www.cosmos.esa.int/web/lisa. Accessed: 2024-10-23.

Grafana Labs. Grafana dashboards. URL https://grafana.com/grafana/dashboards/. Accessed: 2025-06-15.

Apache Hadoop. Hadoop performance monitoring. URL https://hadoop.apache.org/. Accessed: 2025-01-10.

Icinga. Monitor your entire infrastructure. URL https://icinga.com/. Accessed: 2025-01-10.

Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. Hpc i/o throughput bottleneck analysis with explainable local models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1–13, November 2020. doi: 10.1109/SC41405.2020.00037. URL https://ieeexplore.ieee.org/document/9355272/?arnumber=9355272.

Mohaiminul Islam and Shangzhu Jin. An overview of data visualization. In *2019 International Conference on Information Science and Communications Technologies (ICISCT)*, page 1–7, November 2019. doi: 10.1109/ICISCT47635.2019.9012031. URL https://ieeexplore.ieee.org/document/9012031.

Forschungszentrum Jülich. Llview. https://apps.fz-juelich.de/jsc/llview/docu/. Accessed: 2025-02-12.

Alexander Kossiakoff, Samuel J. Seymour, David A. Flanigan, and Steven M. Biemer. *Systems Engineering Principles and Practice*. Wiley, 3rd edition, 2020.

Kubernetes. Kubernetes: Production-grade container orchestration. https://kubernetes.io. Accessed: 2025-06-18.

Grafana Labs. Grafana. URL https://grafana.com/. Accessed: 2025-01-10.

Jie Li, Ghazanfar Ali, Ngan Nguyen, Jon Hass, Alan Sill, Tommy Dang, and Yong Chen. Monster: An out-of-the-box monitoring tool for high performance computing systems. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, page 119–129, September 2020. doi: 10.1109/CLUSTER49012.2020.00022. URL https://ieeexplore.ieee.org/abstract/document/9229641?casa_token=4kGHH1vALVwAAAAA:f8mECM2EdvagZrhWfwSB5TYzh3pNMOmGfZ7awVWAK4E2Zb5QfpyfY2QbmTn11arxpVxhXp_RcqAJ.

Jaelyn Litzinger, Roy Hallquist, and James Tessmer. Hpc monitoring visualization: Understanding usage of hpc systems. In *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, PEARC '23, page 453–456, New York, NY, USA, September 2023. Association for Computing Machinery. ISBN 978-1-4503-9985-2. doi: 10.1145/3569951.3597554. URL https://dl.acm.org/doi/10.1145/3569951.3597554.

LogicMonitor. Logicmonitor. URL https://www.logicmonitor.com/. Accessed: 2025-01-10.

Loki. Loki: Like prometheus, but for logs. https://grafana.com/oss/loki/. Accessed: 2025-06-18.

Miguel A. López-Peña, Jessica Díaz, Jorge E. Pérez, and Héctor Humanes. Devops for iot systems: Fast and continuous monitoring feedback of system availability. *IEEE Internet of Things Journal*, 7(10):10695–10707, October 2020. ISSN 2327-4662. doi: 10.1109/JIOT.2020.3012763.

Microsoft. Azure monitor. URL https://learn.microsoft.com/en-us/azure/azure-monitor/overview. Accessed: 2025-01-10.

Bashir Mohammed, Mariam Kiran, and Bjoern Enders. Netgraf: An end-to-end learning network monitoring service. In *2021 IEEE Workshop on Innovating the Network for Data-Intensive Science (INDIS)*, page 12–22, November 2021. doi: 10.1109/INDIS54524.2021.00007. URL https://ieeexplore.ieee.org/abstract/document/9652600.

MongoDB, Inc. Mongodb. https://www.mongodb.com/. Accessed: 2025-02-10.

Munin. Munin. URL https://munin-monitoring.org/. Accessed: 2025-01-10.

Nagios. Nagios. URL https://www.nagios.org/. Accessed: 2025-01-10.

Netdata. Netdata. URL https://www.netdata.cloud/. Accessed: 2025-01-10.

Alessio Netti, Michael Ott, Carla Guillen, Daniele Tafani, and Martin Schulz. Operational data analytics in practice: Experiences from design to deployment in production hpc environments. *Parallel Computing*, 113:102950, October 2022. ISSN 0167-8191. doi: 10.1016/j.parco.2022.102950.

N. Nguyen, T. Dang, J. Hass, and Y. Chen. HiperJobViz: Visualizing Resource Allocations in High-Performance Computing Center via Multivariate Health-Status Data. In *2019 IEEE/ACM Industry/University Joint International Workshop on Data-Center Automation, Analytics, and Control (DAAC)*, pages 19–24, 2019. doi: 10.1109/DAAC49578.2019.00009. URL https://doi.org/10.1109/DAAC49578.2019.00009.

Adel Noureddine. Powerjoular and joularjx: Multi-platform software power monitoring tools. In *2022 18th International Conference on Intelligent Environments (IE)*, page 1–4, June 2022. doi: 10.1109/IE54923.2022.9826760. URL https://ieeexplore.ieee.org/abstract/document/9826760?casa_token=8vSk8osunAgAAAAA:S0mz_unyHCRMugLiruQuRcyVoKphy4gk40gSbkAfTfccLM-nK5dMlDaUbOCRB5ekAsLoGYnuWTIs.

OpenStack. Openstack telemetry service. URL https://wiki.openstack.org/wiki/Telemetry. Accessed: 2025-01-10.

Opsview. Opsview. URL https://www.opsview.com/. Accessed: 2025-01-10.

Michael Ott, Woong Shin, Norman Bourassa, Torsten Wilde, Stefan Ceballos, Melissa Romanus, and Natalie Bates. Global experiences with hpc operational data measurement, collection and analysis. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, page 499–508, September 2020. doi: 10.1109/CLUSTER49012.2020.00071. URL https://ieeexplore.ieee.org/abstract/document/9229593?casa_token=TvRtnmoLVlkAAAAA:Rt2gJjphQeH42hgg_ms6WrWQCZ4LCc9N6EzlKsdMnTX2aZoFgN1XOf6A4lTmtfG5xIiq3i2h31Xa.

Lisa Pappas and Lisa Whitman. Riding the technology wave: Effective dashboard data visualization. In Michael J. Smith and Gavriel Salvendy, editors, *Human Interface and the Management of Information. Interacting with Information*, page 249–258, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-21793-7. doi: 10.1007/978-3-642-21793-7_29.

Prometheus. Prometheus. URL https://prometheus.io/. Accessed: 2025-01-10.

New Relic. New relic. URL https://newrelic.com/. Accessed: 2025-01-10.

S. Sanchez, A. Bonnie, G. Van Heule, C. Robinson, A. DeConinck, K. Kelly, Q. Snead, and J. Brandt. Design and implementation of a scalable hpc monitoring system. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, page 1721–1725, May 2016. doi: 10.1109/IPDPSW.2016.167. URL https://ieeexplore.ieee.org/abstract/document/7530073.

Alper Sarikaya, Michael Correll, Lyn Bartram, Melanie Tory, and Danyel Fisher. What do we talk about when we talk about dashboards? *IEEE Transactions on Visualization and Computer Graphics*, 25(1):682–692, January 2019. ISSN 1941-0506. doi: 10.1109/TVCG.2018.2864903.

SchedMD. Slurm documentation. https://slurm.schedmd.com. Accessed: 2025-02-10.

Gayane Sedrakyan, Erik Mannens, and Katrien Verbert. Guiding the choice of learning dashboard visualizations: Linking dashboard design and data visualization concepts. *Journal of Computer Languages*, 50:19–38, February 2019. ISSN 25901184. doi: 10.1016/j.jvlc.2018.11.002.

SequenceIQ. Sequenceiq. URL https://github.com/sequenceiq. Accessed: 2025-01-10.

Shinken. Shinken. URL https://shinken.readthedocs.io/en/latest/01_introduction/about.html. Accessed: 2025-01-10.

Konstantin S Stefanov, Sucheta Pawar, Ashish Ranjan, Sanjay Wandhekar, and Vladimir V Voevodin. A review of supercomputer performance monitoring systems.

Thanos. Thanos: Highly available prometheus setup with long term storage capabilities. https://thanos.io. Accessed: 2025-06-18.

Renato Toasa, Marisa Maximiano, Catarina Reis, and David Guevara. Data visualization techniques for real-time information — a custom and dynamic dashboard for analyzing surveys' results. In *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*, page 1–7, June 2018. doi: 10.23919/CISTI.2018.8398641. URL https://ieeexplore.ieee.org/document/8398641/?arnumber=8398641.

Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990.

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2001.

Yiannis Verginadis. A review of monitoring probes for cloud computing continuum. In Leonard Barolli, editor, *Advanced Information Networking and Applications*, page 631–643, Cham, 2023. Springer International Publishing. ISBN 978-3-031-28694-0. doi: 10.1007/978-3-031-28694-0_59.

Vadim Voevodin, Shaikhislamov I., and Serov A. Tasc software for hpc performance analysis: Current state and latest developments. , 13, 10 2024. doi: 10.14529/cmse240304.

Bin Yang, Wei Xue, Tianyu Zhang, Shichao Liu, Xiaosong Ma, Xiyang Wang, and Weiguo Liu. End-to-end i/o monitoring on leading supercomputers. *ACM Trans. Storage*, 19(1):3:1–3:35, January 2023. ISSN 1553-3077. doi: 10.1145/3568425.

Pavel Yermalovich. Dashboard visualization techniques in information security. In *2020 International Symposium on Networks, Computers and Communications (ISNCC)*, page 1–6, October 2020. doi: 10.1109/ISNCC49221.2020.9297291. URL https://ieeexplore.ieee.org/document/9297291/?arnumber=9297291.

Zabbix. Zabbix. URL https://www.zabbix.com/. Accessed: 2025-01-10.

Zotero. Zotero: Your personal research assistant. https://www.zotero.org, 2024. Accessed: 2024-11-24.

Appendix A

Requirements Document

Software Requirements Specification
for LISA Pipeline Runner dashboard

Prepared by:

Marija Jakovleva

Faculty of Engineering, University of Porto
Master of Software Engineering

Contents

Change Log

Date	Changes
June 27th 2025	First issue.

Introduction

This document defines the requirements applicable to the Laser Interferometer Space Antenna (LISA) Pipeline Runner monitoring dashboard.

The Pipeline Runner is a software framework designed to execute scientific workflows. It provides a unified interface for running different data processing pipelines, allowing users to apply diverse computational methods to their datasets.

A dashboard is a visual interface that displays metrics in a clear and organized manner by using charts, graphs, and tables. It allows users to monitor the status, performance and progress of systems and processes in a single view, allowing for quick understanding of information.

Chapter 1

Scope

This document defines the requirements for the development of the Laser Interferometer Space Antenna (LISA) Pipeline Runner dashboard, a software that is developed as part of the LISA DDPC (Distributed Data Processing Centre).

The LISA Pipeline Runner dashboard supports the monitoring and visualization of job execution and resource metrics consumption.

This document is not applicable to other dashboards developed for the LISA mission.

Chapter 2

Normative references

The following ECSS standards are referenced in this requirements document. These references are indispensable for the application of the requirements. For the most up-to-date versions of the referenced documents, the reader should consult the relevant publication sources:

- ECSS-E-ST-40C - Space engineering. Software
- ECSS-D-00-01C - Drafting rules and template for ECSS Standards

Chapter 3

Terms, definitions and abbreviated terms

3.1 Terms specific to the present document

3.1.1 pipeline

a sequence of automated procedures that data undergoes for transformation, processing, and delivery to its final destination.

3.1.2 resource

refers to a real or virtual element that an application or system contains during its functioning. Resources include CPU time, memory, disk storage, network bandwidth and more.

3.1.3 memory

refers to the hardware that temporarily holds data and instructions currently being used by the CPU.

3.1.4 storage

hardware used for the long-term retention of data, applications, and the operating system, even when the computer is powered off.

3.1.5 telemetry

collection and transmission of data about the performance, behavior, and health of an application or system.

3.1.6 Prometheus

an open-source monitoring system that collects and stores time-series data using a pull-based model over HTTP and a flexible query language (PromQL).

3.1.7 Grafana

an open-source analytics and visualization platform that allows users create interactive dashboards from a variety of data sources.

3.1.8 Thanos

a scalable extension to Prometheus that enables long-term storage, deduplication, and combined querying across multiple Prometheus instances.

3.1.9 Loki

a log aggregation system designed to work with Grafana, using the same label-based approach as Prometheus for indexing and querying logs.

3.2 Abbreviated terms

Abbreviation	Meaning
CI/CD	Continuous Integration and Continuous Delivery
CPU	Central Processing Unit
DDPC	Distributed Data Processing Centre
ECSS	European Cooperation for Space Standardization
GB	Gigabyte
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LISA	Laser Interferometer Space Antenna
LLM	Large Language Model
MFA	Multi-factor Authentication
PA	Product Assurance
QA	Quality Assurance
RAM	Random Access Memory
S3	Simple Storage Service
SGS	Science Ground Segment
SQL	Structured Query Language
SSD	Solid State Drive
UI	User Interface
UX	User Experience
YAML	Yet Another Markup Language

Chapter 4

Pipeline Runner

4.1 Introduction

This Chapter 4 introduces the structure of the Pipeline Runner architecture.

4.2 Overview of the Pipeline Runner architecture

The Pipeline Runner is a dedicated Kubernetes cluster, deployed using Kind (Kubernetes-in-Docker), and composed of multiple applications. Its primary purpose is to provide a simpler and more flexible environment for managing Argo workflows. The Pipeline Runner allows users to easily upload, validate, and execute workflow templates through Argo.

Table 4.1: Overview of technologies in the Pipeline Runner infrastructure

Component	**Description**
Argo Workflows	A Kubernetes-native workflow engine for orchestrating jobs and automating complex pipelines by using YAML templates.
Argo Events	An event-driven automation framework that triggers Argo Workflows or other actions in response to external events, such as webhooks or Kafka messages.
Argo UI	A web-based interface to manage, visualize and debug workflows running in Argo.
Kafka	A distributed streaming platform which enables communication between services via publish-subscribe messaging.
Kafdrop	A web-based UI for inspecting Kafka clusters, including topics, partitions, and consumer groups. Used for debugging and monitoring Kafka activity.
MinIO	A high-performance, S3-compatible object storage service used for storing artifacts, logs, and other binary data.
MinIO UI	Provides a simple, web-based interface for managing buckets, uploading and downloading objects and browsing stored files.
MongoDB	A NoSQL document database used to store structured and semi-structured data. Stores Pipeline Runner users, templates and workflow metadata.
Mongo Express	A web-based interface for MongoDB that allows users to view, edit, and manage database collections and documents through a simple UI.
Node.js Backend	The server-side application responsible for uploading templates, launching workflows and passing real-time execution notifications to the frontend application.
React Frontend	A frontend application used to upload or paste workflow templates, monitor real-time status changes of workflows, and provides an overview of all workflows.

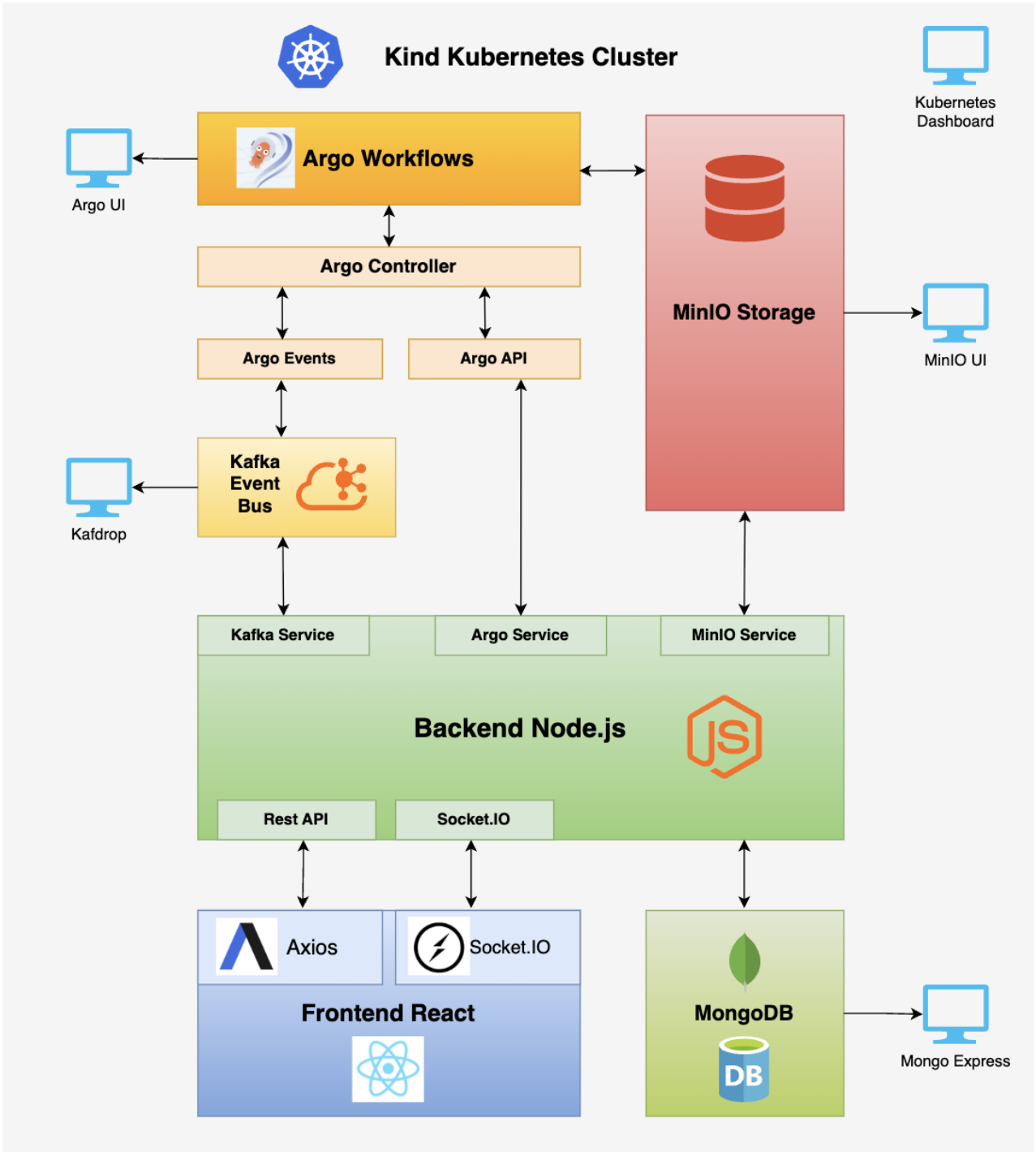


Figure 4.1: Pipeline Runner Architecture Overview

Chapter 5

Requirements

5.1 Introduction

This Chapter 5 defines the requirements of the LISA Pipeline Runner dashboard. Requirements are categorized by priority using the keywords *shall*, *should*, and *may*:

- *Shall* indicates mandatory requirements that are essential for a functional dashboard.
- *Should* refers to recommended features that are desirable but not critical.
- *May* describes optional features that are beneficial but not necessary.

5.2 General requirements

General requirements describe system-wide expectations and emphasize the overarching capabilities and attributes of the monitoring dashboard.

a) The dashboard shall support workflow monitoring, providing visibility into the status, progress, and logs of all monitored workflows.

b) The dashboard shall provide error diagnostics to assist users in identifying, analyzing, and understanding errors occurring in monitored workflows.

c) The dashboard shall ensure that all monitored data, including workflow states and metrics are stored in a retrievable manner for both real-time and historical analysis.

d) The dashboard shall provide a user interface that supports intuitive navigation and visualization suitable for both high-level overviews and detailed analysis.

e) The dashboard shall associate all monitored resource metrics (e.g., CPU, memory, disk usage) with the corresponding workflows, services, or processes to provide insights into resource consumption.

f) The dashboard should support correlation features that allow users to cross-reference workflow events, errors, and resource usage within a shared timeline or analysis view.

g) The dashboard may support scalability insights, providing visibility into system performance under varying loads.

5.3 Functional requirements

Functional requirements focus on specific features, behaviors and interactions of the application that must be implemented in order to satisfy the general requirements.

a) The dashboard shall display a real-time status overview of all workflows, including their current state (Pending, Running, Succeeded, Failed, Error).

b) The dashboard shall continuously collect and display resource metrics (CPU, memory, disk usage, network) for each workflow or service being monitored.

c) The dashboard shall display logs from all detected errors and failures.

d) The dashboard shall store historical workflow data, including statuses, metrics, and events, for a configurable retention period.

e) The dashboard shall support both tabular and graphical representations of data.

f) The dashboard may allow users to query and retrieve historical data, using filters such as time range and job name.

5.4 Performance requirements

Performance requirements cover things like response time, data update rate, system capacity, and retention, all of which are crucial in an operational monitoring system.

a) The dashboard should display workflow status updates with a maximum delay of 5 seconds from the moment data is available in Prometheus.

b) Diagnostic information for any detected error should be made available in the dashboard within 10 seconds of the error occurrence.

c) The dashboard should maintain 99.9% uptime over a rolling 30-day period, excluding scheduled maintenance windows.

d) The dashboard may support at least 50 concurrent users without decline in response time.

5.5 Interface requirements

Interface requirements define how the application interacts with external systems, such as telemetry sources and databases. Interface requirements are crucial in ensuring compatibility and integration.

a) The dashboard shall receive workflow metrics data via Prometheus, Thanos or Loki, depending on the type of data needed.

b) The dashboard shall support long-term storage of the time-series metric data using MinIO and Thanos.

c) The dashboard should be configured to handle schema evolution for incoming metrics, supporting additive changes without breaking compatibility.

5.6 Operational requirements

Operational requirements describe how the system should operate under normal conditions.

a) The dashboard shall support automated archiving of historical data older than a certain threshold (e.g., 10 days) to long-term storage in MinIO.

b) The dashboard should be operable by authorized users through a secure web interface accessible from approved operational environments.

5.7 Resources requirements

Resource requirements describe the computing, storage, network, and other system resources necessary to deploy and operate the dashboard reliably.

a) The dashboard shall support deployment in a containerized environment using Kubernetes.

b) The dashboard should support deployment on Linux-based operating systems.

c) The dashboard may operate on a virtual server with a minimum of: 8 CPU cores, 16 GB RAM, 500 GB SSD storage for application and database.

5.8 Design requirements and implementation constraints

Design requirements focus on how the system should be structured while ensuring it meets functional, usability, and adaptability goals.

a) The dashboard shall use PromQL or LokiQL queries for data retrieval.

b) The dashboard shall include logging by capturing system events and job errors with timestamps using Loki.

c) The dashboard shall be implemented using Grafana to ensure long-term maintainability.

d) All configuration files, scripts, and deployment pipelines shall be stored in a version-controlled repository, such as Gitlab, ensuring proper control of changes.

e) The dashboard should include a visual alerting system, with color-coded status indicators (green for normal, yellow for warnings, red for critical).

f) The dashboard should be designed to support horizontal scaling to handle increased pipeline volumes by deploying additional processing nodes.

5.9 Security and privacy requirements

Security and privacy requirements cover confidentiality, integrity, availability, and traceability.

a) The dashboard should ensure that all user authentication requires strong authentication mechanisms, such as password complexity rules and multi-factor authentication (MFA).

b) The dashboard should not be developed using any public LLMs that collect input data, preventing confidential information leak.

c) The dashboard may ensure that all data transmissions between components are encrypted to prevent interception and unauthorized modification.

5.10 Portability requirements

Portability requirements specify the ability of the system to be deployable across different environments and adaptable to future system changes.

a) The dashboard shall be a web application accessible from the most common browsers (Chrome, Safari, Firefox, Edge).

b) The dashboard shall be containerizable using Kubernetes to support quick deployment.

c) The dashboard should separate configurations from code, ensuring environment variables can be easily adjusted without modifying the source code.

d) The dashboard should not have hard-coded file paths, host addresses, or environment variables within the source code.

5.11 Software quality requirements

Quality requirements are responsible for addressing the expected levels of maintainability, performance, reliability, and usability.

a) The dashboard shall use Git for version control with branching strategies to provide traceability of changes.

b) The dashboard should undergo a final acceptance review before deployment to ensure that all requirements have been satisfied.

c) The dashboard should be developed following documented coding standards for code readability and maintainability.

d) The dashboard should provide meaningful error messages to help with issue diagnosis.

5.12 Software reliability requirements

Reliability requirements are responsible for describing how the software can perform its intended functions under defined conditions for a specified period of time.

a) The dashboard should follow a zero-downtime deployment strategy, ensuring that updates do not interrupt operations.

b) The dashboard should undergo reliability testing, such as stress testing and long-duration operational simulation.

c) The dashboard should detect and log all internal errors and exceptions.

5.13 Software maintainability requirements

Maintainability requirements ensure that the software is easy to maintain, update, and extend over its lifecycle.

a) The dashboard's error messages and logs shall be structured, timestamped, and traceable to the originating component.

b) The dashboard shall support the addition of new metric sources without requiring significant refactor of existing code.

c) The dashboard's visualization graphs and tables shall be well-documented, including descriptions explaining the exact metric displayed.

d) The dashboard's source code should adhere to documented coding standards and be fully documented.

e) The dashboard's external dependencies, such as libraries and frameworks, may have their versions explicitly declared.

f) The dashboard's documentation may be stored in a version-controlled repository.

5.14 Software safety requirements

Safety requirements ensure that the software is safe to operate in different conditions for the mitigation of risks to system integrity.

a) The dashboard should ensure that error messages related to critical failures are displayed clearly on the user interface.

b) The dashboard should have error handling, ensuring that a failure in one component does not compromise the overall state of the system.

c) The dashboard may be tested during both development and deployment phases, with specific safety-critical failure scenarios to ensure the software remains in a safe state.

5.15 Software configuration and delivery requirements

Configuration and delivery requirements ensure that the software is configured and delivered in a controlled way, paying close attention to deployment procedures, version control, and configuration management.

a) The dashboard shall be packaged into reproducible builds, ensuring that each version of the software can be regenerated from the source code and configuration files in an identical state.

b) The dashboard should provide clear and up-to-date release notes with each version.

c) The dashboard should include an automated build and deployment pipeline, allowing for consistent deployment processes.

d) The dashboard may use a consistent naming convention for all modules and components.

e) The dashboard may include a rollback mechanism, enabling operators to revert to a previous stable version in the event of issues with a new deployment.

5.16 Data definition and database requirements

Data definition and database requirements guarantee that data management procedures, such as storage, organization, and retrieval, are clear and effective.

a) The dashboard's databases may support backups and restores, allowing the system to recover data in case of data loss.

5.17 Human factor related requirements

Human factor requirements primarily focus on the user experience (UX) and human-centric design to ensure that the software is intuitive and efficient.

a) The dashboard shall provide an intuitive and user-friendly interface.

b) The dashboard shall be designed for minimal cognitive load, presenting data in a way that allows users to quickly comprehend displayed information.

c) The dashboard shall include a responsive design, ensuring that the dashboard's layout automatically adjusts to different screen sizes and resolutions without compromising usability.

d) The dashboard should ensure visual accessibility by following guidelines for color contrast, font size, and layout design.

e) The dashboard interface should contain clear visual indicators and color-coding to highlight issues or failures.

Chapter 6

Validation requirements

This Chapter defines the validation requirements and associated methods for ensuring that the monitoring dashboard meets the mission objectives and satisfies all user needs under the intended operational conditions.

It is important to note that the validation requirements for the LISA Pipeline Runner monitoring dashboard presented in this document have been derived from general ground segment validation requirements, as the final requirements are still under revision.

6.1 Validation Approach

The validation requirements defined in this chapter have been derived from two key documents:

1. SCI-SX-PA-RS-001 - Product Assurance Requirements for the Science Ground Segment
2. SCI-SX-PA-RS-002 - Engineering Requirements for the Science Ground Segment

These documents are internal and unclassified, to be used for the LISA mission, prepared by the PA/QA team. The Product Assurance Requirements document and the Engineering requirements document are both applicable to the SGS (Science Ground Segment).

6.2 Validation Strategy

Validation shall be performed using a combination of:

- Functional Testing: Confirming that the dashboard displays and updates metrics correctly.
- Performance Testing: Verifying response times, refresh intervals, and query loads under expected workflow execution volumes.

- Scenario-Based Validation: Running workflows to simulate operations and edge cases.
- User Validation: Ensuring target users can use the dashboard effectively.

In order to ensure that the dashboard meets the intended operational requirements, before deployment, the dashboard shall be tested using preliminary set of mission operations data.

6.3 Validation Methods

The validation of the dashboard shall be conducted using a combination of standardized validation methods to guarantee full coverage of functional and non-functional requirements. These methods follow established engineering practices and are selected based on the nature of each requirement.

Validation Type	Description
Analysis	Reviewing Prometheus queries, dashboard panels and data export
Demo	Running Argo workflows and visualizing them in Grafana
Inspection	Reviewing configuration files, YAMLs, and data retention policies

Table 6.1: Overview of validation methods

6.4 Validation Matrix

The following matrices define how each requirement of the Pipeline Runner monitoring dashboard will be validated, including the method, environment, and acceptance criteria.

Table 6.2: Validation Matrix for General Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.2.a	The dashboard shall support workflow monitoring, providing visibility into the status, progress, and logs of all monitored workflows	Demo	HPC staging cluster with running Argo workflows	All running workflows appear within 15 seconds of launch
5.2.b	The dashboard shall provide error diagnostics to assist users in identifying, analyzing, and understanding errors occurring in monitored workflows	Demo	Simulated pipeline failure scenarios	Error logs and failure reasons are displayed within 10 seconds of failure occurrence

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.2.c	The dashboard shall ensure that all monitored data, including workflow states and metrics are stored in a retrievable manner for both real-time and historical analysis	Inspection	Prometheus and Thanos configuration	Monitoring data is queryable for at least 1 year and includes all required workflow metrics
5.2.d	The dashboard shall provide a user interface that supports intuitive navigation and visualization suitable for both high-level overviews and detailed analysis	Analysis	UI/UX review with representative users	Users can easily access detailed and summary views with no prior detailed knowledge of the system
5.2.e	The dashboard shall associate all monitored resource metrics (e.g., CPU, memory, disk usage) with the corresponding workflows, services, or processes to provide insights into resource consumption	Demo	Grafana with running Argo workflows	Resource usage metrics are accurate for all monitored components, with a tolerance of up to 5% deviation.
5.2.f	The dashboard should support correlation features that allow users to cross-reference workflow events, errors, and resource usage within a shared timeline or analysis view	Demo	Live dashboard with simulated workflow issues	Users can cross-reference workflow events, errors, and metrics in a unified dashboard configuration with all components updating synchronously within 5 seconds
5.2.g	The dashboard may support scalability insights, providing visibility into system performance under varying loads	Demo	Load testing environment with simulated workload variation	Dashboard reflects performance changes under load and provides time-series trends

Table 6.3: Validation Matrix for Functional Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.3.a	The dashboard shall display a real-time status overview of all workflows, including their current state (Pending, Running, Succeeded, Failed, Error)	Demo	Staging environment with simulated jobs in various states	Dashboard updates job states within 5 seconds of state change and correctly reflects all job statuses
5.3.b	The dashboard shall continuously collect and display resource metrics (CPU, memory, disk usage, network) for each workflow or service being monitored	Demo	Live metrics collection with Prometheus and node_exporter	Metrics are updated every 10 seconds and accurately reflect monitored job resource usage
5.3.c	The dashboard shall display logs from all detected errors and failures	Demo	Simulated job failures and logging setup	Logs are displayed within 5 seconds of error detection with relevant metadata
5.3.d	The dashboard shall store historical workflow data, including statuses, metrics, and events, for a configurable retention period	Inspection	Prometheus/ Thanos retention configuration	Historical data remains accessible and complete for the full configured retention period (minimum 1 year)
5.3.e	The dashboard shall support both tabular and graphical representations of data	Demo	Dashboard with mixed panel views	Users can toggle between tabular and graphical views and see consistent data in both formats
5.3.f	The dashboard may allow users to query and retrieve historical data, using filters such as time range and job name	Demo	Dashboard with historical metrics and Thanos as a datasource	Filtered historical data queries return accurate results

Table 6.4: Validation Matrix for Performance Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.4.a	The dashboard should display workflow status updates with a maximum delay of 5 seconds from the moment data is available in Prometheus	Demo	Staging environment with Prometheus and simulated workflow data	Status updates appear on the dashboard within 5 seconds of data being available in Prometheus
5.4.b	Diagnostic information for any detected error should be made available in the dashboard within 10 seconds of the error occurrence	Demo	Simulated error scenarios in monitored pipelines	Diagnostic details shown in dashboard within 10 seconds of simulated error occurence
5.4.c	The dashboard should maintain 99.9% uptime over a rolling 30-day period, excluding scheduled maintenance windows	Inspection	Production uptime logs and monitoring reports	Dashboard availability is more or equal to 99.9% over 30 days (excluding scheduled downtime)
5.4.d	The dashboard may support at least 50 concurrent users without decline in response time	Demo	Performance testing setup with 50 simulated concurrent users	No visible lag or decline in performance in the dashboard UI

Table 6.5: Validation Matrix for Interface Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.5.a	The dashboard shall receive workflow metrics data via Prometheus, Thanos or Loki, depending on the type of data needed	Inspection	Prometheus, Thanos and Loki integration configuration	Prometheus, Thanos and Loki are configured as data sources and metrics are visible in dashboard panels
5.5.b	The dashboard shall support long-term storage of the time-series metric data using MinIO and Thanos	Demo	Monitoring setup with MinIO and Thanos	Archived metrics are queryable using Thanos after 10+ days of retention

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.5.c	The dashboard should be configured to handle schema evolution for incoming metrics, supporting additive changes without breaking compatibility	Analysis	Simulated schema evolution by adding new labels to metrics	New metrics are accepted and do not break existing dashboard functionality

Table 6.6: Validation Matrix for Operational Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.6.a	The dashboard shall support automated archiving of historical data older than a certain threshold (e.g., 10 days) to long-term storage in MinIO	Demo	Data retention simulation with MinIO bucket	Data older than 10 days is moved to a MinIO bucket without manual input and remains queryable using Thanos as a data source
5.6.b	The dashboard should be operable by authorized users through a secure web interface accessible from approved operational environments	Demo	Secure environment with role-based access control	Authorized users can access and operate the dashboard from whitelisted IPs only

Table 6.7: Validation Matrix for Resource Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.7.a	The dashboard shall support deployment in a containerized environment using Kubernetes	Demo	Kubernetes cluster with container orchestration	Dashboard is deployed, scaled, and accessible within a Kubernetes-managed environment
5.7.b	The dashboard should support deployment on Linux-based operating systems	Demo	Linux server environment	Dashboard installs and runs successfully on supported Linux distributions

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.7.c	The dashboard may operate on a virtual server with a minimum of: 8 CPU cores, 16 GB RAM, 500 GB SSD storage for application and database	Inspection	Virtual server environment	System resources match or exceed minimum specified configuration

Table 6.8: Validation Matrix for Design Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.8.a	The dashboard shall use PromQL or LokiQL queries for data retrieval	Inspection	Dashboard configuration and query templates	All data panels use PromQL or LokiQL queries to retrieve metrics from Prometheus, Thanos or Loki
5.8.b	The dashboard shall include logging by capturing system events and job errors with timestamps using Loki	Demo	Loki integration with simulated workflow logs	Workflow execution emits logs which are visible in designated Grafana log panels
5.8.c	The dashboard shall be implemented using Grafana to ensure long-term maintainability	Inspection	Dashboard codebase and deployment stack	Grafana is the primary framework used for all visualization components
5.8.d	All configuration files, scripts, and deployment pipelines shall be stored in a version-controlled repository, such as Gitlab, ensuring proper control of changes	Inspection	GitLab repository and CI/CD configuration	All relevant files and pipelines are tracked in GitLab with version history
5.8.e	The dashboard should include a visual alerting system, with color-coded status indicators (green for normal, yellow for warnings, red for critical)	Demo	Live dashboard with simulated status transitions	Color-coded indicators change correctly based on system state
5.8.f	The dashboard should be designed to support horizontal scaling to handle increased pipeline volumes by deploying additional processing nodes	Analysis	Scalability design review and simulated scaling tests	System scales with additional nodes and maintains performance under load

Table 6.9: Validation Matrix for Security and Privacy Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.9.a	The dashboard should ensure that all user authentication requires strong authentication mechanisms, such as password complexity rules and multi-factor authentication (MFA)	Demo	Authentication configuration in production-like environment	Authentication requires complex passwords and MFA for all users
5.9.b	The dashboard should not be developed using any public LLMs that collect input data, preventing confidential information leak	Analysis	Code review and toolchain inspection	No code, data, or interaction involves public LLMs with data collection policies
5.9.c	The dashboard may ensure that all data transmissions between components are encrypted to prevent interception and unauthorized modification	Inspection	Network configuration and TLS certificate setup	All internal communications use TLS or equivalent encryption protocols

Table 6.10: Validation Matrix for Portability Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.10.a	The dashboard shall be a web application accessible from the most common browsers (Chrome, Safari, Firefox, Edge)	Demo	Cross-browser testing environment	Dashboard renders and functions correctly in latest versions of Chrome, Safari, Firefox, and Edge
5.10.b	The dashboard shall be containerizable using Kubernetes to support quick deployment	Demo	Kubernetes environment with container registry	Dashboard is built as a container image and deployed successfully using Kubernetes

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.10.c	The dashboard should separate configurations from code, ensuring environment variables can be easily adjusted without modifying the source code	Inspection	Source code and deployment scripts	Configuration is injected via environment variables and not hardcoded in codebase
5.10.d	The dashboard should not have hard-coded file paths, host addresses, or environment variables within the source code	Inspection	Code review	All file paths, hostnames, and environment variables are dynamically set via configuration

Table 6.11: Validation Matrix for Software Quality Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.11.a	The dashboard shall use Git for version control with branching strategies to provide traceability of changes	Inspection	Version control repository (GitLab)	All changes are tracked with identifiable branches and understandable commit messages
5.11.b	The dashboard should undergo a final acceptance review before deployment to ensure that all requirements have been satisfied	Demo	Staging environment with full validation checklist	Acceptance review confirms fulfillment of all documented requirements
5.11.c	The dashboard should be developed following documented coding standards for code readability and maintainability	Inspection	Codebase and development guidelines	Code follows internal or industry-standard style guide with documentation present
5.11.d	The dashboard should provide meaningful error messages to help with issue diagnosis	Demo	Simulated error conditions in runtime environment	Displayed errors contain descriptive messages and relevant stack tracing for debugging

Table 6.12: Validation Matrix for Software Reliability Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.12.a	The dashboard should follow a zero-downtime deployment strategy, ensuring that updates do not interrupt operations	Demo	CI/CD setup for rolling deployment in staging	Dashboard updates without service interruption or user session loss
5.12.b	The dashboard should undergo reliability testing, such as stress testing and long-duration operational simulation	Demo	Load testing and soak testing environment	Dashboard remains stable and responsive under prolonged and high-stress conditions
5.12.c	The dashboard should detect and log all internal errors and exceptions	Demo	Simulated error conditions during runtime	All internal exceptions are logged with timestamps and relevant context

Table 6.13: Validation Matrix for Software Maintainability Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.13.a	The dashboard's error messages and logs shall be structured, timestamped, and traceable to the originating component	Demo	Simulated error scenarios in test environment	Logs are structured with timestamps and component identifiers
5.13.b	The dashboard shall support the addition of new metric sources without requiring significant refactor of existing code	Analysis	Architecture and code structure review	New sources can be integrated via configuration without altering core logic
5.13.c	The dashboard's visualization graphs and tables shall be well-documented, including descriptions explaining the exact metric displayed	Inspection	Visualization dashboard	All panels on every dashboard configuration have a description that describes the metric

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.13.d	The dashboard's source code should adhere to documented coding standards and be fully documented	Inspection	Code repository, inline documentation review with coding standard guide	All modules, functions, and classes include relevant documentation or comments and follow structure conventions defined in coding standards
5.13.e	The dashboard's external dependencies, such as libraries and frameworks, may have their versions explicitly declared	Inspection	Dependency management files (package.json)	All third-party packages include fixed version declarations
5.13.f	The dashboard's documentation may be stored in a version-controlled repository	Inspection	Documentation repository and version history	Documentation is maintained in Git with tracked commits and change history

Table 6.14: Validation Matrix for Software Safety Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.14.a	The dashboard should ensure that error messages related to critical failures are displayed clearly on the user interface	Demo	Simulated critical error scenarios	Critical errors are shown on-screen with clear, user-friendly messages
5.14.b	The dashboard should have error handling, ensuring that a failure in one component does not compromise the overall state of the system	Analysis	System architecture and fault injection review	Isolated component failures do not propagate or cause system-wide disruptions
5.14.c	The dashboard may be tested during both development and deployment phases, with specific safety-critical failure scenarios to ensure the software remains in a safe state	Demo	Development and staging environments with failure simulation	Dashboard continues to operate safely and predictably during tested failure modes

Table 6.15: Validation Matrix for Software Configuration and Delivery Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.15.a	The dashboard shall be packaged into reproducible builds, ensuring that each version of the software can be regenerated from the source code and configuration files in an identical state	Demo	Build environment with version pinning	Identical builds are generated consistently from the same commit and configuration
5.15.b	The dashboard should provide clear and up-to-date release notes with each version	Inspection	Release artifacts and documentation repository	Each version includes accurate release notes describing features, fixes, and changes
5.15.c	The dashboard should include an automated build and deployment pipeline, allowing for consistent deployment processes	Demo	CI/CD pipeline	New code commits trigger repeatable build and deployment process without manual input
5.15.d	The dashboard may use a consistent naming convention for all modules and components	Inspection	Source code and documentation	All modules and components follow defined naming patterns without exceptions
5.15.e	The dashboard may include a rollback mechanism, enabling operators to revert to a previous stable version in the event of issues with a new deployment	Demo	Staging environment with deployment history	System can be rolled back to previous release without data loss or service downtime

Table 6.16: Validation Matrix for Data definition and Database Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.16.a	The dashboard's databases may support backups and restores, allowing the system to recover data in case of data loss	Demo	Backup and restore simulation in staging environment	Data can be successfully backed up and restored with no loss or corruption

Table 6.17: Validation Matrix for Human factor related Requirements

Req. ID	Requirement Description	Validation Method	Validation Environment	Acceptance Criteria
5.17.a	The dashboard shall provide an intuitive and user-friendly interface	Analysis	UI/UX review with target users	Users report high usability in feedback sessions or usability tests
5.17.b	The dashboard shall be designed for minimal cognitive load, presenting data in a way that allows users to quickly comprehend displayed information	Analysis	UI design evaluation and expert review	Layout, groupings, and hierarchy support fast understanding with minimal interpretation needed
5.17.c	The dashboard shall include a responsive design, ensuring that the dashboard's layout automatically adjusts to different screen sizes and resolutions without compromising usability	Demo	Multiple screen resolutions and device emulation	Layout adapts seamlessly without loss of usability or visual consistency across various screen resolutions, including common desktop resolutions (1920×1080) and tablet displays (1280×800 and 1024×768)
5.17.d	The dashboard should ensure visual accessibility by following guidelines for color contrast, font size, and layout design	Inspection	Design artifacts and implemented interface	Dashboard complies with accessibility guidelines for contrast and readability
5.17.e	The dashboard interface should contain clear visual indicators and color-coding to highlight issues or failures	Demo	Simulated failure on a live dashboard	Warnings and errors are clearly highlighted with visual indicators and distinct colors

6.5 Conclusion

All validation activities will be performed prior to system deployment in the mission control infrastructure. Any critical findings will be documented and addressed before handover to operational teams.

Appendix B

ICD Document

Interface Control Document
for LISA Pipeline Runner dashboard

Prepared by:

Marija Jakovleva

Faculty of Engineering, University of Porto
Master of Software Engineering

Contents

Change Log

Date	Changes
June 27th 2025	First issue.

Introduction

This document defines and specifies the interfaces between the LISA Pipeline Runner monitoring dashboard component within the Kubernetes cluster.

Chapter 1

Terms, definitions and abbreviated terms

1.1 Terms specific to the present document

pipeline

a sequence of automated procedures that data undergoes for transformation, processing, and delivery to its final destination.

dashboard

a visual interface that aggregates, displays and organizes key metrics, logs, and statuses from infrastructure in real time.

interface

a defined point of interaction between two systems or components, specifying how they communicate and exchange data.

Prometheus

an open-source monitoring system that collects and stores time-series data using a pull-based model over HTTP and a flexible query language (PromQL).

Grafana

an open-source analytics and visualization platform that allows users create interactive dashboards from a variety of data sources.

Thanos

a scalable extension to Prometheus that enables long-term storage, deduplication, and combined querying across multiple Prometheus instances.

Loki

a log aggregation system designed to work with Grafana, using the same label-based approach as Prometheus for indexing and querying logs.

1.2 Abbreviated terms

Abbreviation	Meaning
AD	Applicable Document
API	Application Programming Interface
ECSS	European Cooperation for Space Standardization
HTTP	Hypertext Transfer Protocol
IF	Interface
ICD	Interface Control Document
LISA	Laser Interferometer Space Antenna
PromQL	Prometheus Query Language
RPC	Remote Procedure Call
RF	Reference Document
TSDB	Time Series Database
UID	Unique Identifier
URL	Uniform Resource Locator
YAML	Yet Another Markup Language

Chapter 2

Introduction

2.1 Purpose and objective

This document defines the interfaces of the LISA Pipeline Runner Monitoring Dashboard, a component deployed within a Kubernetes cluster. The dashboard is part of the LISA Pipeline Runner, a software system designed to execute scientific workflows.

The Pipeline Runner was designed to execute multiple scientific workflows concurrently, allowing parallel processing of complex tasks. Given the nature of these workflows, users must be able to oversee their successful execution and verify the overall health of the system that is running them. To support this, a dedicated monitoring dashboard needs to be developed. This dashboard can provide real-time visibility into both system-level metrics, such as CPU, memory, and storage usage and workflow-specific data, such as status, progress and logs. By collecting this information into a centralized interface, the dashboard allows users to detect issues early, maintain operational reliability and optimize system performance.

This document is not applicable to other dashboards developed for the LISA mission.

Chapter 3

Responsibility

3.1 Interface Responsibilities

- **Monitoring Team**: Responsible for defining and maintaining the interfaces between Prometheus, Loki, Thanos and Grafana. This includes the configuration of visualization panels, graph description, Grafana data source integration, and overall dashboard development.
- **Pipeline Runner Team**: Responsible for exposing Prometheus-compatible metrics from Argo Workflows, ensuring that metrics are named, labeled and exported according to the conventions defined in this ICD.

The **Monitoring Team** is the designated owner of this Interface Control Document (ICD) and is responsible for coordinating all proposed interface changes and ensuring adherence to procedures.

3.2 Document Approval Authority

This Interface Control Document (ICD) is subject to configuration control. All updates, modifications, and releases shall be approved in accordance with the project's document control and interface management processes.

- **Approval Authority:** The Monitoring Lead Engineer is the designated approval authority for the initial release and subsequent revisions of this ICD. Final approval shall be coordinated with the System Engineering Manager.
- **Change Control Authority:** All proposed changes to this ICD shall be reviewed and approved by the Interface Control Board (ICB), which includes representatives from the Monitoring Team Pipeline Runner Team. Changes shall not take effect until formal approval is recorded.

- **Configuration Management:** This document is maintained under configuration control. Change dates and requests are maintained in the change log of this document.

3.3 Interface End Responsibilities

The responsibilities for each end of the interfaces described in this document are explicitly defined. This ensures that the development, implementation, and maintenance of each interface is properly coordinated and traceable.

Table 3.1: Interface End Responsibilities

Interface ID	**Description**	**Producer**	**Consumer**
IF-01	Prometheus scraping Argo metrics	Pipeline Runner Architecture Team	Monitoring Team
IF-02	Thanos extracting historical Prometheus metrics from MinIO	Pipeline Runner Architecture Team	Monitoring Team
IF-03	Grafana reading from Prometheus	Monitoring Team	Monitoring Team
IF-04	Loki collecting logs from Kubernetes workloads	Monitoring Team	Monitoring Team
IF-05	Grafana querying logs from Loki	Monitoring Team	Monitoring Team
IF-06	Thanos aggregating Prometheus metrics	Monitoring Team	Monitoring Team
IF-07	Grafana querying metrics from Thanos	Monitoring Team	Monitoring Team

Each interfacing party is responsible for the correct implementation, availability, and maintenance of their respective interface endpoints. The Monitoring Team is responsible for ensuring all consumers are correctly configured and aligned with the expected data models and formats defined in this ICD.

Chapter 4

Applicable and Reference Documents

4.1 Documents

This Interface Control Document (ICD) has been developed in accordance with the applicable standards and guidelines established by the European Cooperation for Space Standardization (ECSS). The documents listed below provide supporting references used during the preparation and structuring of this ICD.

The applicable documents define the mandatory requirements that this document complies with, while the reference documents provide background information relevant to the monitoring dashboard and its integration within the LISA Pipeline Runner system.

A complete list of applicable and reference documents is provided in Table 4.1.

Table 4.1: Applicable and Reference Documents

ID	**Document Title**	**Description**
AD-01	ECSS-E-ST-40C	Software Engineering
AD-02	ECSS-M-ST-40C	Configuration and Information Management
AD-03	ECSS-E-ST-10-24C	Interface Management
RD-01	Prometheus Documentation	Prometheus API specifications
RD-02	Grafana Documentation	Grafana data source and dashboard configuration
RD-03	Thanos Documentation	Thanos API configuration
RD-04	Loki Documentation	Loki configuration
RD-03	LISA Pipeline Runner Requirements Document	Internal document outlining all requirements of the monitoring dashboard

4.2 Relationship to Other Programme Documents

This ICD is a configuration document that defines the internal software and data interfaces of the LISA Pipeline Runner Monitoring Dashboard. It shall be considered the authoritative reference for all interface-related specifications involving Prometheus, Grafana, Loki, and Thanos within the monitoring infrastructure.

This document takes precedence over informal documentation, implementation-specific notes, or ad hoc interface agreements. In case of conflict between this ICD and lower-level implementation documents, the ICD shall prevail. If inconsistencies arise between this ICD and higher-level system architecture or the LISA Pipeline Runner monitoring dashboard software requirements specification, the issue shall be raised and resolved through the formal configuration management and change control process.

This document is complementary to, and shall be read in conjunction with the LISA pipeline runner monitoring dashboard software requirements specification document and the applicable ECSS standards listed in Section 4.1.

4.3 Product Tree

The Product Tree provides a hierarchical breakdown of the LISA Pipeline Runner system (See Figure 4.1). This ICD focuses on the Monitoring Subsystem, which is a distinct configuration item within the overall product structure.

The monitoring subsystem includes Prometheus for time-series data collection, Grafana for visualization, Loki for log aggregation, and Thanos for long-term storage and global querying. Supporting components such as `node_exporter`, `kube-state-metrics`, `promtail`, `thanos-query`, `thanos-store-gateway`, `thanos-compactor` and `thanos-sidecar` are considered to be auxiliary elements that enable core functionality and supply necessary data to the monitoring system. These supporting components are further explained later in this document (See Sections 5.2.1, 5.4.4 and 5.5.1). Each component is treated as an identifiable item within the configuration management process, with defined interfaces and responsibilities as outlined in this document.

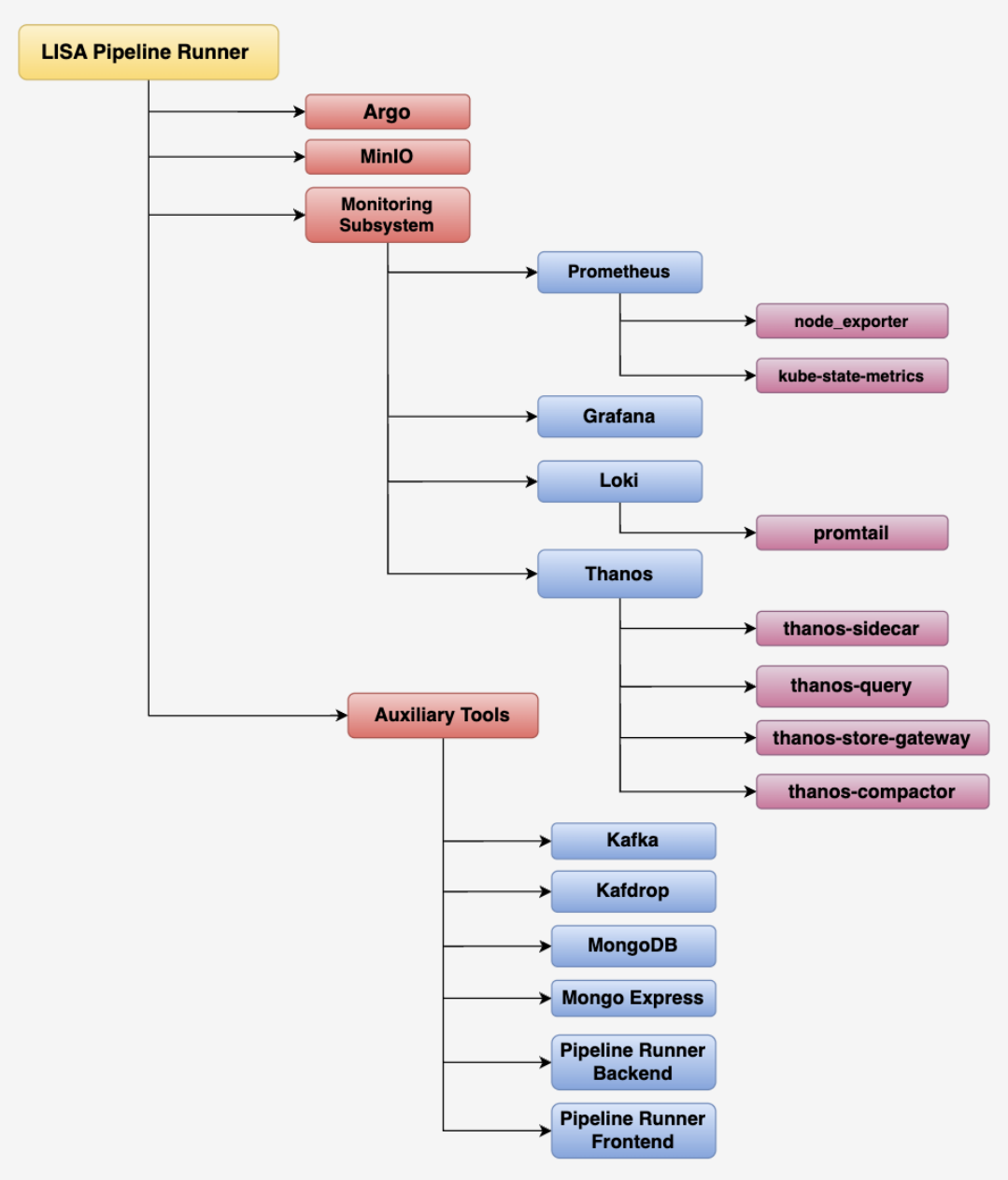


Figure 4.1: LISA Pipeline Runner Monitoring Dashboard Product Tree

Chapter 5

Interface Definition

5.1 Overall Naming Schema and Constraints

To ensure consistency across the LISA Pipeline Runner monitoring dashboard, all naming conventions used for interfaces, metrics, and dashboards follow predefined schemas:

- All Argo workflows executed within the LISA Pipeline Runner environment shall be prefixed with `lisa-workflow-`.
- All infrastructure components that form part of the monitoring dashboard subsystem are deployed within the dedicated Kubernetes namespace `scientific-workflow`. All service discovery, metric scraping, and logging configurations assume this namespace context unless explicitly set otherwise.

Workflow names are always lowercase and constructed by first adding the fixed prefix `lisa-workflow-` for identification, then converting the user input to lowercase, replacing any non-alphanumeric characters with hyphens, and finally appending a suffix of five randomly generated alphanumeric characters to ensure uniqueness across workflow instances. The general naming pattern is:

```
lisa-workflow-{user_input}-{rand5}
```

Since each character in the suffix can be one of 36 possibilities (26 letters + 10 digits), the total number of possible combinations is $36^5 = 60{,}466{,}176$. The large amount of variations makes it extremely unlikely for two workflows to end up with the same name, even if the `user_input` is identical.

Dashboard panels and graph visualizations that are specific to a given workflow only reference workflows whose names begin with the `lisa-workflow-` prefix. This constraint ensures that workflow-specific metrics and logs are scoped accurately and visualized consistently across the monitoring stack. This filtering is implemented in Grafana through the use of a templated variable named `$Workflow`, which dynamically resolves to the selected workflow name.

In addition to workflow naming, all Prometheus metric names and labels follow the Prometheus conventions, using `snake_case`. Prometheus metric label keys and values shall be limited to lowercase alphanumeric characters, underscores, dashes and must avoid dynamic or high-cardinality values unless explicitly justified.

5.2 Prometheus Interfaces

The Prometheus component of the monitoring dashboard system is deployed and managed via the Prometheus Operator within the `scientific-workflow` namespace. Its primary function is to collect metrics from both the cluster infrastructure and workflow-level components.

5.2.1 Scrape Configuration and Service Discovery

Prometheus is configured to automatically discover and scrape metrics from the following cluster components:

- **kube-state-metrics** – Kubernetes object state metrics (e.g., deployments, pods, nodes)
- **node_exporter** – Node-level OS metrics (e.g., CPU, memory, disk)
- **kubelet** – Node-level container runtime and cAdvisor metrics
- **cAdvisor** – Per-container CPU, memory, filesystem usage

All metrics endpoints are discovered through Kubernetes service annotations and relabeling rules defined by the Prometheus Operator.

5.2.2 Scrape Protocol Version Constraint

Due to a compatibility constraint with Thanos, Prometheus is explicitly configured to use only the `PrometheusText0.0.4` scrape protocol. Attempting to use `PrometheusText1.0.0` or not explicitly specifying the protocols results in runtime errors. This restriction is applied uniformly to all scrape jobs via the `scrapeProtocols` directive in the `values.yaml` file:

```
scrapeProtocols:
  - PrometheusText0.0.4
```

All monitored components must expose metrics in this protocol version to ensure compatibility with the `thanos-sidecar` and `thanos-store-gateway` components.

5.2.3 Retention Policy

Prometheus is configured with a default data retention period of **10 days**. This setting limits the duration for which metrics are stored locally, after which they are exported by Thanos to the MinIO bucket.

5.2.4 Metric Format and Conventions

All scraped metrics are expected to conform to Prometheus standards:

- Metric names shall be in `snake_case`
- Unit suffixes (e.g., `seconds`, `bytes`) must be included
- Labels must use lowercase alphanumeric characters, underscores, and hyphens
- High-cardinality or dynamic labels (e.g., pod UID) should be avoided where possible

5.2.5 Custom Metrics Integration

In addition to infrastructure-level metrics, this Prometheus instance supports the collection of custom application-specific metrics from LISA workflows. There are currently two supported methods for introducing such metrics:

Workflow-Defined Metrics

Custom metrics can be defined directly within the Argo workflow YAML template under the `spec` field, for example:

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: lisa-workflow-custom-metrics
  namespace: scientific-workflow
spec:
  metrics:
    prometheus:
      - name: duration_gauge
        labels:
          - key: name
            value: "{{workflow.name}}"
        help: "Duration gauge by name"
        gauge:
          realtime: true
          value: "{{workflow.duration}}"
```

This allows workflow authors to specify and expose any runtime metrics relevant to their execution logic without having to modify the Pipeline Runner source code.

Backend-Defined Metrics

Alternatively, metrics can be injected programmatically from within the Pipeline Runner backend source code. Currently, there is a single example of this approach: a custom Prometheus metric that exposes the name of the workflow author. This metric is inserted into the Argo API call directly by the backend service like so:

```
const argoWorkflow = {
     apiVersion: 'argoproj.io/v1alpha1',
     kind: 'Workflow',
     metadata: {
       name: generatedName,
     },
     spec: {
       },
       metrics: {
         prometheus: [
           ...(templateSpec.spec?.metrics?.prometheus ||
           {
             gauge: {
               realtime: true,
               value: '{{workflow.duration}}',
             },
             help: 'The user who created the workflow',
             labels: [
               {
                 key: 'author_id',
                 value: req.user.email,
               },
               {
                 key: 'workflow_name',
                 value: generatedName,
               },
             ],
             name: 'created_by',
           },
         ],
       },
     },
   };
```

5.3 Grafana

5.3.1 Grafana Interfaces

Grafana is the primary visualization component within the monitoring subsystem. It provides interactive dashboards for real-time and historical monitoring of Argo workflows, node-level and cluster-level metrics. Grafana is deployed as a standalone application in the `scientific-workflow` namespace.

5.3.2 Authentication

Grafana comes preconfigured with basic authentication enabled by default, using the standard credentials `admin:admin`. This default login can be changed either through the UI on first login or directly in the configuration files or source code before deployment. For enhanced security and access control, Grafana supports role-based authentication, allowing administrators to create users with different permission levels: Admin, Editor, and Viewer. Additional users can be created manually through the Grafana UI or programmatically through the API to ensure that sensitive configuration or data is only accessible to authorized users.

5.3.3 Data Source Configuration

Grafana is connected to the following data sources:

- **Prometheus**: used for short-term metrics
- **Thanos**: used for long-term metric queries and historical data
- **Loki**: used for log aggregation

Data sources are configured inside the `deployment.yaml` file ConfigMap in the `grafana` directory. Connections with the data sources are made using HTTP, and no authentication is enforced within the internal cluster network.

5.3.4 Templating and Workflow Filtering

To support dynamic querying in dashboards, Grafana utilizes templating variables. The key variable used to isolate workflow-specific data is `$Workflow`, which is defined globally across all workflow monitoring dashboards.

This variable is configured to retrieve values from the pod labels where the value begins with the prefix `lisa-workflow-`. This variable allows to differentiate between infrastructure pods and workflow pods.

5.3.5 Dashboard Structure and Naming Conventions

All Grafana dashboards in this project are pre-configured and version-controlled. They are stored in the project repository inside the `grafana/dashboards` directory. These dashboards are automatically loaded into Grafana during deployment.

The following four dashboards are included by default:

- **Main Dashboard**: Provides a general overview of the system, including workflow statuses, currently running workflow count and infrastructure error logs.
- **Kubernetes Dashboard**: Displays metrics related to cluster health. Has different filtering options based on nodes, instances or pods.
- **Argo Dashboard**: Focuses on Argo Workflow execution metrics, including workflow authors, logs, statuses and resource consumption.
- **Historical Dashboard**: Retrieves long-term resource metric consumption via Thanos, allowing comparison and trend analysis beyond Prometheus's default retention window.

When new dashboards are created through the Grafana UI, they can be exported as JSON files and added to the `grafana/dashboards` directory in the source code.

5.4 Thanos

Thanos is deployed using Prometheus Operator as a part of the monitoring subsystem to extend Prometheus with long-term metric storage and historical data access. It is deployed in the `scientific-workflow` namespace and consists of several cooperating components: Sidecar, Store Gateway, Query, and Compactor. All components interact using gRPC and HTTP interfaces and are configured to use a dedicated `prometheus-data` MinIO bucket as the object storage.

5.4.1 Prometheus Integration

Each Prometheus instance is accompanied by a Thanos Sidecar, which serves two primary functions:

1. Exposes Prometheus's TSDB data to the Thanos Query and Store components via gRPC
2. Uploads block-based metric data to MinIO in a format compatible with the Thanos Store Gateway

To ensure compatibility with Thanos Sidecar, Prometheus is explicitly configured to use the `PrometheusText0.0.4` scrape protocol (see Section 4.2). The Sidecar component exposes gRPC endpoints that allow the Thanos Query component to retrieve both live and historical data.

5.4.2 Object Storage

Thanos uses MinIO as an object storage. The Thanos Sidecar uploads metric blocks to MinIO at regular intervals, and the Thanos Compactor and Store Gateway retrieve data.

Object storage is configured in the `objstore.yml` file in the `thanos` directory. All Thanos components use internal cluster networking to communicate with MinIO.

5.4.3 Grafana Integration

The Thanos Query component provides a Prometheus-compatible HTTP API that collects data from Prometheus instances and long-term storage through the Store Gateway. The Grafana historical dashboard is configured to use Thanos as a data source for metrics beyond the 10-day retention window of Prometheus.

The Thanos Query API is added as a Grafana data source with a base URL `http://thanos-query:9090`, allowing transparent access to historical data via the same query language that Prometheus uses (PromQL).

5.4.4 Component Responsibilities

Each Thanos component has a specific role:

- **Sidecar**: Connects to Prometheus and uploads TSDB blocks to the object storage
- **Store Gateway**: Reads the metric blocks from object storage and exposes them through a gRPC endpoint
- **Compactor**: Performs downsampling and deduplication of stored metric blocks
- **Query**: Provides a query interface across all data sources (Prometheus instances + Store)

All components communicate within the `scientific-workflow` namespace using gRPC and HTTP APIs.

5.5 Loki

Loki is used within the monitoring dashboard subsystem to collect and query logs from Kubernetes. It provides lightweight, label-based log aggregation that is aligned with Prometheus's data model. It is deployed in the `scientific-workflow` namespace and is integrated with Grafana.

5.5.1 Log Ingestion

Logs are collected using Promtail, which runs as a DaemonSet on each Kubernetes node. Promtail reads logs from pod log files under `/var/log/containers` and forwards them to Loki over HTTP. Each log entry is labeled with key metadata:

- `namespace`
- `pod`
- `container`
- `workflow` (if applicable)

This label-based approach allows logs to be filtered and queried in a similar fashion to Prometheus metrics.

5.5.2 Grafana Integration

Grafana is configured with Loki as a data source and uses it in both dashboard panels and the Explore view. Log queries can be dynamically filtered using the `$Workflow` variable to view logs specific to a specific workflow execution.

End of Document

Appendix C

Workflow Templates

The following Argo workflow templates were used for the evaluation of the dashboard.

OOMKill Workflow Template

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: oomkill
  namespace: scientific-workflow
spec:
  entrypoint: cause-oom
  templates:
  - name: cause-oom
    container:
      image: python:3.10
      command: ["python"]
      args:
        - -c
        - |
          import time
          a = []
          while True:
              a.append(' ' * 10**6)
              time.sleep(0.1)
      resources:
        limits:
          memory: 64Mi
          cpu: "0.1"
  arguments:
    parameters:
      - name: artifactOutputPath
        value: /tmp
```

Missing Output Path Workflow

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: output-missing
  namespace: scientific-workflow
spec:
  entrypoint: bad-output
  templates:
  - name: bad-output
    outputs:
      artifacts:
      - name: result
        path: /tmp/result.txt
    container:
      image: alpine
      command: ["sh", "-c"]
      args: ["echo 'This won't create /tmp/result.txt'; sleep 60"]
      resources:
        limits:
          memory: "128Mi"
          cpu: "250m"
        requests:
          memory: "64Mi"
          cpu: "100m"
  arguments:
    parameters:
      - name: artifactOutputPath
        value: /tmp
```

Error Exit Workflow

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: exit-code-error
  namespace: scientific-workflow
spec:
  entrypoint: fail-step
  templates:
  - name: fail-step
    container:
      image: alpine
      command: ["sh", "-c"]
      args: ["echo 'Failing soon...'; sleep 60; exit 1"]
      resources:
        requests:
```

```
          memory: 64Mi
          cpu: "0.1"
        limits:
          memory: 128Mi
          cpu: "0.5"
  arguments:
    parameters:
      - name: artifactOutputPath
        value: /tmp
```

Long-Running Workflow

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: lisa-workflow-with-metric
  namespace: scientific-workflow
spec:
  metrics:
    prometheus:
      - name: duration_gauge
        labels:
          - key: name
            value: "{{workflow.name}}"
        help: "Duration gauge by name"
        gauge:
          realtime: true
          value: "{{workflow.duration}}"
  arguments:
    parameters:
      - name: artifactOutputPath
        value: /tmp
  ttlSecondsAfterFinished: 300
  entrypoint: long-task
  templates:
    - name: long-task
      container:
        image: busybox
        command: [sh, -c]
        args:
          - |
            echo "Starting long-running task...";
            for i in $(seq 1 60); do
              echo "Tick $i";
              sleep 5;
            done;
            echo "Finished!";
        resources:
          requests:
```

```
            memory: 64Mi
            cpu: "0.1"
          limits:
            memory: 80Mi
            cpu: "0.125"
```

Higher CPU and RAM Load

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  name: simulate-macos-load
spec:
  arguments:
    parameters:
      - name: artifactOutputPath
        value: /tmp
  entrypoint: simulate
  templates:
    - name: simulate
      container:
        image: ubuntu:22.04
        command: [sh, -c]
        args:
          - |
            apt-get update && \
            apt-get install -y stress-ng && \
            echo "Starting 80% CPU and memory load..." && \
            stress-ng \
              --cpu 1 \
              --cpu-load 80 \
              --vm 1 \
              --vm-bytes 480M \
              --timeout 6m && \
            echo "Done. Sleeping to let Prometheus scrape..." && \
            sleep 180
        resources:
          limits:
            cpu: "1000m"
            memory: "600Mi"
          requests:
            cpu: "1000m"
            memory: "600Mi"
```

Appendix D

Repository

The repository for the code of this dissertation is available at
`https://github.com/MaryJaneLV/LISA-pipeline-runner-monitoring-dashboard`

Appendix E

Demo

A demo video of the dashboard in use is available at
https://drive.google.com/drive/folders/12-Bej0R2qRhTTsqIQJNgv7jWIoB87OYJ