EXTRACT

A distributed data-mining software platform for extreme data across the compute continuum

D3.1 Data-driven orchestration requirements

Version 1.0

Documentation Information

Contract Number	**101093110**
Project Website	www.extract-project.eu
Contractual Deadline	M6, June 2023
Dissemination Level	Public
Nature	Report
Author	IKL
Contributors	BSC, IBM, IKL, SIX
Reviewer	BSC
Keywords	Data, orchestration, mining, parallel, programming model

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

EXTRACT
A distributed data-mining software platform for extreme data across the compute continuum

Change Log

Version	Description Change
V0.1	Initial draft of the table of contents
V0.2	Deployment scheduling requirements and COMPSs
V0.3	Introduction and monitoring requirements
V0.4	Added Ray for specific ML/BD tasks, comment, restructured some text
V1.0	Internal Review, BSC

Table of Contents

1. Introduction

This deliverable represents the contributions made for the first stage of the project in Work Package 3 (WP3). It encompasses the tasks undertaken in T3.1, which involved establishing the requirements of the orchestrator and monitoring system for the EXTRACT platform, ultimately aiming to achieve Milestone MS1.

This deliverable mainly focuses on the definition of a data driven orchestration structure and how to integrate it inside EXTRACT project, which forms the fundamental basis for the research activities conducted within EXTRACT, shaping the projects goals and methodologies. It starts with an in-depth analysis of the current state of the art, offering an overview of the main concepts to be considered of a scheduling, deployment and monitoring architecture and exploring various of its approaches.

The following four key points aim to contribute to the specification of an efficient and effective monitoring infrastructure for the data driven orchestrator while utilizing selected open-source orchestration technologies. This infrastructure will be capable of collecting data pertaining to the execution of the data mining workflow throughout the compute continuum, enabling optimized orchestration and deployment decisions.

- Deployment scheduling requirements based on workflow and continuum definition.
- Selection of monitoring requirements. Highlighting the interoperability aspect between monitoring frameworks in various execution environments where workflows can be deployed, the objective is to develop mechanisms for monitoring the edge, cloud, and high-performance computing (HPC) environments.
- Selection of deployment and redeployment mechanisms provided by Nuvla. Extensions related to smart policy placement and workflow considering data location heuristics.
- Integration of security and privacy mechanisms to prevent information leaks that could compromise the data-mining workflow or the continuum infrastructure.

This document aims to provide a specification and description of a scheduling, deployment and monitoring architecture and its workflow inside an Edge-to-Cloud Continuum and extreme data scenario. The established objectives around which the entire WP specification will be executed are described in the following lines:

1. With the objective of obtaining the most accurate and relevant computing resources develop data-driven deployment and scheduling methods across compute continuum, considering the necessities of the data processes and analytics methods, and fulfilling the extreme data characteristics, security and energy requirements.
2. With the objective of effectively monitor the execution of data-mining workflows, develop a monitoring infrastructure for the data driven orchestrator capable of gathering information related to the execution and focusing on

aspects as latency, available resources, performance, security and energy consumption.

1.1 Relationship with other WPs

Deliverable	Task	Relation
D1.1	**T1.1**	D1.1 describes the use-cases and their functional and non-functional requirements.
D2.1	**T2.1**	Technical requirements of the data infrastructure and data the mining framework.
D4.1	**T4.1**	D4.1 describes the Compute Continuum and its requirements.

Table 1: Relationship with other WPs

1.2 Document structure

This document is organized in following sections:

- Section 1 comprises a concise summary of the document, giving a main view of the structure of the document and its contents.
- Section 2 describes the context, giving a general overview of a data driven monitoring infrastructure within a dynamic Edge-to-Cloud integration and an extreme and high-volume data context.
- Section 3 lists the functional deployment and scheduling requirements, including the analysis and selection of the open-source technologies to be used in this work package.
- Section 4 details the monitoring requirements and their components.
- Section 5 describes deployments mechanisms and their integration with the scheduling system, adding functionalities to implement smart policy placement mechanisms, and an interoperability abstraction layer.
- Section 6 provides an overview of the conclusions derived from this document.

The document concludes by listing the acronyms and abbreviations.

2. High level view of the scheduling, deployment and monitoring architecture

In all the components of the Compute Continuum, that is the edge, the cloud or HPC, the deployment and scheduling of applications will be managed by an orchestrator. The orchestrator will be responsible for managing the deployment of applications,

including scaling up and down, and scheduling tasks on available resources. In addition, it will also be responsible for monitoring the state of the applications and the resources they are using in order to replicate or revive pods in case anything were to happen. To do so, it will need to take into account the available resources, including CPU, memory, and storage, as well as any additional constraints imposed on these resources, such as a given factor of utilization or time-based constraints. Additionally, some devices may have limited storage capacity, which could impact the ability to deploy certain applications. Similarly, some devices may have limited processing power or memory, which could impact the performance of certain applications.

In addition to resource constraints, the orchestrator will also need to consider the requirements of the applications being deployed. This could include factors such as the type of application, the amount of data being processed, and the expected execution time. By considering these factors, the orchestrator can ensure that applications are deployed in a way that meets their specific requirements.

To ensure optimal performance, the orchestrator will also need to consider factors such as network latency and bandwidth too. This will be particularly important in those cases where data needs to be transferred between the edge and the cloud or the HPC components of the compute continuum. By minimizing network latency and maximizing bandwidth, the orchestrator can help to ensure that data is transferred quickly and efficiently.

It is important to remark that in EXTRACT, the orchestration will be formed by two different entities. We will have an orchestrator at application-level and a platform-level orchestrator. In Section 3.3 of this document, we focus on the application-level orchestrator, whereas platform-level orchestration technologies are further described in Deliverable D4.1.

To enhance parallelism, a parallel programming model or analytics framework will be used. This will enable the development of applications that can be executed in parallel across multiple devices, allowing for faster and more efficient processing. This analytics framework will be responsible for describing how a certain application or task can exploit its parallel activities, by defining what functions can be executed in parallel by different resources while managing data dependencies at the same time. This programming model can be seen as the application-level orchestrator, as it will define how a task can optimally be executed in a distributed environment. The actual deployment of the resulting application described by the parallel programming model will be done by the platform-level orchestrator, which will see the described analytics as one more task to deploy, managing the scheduling and execution of all tasks across the available resources accordingly.

The platform-level orchestrator might only expose a subset of the available resources to the programming model. The rationale for doing so, is that not all the clusters or nodes of the entire continuum might be suitable for a given task. The criteria will be based on 3 factors:

1. Data location: especially in the case of data-intensive tasks, the location of where the data is stored will be taken into account to potentially minimize data transfers from distant clusters and hence minimize unnecessary latencies.
2. Nature of the task: certain tasks might need specific hardware for an optimal execution. An example would be a cluster/node with GPUs for a ML application.
3. Environment: information about the utilization of clusters/nodes will be used to take better scheduling and deployment decisions. This information will be gathered by the monitoring system.

The nodes will communicate with each other through a local network infrastructure, such as Ethernet or Wi-Fi, depending on the specific deployment in the edge and its most convenient topology. The communication can be facilitated by a master node that manages the edge nodes and orchestrates the communication between them, which can be located in the edge itself or in the cloud or HPC, especially if the bandwidth between the cloud or HPC clusters and the edge allows so. A multi-cluster overlay network will be responsible for managing the communication between the edge nodes and the cloud or HPC resources, as further detailed in D4.1. The edge devices and cloud service endpoints within the edge-to-cloud continuum often exist in diverse network domains. This includes restricted domains where devices and equipment are not readily available for remote access, such as those behind NAT or within LTE/5G RAN networks. To address these scenarios, the deployment and lifecycle management of applications in restricted edge networks require the capability to pull workload deployment and management tasks from the platform orchestrator's control plane. At a high level, the deployment and scheduling in the edge will be designed to maximize the utilization of resources and minimize latency, while also ensuring that the applications are deployed in a way that meets their requirements and provides the necessary level of performance.

The EXTRACT project focuses on addressing the compute continuum by integrating Cloud Computing and Internet of Things. This integration encompasses various entities and resources at different levels, including data centers, cloud computing infrastructures, HPCs, edge devices, networks, and on-premise devices. To effectively coordinate and manage the diverse resources and entities within the compute continuum ecosystem, it is necessary to specify and deploy an advanced monitoring system.

The monitoring system developed during the EXTRACT project will manage the collection, processing, storage, and reporting of the status and operation of each resource or entity within the compute continuum. The gathered data will be processed and transmitted to the compute continuum orchestrator.

The monitoring system is a key outcome of the EXTRACT project and will significantly enhance the management and orchestration within the compute continuum. Orchestration enables the coordination and management of different resources, technologies, and services across various stages. By facilitating improved and optimized orchestration, the monitoring system ensures efficient and effective operation within the computing ecosystem.

The monitoring system consists of several components at various levels of the compute continuum: Data Collection, Data Storage, Data Transport, Data Processing, Visualization and Alerting. These components will be implemented using state-of-the-art technologies in accordance with the monitoring component's requirements and metrics.

The monitoring platform will offer near real-time monitoring of the system, providing scalable, efficient, flexible, and extensible mechanisms for monitoring. It will persist historical data related to monitored metrics, which can be accessed through APIs or hooks to seamlessly integrate with the orchestrator and any EXTRACT application that has permissions to access them. Additionally, the monitoring platform will prioritize reliability, security, and compliance of the data and system. In terms of metrics, the monitoring platform will focus on two areas: (a) nodes and infrastructure within the compute continuum, and (b) deployed applications and systems. The monitoring platform will integrate with the compute continuum ecosystem and it will be designed and developed with interoperability and integration on mind.

Section 4 provides a more detailed description of the requirements, metrics and interoperability of the monitoring system to be developed in the EXTRACT project.

3. Deployment scheduling requirements

3.1. General Requirements of the scheduling subsystem

EXTRACT addresses a so-called compute continuum, that is, a distributed computing paradigm that spans multiple computing domains, such as edge devices, cloud infrastructure, and high-performance computing (HPC) resources. By leveraging the strengths of each computing domain, a compute continuum can deliver high-performance computing capabilities to a wide range of applications, from data processing and analysis to real-time critical tasks, and to machine learning and artificial intelligence. The main challenge in building a compute continuum is to ensure that data and processing resources are seamlessly integrated and can communicate with each other efficiently and securely. Furthermore, that processing units are effectively used aiming at optimal performance. To guarantee that, the following are general requirements which the deployment should count on:

- **Scalability**: Applications deployed in an Edge-Cloud-HPC environment need to be able to scale up/down or in/out dynamically based on changing demands. This requires a flexible and scalable infrastructure that can handle spikes in traffic and workload without a significant loss in performance or effect on other functional and non-functional requirements. Automated resource reduction such as scale-in/down and auto-deletion is required for releasing resources from each application to others. Also, for public cloud or other pay-per-use infrastructures, releasing resources helps reduce operating costs.

- **Resilience**: Applications must be designed to withstand failures and disruptions that may occur in the compute continuum. This requires redundancy, fault tolerance, and disaster recovery mechanisms that can ensure high availability and data integrity.
- **Security**: Security is a critical requirement in any EXTRACT application or use case where sensible data may be collected at the edge. Applications must be designed to protect against cyber-attacks and data breaches, and must include measures such as encryption, access control, and identity management to ensure data confidentiality, integrity and availability. For orchestration and scheduling, this implies secure communication between the orchestrator / scheduler and the workers.
- **Interoperability**: Given that different applications across the compute continuum might use different technology stacks, they must be able to interoperate with each other and with the rest of the systems in the environment. This requires adherence to uniform APIs and protocols, as well as the use of middleware and integration technologies that can facilitate data exchange and communication between different components. This is a key requirement especially when selecting what technology to use at each layer and possibly different implementations at distinct locations (edge/cloud/HPC).
- **Performance**: both use cases addressed in EXTRACT must be able to deliver high performance and low latency to meet the potential real-time demands. This requires not only optimization of the application code, which falls outside of the scope of EXTRACT, but the use of efficient data processing technologies and performance-enhancing computation too. High volume and speed characteristics are effectively addressed by the use of High-Performance Computing technologies, which support massive parallel processing capabilities and advanced acceleration features (e.g., GPU, FPGA, many-core fabrics or AI-cores). However, HPC systems are not suitable for: (1) real-time operations and geographically disperse data sources, due to the potentially large communication latencies and single-point resource contention; (2) energy-efficient solutions; (3) extreme large-stored volumes. The dispersity of data enforces an interoperable system and data processing at the edge as well.
- **Monitoring and Management**: Finally, applications must be designed to enable effective monitoring and management on both the edge, the cloud and the HPC. The outcome of this monitoring activity can be fed back to take better orchestration and scheduling decisions, as well as help users identify and troubleshoot application issues and allow applications themselves to operate based on metrics. This requires the use of tools and technologies such as log analysis, performance monitoring, and automated alerting to ensure that issues are identified and addressed in a timely manner.

3.2. Analysis of feasible technologies to be used

As already stated, the programming model will be the responsible component to describe the analytics and how a certain application should be optimally executed, for instance exploiting its parallel capabilities. At a later stage, these analytics will be used by the platform orchestrator to actually deploy tasks into the most suitable resources.

As parallel programming models, these are some open-source options that came into consideration:

- **Message Passing Interface (MPI)**: MPI is a widely used model for parallel programming in distributed-memory systems, where multiple processors communicate and synchronize by sending and receiving messages.
- **OpenMP**: a popular shared-memory parallel programming model for multi-core and multi-processor systems that relies on compiler directives to parallelize code.
- **CUDA**: a parallel programming model designed for NVIDIA GPUs that provides a set of APIs and tools to program and optimize GPU-accelerated applications.
- **OpenCL**: allows for the development of cross-platform applications that can be executed on different types of hardware, including CPUs, GPUs, and FPGAs.
- **Apache Spark**: a parallel programming model for large-scale data processing that supports in-memory data processing and offers high-level APIs for building parallel applications.
- **COMPSs** (COMmunity Parallel Support Service): COMPSs is a parallel programming model and runtime system developed at BSC that enables the development of high-performance computing applications for distributed infrastructures. COMPSs is designed to simplify the development of parallel applications by abstracting the complexity of distributed computing and providing a high-level API that allows developers to express parallelism in their applications.
- **Ray:** Ray is an orchestration platform for Python that enables efficient scale-out of Python applications across multiple cores in the same node and across multiple nodes in the same cluster. It supports an annotation-based approach, stateless remote computation (tasks) stateful remote computation (actors), and nested distributed invocations. Ray particularly excels in integration with most major ML platforms such as PyTorch, TensorFlow, XGBoost etc. and offers native support for specific domains such as RL (Reinforcement Learning) and hyperparameter tuning.

  The system takes care of the distribution of tasks and data across the computing infrastructure. This allows developers to write parallel code that can be executed on different distributed computing infrastructures without having to modify the code, only adding some decorators to the functions that can be parallelized.

3.3. Selection of open-source technologies

Given that we needed a parallel programming model that could handle the heterogeneity and distributed nature of the edge environment, and after comparing the various options described in the previous subsection, we find COMPSs and Ray to be the most suitable programming platforms, but at different roles in EXTRACT, as explained further below.

Both platforms offer simple programming abstractions that mask away the complexities of distributed computing and allow for efficient resource utilization. Both support multiple programming models such as Map/Reduce and Task-based programming. Both can scale well and offer fault-recovery. However, they are still considered complementary in the following aspect. COMPSs applications can execute on a mix of distributed computing infrastructures, including clusters, grids, and clouds, i.e., in the entire compute continuum. Ray, in contrast, is typically limited to running at cluster scope – a group of nodes closely located together and sharing a network, upon which Ray establishes its own application cluster. On the other hand, Ray offers excellent proven support for (almost) all existing third-party big-data and ML platforms to-date. For example, specifically for EXTRACT (PER use-case), Ray has native SoTA support for RL (Reinforcement Learning), with scalable implementations of many of the leading published algorithms.

For these reasons, we believe COMPSs is most suitable as the top-level and default orchestration solution in EXTRACT, where all EXTRACT applications will essentially be structured as COMPSs workflows using its API and decorators, allowing tasks to be deployed and scheduled across the compute continuum. However, for some ML/BD tasks in the workflow, we may consider Ray to be the appropriate platform, such that the task may be internally defined as a Ray distributed application. This aligns well with the SkyPilot tool being used for the compute continuum in D4.1 since that tool automatically deploys Ray in each of the compute continuum clusters, but can add COMPSs components (and any other components) just as well.

3.3.1. Introduction to COMPSs

COMPSs is an orchestrator and scheduler for Python workflows that will be used in EXTRACT applications. As explained above, it offers easy distribution of Python code across a mix of backend platforms by embedding the code with COMPSs decorators and APIs. It can be used within container technologies that allow the implementation of micro-service scalable solutions.

COMPSs is a framework that acts under the behavior of the master/worker architecture, is highly scalable and can handle large-scale applications running on multiple cores. It is also fault-tolerant, meaning that it can recover from errors and continue executing the application with only a small payment in the cost of execution time of the application running, only if the master node does not fail.

Another advantage of COMPSs is its support for data-centric computing, which allows the user to define workflows in terms of data dependencies rather than task dependencies. This makes it easier to express complex data flows and ensures that computations are executed in the correct order.

In addition, COMPSs provides a few advanced optimization techniques that can improve the performance of applications. These include dynamic task scheduling, which enables the runtime system to adapt to changes in the workload, and data-aware scheduling, which optimizes the placement of data and computations to minimize data movement.

This is possible thanks to the characterization of the workflow to be executed before running it in the real case scenario. COMPSs captures all the metrics for each of the tasks that are going to be executed and the data structure information where it is executed, and it uses this information to select the best execution path that minimizes the total workflow execution time respecting the user specifications and trying to balance the system load. This characterization can be benefited through the data obtained in the monitoring layer to implement better scheduling decisions. As a high computational cost is required to find the best execution path that minimizes the total workflow execution time, COMPSs gives us the possibility to choose different heuristics, related to task metrics, that can lead to a good result without necessarily being the best.

The heuristics mentioned can be divided into two different categories depending on the task metrics:

- A category that prioritizes by considering the execution time of each of the tasks. These heuristics are Shortest Process Time (SPT), which prioritizes tasks with less execution time and, on the other hand, Longest Process Time (LPT), that prioritizes tasks with longest execution time.
- A category that prioritizes by the number of successors of each of the tasks. Those heuristics are Largest Number of Successors in Next Level (LNSNL), that prioritizes tasks with the largest number of direct successors, and Largest Number of Successors (LNS) which prioritizes the tasks with the largest number of successors in the next multiple levels of depth in the graph.

The responsibility of which is the best heuristic to select for the execution of the workflow falls on the user.

Considering the aforementioned heuristics, an overview of how COMPSs executes the workflow by finding the best suitable execution configuration is explained below. It is important to remark that these steps will be repeated multiple times until the workflow is finished.

1. It takes a general view of the structure of the system where the workflow is executed. It establishes the master/worker connections and the connection to the specified monitoring tools.

2. It starts to schedule by taking the current level of tasks (non-dependent free tasks to compute) in the complete graph and ordering them in a priority ready queue.
3. Once the ready queue is set, the best resources (computing node and core) for each of the tasks in the ready queue will be selected.
4. Finally, it releases the dependencies of the next level tasks in the graph by the tasks that have already been computed.

From the user view side, the high-level steps to effectively parallelize an application using COMPSs are:

1. Define the application as a set of tasks: Start by defining the tasks that make up the application, along with their dependencies and inputs/outputs.
2. Annotate the tasks with COMPSs directives, which indicate COMPSs which tasks can be executed in parallel and which dependencies need to be satisfied before a task can be executed.
3. Compile the application with COMPSs to generate a set of executable files (when using compiled languages). That step can be skipped for interpreted languages.
4. Deploy the compiled application to the compute continuum.
5. Run the application and let COMPSs take care of scheduling the tasks and managing their dependencies. As the application runs, COMPSs will monitor its progress and resource usage.
6. Analyze the monitored results once it has finished running.

In the following code snippet, we show a sample application that uses COMPSs to enhance parallelism:

```
# sample_app.py

@task(returns=int)

@constraint(cpu_cores=2)

@storage("memory")

def square(x):

      return x * x


@task(returns=int)

@depends(square)

def sum_list(x_list):

      return sum(x_list)


if __name__ == "__main__":
```

```
# Declare the input data
input_data = [1, 2, 3, 4, 5]

# Divide the input data into two parts
input_data_1 = input_data[:3]
input_data_2 = input_data[3:]

# Invoke the square tasks for each part of the input data in parallel
result_1 = [square(x) for x in input_data_1]
result_2 = [square(x) for x in input_data_2]

# Combine the results using a sum task
final_result = sum_list([result_1, result_2])
```

In the above example, the input data is divided into two parts and we then invoke the square task in parallel for each part using two separate lists, `result_1` and `result_2`. We then use the `sum_list` function to combine the results of the two lists and compute their sum.

COMPSs leverages data-awareness by means of the decorators. Some of the most common ones are used in this code snippet:

`@task(returns=...)`: This decorator is used to indicate that a function should be turned into a COMPSs task. The `returns` argument specifies the type of the value that the task returns. For example, `returns=int` indicates that the task returns an integer. Any other data type could be used as well.

`@constraint(...)`: This decorator is used to specify task constraints. Task constraints are requirements that a task must satisfy in order to run on a specific resource. For example, we can use `@constraint(cpu_cores=2)` to indicate that a task requires at least two CPU cores. Other types of constraints include memory (`@constraint(memory="2G")`), network (`@constraint(network=True)`), and disk (`@constraint(disk="10G")`). This information is very useful for the scheduler to allocate a certain job to a specific computing device.

`@depends(...)`: This very important decorator is used to specify task dependencies. Task dependencies indicate that a task must wait for another task to complete before it can start. The `@depends()` parameter can indicate either a task ID or a specific output parameter of another task.

The ***`@storage`*** decorator specifies that the task's input and output data should be stored in memory, which again is information the EXTRACT can leverage in order to better decide whether the data is stored in HPC, the cloud or in the edge.

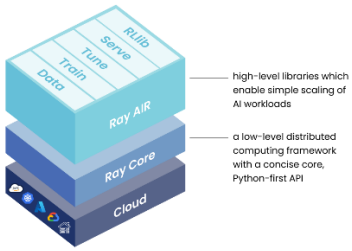
For a complete list of the available COMPSs decorators, please refer to https://compss-doc.readthedocs.io/en/stable/. By using these decorators, we can specify more complex workflows that take advantage of the parallelism and distributed computing capabilities of COMPSs.

By running this example with COMPSs, the `square` tasks would be distributed across the available processing nodes, allowing us to take advantage of the parallelism and distributed computing capabilities of COMPSs.

3.3.2. Introduction to Ray

Ray is a framework for easy, semi-transparent scaling-out of Python applications over many cores inside a single node and over many nodes in the same cluster. Ray provides annotations and API for converting functions into remote tasks and classes into remote actors. Tasks are stateless, in the sense that a task's initial state depends almost entirely on the values of the parameters in the function invocation that constitutes the task. Actors are stateful, in the sense that invoked methods may use both the invocation parameters and the object's internal state. Distributed actors and tasks can interachangeably invoke one another (nested invocations). In addition, Ray supports a scalable and efficient method for global state/variables that can be used anywhere (incl. tasks and actors) using a shared-memory key-value store in each node (Plasma) that can replicate efficiently across the cluster and is used also for storing remote invocation results.

Ray has been designed for and used in machine learning and big-data computation. To that end, most popular Python big-data (Modin / Pandas / Numpy etc.) and machine learning (PyTorch / TensorFlow / HuggingFace, etc.) frameworks ported onto it. A typical deployment of Ray contains the stack shown below:



Ray Core is the basic framework that provides the task, actor and object (global data) abstractions. Above it, **Ray AIR** is the basis of the 3rd-party eco-system that contains both easy integrations with ML frameworks (Train/Serve), BD computation (Data) and some Ray-native highly-efficient libraries for specific domains, such as RLlib for Reinforcement Learning and Tune for hyperparameter tuning. Finally, at the bottom, are the many backends upon which a Ray cluster can be deployed. Ray further supports auto-scaling of its cluster on top of the current backend to dynamically match the cluster size and operational costs with the demands of the applications inside.

Developing a Ray application can be done either by writing explicitly with a high-level Ray API (e.g., Ray AIR), or by coding a sequential algorithm using various (incl. non-Ray) libraries and scaling it out by adding Ray annotations and synchronization API. Since API is a familiar method, we will briefly present here only the annotations/synchronization method. Both methods can be seamlessly combined.

The simple application below demonstrates basic Ray sync API and annotations. It computes a distributed Monte-Carlo estimation of Pi, with the Ray statements marked in bold. The algorithm itself is quite straightforward – estimate Pi as a ratio of sampled area of a circle with radius of 1 divided by the sampled area of the 2x2 square that surrounds the circle. Sampling is actually done only in the first quadrant, so the resulting ratio is multiplied by 4 (line 33). All the sampling is completely independent, so the algorithm can be arbitrarily parallelized. In the example below, there are 100 billion samples divided into batches of 1 million each that can be run concurrently. As one can expect, changing these settings can affect the global computation time subject to resource availability.

Let us now consider the code. Ignoring the imports, the first relevant statement for working with Ray is **ray.init()** at line 6, which connects the application to the Ray cluster – either the existing cluster and if not, by creating a new one. Next, the **@ray.remote** annotation (line 8) defines a function or a class as a remote task or actor (that can run in a different process or a different node in the cluster). All type information for transferring parameters and return values is inferred automatically by Ray from the Python runtime, so it's typically not required.

Invoking remote tasks and methods is done by appending the **.remote** attribute to the function or methods reference, everything else remaining the same, as seen in line 28. However, all remote invocations are asynchronous in nature, returning a global future-like result object. To further compute based on result objects, they need to be waited for and retrieved, which is what the **ray.get()** call at line 29 does. This call can operate on a single result or a list of results. Invoking and waiting also automatically infers the dependency order between tasks.

```
1     import ray
2     import random
3     import time
4     import math
5     from fractions import Fraction
6     ray.init()
7
8     @ray.remote
9     def pi4_sample(sample_count):
10     """pi4_sample runs sample_count experiments, and returns the
11     fraction of time it was inside the circle.
12     """
13     in_count = 0
```

```
    for i in range(sample_count):
    x = random.random()
    y = random.random()
    if x*x + y*y <= 1:
    in_count += 1
    return Fraction(in_count, sample_count)

SAMPLE_COUNT = 1000 * 1000
FULL_SAMPLE_COUNT = 100 * 1000 * 1000 * 1000 # 100 billion samples
BATCHES = int(FULL_SAMPLE_COUNT / SAMPLE_COUNT)
print(f'Doing {BATCHES} batches')
start = time.time()
results = []
for _ in range(BATCHES):
 results.append(pi4_sample.remote(sample_count = SAMPLE_COUNT))
output = ray.get(results)
end = time.time()
dur = end - start
print(f'Running {FULL_SAMPLE_COUNT} tests took {dur} seconds')
pi = sum(output)*4/len(output)
print(f'Pi estimated value: {float(pi)}')
print(f'Estimation error: {abs(pi-math.pi)/pi}')
```

There are several other useful annotations and sync APIs, such as defining resource requirements for tasks and actors, handling fault-tolerance etc. Full detailed documentation for all Ray components, as well as many examples, tutorials and tooling can be found at https://docs.ray.io/en/latest/index.html.

4. Monitoring requirements

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 and provide it to the orchestrator and other EXTRACT applications that might use this information to enable the optimal performance, availability, scalability, security and the management of the compute continuum and the applications.

The main requirements that the monitoring system of the EXTRACT platform needs to meet are:

- **Near real-time monitoring**: the platform needs to collect and process metrics from the EXTRACT platform in near real-time to be able to report them to the orchestrator.
- **Flexibility and extensibility**: it should allow easy integration of new nodes, components, services and protocols within the compute continuum into the monitoring system. Therefore, it should be designed in a modular and extensible

way, allowing the addition of new metrics, data sources and analytical capabilities without significant readjustments.

- **Integration**: it should provide APIs or hooks to integrate with the orchestrator system for streamlined operations.
- **Historical Data**: it must store and retain historical metrics data for trend analysis, intelligent capacity planning, and retrospective analysis that can be useful for the orchestrator.
- **Scalability and efficiency**: the monitoring system should handle a large number of monitoring agents and scale seamlessly with the growth of the compute continuum. In addition, these monitoring agents should have minimal impact on the consumed resources such as low CPU and memory overhead.
- **Reliability**: ensure the system is resilient to failures and can recover gracefully.
- **Security and compliance**: it should follow secure practices for data collection, storage, and transport, especially when dealing with sensitive metrics. Therefore, the monitoring system needs to comply with relevant data privacy regulations and security standards.

4.1. Monitoring components

After defining the requirements of the monitoring system, these are the components that it must contain to meet them:

- **Data Collectors**: the monitoring system must deploy monitoring agents across the compute continuum to collect metrics. These agents or collectors should be installable on different instances of the EXTRACT platform, on Kubernetes nodes, pods and services. They should be capable of gathering system-level and application-specific metrics.
- **Data Storage**: it should have a centralized storage system or a distributed database to store the collected metrics. In this case, the most realistic would be a time series database such as Prometheus or InfluxDB. However, another solution could be a scalable storage solution such as Apache Cassandra.
- **Data Transport**: for sending the collected metrics from the data collectors to the storage system it is necessary to have a messaging or streaming system. Apache Kafka or RabbitMQ can be suitable choices for this purpose.
- **Data Processing procedures**: implement data processing and analytics components to perform, aggregate, and clean the collected metrics before storing them.
- **Visualization**: it would be useful to have a monitoring dashboard or visualization tool to present the collected metrics in a user-friendly and interactive manner. Grafana, Kibana, or custom-built dashboards can be used for this purpose.

In Figure 1 we illustrate the architecture proposal for the monitoring system. As data collectors, we can distinguish two types of Node daemons: Node containers daemons

which oversee the containers and applications of their corresponding node, and Node monitoring daemons which are in charge of constantly monitoring the state and the surrounding network of the node. In this context, the Node state is defined by dynamic properties, such as the amount of available memory or the CPU load of a node. These daemons are responsible for collecting metrics from the nodes, which are then transported via an API to the metrics database through the messaging system. Before storing the metrics, they undergo processing procedures. Once the metrics are stored, they are sent to the Orchestrator system to analyze the status of the nodes and applications, and subsequently schedule the processes. Additionally, a monitoring dashboard will be available to visualize the progress of the system.

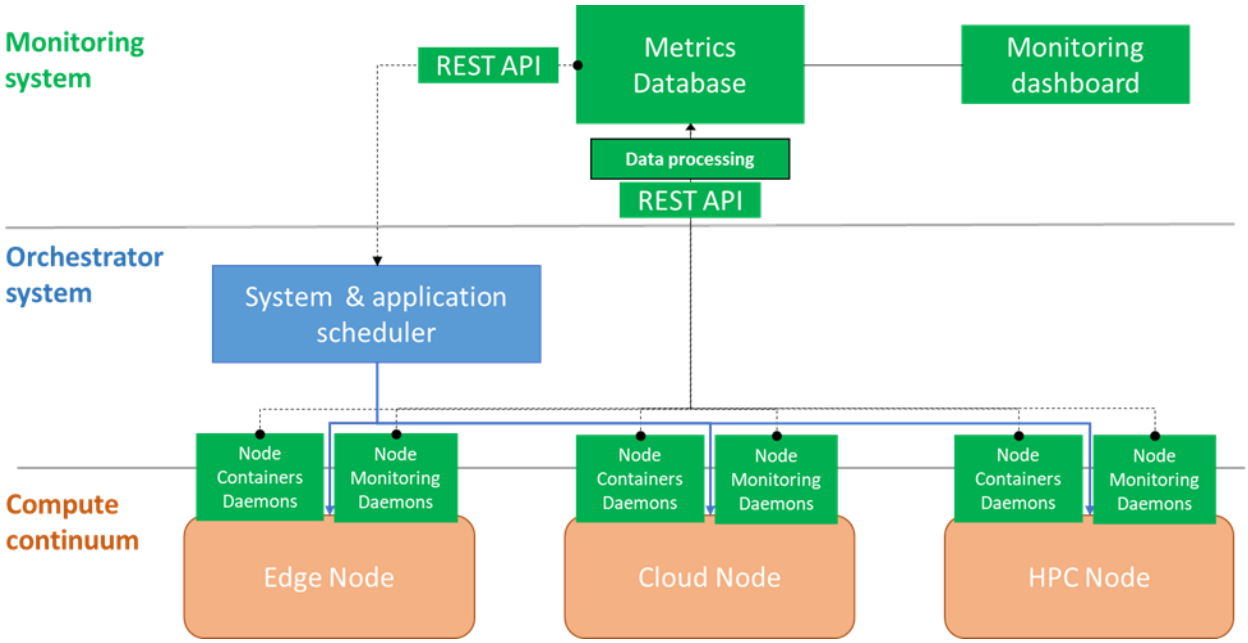


Figure 1 Monitoring system architecture

4.2. Metrics requirements

In this section we are going to define the metrics requirements to be met by the monitoring agents. First of all, it should be considered that the monitoring system serves as a tool to provide information about the EXTRACT platform's state to the Orchestrator in order to schedule the applications guaranteeing the application's quality of service. Consequently, the metrics collected must be useful to improve the performance of the Edge-to-Cloud continuum architecture, optimizing resource allocation, and enabling informed decision-making.

To effectively monitor the EXTRACT platform, it is necessary to collect a variety of metrics from Edge, Cloud, and HPC nodes, as well as the applications and services deployed on them. The key metrics that can help improve the performance of the Edge-to-Cloud continuum architecture, optimize resource allocation, and enable informed decision-making are the following:

Metrics related to the nodes and the infrastructure of the compute continuum:

- **Available nodes**: 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.
- **CPU usage**: Monitoring CPU usage of nodes helps in optimizing resource allocation and identify performance bottlenecks.
- **Memory usage**: Tracking memory usage and availability of nodes aids in efficient resource allocation and capacity planning.
- **Network Throughput**: Monitoring network traffic and throughput between nodes helps optimize data transfer and identify network-related issues.
- **Storage Utilization**: Keeping track of storage usage on nodes ensures efficient allocation and helps identify capacity constraints.
- **Node Health**: Metrics such as temperature, power consumption, and hardware failures provide insights into the overall health and reliability of the nodes.
- **Workload Distribution**: Tracking the distribution of workloads across nodes helps identify imbalances and optimize resource allocation.
- **Resource Efficiency**: Metrics like CPU and memory usage per node help identify opportunities for resource optimization and cost savings.

Metrics related to the applications and systems deployed:

- **Application Availability**: Monitoring the availability of deployed applications ensures they are accessible and functioning as expected.
- **Application Response Time**: Measuring the response time of applications helps evaluate their performance and identify areas for improvement. This metric can surface errors that should be fixed within the application layer.
- **Container Metrics**: Collecting container-specific metrics, such as CPU and memory usage per container, aids in optimizing resource allocation and performance within Kubernetes-like environments.
- **Network Throughput**: Monitoring sent and received data packets as well as how much bandwidth is being used.
- **Storage Utilization**: Keeping track of storage usage helps ensure efficient storage allocation and identify potential capacity constraints.
- **Workload-specific Metrics**: Metrics tailored to specific workloads or services deployed on the nodes provide insights into their performance and resource utilization.
- **Service Health**: Metrics related to the health and availability of services deployed on the nodes enable proactive monitoring and timely issue resolution.
- **Node Health**: Monitoring node health metrics, including temperature, power consumption, and hardware failures, ensures the reliability and stability of the infrastructure.
- **Workload Distribution**: Tracking the distribution of workloads across nodes helps identify imbalances and optimize resource allocation.

- **Resource Efficiency**: Metrics related to resource utilization efficiency, such as CPU and memory usage per workload or application, help identify opportunities for optimization and cost savings.

These metrics, when collected and analyzed effectively, provide valuable insights into the performance and resource utilization of the Compute Continuum platform. They enable proactive monitoring, timely decision-making, and optimization of the Edge-to-Cloud continuum architecture.

4.3. Metrics collection methods and protocols

After evaluating multiple existing open-source options for observability and metrics collection, Prometheus and Open Telemetry have been considered as the main alternatives. Both projects are part of the Cloud Native Computing Foundation (CNCF) and aim to simplify how generate, collect and monitor.

Prometheus relies on the kube-state-metrics (KSM) service. This is a simple service which uses a Go client and allows listening the Kubernetes API and generates metrics about the state of deployed nodes, pods and services. These metrics are served as plaintext and accessible through HTTP protocol and intended for consumption by Prometheus or any scraper that can scrape a Prometheus client endpoint. Prometheus has its own time series database where it stores the metrics that can later be queried with its own query language, PromQL (Prometheus Query Language). Although this database can handle a lot of data, it is not officially meant to be a long-term storage solution, so data is often sent to another storage solution like Promscale or InfluxDB.

OpenTelemetry has a smaller scope. It collects metrics using a set of tools, APIs, and SDKs to create and manage telemetry data which is sent to other systems for storage or query. OpenTelemetry decouples the generation of signals from the operational considerations of storage and querying. This means that collected metrics often end up back in Prometheus or a Prometheus compatible system.

4.4. Interoperability

Currently there is a de-facto standard denominated OpenMetrics for transmitting cloud-native metrics at scale, with support for both text representation and Protocol Buffers. This standard begins to emerge when Prometheus becomes the default tool for cloud-native observability and releases a metric exposition format called Prometheus exposition format 0.0.4.

The OpenMetrics standard is a specification for the exposition and exchange of metrics data in a standardized format. It aims to provide a common, interoperable format for metrics collection, storage, and querying across different monitoring and observability systems.

Key features and aspects of the OpenMetrics standard include:

- **Text-based Format**: OpenMetrics adopts a text-based format that is human-readable and easy to generate, parse, and understand. It uses plain text, specifically the line-based Text-based Exposition Format (TBEF).

- **Data types**: OpenMetrics define various data types such as Integer, Float, Timestamps, Strings, Label, LabelSet, MetricPoint, Exemplars, Metric and MetricFamily.
- **Metric Types and Labels**: OpenMetrics supports various metric types such as counters, gauges, histograms, and summaries. Metrics can also have labels or tags, which provide additional context and allow for more flexible querying and aggregation.
- **Communication**: OpenMetrics defines that the communication between ingestors and exposers must be done using the HTTP protocol and can be secured with TLS 1.2 or later.

In conclusion, Prometheus and OpenTelemetry can use OpenMetrics format for metrics exposition. It leverages the OpenMetrics specification to gather, store, and query metrics in a standardized and interoperable manner. This allows to seamlessly integrate with other systems that support the OpenMetrics format, enabling easy exchange and consumption of metrics across various monitoring and observability tools.

4.5. Interaction with scheduler / orchestrator

In this architecture, the scheduler / orchestrator will have access to the monitoring database through an API. This will expose a series of endpoints that allow us to know the status and desired metrics of the different nodes and services deployed. In this way, the scheduler / orchestrator is able to perform the following tasks:

- **Resource Allocation:** The scheduler is responsible for allocation and management of available resources such as CPU, memory, storage, and network bandwidth.
- **Tasks Scheduling:** Considering factors such as dependencies, priorities, resource requirements and constraints to determine the most appropriate order and timing for task execution. This ensures that tasks are executed in a coordinated manner and according to desired behavior.
- **Scaling and Load Balancing:** The scheduler manages the scaling of resources and load balancing across multiple nodes or applications. This helps distribute work evenly, prevent overloading of resources, and ensures scalability as demand fluctuates.
- **Fault Tolerance and Resilience:** The scheduler plays a role in ensuring fault tolerance and resilience. It can detect failures or disruptions in nodes or services and take appropriate actions, such as rescheduling tasks, reallocating resources or triggering automated recovery processes. This helps maintain system availability and minimize the impact of failures.

5. Deployment mechanisms

5.1. Deployment requirements

The below requirements are based on the premise that users package their applications as Docker/OCI containers to run under Container Orchestration Engine (COE) on the edge devices and cloud. The following outlines the elements of the edge and cloud management system that provides the runtime and management capabilities for execution of the user-defined applications at the edge and cloud.

Edge application deployment and management stack:

- SaaS/PaaS with edge device and application deployment and management functionality (Edge SaaS/PaaS)
- Edge device and application management Agent (Edge Agent)
- Container Orchestration Engine (COE)
- Container Runtime (CR)
- Operating System (OS)
- Edge device hardware (edge HW)

Cloud application deployment and management stack:

- PaaS with cloud resource provisioning and application deployment and management functionality (Cloud PaaS)
- Container Orchestration Engine (COE)
- Container Runtime (CR)
- Operating System (OS)
- Virtual Machine running on the cloud (cloud VM)

On the level of the applications deployment and management actions, there are a lot of commonalities between cloud and edge. Hence, it is reasonable from the implementation point of view to combine Edge and Cloud SaaS/PaaS in a single Edge-to-Cloud SaaS/PaaS service.

Applications as containers: The deployment management system expects user applications packaged as Docker/OCI containers and the deployment expressed in the DSL(s) supported by the COE(s) selected by the project. Example: COE - Kubernetes and DSL - Kubernetes manifest.

COE on edge and cloud: Container Orchestration Engine (COE; e.g. Kubernetes, Docker, Docker Swarm) must be running on each edge or cloud resource to be integrated to the system for workload placement.

Pool of independent resources: The system must allow registration of independent edge devices (running COE) and cloud endpoints to form a pool of independent resources on which user applications can be deployed.

Agent-based edge and application management: For integration of the edge resources with the control plane, the system must provide agents running under supervision of COE on the edge devices.

Mixed deployment model: To satisfy working with network restricted edge devices and cloud endpoints the solution must support PULL and PUSH modes for deployment and lifecycle management of applications.

Comprehensive set of deployment management actions: The system should allow users to perform all application management actions like registration of the definitions of the application, deployment, update of the container image, update of the configuration parameters via environment variables and configuration files, get logs of each deployed component, get load metrics of each deployed component and termination of the deployment.

Deployment via RESTful API: The system must expose comprehensive and secure RESTful API allowing to perform all deployment related actions from "**Comprehensive set of deployment management actions**" requirement.

Deployment via web UI: The user facing interface must feature rich web GUI and expose all application deployment functionality available via API.

Deployment history: The system must keep record of the user deployed application (including state transitions) and allow for updates or re-deployments of it at the later stages.

Deployment telemetry: The system must maintain a reference to the load and performance metrics of the deployments it is managing.

5.2. Integration with the scheduling system

After matching the project requirements to the edge-to-cloud continuum management, application workload placement and lifecycle orchestration against the above listed tool, we think Nuvla could be a very suitable option for the project to fulfill the function of the edge and cloud orchestrator of the EXTRACT project.

Below is the list of considered tools along with their pros and cons based on the project requirements.

- Open Cluster Manager
  - Pros
    - Management of multiple remote COE clusters.
    - PULL/PUSH mode (PULL via *klusterlet*s deployed on remote edge or cloud clusters.)
    - Open Source.
  - Cons
    - No web GUI.
    - Kubernetes only.

- Doesn't have the concept of edge device and doesn't provide edge device level management functionality.
- Requires wrapping of user application manifests into ad-hoc *ManifestWork* for describing placement on remote clusters.
- No meta-data catalogue for workload scheduling based on data location.

- Rancher
  - Pros
    - Deployed as PaaS service.
    - Rich web GUI.
    - Can manage Docker Swarm, but first-class support was removed. Available as Catalog application.
    - Manages remote resources as independent COE endpoints.
    - Open Source.
  - Cons
    - No PULL mode.
    - Doesn't have the concept of edge device and doesn't provide edge device level management functionality.
    - No meta-data catalogue for workload scheduling based on data location.
- Nuvla
  - Pros
    - PULL/PUSH mode (PULL via NuvlaEdge Agent on remote edge COE).
    - Rich web GUI.
    - Supports Kubernetes and Docker Swarm on the cloud.
    - Supports Kubernetes, Docker, and Docker Swarm at the edge (allowing for management of resource restricted devices).
    - Provisions and lifecycle-manages Kubernetes and Docker Swarm clusters on clouds.
    - Manages remote resources as independent COE endpoints.
    - Provides integrated meta-data catalogue for workload placement based on data location.
    - Provides edge device level management functionality (OS, COE).
    - Open Source.
  - Cons
    - Learning curve

It is expected for the scheduling system to connect to the workloads placement and management part of the orchestrator for enforcement of the deployment scheduling decisions. The project is aiming at using Nuvla for various tasks including the process of the orchestration of the workloads on the edge and cloud. For this purpose, Nuvla provides RESTful API for definitions and various operations on:

- Data records and data sets

- User applications
- Inventory of edge devices and cloud resources (as COE endpoints)
- Deployments of the user applications on the remote resources
- Metrics of the edge device and user applications

The above set of RESTful resources and operations on them should provide the basis for the scheduling system first, to fulfill the initial data-based workload placement requests using hard (predicate-based) and soft (priority-based) requirements and which initially might be sub-optimal, and second, later on to provide reallocation decisions for more optimal placement of the workloads.

Nuvla provides a rich set of resources describing the workload related elements along with the corresponding actions on them that can be utilized already now by the EXTRACT scheduling system. However, if design and implementation of the deployments scheduling sub-system requires extra actions on the current set of resources or completely new resources with their own set of actions, this will be possible to add.

Nuvla is a highly modular and extensible system. If found useful, the deployment scheduler sub-system of EXTRACT can be integrated into Nuvla and exposed through its API.

5.3. Integration with interoperability abstraction layer

Nuvla PaaS uses RESTful API as established by industry standard to expose its edge, cloud, and application management functionality. The API is clear and well structured around the edge-to-cloud continuum management domain. It uses standard CRUD for data operations and JSON as the data format for exchange between client and server. The Python library *nuvla-api* and command line client *nuvla-cli* are available to the developers and users for better integration and communication with the service. This usage of the well-established standards will facilitate the integration of the Nuvla service with the EXTRACT components and services.

When user application deployments on edge devices are done with Nuvla, Nuvla keeps all the state, state transitions, and operations performed on the applications. This provides a good historical overview of the application's lifetime. Edge device and application utilization and load metrics are collected and stored on Nuvla as well. At the moment, Nuvla uses custom telemetry collector implemented as part of the NuvlaEdge agent running on the edge devices.

We envision an extension to Nuvla's currently employed telemetry collection solution by the introduction of the industry de-facto standards and tools described in section **Error! Reference source not found.**. As part of the extension, we plan to:

- Host telemetry data storage service as part of Nuvla service
- Support deployment of the COE-level and application metrics collectors on edge devices and cloud clusters and secure connection of them to the internal data storage service

- Expose per edge device / cloud COE endpoint collected metrics along with the applications' ones for visualization and consumption by other services e.g., deployment scheduling service.

6. Conclusions

This deliverable encompasses the initial phase of the project within Work Package 3 (WP3). It covers the tasks performed in T3.1, focused on the determination of the prerequisites for the orchestrator and the monitoring system of the EXTRACT platform. In short, it focuses on the definition of a data-driven orchestration structure and how to integrate it into the project.

This deliverable details the different technologies available that the orchestrator may use for deployment and scheduling of applications. In addition, it details the factors that the orchestrator must consider to ensure optimal performance. That is why this document also aims to detail monitoring requirements identifying how to collect, process, store and communicate information with regard to performance, status, and resources of the compute continuum and provide this information to the orchestrator.

The definitions and requirements established in this document will serve as a baseline for the developments to be carried out in the subsequent tasks of the WP3.

7. Acronyms and Abbreviations

- AI - Artificial Intelligence
- API - Application Programming Interface
- AWS - Amazon Web Services
- COMPSs - COMP Superscalar
- COE - Container Orchestration Engine
- CPU - Central Processing Unit
- CR - Container Runtime
- CSV - Comma-Separated Values
- DL - Deep Learning
- D - Deliverable
- EC - European Commission
- EXTRACT - A distributed data-mining software platform for extreme data across the compute continuum
- GPU - Graphics Processing Unit
- GPL - General Public License
- HE - Homomorphic Encryption
- HPC - High-Performance Computing
- HTTP - Hypertext Transfer Protocol
- IoT - Internet of Things
- K8s - Kubernetes
- KPI - Key Performance Indicator
- ML - Machine Learning
- PER - Personalized Evacuation Route
- RE - Restful - Representational State Transfer
- RL - Reinforcement Learning
- S3 - Simple Storage Service
- TASKA - Transient Astrophysics with a Square Kilometre Array
- VPN - Virtual Private Network
- WP - Work Package