D3.2 Data-driven orchestration and monitoring (first release)

Version 1.0

Documentation Information

Contract Number	**101093110**
Project Website	www.extract-project.eu
Contratual Deadline	M15, 31st March 2024
Dissemination Level	Public
Nature	Report
Author	IKERLAN (IKL)
Contributors	BSC, IBM, IKL, SIX
Reviewer	LRI
Keywords	Data, orchestration, monitoring, first-release

Change Log

Version	Description Change
V0.1	Initial draft of the table of contents
V0.2	Distributed Monitoring Architecture
V0.3	Code addition for justification
V0.4	Review (LRI)
V0.5	Apply comments from review (IKL)
V1.0	Ready to submit

The EXTRACT Project has received funding from the European Union's Horizon Europe programme under grant agreement number 101093110.

Table of Contents

1. Introduction 3
1.1. Structure 3
1.2. Relationship with other WPs 3
2. Distributed Monitoring Architecture 4
2.1. Monitoring Components 4
2.1.1. Data Storage 5
2.1.2. Data Collectors 6
2.1.3. Data Transport 8
2.1.4. Data Processing Procedures 8
2.1.5. Visualization 10
2.2. Metrics 12
2.2.1. CPU 13
2.2.2. Memory 14
2.2.3. Networking 16
2.2.4. Storage 17
2.2.5. System 18
2.3. Validation of requirements 18
3. Data-driven Workflow Deployment and Scheduling 20
3.1. Orchestrator architecture 20
3.2. Technology description 22
3.2.1. Kubernetes 22
3.2.2. COMPSs 23
3.2.3. Nuvla 24
3.3. Interaction with Monitoring Platform 24
4. Next Steps 25
4.1. Prometheus Service Discovery 25
4.2. Metric Candidates 26
4.3. Exporter Candidates 26
4.4. Monitoring API 27
4.5. Scheduling algorithms 27
5. Conclusion 28
6. Acronyms and Abbreviations 29
7. References 29

1. Introduction

This document provides a comprehensive account of advancements made thus far within Extract project´s Work Package 3. To be precise, it mainly encompasses the work undertaken in the context of two tasks: T3.2 Data-driven Workflow Deployment and Scheduling, devoted to the development of the EXTRACT orchestrator, and T3.3 Distributed Monitoring Architecture, which aims to develop a monitoring infrastructure that will collect information from the execution of data mining workflows across the compute continuum. D3.2 marks a significant milestone in the ongoing Extract research project, providing a comprehensive account of advancements made thus far.

In order to meet Objective 2 (which addresses the development of novel data-driven orchestration mechanisms to deploy and run data-mining workflows) together with all the corresponding technical objectives, the primary focus of this deliverable lies in detailing a monitoring system designed to capture metrics related to various features of the Extract platform. These metrics serve as crucial inputs for an orchestrator, enabling it to make optimized decisions. The orchestrator itself will be thoroughly described, outlining its components and functionalities. Furthermore, this document delves into the intricate integration between the orchestrator and the monitoring system, offering a holistic understanding of the data-driven orchestration and monitoring framework developed within the project.

1.1. Structure

This document is organized in 4 sections:

- Section 1 introduces the document and gives a main view of the structure of the document.
- Section 2 details the distributed monitoring architecture, enumerating its components, discussing the metrics employed, examining the prerequisites for validation, and outlining the subsequent actions.
- Section 3 covers the implementation and arrangement of the data-driven workflow, outlining the architecture of the orchestrator and explaining the technology utilized.
- Section 4 gives a summary of the conclusions drawn from this document.

The document concludes by listing the acronyms, abbreviations, and bibliography references.

1.2. Relationship with other WPs

Deliverable	Task	Relation
D2.2	T2.3	Data-Mining Framework (WIP)
D3.1	T3.1	Data-driven orchestration requirement specification
D4.1	T4.1	Compute continuum requirement specification and EXTRACT platform integration plan
D4.2	T4.2	Programming and Execution Models Interoperability

Table 1. Relationship with other WPs

2. Distributed Monitoring Architecture

The distributed monitoring architecture was initially proposed in deliverable D3.1. The current section provides further details on the current state of the monitoring architecture. The different technologies used to implement each proposed components are defined and the different metrics that are being monitored are listed. Additionally, the current fulfillment of the requirements presented in deliverable D3.1 are specified which allows identifying the future steps towards the development of the monitoring architecture.

2.1. Monitoring Components

The monitoring architecture proposed in deliverable D3.1 has been updated as seen in Figure 1. The components necessary to implement the monitoring system can be identified and will be explained in more detail in the following subsections.

Indicate that this monitoring system will be deployed in the Kubernetes architecture of the project using Ansible. This software tool provides simple but powerful automation including provisioning, configuration management, application deployment and orchestration. In the following subsections also include playbooks and roles which are the files where the Ansible tool defines the tasks to be executed on the nodes of architecture.

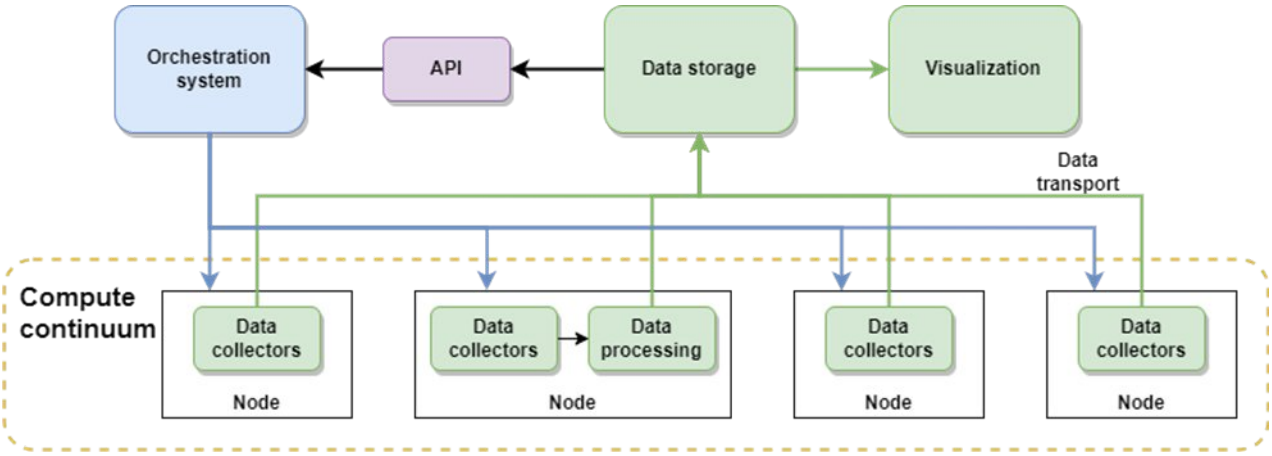

Figure 1 Monitoring Architecture

Prometheus [1] has been used as the cornerstone technology to implement this monitoring system. Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability in modern and dynamic infrastructures, particularly those using container orchestration systems like Kubernetes. The architecture of Prometheus and some of its ecosystem's components are illustrated in the next figure.

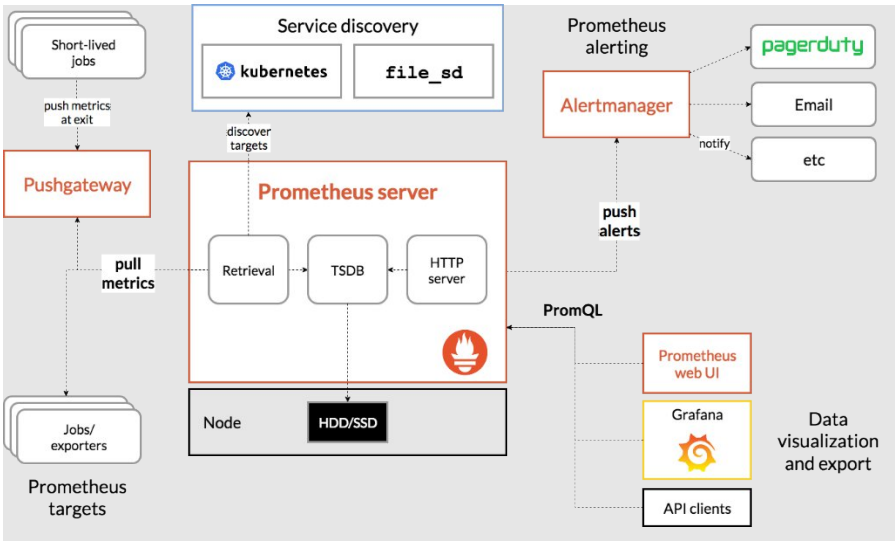

Figure 2 Prometheus Architecture

The following code snippet is a Ansible playbook that deploys the whole monitoring architecture on the Kubernetes infrastructure. The first role installs helm which is a Kubernetes deployment package manager that is required for the rest of the roles.

```
---
- name: Install Prometheus
  hosts: master
  roles:
    - install-helm
    - install-prometheus
    - install-metrics-server
    - install-grafana
```

Prometheus capabilities, combined with other technologies, are leveraged to implement the different components in the monitoring architecture defined in deliverable D3.1. In the following sections, each of these components is detailed, including the corresponding role.

2.1.1. Data Storage

The Prometheus ecosystem includes multiple components but the most important is the Prometheus Server. This service includes a time series database where captured metrics are stored.

To consult or retrieve information from this database, PromQL (Prometheus Query Language) is used. This query language is designed specifically for querying and manipulating time series data collected by Prometheus. Some of the most important features or key aspects that can be highlighted is that PromQL supports instants queries and range queries where aggregation functions and arithmetic and binary operations like addition, subtraction, multiplication, division, and comparisons can be apply. Furthermore, it includes functions for calculating the rate of change and the total increase and functions to work with histogram and summary metrics, which are used to measure the distribution of values.

The following code snippet shows the role for installing Prometheus.

```
---
- name: Add Prometheus Helm chart repository
  kubernetes.core.helm_repository:
    name: prometheus-community
    repo_url: https://prometheus-community.github.io/helm-charts

- name: Install Prometheus Helm chart
  kubernetes.core.helm:
    release_name: my-prometheus
    chart_ref: prometheus-community/prometheus
    chart_version: 22.7.0
    state: present # present / absent: use this to remove installation
    release_namespace: monitoring
    create_namespace: true
    values:
      server:
        service:
          type: NodePort
          nodePort: 31000
```

2.1.2. Data Collectors

The data collectors are basically software components called exporters that collect and expose metrics from system, services, and applications in a format that Prometheus can scrape and store. Therefore, they act as bridges between Prometheus and system monitor allowing to gather information about health, performance, and other relevant metrics.

In this case, the exporters that are used for collecting and exposing the architecture metrics are kube-state-metrics [2] and metrics-server [3]. The first one is used for exposing the state of various Kubernetes objects as metrics. This object includes nodes, pods, services and more and takes information from Kubernetes API server about the current state of these objects and exposes them as Prometheus style metrics. The last one collects resource utilization metrics from the various components of Kubernetes clusters using Kubelets and exposes them as Prometheus style metrics.

In addition, the use of OpenTelemetry [4] for exposing custom metrics is also considered. OpenTelemetry, also known as OTel is an open-source observability framework for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, and logs. In this case this software can be implemented in the different services for exposing custom metrics to Prometheus server. For example, it is possible to export the number of times certain code has been executed or has failed.

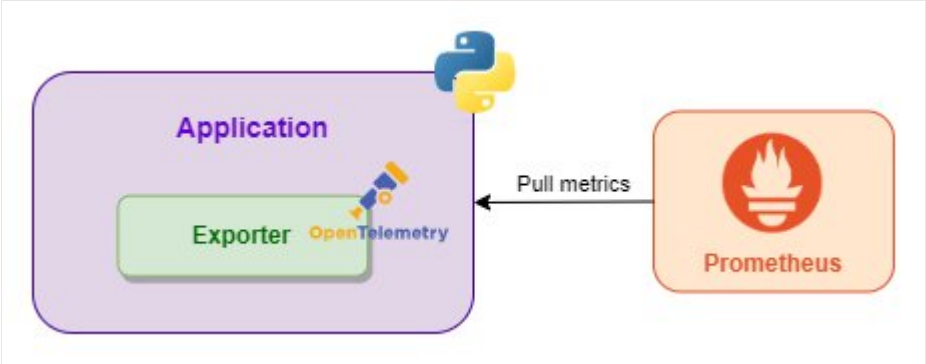

Figure 3 OpenTelemetry as Prometheus Exporter

The following code snippet shows the role for installing metrics-server exporter.

```
---
- name: Add Metrics server Helm chart repository
  kubernetes.core.helm_repository:
    name: metrics-server-repo
    repo_url: https://kubernetes-sigs.github.io/metrics-server/

- name: Install Metrics Server Helm chart
  kubernetes.core.helm:
    release_name: metrics-server
    chart_ref: metrics-server-repo/metrics-server
    chart_version: 3.11.0
    state: present # present / absent: use this to remove installation
    release_namespace: kube-system
    values:
      args:
```

```
        - --kubelet-insecure-tls
```

2.1.3. Data Transport

There are two fundamentally different approaches by which the Prometheus server collects metrics. The "push approach" when the application actively pushes its metrics to designated Prometheus Pushgateway. Then the Pushgateway becomes responsible for exposing metrics over HTTP. And the "pull approach" when Prometheus Server periodically pulls metrics from the exposed HTTP endpoint. Typically, metrics are exposed at a designated endpoint (e.g., '/metrics') and formatted using a simple text-based format that includes key-value pairs, and each metric is associated with a timestamp.

2.1.4. Data Processing Procedures

As already indicated in a previous section there is the possibility of using OpenTelemetry to export custom metrics. The main advantage of its use is the opportunity to implement data processing, analytics, aggregation, and clean before exporting the metrics, although they must maintain the OpenTelemetry protocol data model (OTLP).

The OpenTelemetry metrics data model is defined by a protocol specification and semantic conventions, specifically designed for delivering pre-aggregated metric timeseries data. This model serves a dual purpose: enabling the seamless import of data for existing systems and the effortless export of data into established systems.

2.1.4.1. Open Telemetry example

The following code snippet shows an example of a simple application what uses Open Telemetry to expose a counter that is updated every second with a random value between -100 and 100.

```
import sys
import time
import random
from opentelemetry import trace, metrics
from opentelemetry.sdk.resources import Resource, SERVICE_NAME
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.exporter.prometheus import PrometheusMetricReader
from prometheus_client import start_http_server


# Service name required for the Prometheus
```

```
resource = Resource.create(attributes={SERVICE_NAME: "example_application"})

# Start the Prometheus client
start_http_server(port=8000, addr="0.0.0.0")

# Initialize PrometheusMetricReader which pulls metrics from the SDK
# on-demand to respond to scrape requests
reader = PrometheusMetricReader()
provider = MeterProvider(resource=resource, metric_readers=[reader])
metrics.set_meter_provider(provider)

# Acquire a tracer
tracer = trace.get_tracer("example_aplication.tracer")

# Acquire a meter.
meter = metrics.get_meter("example_aplication.meter")

# Create a random counter
random_meter = meter.create_up_down_counter(
    name="example_aplication.random",
    description="Random number of example application",
)

# Start the application
sys.stdout.flush()
while True:
    # Create a new span
    with tracer.start_as_current_span("do_random") as random_span:
        time.sleep(1)
        random_number = random.randint(-100, 100)
        # Set the random number as an attribute
        random_span.set_attribute("random.value", random_number)
        # Add the random number to the random counter
        random_meter.add(random_number)
```

```
        sys.stdout.flush()
```

2.1.5. Visualization

Prometheus itself does not provide native visualization capabilities so Grafana [5] has been used as monitoring platform to visualize and analyze metrics. Using this visualization tool, it is possible to connect Prometheus as a data source and create comprehensive dashboards. There is also the possibility of importing different dashboards made by the community that already have the queries and charts defined. In this case several dashboards have been imported. On the one hand, dashboards have been imported to visualize the different metrics of the distinct entities that make up a Kubernetes cluster. And on the other hand, other dashboards that allow to view metrics exposed by custom nodes like the Jetson Nano.

The following code snippet shows the role for installing Grafana.

```
---
- name: Add Prometheus Helm chart repository
  kubernetes.core.helm_repository:
    name: grafana-community
    repo_url: https://grafana.github.io/helm-charts

- name: Install Grafana Helm chart
  kubernetes.core.helm:
    release_name: my-grafana
    chart_ref: grafana-community/grafana
    chart_version: 7.0.19
    state: present # present / absent: use this to remove installation
    release_namespace: monitoring
    create_namespace: true
    values:
      adminPassword: extract2024
      datasources:
        datasources.yaml:
          apiVersion: 1
          datasources:
            - name: Prometheus
              type: prometheus
              url: http://my-prometheus-server
              access: proxy
```

```
            isDefault: true
            editable: true
      dashboardProviders:
        dashboardproviders.yaml:
          apiVersion: 1
          providers:
          - name: 'default'
            orgId: 1
            folder: ''
            type: file
            disableDeletion: false
            updateIntervalSeconds: 10
            options:
              path: /var/lib/grafana/dashboards
              folderFromFilesStructure: true
      dashboards:
        default:
          kubernetes-monitoring:
            gnetId: 315
            revision: 3
            datasource: Prometheus
          k8s-system-api-server:
                  url:  https://raw.githubusercontent.com/dotdc/grafana-dashboards-
kubernetes/master/dashboards/k8s-system-api-server.json
            token: ''
          k8s-system-coredns:
                  url:  https://raw.githubusercontent.com/dotdc/grafana-dashboards-
kubernetes/master/dashboards/k8s-system-coredns.json
            token: ''
          k8s-views-global:
                  url:  https://raw.githubusercontent.com/dotdc/grafana-dashboards-
kubernetes/master/dashboards/k8s-views-global.json
            token: ''
          k8s-views-namespaces:
                  url:  https://raw.githubusercontent.com/dotdc/grafana-dashboards-
kubernetes/master/dashboards/k8s-views-namespaces.json
```

```
        token: ''
    k8s-views-nodes:
        url: https://raw.githubusercontent.com/dotdc/grafana-dashboards-kubernetes/master/dashboards/k8s-views-nodes.json
        token: ''
    k8s-views-pods:
        url: https://raw.githubusercontent.com/dotdc/grafana-dashboards-kubernetes/master/dashboards/k8s-views-pods.json
        token: ''
  service:
    type: NodePort
    nodePort: 31001
```

Figure 4 Grafana Kubernetes dashboard

2.2. Metrics

This following chapter contains the metrics already implemented and available in the Prometheus database. Additionally, a set of potential metric candidates are presented in the chapter 2.5.1 to share them with the partners and to decide which metric candidates could be interesting to be implemented.

2.2.1. CPU

The following CPU related metrics have already been implemented. Monitoring CPU usage of nodes helps optimize resource allocation and identify performance bottlenecks.

- **Cluster CPU Usage:** The current usage of the CPUs of the whole cluster. The cluster CPU usage metric is a percent value (0% to 100%). The metric is calculated by dividing the sum of the current used CPU resources by the sum of the current free CPU resources in the cluster. The metric englobes all the nodes running in the cluster and is a metric to get a quick overview on the current CPU usage on the whole cluster. Once identified a possible bottleneck, a more detailed analysis can be done using the more detailed metrics.

Query:

```
sum (rate (container_cpu_usage_seconds_total {id="/", kubernetes_io_hostname = ~"^.*$"}[1m])) / sum ( machine_cpu_cores { kubernetes_io_hostname =~"^.*$"}) * 100
```

- **Containers CPU Usage:** The current CPU usage of each of the Containers. The Container's CPU usage metric is a percentage value whose maximum value depends on the number of cores assigned to the Container (0% to x00% being x the number of cores). The metric is calculated by adding up the current CPU usage value of the Container for each of the cores assigned to the Container.

Query:

```
sum (rate (container_cpu_usage_seconds_total {image!="",name!~"^k8s_.*", kubernetes_io_hostname =~"^.*$"}[1m])) by (kubernetes_io_hostname, name, image)
```

- **Processes CPU Usage:** The current usage of the CPUs by each of the processes. The process CPU usage metric is a percentage value whose maximum value depends on the number of cores assigned to the process (0% to x00% being x the number of cores). The metric is calculated by adding up the current CPU usage value of the process for each of the cores assigned to the process.

Query:

```
sum (rate (container_cpu_usage_seconds_total {id!="/", kubernetes_io_hostname = ~"^.*$" }[1m])) by (id)
```

- **PODs CPU Usage:** The current CPU usage of each of the Kubernetes PODs. The PODs CPU usage metric is a percentage value whose maximum value depends on the number of cores assigned to the Kubernetes POD (0% to x00% being x the number of cores). The metric is calculated by adding up the current CPU usage value of the POD for each of the cores assigned to the POD.

Query:

```
sum (rate (container_cpu_usage_seconds_total {image!="", name=~"^k8s_.*", kubernetes_io_hostname =~"^.*$"}[1m])) by (pod_name)
```

- **System Services CPU Usage:**The current CPU usage of the system services. The System Services CPU usage metric is a percentage value whose maximum value depends on the number of cores used by each of the system services (0% to x00% being x the number of cores). The metric is calculated by adding up the current CPU usage value of each of the System Services.

Query:

```
sum (rate (container_cpu_usage_seconds_total {systemd_service_name!="", kubernetes_io_hostname=~"^.*$"}[1m])) by (systemd_service_name)
```

- **Namespace CPU Usage:** The current CPU usage of a Kubernetes namespace. The namespace CPU usage metric is a percentage value whose maximum value depends on the number of cores used by each of the entities executed in this namespace (0% to x00% being x the number of cores). The metric is calculated by adding up the current CPU usage value of each of the entities assigned to the namespace. This metric can used to obtain the CPU usage of an application which is executed on the cluster. For this all entities composing the application needs to be assigned to the same namespace.

2.2.2. Memory

The following memory related metrics have already been implemented. Tracking memory usage and availability of nodes aids in efficient resource allocation and capacity planning.

- **Cluster Memory Usage:** The current usage of memory of the whole cluster. The cluster memory usage metric is a percentage value (0% to 100%). The metric is calculated by dividing the sum of the current used memory by the sum of the current free memory in the cluster. The metric englobes all the nodes running in the cluster and is a metric to get a quick overview on the current memory usage on the whole cluster. Once identified a possible bottleneck, a more detailed analysis can be done using the more detailed metrics.

Query:

```
sum (container_memory_working_set_bytes {id="/",kubernetes_io_hostname=~"^.*$"}) / sum (machine_memory_bytes{kubernetes_io_hostname=~"^.*$"}) * 100
```

- **Containers Memory Usage:** The current usage of memory in each of the containers. The container memory usage metric measures the number of bytes of main memory used by each container. The metric englobes all the containers existing in the cluster and provides a usage value for each of them.

Query:

```
sum (container_memory_working_set_bytes {image!="",name!~"^k8s_.*", kubernetes_io_hostname =~"^.*$"}) by (kubernetes_io_hostname, name, image)
```

- **Processes Memory Usage:** The current usage of memory by each of the processes. The processes memory usage metric measures the number of bytes of main memory used by each process. The metric englobes all the processes running on the cluster and provides a usage value for each of them.

Query:

```
sum (container_memory_working_set_bytes {id!="/",kubernetes_io_hostname=~"^.*$"}) by (id)
```

- **Pods Memory Usage:** The current usage of memory by each of the Kubernetes PODs. The PODs memory usage metric measures the number of bytes of main memory used by each Kubernetes POD. The metric englobes all the PODs running on the cluster and provides a usage value for each of them.

Query:

```
sum (container_memory_working_set_bytes {image!="",name=~"^k8s_.*", kubernetes_io_hostname=~"^.*$"}) by (pod_name)
```

- **System Services Memory Usage:** The current memory usage of the system services on the nodes. The system services memory usage metric measures the number of bytes of main memory used by each of the system services. The metric englobes all the system services running on the cluster and provides a usage value for each of them.

Query:

```
sum (rate (container_cpu_usage_seconds_total {systemd_service_name!="", kubernetes_io_hostname =~"^.*$"}[1m])) by (systemd_service_name)
```

- **Namespace Memory Usage:** The current memory usage of a Kubernetes namespace. The namespace memory usage metric measures the number of bytes of main memory used by each of the entities executed in a namespace. The metric is calculated by adding up the current memory usage of each of the entities assigned to the namespace. This metric can used to obtain the memory usage of an application which is executed on the cluster. For this all entities composing the application needs to be assigned to the same namespace.

Query:

```
sum (container_memory_working_set_bytes {container!="", kubernetes_io_hostname=~"^.*$"}) by (namespace)
```

2.2.3. Networking

The following network related metrics have already been implemented. Monitoring network traffic and throughput between nodes helps optimize data transfer and identify network-related issues.

- **Cluster Network I/O Pressure:** The current amount of incoming and outgoing network traffic. The cluster network I/O pressure metric measures the total number of bytes per second transferred through the network by the applications executed on the cluster.

Query:

```
sum (rate (container_network_receive_bytes_total {kubernetes_io_hostname=~"^.*$" }[1m]))
```

```
sum (rate (container_network_transmit_bytes_total {kubernetes_io_hostname=~"^.*$"} [1m]))
```

- **Containers Network I/O Pressure:** The current network usage of each of the containers. The containers network I/O pressure metric measures the number of bytes per second transferred through the network by each of the containers executed on the cluster.

Query:

```
sum (rate (container_network_receive_bytes_total{image!="",name!~"^k8s_.*", kubernetes_io_hostname=~"^.*$"}[1m])) by (kubernetes_io_hostname, name, image)
```

```
- sum (rate (container_network_transmit_bytes_total{image!="",name!~"^k8s_.*", kubernetes_io_hostname=~"^.*$"}[1m])) by (kubernetes_io_hostname, name, image)
```

- **Processes Network I/O Pressure:** The current network usage of each of the processes. The processes network I/O pressure metric measures the number of bytes per second transferred through the network by each of the processes executed on the cluster.

Query:

```
sum (rate (container_network_receive_bytes_total{id!="/",
kubernetes_io_hostname =~"^.*$"}[1m])) by (id)
```

```
- sum (rate (container_network_transmit_bytes_total{id!="/",
kubernetes_io_hostname =~"^.*$"}[1m])) by (id)
```

- **Pods Network I/O Pressure:**The current network usage of each of the PODs. The PODs network I/O pressure metric measures the number of bytes per second transferred through the network by each of the Kubernetes PODs executed on the cluster.

Query:

```
sum (rate
(container_network_receive_bytes_total{image!="",name=~"^k8s_.*",
kubernetes_io_hostname=~"^.*$"}[1m])) by (pod_name)
```

```
sum (rate
(container_network_transmit_bytes_total{image!="",name=~"^k8s_.*",
kubernetes_io_hostname=~"^.*$"}[1m])) by (pod_name)
```

- **Namespace Networks I/O Pressure:** The current network usage of a Kubernetes namespace. The namespace network usage metric measures the number of incoming and outgoing bytes per second transferred over the network by each of the entities executed in a namespace. The metric is calculated by adding up the current bytes transferred by each of the entities assigned to the namespace. This metric can used to obtain the current network usage of an application which is executed on the cluster. For this all entities composing the application needs to be assigned to the same namespace.

Query:

```
sum (rate
(container_network_receive_bytes_total{container!="",name=~"^k8s_.*",
kubernetes_io_hostname=~"^.*$"}[1m])) by (namespace)
```

```
sum (rate
(container_network_transmit_bytes_total{container!="",name=~"^k8s_.*",
kubernetes_io_hostname=~"^.*$"}[1m])) by (namespace)
```

2.2.4. Storage

The following storage related metrics have already been implemented. Keeping track of storage usage on nodes ensures efficient allocation and helps identify capacity constraints.The following storage related metrics have already been implemented. Keeping track of storage usage on nodes ensures efficient allocation and helps identify capacity constraints.

- **Node disk throughput:** The current throughput of the filesystem on node level. The node disk throughput metric measures the number of bytes per second read or written to the disk by each of the nodes executed of the cluster.

Query:

```
sum (rate (node_disk_io_now{}[$__rate_interval])) by (node)
```

- **Namespace disk throughput**: The accumulated throughput on namespace level. The namespace disk throughput metric measures the number of bytes per read or written to the disk by each of the entities executed in a namespace. The metric is calculated by adding up the current bytes read or written by each of the entities assigned to the namespace. This metric can used to obtain the current disk throughput of an application which is executed on the cluster. For this all entities composing the application needs to be assigned to the same namespace.

Query:

```
sum (rate (node_disk_io_now{}[$__rate_interval])) by (namespace)
```

2.2.5. System

The following system related metrics have already been implemented. The system needs to be dynamic to adapt its deployment to these infrastructure availability changes; therefore, it is important to monitor the available compute nodes.

- **Available compute nodes:** The current number of compute nodes available on the cluster. This metric changes for example, when there are infrastructure availability changes on the cluster or new infrastructure is added.

Query:

```
count (count by (node) (kube_node_info{cluster=""}))
```

2.3. Validation of requirements

This chapter will review the requirements defined in deliverable D3.1, section 4. “Monitoring requirements” to validate that they have been considered during the development of the architecture of the monitoring system for the EXTRACT platform.

According to deliverable D3.1, the main objective of the monitoring system is to collect, process, store, and report information about the operation, the status, and the resources of the compute continuum. This information will be used by the orchestrator and other EXTRACT applications to optimize performance, availability, scalability, security, and management of the compute continuum and associated applications.

The following table shows the list of the previously defined requirements and the tools that have been used to fulfill them, as well as an update of the implementation status. More detailed information on the requirement and why it is necessary for the system can be found in section 4 of deliverable D3.1.

Monitoring requirements		
Requirement	**Tool used to fulfill the requirement**	**Implementation status**
Near real-time monitoring	Prometheus	Implemented
Flexibility and extensibility	Prometheus + Open Telemetry	Implemented
Integration	API	Not Implemented
Historical Data	Prometheus (time series database)	Implemented
Scalability and efficiency	Related to architecture	Implemented
Reliability	Related to architecture	Implemented
Security and compliance	Prometheus	Implemented

Table 2. Monitoring requirements and tools

On the other hand, the requirements of the metrics to be met by the monitoring agents were also defined. The following table shows the list of the metric requirements, as well as an update of the implementation status. More detailed information on the requirement and why it is necessary for the system can be found in section 4.2. of deliverable D3.1.

Metric requirements		
Requirement	**Tool used to fulfill the requirement**	**Implementation status**
Metrics related to the nodes and the infrastructure of the compute continuum:		
Available nodes	kube-state-metrics metrics-server	Implemented
CPU usage		Implemented
Memory usage		Implemented
Network		Implemented

Throughput		
Storage Utilization		Partially Implemented. Throughput related metrics have been implemented. Storage usage metrics are still pending to be implemented.
Node Health	TBD	Not Implemented
Workload Distribution		Not Implemented
Resource Efficiency		Not Implemented
Metrics related to the applications and systems deployed:		
Application Availability	TBD	Not Implemented
Application Response Time		Not Implemented
Container Metrics	kube-state-metrics metrics-server	Implemented
Network Throughput		Implemented
Storage Utilization		Partially Implemented. Throughput related metrics have been implemented. Storage usage metrics are still pending to be implemented.
Workload-specific Metrics	TBD	Not Implemented
Service Health		Not Implemented
Node Health		Not Implemented
Workload Distribution		Not Implemented
Resource Efficiency		Not Implemented

Table 3. Metric requirements and tools

3. Data-driven Workflow Deployment and Scheduling

3.1. Orchestrator architecture

As stated in D3.1, the EXTRACT platform will be composed of two orchestration layers: the application layer and the infrastructure layer. The application layer is in charge of scheduling the data-driven workflow and the infrastructure layer of its deployment.

In the EXTRACT context, the application layer is implemented with COMPSs, which is in charge of creating the Task Dependency Graph and executing the tasks of the workflows in its workers. On the other hand, the infrastructure layer is implemented with Kubernetes and is in charge of deploying COMPSs as Pods and ensuring the full workflow execution (e.g. Pods restarting) and allowing for Pods communication among them.

It is worth mentioning that both, COMPSs and Kubernetes, have their own scheduler and orchestrator, and it may be easy to confuse the terms. In order to have a clearer vocabulary, we propose the next definitions:

- **Application scheduler:** The COMPSs scheduler that decides in which node a task of the TDG will be executed, taking into account network and data-locality. The default COMPSs scheduler is *es.bsc.compss.scheduler.orderstrict.fifo.FifoTS*, that prioritizes task generation order in FIFO.

- **Application orchestrator:** The COMPSs orchestrator that is in charge of offloading a task to the decided COMPSs worker.

- **Infrastructure scheduler:** The Kubernetes scheduler that will determine where the COMPSs master and workers are deployed. The default scheduler is *kube-sched* and affinity and anti-affinity rules are used such that every worker is deployed in a different node.

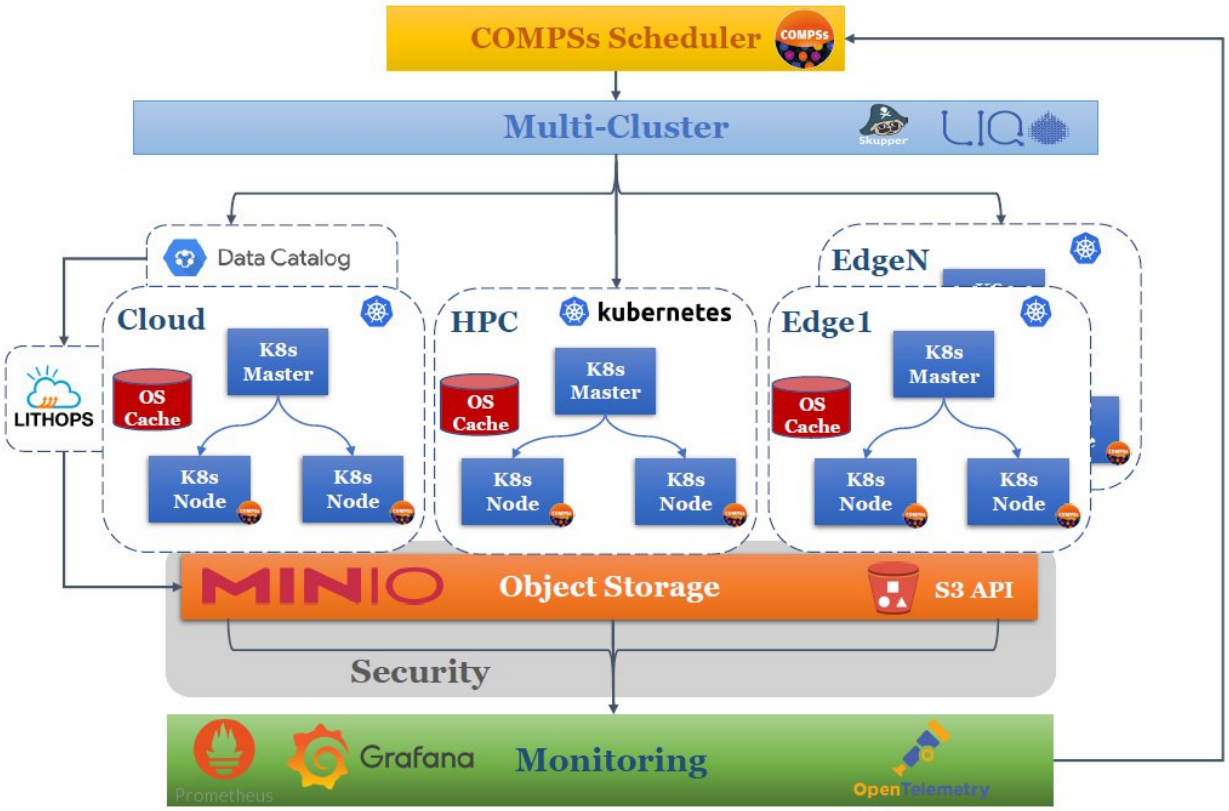

Figure 5 Orchestration Architecture Overview

- **Infrastructure orchestrator:** The Kubernetes orchestrator is a broader term that encompasses the entire Kubernetes platform, which is responsible for managing and coordinating the containerized workloads across the continuum.
- **Monitoring**: The monitoring layer described in detail in the previous sections, plays a pivotal role in providing real-time insights into the system's health and performance. It collects metrics related to resource utilization, application performance, and system state, which are then fed back to the scheduler. This feedback loop enables to adjust the scheduling strategies dynamically, prioritizing tasks based on the current operational context and resource availability.

3.2. Technology description

3.2.1. Kubernetes

Kubernetes is a commonly used open-source platform designed for automating the deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. As a highly flexible container orchestration tool, it enables the efficient handling of workloads by using the concept of pods, which are the smallest deployable units that can be created and managed in Kubernetes.

The integration of Kubernetes clusters with a global, multi-cluster orchestrator/scheduler for EXTRACT such as COMPSs poses a conflict for the scheduling of tasks. Naturally, a task requires resources such as CPU/GPU/memory/storage/network to be available for it out of a Kubernetes cluster when the task needs to run. Thus, the task needs to to be scheduled for execution in the cluster. On one hand, COMPSs needs to be the scheduler of the task because it orchestrates the overall workflow that the task belongs to and may have sensitive timing constraints, as discussed in D3.1 []. On the other hand, a Kubernetes cluster is equipped with a native scheduler [] of its own, which facilitates usage of the cluster's entire resource pool, as well as enabling additional capabilities such as high-availability etc.

As both schedulers, COMPSs and Kubernetes scheduler, contend for the same pool of resources, there need to be a consistent strategy that allows both of them to work together, while avoiding conflicting decisions. In addition, EXTRACT requirements of real-time execution need to be observed.

There are two common strategies that have been considered for resolving the above issue. One is *delegation*, in which COMPSs serves as a top-level orchestrator and scheduler. When COMPSs needs to execute a task in a particular cluster, it delegates the further scheduling and execution of the task to that cluster, by sending the task to execute as a Kubernetes resource – e.g., a pod, or a Knative Service request []. For a scheduling decision, COMPSs considers the free resource capacities of the entire cluster.

A second common strategy that was considered for scheduler cooperation is *partitioning*. The cluster's resource pool is split into partitions, and each partition is governed by a separate scheduler. In that case, the task is sent by COMPSs to the partition that it controls, so the task may assume any form that fits the design of the COMPSs execution. For a scheduling decision, COMPSs considers the free resource capacities only within its partition.

There are pros and cons to both strategies above. In the process of arriving at a design decision, we consider the following key aspects:

1. Flexibility – how much work is needed to accommodate changes in the cluster size and capacity.
2. Timing overhead – how much extra time (beyond net task execution) is needed to get a task executed in a given cluster.
3. Resource overhead – how much extra resources (beyond what the task requires) are needed to get a task executed in a given cluster.
4. Interference – how much may regular Kubernetes operation interfere with the EXTRACT scheduling and execution.

Note that isolation, while being a common property of containers, is not considered. This is because EXTRACT is not about multi-tenancy. In other words, an EXTRACT application is assumed to be a single tenancy domain, using the architecture exclusively for its own purposes.

We now consider delegation in light of the above aspects. Flexibility is a clear advantage, since the entire cluster always remains a single worker for COMPSs, capable of executing concurrent tasks up to the cluster's capacity, regardless of changes. There is some limitation here, given that resources are split across multiple nodes, but this is typically an issue at high load and can be mitigated. On the other hand, the other aspects are disadvantages. Timing overhead: spawning tasks as pods can be slow due to e.g., container warm-up, which can be mitigated to a degree by cluster tuning or by using Knative Serving with minimal pool size. Similarly, container may pose significant resource overhead beyond a task's dependency due to mandatory OS and platform libraries. Last, invoking operations on the Kubernetes clusters not through COMPSs may clearly affect the free resource of the clusters, causing interference - possible scheduling delays and/or failures.

When considering partitioning with relation to the above aspects, we see a different picture. It clearly is more complex in terms of flexibility since the partition needs to be redefined whenever nodes are added or removed from the cluster. One simple way of implementing partitioning is setting up a single long-running COMPSs worker pod in each Kubernetes node, and then sending the task to execute as a thread or a process within that pod. Resource and timing overheads can be quite minimal in this strategy (process or thread overhead). Last, interference is also minimal, since the worker pod is pre-allocated with capacity, so any cluster operation affects outside the pod.

To conclude this discussion, it is now clear that partitioning is the superior strategy for meeting EXTRACT requirements. It is therefore selected for use going forward with EXTRACT implementation. The actual implementation is as suggested above – a single worker pod for COMPSs in each Kubernetes node. Further services or components in the cluster that need to be invoked as part of the task execution should also be pre-allocated to minimize interference to a desired degree.

3.2.2. COMPSs

COMPSs is a task-based parallel computing model developed by BSC, that efficiently schedules tasks across the entire compute continuum. COMPSs specializes in optimizing task execution by dynamically scheduling and placing tasks based on data locality and computing resources, thereby enhancing performance and scalability. COMPSs was already introduced with a more detailed description in D3.1. Furthermore, in D4.2, we provide an extensive explanation on how COMPSs is being used in a first MVP in which it acts as the application orchestrator.

3.2.3. Nuvla

Nuvla is an edge and a container management platform built upon open-source software and open standards. The Nuvla platform allows you to configure any number of Container-as-a-Service (CaaS) (e.g. Docker Swarm, Kubernetes) endpoints. This means you can mix and match public clouds, private clouds and infrastructure, as well as edge devices.

The Nuvla platform exposes a powerful REST API. This API allows developers to integrate Nuvla into third-party systems, script it and even use it as Infrastructure as code (IaC). This enables a simple and effective edge-to-multi-cloud solution. The platform is application centric, hardware agnostic, cloud neutral and container native. This allows end users to manage any containerized application across a fleet of edge devices and container-orchestration engines.

In addition, Nuvla supports a data management platform that leverages the positive attributes of S3-based services and introduces a comprehensive global management system for metadata. The goal is to enhance the efficiency of search functionalities across different service providers. In terms of implementation, the model consists of three core Nuvla resources:

1. data-object: This resource acts as a proxy for data stored in an S3 bucket/object from a specific provider. It manages the lifecycle of S3 objects, simplifying data upload and download processes.
2. data-record: This resource allows users to add additional, user-specified metadata for an object. Enabling the attachment of rich, domain-specific metadata to objects enhances the precision of searching for relevant data.
3. data-set: This resource defines dynamic collections of data-object and/or data-record resources through filters. Administrators, managers, or users can define these collections, providing a flexible and customizable approach to data organization.

Collectively, these resources establish a versatile data management framework applicable to a broad range of use cases. The typical workflow involves creating a data-object (implicitly creating the S3 object), optionally adding metadata using a data-record object, and finally, finding and using the relevant data-object resources included in a data set.

Nuvla facilitates the "using" element by binding data types to user applications capable of processing the data, offering seamless integration between data management and application utilization.

3.3. Interaction with Monitoring Platform

This section outlines how the metrics collected by Prometheus will be utilized to inform the system and refine scheduling decisions, thereby enhancing the efficiency and effectiveness of resource utilization across the continuum.

The array of metric candidates discussed in section 2.2 seem a very good place to start in order to use them in the implementation of a scheduling algorithm.

However, the specific set of metrics that will be utilized falls outside of the scope of the MVP, and as a result, the concrete set of metrics and their weights in the decision-making will be selected during the next phase of the project. In any case, COMPSs can leverage different schedulers at this time. Currently, the version we are using for our preliminary tests, does not integrate in its logic the hereby presented metrics, but rather builds on heuristics. These heuristics build a static graph, which means that no run-time aspects are considered in order to dynamically modify the workflow deployment.

To that end, in Extract we want to leverage all the available information that we can gather from the system and its resources, so that the scheduler has the ability to scale both horizontally and vertically in a multitenant environment such as our platform's, meeting the most stringent application demands and striving for the most optimal utilization and non-functional requirements (e.g. timing constraints).

As stated in Section 2.1.3, there are two approaches by which the Prometheus server can collect metrics. COMPSs has been adapted such that an application proactively sends (push policy) its metrics to a designated Prometheus Pushgateway at runtime through the COMPSs framework.

Despite the availability of numerous metrics, the only one currently being sent measures the frequency with which tasks fail to complete within their expected execution time. This expected execution time, determined through either application-specific profiling or dynamically during the runtime of the application, incorporates a margin to counteract the potential temporal variability in data packet delivery across computer networks and telecommunications systems, commonly known as jitter, as well as variability in execution time attributable to the computational load of a task on a specific node. Upon the COMPSs framework pushing metrics to the Pushgateway, it becomes the Pushgateway's responsibility to make these metrics accessible over the HTTP protocol for subsequent retrieval by the Prometheus Server in the "pull approach."

4. Next Steps

The following chapter identifies the next steps and future improvements to what has already been implemented.

4.1. Prometheus Service Discovery

Service discovery in Prometheus provides the capability to automatically identify and monitor services as they dynamically appear or disappear within a system. This feature enables Prometheus to adapt seamlessly to changes in the environment, ensuring continuous monitoring of services. Furthermore, this feature is particularly useful in dynamic environments such as within a Kubernetes architecture.

To solve this challenge, Prometheus supports different service discovery mechanisms, although the most used are File-Based Service Discovery (file_sd) and HTTP Service Discovery (http_sd). Both methods are very similar. The main difference is that with file_sd, a simple modification of the file will notify Prometheus, while http_sd will check changes periodically. With both methods, it will not be necessary to restart the Prometheus server to start monitoring new services.

In addition, since the Extract platform consists of a Kubernetes architecture, there is a specific service discovery tool, native to Kubernetes, for discovering and monitoring new services running in a cluster. Therefore, it will be possible to also use the Kubernetes API to discover new pods and services.

4.2. Metric Candidates

In this section we present a set of metric candidates which could be interesting for the project but have not been implemented yet. The following versions will discuss this list with the partners and decide which metrics should be implemented.

- **CPU Quota**: The CPU consumption limit for an application.
- **Memory Quota**: The memory consumption limit for an application.
- **Disk space availability**: Metrics regarding the available storage space.
- **Node/Application health**: Metrics regarding the current health status of the computing nodes and/or applications.
- **Application response time**: Metrics regarding the response time of the applications.
- **Networking dropped packages**: The amount of dropped packages in incoming and outgoing network traffic.

It should be remarked that most of these metrics can be obtained on an application level or on a physical node level. Whenever possible, both metrics will be provided. For example, health metrics could, on one side, reflect the current application health state (Application up/down) and, on the other side, could reflect the current state of the actual physical nodes available in the system (Node up/down).

There could be relations between both metrics. But it is not mandatory. For example, a fault-tolerant application could be healthy even when there are one or more physical nodes in an unhealthy state.

4.3. Exporter Candidates

In this section we present a set of metrics exporters which could be useful to gather additional metrics from the Extract platform performance.

- **Power consumption exporter**: a set of metrics to measure the power consumption of the system. We will analyze using Scaphandre software to collect this type of information [6].
- **DCGM-Exporter**: an exporter dedicated to monitor the health and performance metrics of NVIDIA GPUs resources. By integrating with Prometheus, it allows for comprehensive monitoring and optimization of GPU usage within the Extract platform.

The addition of these new exporters represents a significant enhancement to the Extract platform's monitoring infrastructure.

4.4. Monitoring API

Currently, the orchestration obtains the data needed from the monitoring architecture using the Standard API offered by Prometheus, the PromQL Query Language. While this approach is perfectly functional, it keeps both the monitoring and orchestration tightly bound to each other, and changing any of them requires a high degree of modification in the other to accommodate for the change.

A loosely coupled relationship between the two will be implemented through a general monitoring API. This will greatly reduce the work effort needed to adapt one of them if the other gets replaced.

4.5. Scheduling algorithms

The current scheduling algorithm does not consider run-time information. Different alternatives will be studied to improve this approach that are based on the monitoring architecture data through the aforementioned API. Some of the alternatives that will be considered are:

- **Deterministic algorithm**: an algorithm in which the set of metrics to be considered are selected, and via some weighing and priorization, a scheduling algorithm is built.
- **Heuristics-based algorithm**: we would like to explore how certain knowledge of the system could be incorporated to this approach to fine-grain tune it.
- **AI algorithm trained with the collected metrics**: by providing a large test bench of collected metrics during different kind of applications executed in our platform, we could explore some training models and compare the results with some of the other approaches.

5. Conclusion

This deliverable encompasses the description of the first release of the Data-driven orchestration and monitoring platform, developed within Work Package 3 (WP3). It mainly covers the tasks performed in two tasks of this work package: T3.2, related to the deployment and scheduling of workflow steps such that various goals are optimized in a holistic manner, and T3.3, focused on the development of a monitoring infrastructure capable of gathering information related to the execution of data mining workflows across the compute continuum for optimized orchestration and deployment decisions. In summary, the described development and integration of the first release demonstrates a coherent execution of Work Package 3 objectives, paving the way for optimized workflow deployment and monitoring capabilities as explained in the presented sections. Section 2 introduces the monitoring platform currently under development within the project's framework. After thoroughly analyzing which metrics could be the most valuable to serve as input for the orchestration algorithm, the selection of metrics that the monitoring platform will target are presented and briefly described. Following this, an assessment of the specific requirements outlined in document D3.1 is presented, in order to showcase that all the tools that have been selected within this project can guarantee their fulfillment. Section 3 delves into the planning and deployment of workflows. Initially, the orchestrator is described, unveiling the selected components and functionalities. Following this, a comprehensive overview of the technologies earmarked for deployment is provided. Then, the interaction with the monitoring platform is meticulously detailed, outlining the seamless integration between the orchestrator and the monitoring infrastructure.

6. Acronyms and Abbreviations

- WP – Work Package
- WPL – Work Package Leader

7. References

[1] "https://prometheus.io/".

[2] "https://github.com/kubernetes/kube-state-metrics".

[3] "https://github.com/kubernetes-sigs/metrics-server".

[4] "https://opentelemetry.io/".

[5] "https://grafana.com/".

[6] "https://github.com/hubblo-org/scaphandre".