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

D4.5 Final release of the EXTRACT Software Platform

Version 1.0

Documentation Information

Contract Number	**101093110**
Project Website	www.extract-project.eu
Contractual Deadline	M39, 31 March 2026
Dissemination Level	Public
Nature	Other
Authors	Alba Cañete, Eduardo Quiñones (BSC)
Contributors	URV, IBM, SIXSQ, IKERLAN, BINARE
Reviewer	Simone Naldini (MATH)
DOI	10.5281/zenodo.19208148
Keywords	EXTRACT Software Platform

Change Log

Version	Description Change
V0.1	Initial draft of the table of contents and collaboration assignments
V0.2	BSC contributions
V0.3	Partners contributions
V0.4	Internal review
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

This document presents the final release of the EXTRACT Software Platform, describing the software components developed within the project under Work Packages 2, 3, and 4. The platform integrates the technologies produced during the project into a unified software environment that supports data management, distributed computation, and orchestration across heterogeneous computing infrastructures.

The EXTRACT Software Platform is composed of several components that together enable the execution of data-driven applications across the compute continuum. These components span programming models, data infrastructure services, orchestration mechanisms, and infrastructure abstraction technologies. The integration of these elements forms the basis of the platform delivered by the project.

1.1. Purpose and scope of the deliverable

The main objective of D4.5 is to provide the installation guidelines for all components that form the EXTRACT Software Platform. The document consolidates the technical procedures required to install, configure, and execute the platform in a reproducible manner. The components and their repositories are:

WP	Software	Git repository
WP3	COMPSs	COMPSs runtime : https://gitlab.bsc.es/extract/extract-sa/compss kubecompss: https://gitlab.bsc.es/extract/extract-sa/kubecompss Helm Charts: https://gitlab.bsc.es/extract/extract-use-cases/taska/use-case-c/helm-app
WP2	Lithops	https://gitlab.bsc.es/extract/extract-sa/lithops
WP2	Flexecutor	https://gitlab.bsc.es/extract/extract-use-cases/taska/use-case-c/flexecutor
WP2/WP4	SkyStore	https://github.com/gilv/skystore/
WP2	Dataplug	https://github.com/CLOUDLAB-URV/dataplug
WP2	Nuvla Data Catalog	https://gitlab.bsc.es/extract/extract-use-cases/taska/use-case-c/data-catalogue
WP2	K-serve	https://github.com/revit13/per-demo-sept

WP4	Kubernetes	Ansibles: https://gitlab.bsc.es/extract/extract-sa/ansible Skupper: https://gitlab.bsc.es/extract/extract-sa/skupper Linkerd: https://gitlab.bsc.es/extract/extract-sa/linkerd

By documenting the installation process in a structured and standardized way, this deliverable contributes to the reproducibility, maintainability, and portability of the EXTRACT Software Platform. It also facilitates collaboration among project partners by ensuring that all components can be deployed using consistent procedures across different computing environments.

1.2. Structure of the document

The remainder of the document is organised as follows. Section 2 provides an overview of the EXTRACT Software Platform, describing its architectural organization and the main functional layers that compose the system. The section introduces the Programming Models Layer, the Data Infrastructure Layer, the Orchestration Layer, and the Compute Continuum Abstraction Layer, outlining their roles within the platform. Section 3 presents the installation procedures for the different components of the platform. The section is structured according to the architectural layers introduced previously and includes detailed installation instructions for the main technologies used in the platform. Section 4 summarizes the document and highlights the role of the installation guidelines in supporting the deployment and reproducibility of the EXTRACT Software Platform. Section 5 lists the acronyms and abbreviations used throughout the document. Finally, Section 6 provides the references cited in the deliverable.

2. Overview of the EXTRACT Software Platform

This section provides a concise overview of the EXTRACT Software Architecture. The Software Architecture is organized in four main layers:

1. **Programming models**, composed of Lithops, Flexecutor and COMPSs
2. **Data infrastructure**, composed of SkyStore, Data plug, and the Nuvla Data Catalogue, with S3 as the common data substrate. Also K-serve as model serving solution.
3. **Orchestration**, composed of a monitoring system that includes Prometheus, together with the runtime systems of the programming models

4. **Compute continuum abstraction**, composed of Kubernetes multicluster deployments that provide the infrastructure abstraction for the platform.

2.1. Programming Models Layer

The Programming Models Layer provides the frameworks used to develop and execute distributed applications on the EXTRACT platform. These programming models abstract the complexity of parallel and distributed execution, allowing developers to express applications at a higher level while delegating task scheduling, execution management, and resource utilization to the underlying runtime systems.

The layer includes Lithops, Flexecutor, and COMPSs, which support different execution paradigms and enable applications to run across heterogeneous computing environments, including cloud and edge infrastructures. These frameworks provide mechanisms for parallel task execution, data management, and interaction with the orchestration and infrastructure layers of the platform.

2.2. Data Infrastructure Layer

The Data Infrastructure Layer provides the services required for data storage, management, and access within the EXTRACT Software Platform. Its objective is to ensure that data generated or consumed by applications can be stored, discovered, transferred, and accessed consistently across the different components of the platform.

This layer is composed of SkyStore, Data Plug, and the Nuvla Data Catalogue, which together support data management and integration across the platform. SkyStore provides distributed storage capabilities for datasets and intermediate results. Data Plug enables data access and movement between different storage systems and platform components. The Nuvla Data Catalogue supports data discovery and metadata management, allowing datasets to be registered, described, and located within the platform.

The layer relies on S3-compatible object storage as the common data substrate, providing a unified interface for storing and accessing data across the platform components. In addition, KServe is included in this layer to support the deployment and serving of machine learning models, enabling models to be exposed as scalable services that can be accessed by applications running on the platform.

2.3. Orchestration Layer

The Orchestration Layer manages the coordination, monitoring, and execution of workloads running on the platform. It provides the mechanisms required to supervise the system, collect operational metrics, and support the execution environments of the programming models.

This layer includes a monitoring system based on Prometheus, which collects and stores metrics related to system performance and resource usage. In addition, the runtime systems associated with the programming models operate within this layer, coordinating task execution and interacting with the underlying infrastructure to allocate resources and manage workloads.

2.4. Compute Continuum Abstraction Layer

The Compute Continuum Abstraction Layer provides the infrastructure foundation on which the EXTRACT platform operates. It enables applications and services to run across distributed and heterogeneous computing resources while providing a unified abstraction for resource management.

This layer is based on Kubernetes multi-cluster deployments, which allow the platform to orchestrate containers and services across multiple computing environments. Through this abstraction, the platform can manage workloads across different infrastructure domains while maintaining a consistent deployment and execution model.

3. Installation of the EXTRACT Software Platform

This section describes the installation instructions of all the components that form the EXTRACT Software Platform.

3.1. Programming Models Layer

3.1.1. COMPSs

3.1.1.1. Installation

The COMPSs container images used in the EXTRACT Software Platform are built through an automated workflow based on Docker Buildx, Docker Bake, and Kubernetes. The build process is designed to support both amd64 and arm64 architectures and to generate a consistent set of images for deployment in the target Kubernetes environment.

The installation is performed through the Makefile, which provides the main entry points for preparing the build environment and generating the required images. The most relevant commands in this process are make prepare, make all_arch, and make compss.

Build environment preparation

Before launching a multi-architecture build, the Kubernetes-based build environment must be prepared. This is done with the following command:

```
make prepare
```

This step configures a Docker Buildx builder using the Kubernetes driver. The command first checks whether the Kubernetes cluster is reachable through kubectl. It then detects which node architectures are available and in Ready state, specifically amd64 and arm64.

If at least one supported architecture is available, the process creates the namespace used by the builder, if it does not already exist, and deploys a Buildx builder named kube-builder. A specific builder node is attached for each detected architecture, using Kubernetes node selectors so that each image is built on a native node of the corresponding architecture.

This preparation step is required for distributed multi-architecture builds in the cluster.

Complete image generation

Once the builder has been configured, the complete image set can be generated with:

```
make all_arch
```

This command builds the full COMPSs image stack for all supported architectures available in the Kubernetes cluster. The process is divided into two stages.

First, the base images are generated:

- *min_base*
- *full_base*

The min_base image contains the runtime dependencies required by COMPSs, including Java, Python, MPI-related packages, SSH support, and auxiliary system tools. The *full_base* image extends this environment with the additional dependencies required for compilation, testing, and integration tasks, such as Gradle, browser testing tools, Docker CLI, and CI-oriented packages.

After the base images are generated, the final compss image is built. This image contains the COMPSs framework and is produced from a multi-stage Docker build. The compilation is performed in an intermediate stage based on *full_base*, while the final runtime image is assembled on top of *min_base*. This keeps the final image smaller and separates build-time dependencies from runtime dependencies.

All generated images are pushed to the configured container registry and published as multi-architecture images.

Regeneration of the COMPSs image only

If the base images are already available, it is possible to rebuild only the COMPSs layer with:

```
make compss
```

This command generates only the final COMPSs image, reusing the previously built base images. Internally, the process uses the *full_base* image as the compilation environment, where the COMPSs source tree is copied, submodules are retrieved, and the framework is compiled. The resulting installation is then copied into a final image based on *min_base*.

This option is useful when changes affect the COMPSs source code but do not require rebuilding the complete software stack.

In summary, *make prepare* configures the cluster-side builder, *make all_arch* generates the complete set of COMPSs images, and *make compss* rebuilds only the final COMPSs container layer on top of already existing base images.

Application Docker image creation

This step consists in generating the application image on top of the COMPSs base image, including the workflow source code and the additional Python dependencies required by the execution environment. Once the image has been built, it must be pushed to a container registry that is accessible from the target Kubernetes cluster.

```
cd compss/compss/runtime/scripts/utils


./compss_docker_gen_image \
    --image-base="bernatxz/compss_taska_c" \
    --image-name="user/experiment" \
    --context-dir="/home/user/context_dir" \
    --python-packages="git+https://github.com/Oct-HI/dataplug.git     lithops
boto3" \
    --specific-arch="linux/amd64"
```

3.1.1.2. Execution

Execution of COMPSs is described in Sections 3.3.1 and 3.3.2 of this document.

3.1.2. Lithops

3.1.2.1. Installation

Lithops is deployed on top of the COMPSs container image generated in the previous section. Instead of modifying the COMPSs build process itself, a new application container image is created using the COMPSs runtime image as the base layer. This derived image includes Lithops and the Python dependencies required by the TASKA-C application.

The generation of the Lithops-enabled container image is performed using the script *compss_docker_gen_image.* This script extends an existing COMPSs container by copying the application context and installing the Python dependencies specified by the user.

The script internally generates a temporary Dockerfile and builds the resulting container image using Docker Buildx. If the Kubernetes builder prepared through *make prepare* is available, it is reused for the build process; otherwise, the default local Docker builder is used.

The container image for the TASKA-C application is generated using the compss_docker_gen_image script provided within the COMPSs framework. This script is located in the following directory relative to the COMPSs installation:

```
./compss/runtime/scripts/utils/compss_docker_gen_image
```

From this location, the image can be generated using the following command:

```
./compss_docker_gen_image \
 --image-base="<compss_base_image>" \
 --image-name="<registry/application_image>" \
 --context-dir="<application_context_directory>" \
 --requirements-path="<requirements_file>"
```

The parameters have the following meaning:

- *--image-base* specifies the COMPSs container image used as the base runtime environment.
- *--image-name* defines the name and tag of the generated application image.
- *--context-dir* indicates the directory containing the application source code and input resources.
- *--requirements-path* specifies the Python dependency file required by the application.

The Python dependencies are installed through a standard requirements.txt file. In the EXTRACT use case, the requirements include Lithops and additional packages required by the TASKA-C workflow.

Content of requirements.txt:

```
git+https://github.com/kikemolina3/lithops.git
s3path
psutil
python-casacore
adjustText
boto3==1.28.1
kubernetes
```

```
paho-mqtt
./TASKA-C
```

The resulting image therefore contains the COMPSs runtime, the Lithops framework, and the TASKA-C application environment in a single deployable container image.

3.1.3. Flexecutor

3.1.3.1. Installation

Flexecutor is available as an open-source Python package and can be installed directly from its source repository. Before installing, ensure you have Python 3.10 or later. Since the demonstration example will use one of the TASKA-C pipelines as an example, we must also install the use case dependencies.

```
# Install TASKA-C use case
git clone https://gitlab.bsc.es/extract/extract-use-cases/taska/use-case-c/pipeline.git
cd pipeline
pip install .
# Install Flexecutor
git clone https://gitlab.bsc.es/extract/extract-use-cases/taska/use-case-c/flexecutor.git
cd flexecutor
pip install .
```

3.1.3.2. Execution

1. We need to configure the Lithops config (~/.lithops/config) file for connect out host to k8s and ceph (OVH cluster) backends:

```
lithops:
  backend: k8s
  storage: ceph
  log_level: DEBUG
  monitoring_interval: 1

ceph:
   bucket_name: flexecutor-demo
   endpoint: https://s3.gra.perf.cloud.ovh.net/
   access_key_id: <>
   secret_access_key: <>
   region: gra
```

```
k8s:
  docker_server: registry.gitlab.bsc.es
  runtime_timeout: 18000
  runtime: registry.gitlab.bsc.es/extract/extract-use-cases/taska/use-case-c/pipeline:310
  docker_user: <>
  namespace: taska-c
  docker_password: <>
```

- Replace the placeholders <> by the correct credentials

2. For launch the profiling of the pipeline, execute in bash:

```
PYTHONPATH=. python3 examples/radio_interferometry/scripts/main_classic.py --profile
```

3. For launch the optimization of the pipeline for predict the best configuration, execute in bash:

```
PYTHONPATH=. python3 examples/radio_interferometry/scripts/main_classic.py --optiimize
```

3.2. Data Infrastructure Layer

3.2.1. SkyStore

3.2.1.1. Installation

SkyStore has detailed installation documentation in CONTAINER.md file in the root of the repository folder. This section presents specific setup data and setup choices for deploying SkyStore similarly to what has been demonstrated in EXTRACT, namely, one cloud CCS, one HPC CCS (in BSC premises) and one edge CCS, which was deployed in Venice during the demo.

This section assumes using the tunneler branch of SkyStore repository, which contains explicit support for using non-AWS public and private S3 endpoints, such as OVH S3 and local Minio, by using a custom provider in SkyStore. The details of these features are described in D4.4.

The rest of this section is as follows. Each numbered bullet matches a numbered step in CONTAINER.md. It adds the specific data and/or decisions that need to be used during installation. The overall theme is, of course, multi-cluster Kubernetes installation. The cloud CCS hosts the metadata service, since it can easily export public IP. Each CCS hosts a SkyStore s3-proxy that coordinates through the metadata service.

1. SkyStore container images can be built locally and pushed to a registry of choice. Alternatively, they can be pulled from the EXTRACT private registry on OVH at https://y3z2bluo.gra7.container-registry.ovh.net/harbor/projects with the prefix of y3z2bluo.gra7.container-registry.ovh.net/skystore and the suffix of tun_v6 .
2. The server env.base file is as below. The cloud CCS is using the S3 service of custom:ovh-eu-west. HPC CCS is using aws:eu-west-1 and edge CCS is using custom:minio-local.

```
INIT_REGIONS=custom:minio-local,aws:eu-west-1,custom:ovh-eu-west
SKYSTORE_BUCKET_PREFIX=skystore-extract
```

The s3-proxy env.base files are all the same, as below. Note that SKYSTORE_SRV_ADDR is to be filled after the metadata service is up and has a public IP address.

```
SKYSTORE_SRV_ADDR=
SSH_PORT=32222
SKY_ACCESS_KEY_ID=extract
SKY_SECRET_ACCESS_KEY=SkyStore4Extract
```

Finally, the s3-proxy s3proxy.json files are similar, but not exactly the same. Below is the file for the cloud CCS. The other files have the same content except for the client_from_region field, which changes each time to the specific S3 region used for the respective CCS. For privacy concerns, OVH credentials are not included and can be obtained from EXTRACT coordinator upon request.

```
{
 "init_regions": [
  "custom:minio-local",
  "aws:eu-west-1",
  "custom:ovh-eu-west"
 ],
 "client_from_region": "custom:ovh-eu-west",
 "skystore_bucket_prefix": "skystore-extract",
 "policy": "write_local",
 "server_addr": "127.0.0.1",
 "custom_endpoints": {
  "custom:ovh-eu-west": {
   "endpoint_url": "https://s3.gra.cloud.ovh.net",
   "aws_access_key_id": "<OVH S3 username>",
   "aws_secret_access_key": "<OVH S3 password>",
   "region": "gra"
  },
  "custom:minio-local": {
   "endpoint_url": "http://10.3.86.146:9000",
   "aws_access_key_id": "minio",
   "aws_secret_access_key": "minio123",
   "region": "us-east-1",
   "local_port_fwd": 8012
  }
 }
}
```

3. The Docker deployment step is skipped, since we choose Kubernetes deployment – step 4.

4. Below is a server.yaml file used for deploying the metadata service.

```
# Server architecture similar to S3Proxy - service over deployment
# For now, maximum replica count is 1
# Enables in-place restart and force-pull image
apiVersion: apps/v1
kind: Deployment
metadata:
 namespace: per
 name: skystore-server-dep
spec:
 replicas: 1
 strategy:
  type: Recreate
 selector:
  matchLabels:
   app.kubernetes.io/name: skystore-server
 template:
  metadata:
   labels:
    app.kubernetes.io/name: skystore-server
  spec:
   containers:
    - name: skystore-server-container
      # Change image URL as needed
      image: y3z2bluo.gra7.container-registry.ovh.net/skystore/skystore-server:amd64_tun_v2
      ports:
      - containerPort: 8222
        name: server-c-port
      envFrom:
      - configMapRef:
          name: conf-skystore-server
    # Change or remove serviceAccountName as needed
    imagePullSecrets:
    - name: quay-erezh
---
apiVersion: v1
kind: Service
metadata:
 namespace: per
 name: skystore-service
spec:
 # Change to ClusterIP / NodePort - as needed
 type: LoadBalancer
 selector:
  app.kubernetes.io/name: skystore-server
 ports:
 - name: skystore-service-port
  protocol: TCP
  port: 32222
  targetPort: server-c-port
```

Once the metadata server is deployed and the service is up, take its assigned IP address and update the SKYSTORE_SRV_ADDR field in env.base files for all S3-proxy instances. To complete the deployment of S3-proxy in each CCS, here are the s3proxy.yaml files used for deploying on K8s – first cloud, then HPC, then edge. As in previous examples, the yaml is quite similar for all CCS, with one important difference: the config map in configMapRef, which is created from the respective env files. This is the configuration difference for each service.

Cloud s3proxy.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 namespace: per
 name: s3proxy-ovh-ovh-west-public-dep
spec:
 replicas: 1
 strategy:
  type: Recreate
 selector:
  matchLabels:
   app.kubernetes.io/name: s3proxy-ovh-ovh-west-public-lbl
 template:
  metadata:
   labels:
    app.kubernetes.io/name: s3proxy-ovh-ovh-west-public-lbl
  spec:
   containers:
   - name: s3proxy-ovh-ovh-west-public-ctr
    # Change image URL as needed
    image: y3z2bluo.gra7.container-registry.ovh.net/skystore/skystore-s3proxy:tun_v6
    ports:
    - containerPort: 8002
      name: s3p-ow1-port
    envFrom:
    - configMapRef:
       name: conf-skystore-s3proxy-ovh-ovh-west-public
---
apiVersion: v1
kind: Service
metadata:
 namespace: per
 name: s3proxy-ovh-ovh-west-public-svc
spec:
 type: LoadBalancer
 selector:
  app.kubernetes.io/name: s3proxy-ovh-ovh-west-public-lbl
 ports:
 - name: s3p-ow1-pub-port
  protocol: TCP
  port: 8002
  targetPort: s3p-ow1-port
```

HPC s3proxy.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: per
  name: s3proxy-ovh-eu-west-1-public-dep
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app.kubernetes.io/name: s3proxy-ovh-eu-west-1-public-lbl
  template:
    metadata:
      labels:
        app.kubernetes.io/name: s3proxy-ovh-eu-west-1-public-lbl
    spec:
      containers:
      - name: s3proxy-ovh-eu-west-1-public-ctr
        # Change image URL as needed
        image: y3z2bluo.gra7.container-registry.ovh.net/skystore/skystore-s3proxy:tun_v6
        ports:
        - containerPort: 8002
          name: s3p-ew1-port
        envFrom:
        - configMapRef:
            name: conf-skystore-s3proxy-ovh-eu-west-1-public
---
apiVersion: v1
kind: Service
metadata:
  namespace: per
  name: s3proxy-ovh-eu-west-1-public-svc
spec:
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: s3proxy-ovh-eu-west-1-public-lbl
  ports:
  - name: s3p-ew1-pub-port
    protocol: TCP
    port: 8002
    targetPort: s3p-ew1-port
```

Edge s3proxy.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 namespace: per
 name: s3proxy-ovh-eu-central-1-public-dep
spec:
 replicas: 1
 strategy:
  type: Recreate
 selector:
  matchLabels:
   app.kubernetes.io/name: s3proxy-ovh-eu-central-1-public-lbl
 template:
  metadata:
   labels:
    app.kubernetes.io/name: s3proxy-ovh-eu-central-1-public-lbl
  spec:
   containers:
   - name: s3proxy-ovh-eu-central-1-public-ctr
    # Change image URL as needed
    image: y3z2bluo.gra7.container-registry.ovh.net/skystore/skystore-s3proxy:tun_v6
    ports:
    - containerPort: 8002
     name: s3p-ec1-port
    envFrom:
    - configMapRef:
       name: conf-skystore-s3proxy-ovh-eu-central-1-public
---
apiVersion: v1
kind: Service
metadata:
 namespace: per
 name: s3proxy-ovh-eu-central-1-public-svc
spec:
 type: ClusterIP
 selector:
  app.kubernetes.io/name: s3proxy-ovh-eu-central-1-public-lbl
 ports:
 - name: s3p-ec1-pub-port
  protocol: TCP
  port: 8002
  targetPort: s3p-ec1-port
```

5. From this point on, continue deployment as described in CONTAINER.md to reach a fully-working system.

3.2.1.2. Execution

Each S3-proxy of SkyStore is a fully-functional S3 service. As such, it can be used by all standard S3 clients, both code, CLI and GUI. The credentials are those defined in the above env.base files, e.g., extract/SkyStore4Extract. In CONTAINER.md and in previous reports (D4.2, D4.3) there are several procedures for testing the s3-proxies against the "gold standard" client – the aws CLI. However, in code, EXTRACT code uses either the standard boto3 client library and/or the Nuvla data catalog client, which internally also uses boto3.

3.2.2. Dataplug

3.2.2.1. Installation

The Dataplug library is a client-only framework for dynamic object partitioning, which can be installed with a single pip command.

```
pip install git+https://github.com/CLOUDLAB-URV/dataplug
```

3.2.2.2. Execution

As execution, a preprocessing and dynamic partitioning of a sample MS file can be performed. For execute this task, please see examples/ms_example.py. Modify the target file to point to your MS file and executing the script you'll preprocess and chunk dynamically the object.

3.2.3. Nuvla Data Catalogue

The Nuvla Data Catalogue is a specialized metadata management and discovery service designed to handle distributed data assets across multi-cloud and edge environments. It functions as a centralized control plane that decouples metadata (the descriptive information about the data) from the actual data storage. By providing a "Single Source of Truth," it allows users to index, search, and orchestrate data workflows without needing to move massive datasets into a central repository. This architecture includes optional S3 integration, where Nuvla manages "data-object" resources that point directly to S3-compatible buckets, providing a unified interface for distributed object storage.

3.2.3.1. Integration

As a cloud-native solution, the integration of the Nuvla Data Catalogue primarily involves the configuration of the management environment and the connection of the API into the project's software stack:

- Service Access: Nuvla is accessed via a managed SaaS instance (Nuvla.io) or deployed as a dedicated instance using Docker Compose or Kubernetes for environments requiring strict data sovereignty.
- API Client Setup: Integration on the client side consists of utilizing the "nuvla-api" library. This allows external applications to authenticate and communicate with the catalogue via secure REST endpoints for all data lifecycle operations.

- Infrastructure Configuration: The process is completed by registering the project's storage backends as Infrastructure Service resources within the platform. This includes the linking of S3-compatible buckets, enabling the platform to track and manage the physical location of data files across different providers.

TASKA use case

For the TASKA use case, the python API that "consumes" and "produces" the data records and data objects, has been adapted to serve as a function decorator around the workflow tasks of TASKA-C. First, data ingestion step already filled the data catalogue with data and metadata pointing to the data location on storage nodes. Then, each task is decorated with a Nuvla decorator. The decorator modify the task that are now preceded by a before_func() (generic name) and an after_func() (generic name) that are respectively execute before and after the scientific task. All communication between the data catalogue and the task is performed inside the decorator (see Figure 0). The before_func() role is to query the data catalogue to retrieve the necessary input files required by the task (metadata, S3 location and path). The role of the after_func() is to retrieve the products from the task and publish them to the data catalogue during exporting them to the storage nodes. Each query or publication to the data catalogue is ruled by field values and Nuvla tags.

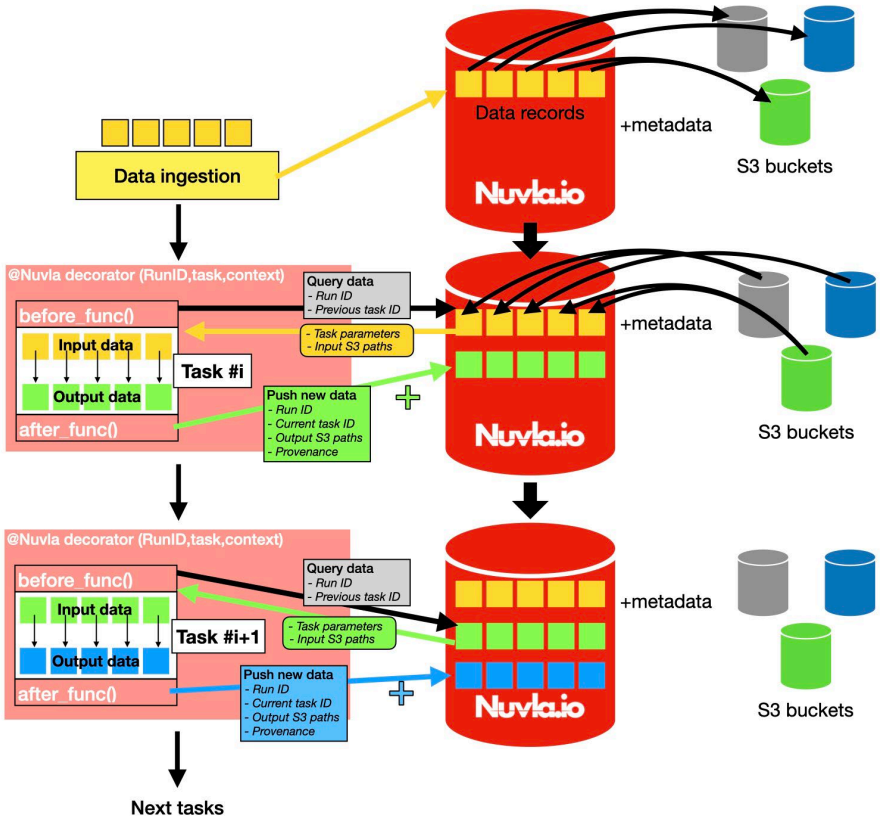

Figure 0: Use Case C implementation of the data catalogue with Nuvla showing data ingestion and the execution of two consecutive tasks with the Nuvla decorator around each task. The before_func() role is to send a query to the data catalogue to ask for the data and metadata necessary for the task. The after_func() role is to retrieve the products of the task and publish them as new data records on the data catalogue while pushing the data to their storage node.

3.2.3.2. Execution

Execution within the Data Catalogue is centered on a metadata-driven lifecycle managed through the REST API. This allows for automated discovery and event-based triggers:

- Registration & Indexing: Data producers execute API calls to create "data-object" and "data-record" resources. While the "data-record" stores JSON-based metadata (such as timestamps or geospatial coordinates), the "data-object" acts as a pointer to the physical file. In S3-integrated setups, this object tracks the bucket location and object key, allowing Nuvla to manage file availability and access permissions centrally.
- Automated Notifications: Nuvla features a robust Notification and Subscription engine that allows the system to react to data events in real-time. By setting up subscriptions via the API, the platform can trigger automated alerts—delivered via MQTT, email, or webhooks—whenever new data matching specific criteria is registered. This enables event-driven architectures where downstream processing starts immediately upon data availability.
- Discovery and Filtering: Users and applications define "data-set" resources, which act as dynamic, saved queries. These sets provide a virtualized view of data across multiple storage providers, filtered by any metadata attribute stored in the catalogue.
- Data Access: When an application identifies the required data, Nuvla can execute a request for a Pre-signed URL. For S3-backed data objects, this provides secure, time-limited access to the raw data directly from the bucket, ensuring high performance and security without the data ever passing through the Nuvla control plane.

Use case C demo

Prerequisites: uv (https://github.com/astral-sh/uv)

```
pip install uv
```

First clone the data catalogue library (standalone library).

```
git clone https://gitlab.bsc.es/extract/extract-use-cases/taska/use-case-c/data-catalogue.git
```

Create an .env file in the data-catalogue folder containing the NUVLA credentials and S3 credentials (See demo information to replace the <KEY> placeholder with demo keys).

```
NUVLA_ENDPOINT = "https://nuvla.io"
NUVLA_KEY = "<KEY>"
NUVLA_KEY_SECRET = "<KEY>"
S3_BUCKET = "testnuvla"
```

```
S3_CREDENTIAL = "<KEY>"
S3_INFRA_SERVICE_ID = "<KEY>"
S3_RUN_BUCKET = "testnuvla-run"
```

Go to the data-catalogue folder and build the package.

```
cd data-catalogue
uv sync
uv pip install -r .
```

The demo scripts are located in data-catalogue/sandbox:

- **demo_create_rawdata.py**: contains the demo script mimicking data ingestion step.

```
uv run python demo_create_rawdata.py
```

- **demo_provenance_with_queries.py:** contains a demo (without use case C code) of a mock data processing workflow consisting of three consecutive tasks. Each tasks use the Nuvla decorator to query or feed the data catalogue with a minimal set of metadata including provenance.

```
uv run python demo_provenance_with_decorators.py
```

- **clear_nuvla.py:** contains the script to query all test data records that has been created during the demo with the field "erasable=true" and clean data records and data objects from the data catalogue.

```
uv run python clear_nuvla.py
```

3.3. Orchestration Layer

3.3.1. Monitoring API

The EXTRACT monitoring infrastructure, developed by Ikerlan, relies on a Kubernetes-based environment to ensure scalability, resilience, and efficient resource management. This section provides detailed instructions on how to install the necessary software and execute the monitoring stack on the cluster. The deployment uses Ansible playbooks to automate installation and configuration tasks.

Prerequisites:

- Ansible
- SSH access to all nodes
- A working Kubernetes cluster
- The inventory file for the cluster, as described previously.

3.3.1.1. Installation

Resource usage

The resource usage of the monitoring infrastructure depends largely on the size of the Kubernetes cluster and the number of monitored nodes. The main components consuming resources are the applications deployed as part of the monitoring stack. Prometheus is the core service responsible for collecting and storing time-series metrics; it requires significant CPU and memory for scraping targets and retaining data. Grafana, used for visualization and dashboard rendering, consumes resources when executing queries and serving user interfaces. Several exporters, such as Kepler Exporter (energy efficiency metrics), NVIDIA Exporter (GPU utilization), and Ping Exporter (network latency), run as pods or DaemonSets on cluster nodes, adding overhead to compute and memory resources. Additionally, the Monitoring API provides integration and data access capabilities, while Helm is used for orchestrating deployments. These components collectively impact CPU, memory, and storage utilization, especially under high-frequency metric collection and intensive dashboard queries.

Deployment

The installation, deployment and execution of the monitoring infrastructure have been automated using Ansible. The following command will prompt for the root password of the target node.

```
ansible-playbook -i inventories/inventory.yaml
playbooks/monitoring/install.yaml --ask-become-pass
```

Roles overview

The automation is organized into roles, each performing a specific task:

- install-helm: Installs Helm, the Kubernetes package manager, enabling the deployment and management of monitoring applications through Helm charts.
- define-storage-class: Creates and configures a Kubernetes StorageClass to manage persistent volumes required by monitoring components.
- install-prometheus: Installs Prometheus, the core monitoring system responsible for scraping metrics from exporters and storing time-series data.
- install-nvidia-exporter: Installs the NVIDIA exporter to gather GPU utilization and performance metrics from nodes equipped with NVIDIA GPUs.
- install-kepler-exporter: Installs the Kepler exporter, which collects energy efficiency and power consumption metrics from cluster nodes.
- install-ping-exporter: Deploys the Ping exporter to measure network latency and connectivity between nodes and external endpoints.
- install-swim-nsm: Installs SWIM NSM (Network State Monitoring) module to monitor latencies and failure rates between the nodes of the architecture.
- install-grafana: Deploys Grafana in the cluster to provide visualization dashboards for metrics collected by Prometheus.
- install-monitoring-api: Deploys the Monitoring API service, which provides an interface for accessing and integrating monitoring data from different sources.

Playbooks

The monitoring platform uses a single playbook to automate the deployment process. This playbook orchestrates all the roles required to install and configure the monitoring stack on the Kubernetes cluster.

- Install.yaml: performs two main tasks;
  - Install Helm on all nodes to enable the deployment and management of Kubernetes resources using Helm charts.
  - Deploy the monitoring architecture on the master node by executing the previously described roles:
    - define-storage-class
    - install-prometheus
    - install-nvidia-exporter
    - install-kepler-exporter
    - install-ping-exporter
    - install-swim-nsm
    - install-grafana
    - install-monitoring-api

Further information about the playbooks and roles that are part of this deployment can be found in the Gitlab repository: https://gitlab.bsc.es/extract/extract-sa/ansible.

3.3.1.2. Execution

Once the cluster is running, there are two main entry points to access the monitoring information:

1. Grafana UI
2. REST API

Grafana UI

The first entry point, Grafana, is a graphical interface where multiple dashboards can be found to visualize different monitoring metrics. Its goal is to provide users with graphical information about the performance of the cluster. Additional dashboards and alerts can be configured from the graphical interface itself to better adjust to the requirements of each use case. Grafana can be accessed from IP:31001.

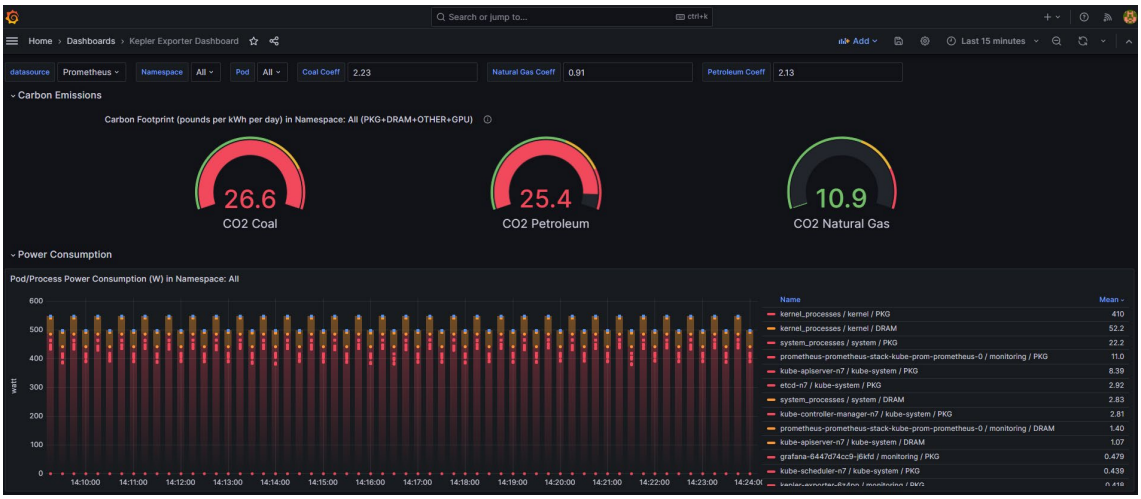

Figure 1: Grafana dashboard for Kepler Exporter

Monitoring REST API

The second entry point consists of a REST API that provides a programmatic interface to access the data required by other components of the Extract architecture. The API has been implemented using Python and FastAPI frameworks ensuring lightweight communication, high performance and automatic generation of OpenAPI documentation. Its main purpose is to expose monitoring, system and infrastructure related data in a standardized format.

The API is organized into three main functional groups: API health, system metrics and Kubernetes infrastructure information, as described below.

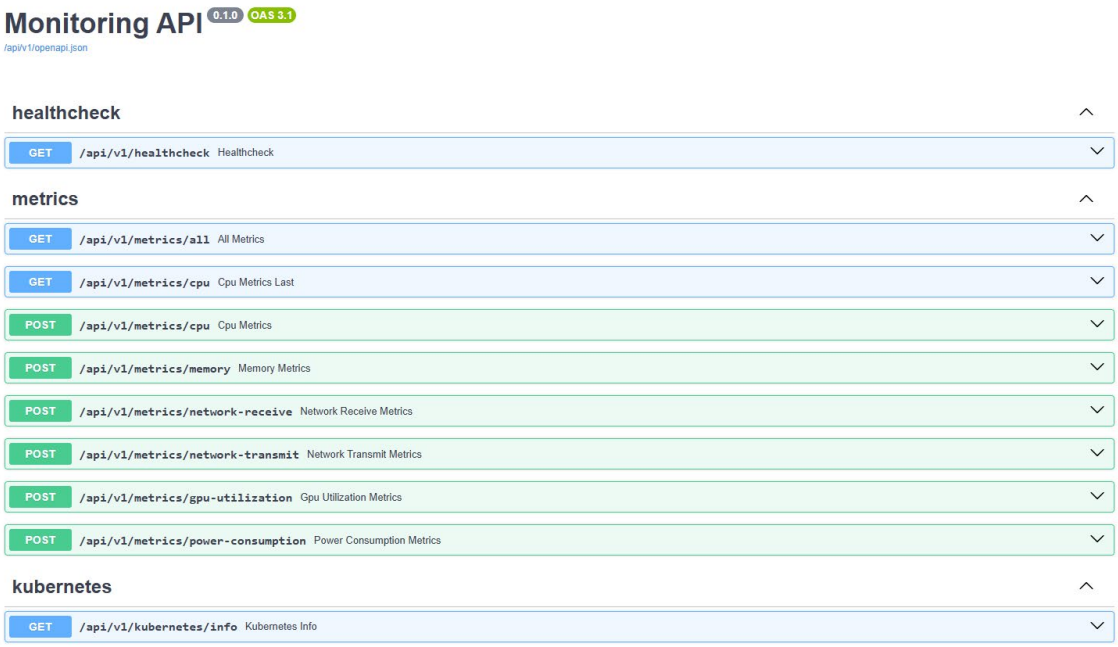

Figure 2: Monitoring API endpoints

The Healthcheck endpoint allows clients to verify the availability and operational status of the API service.

- GET /api/v1/healthcheck: Returns a status indicating whether the API service is operational.

The metrics endpoints provide monitoring data collected by the Prometheus server. External components, such as COMPSS, can leverage these metrics to perform analytics and support decision-making processes.

- GET /api/v1/metrics/all: Returns all available monitoring metrics in a single response.
- GET /api/v1/metrics/cpu: Retrieves the latest CPU usage metrics for specific service. The target service is specified via a query parameter (service_name), allowing clients to obtain CPU metrics associated with an individual component of the system.

All POST metric endpoints share a common request body that defines the time range and resolution for metric retrieval. The request includes the start and end timestamps, the query step used to define the sampling resolution, and the rate interval applied when calculating the rate-based metrics.

- POST /api/v1/metrics/cpu: Retrieves CPU usage metrics from all cluster nodes.
- POST /api/v1/metrics/memory: Retrieves memory usage metrics for all cluster nodes.
- POST /api/v1/metrics/network-receive: Retrieves metrics related to incoming network traffic for all cluster nodes.
- POST /api/v1/metrics/network-transmit: Retrieves metrics related to outgoing network traffic from all cluster nodes.

The Kubernetes endpoints provide information about the Kubernetes Infrastructure.

- GET /api/v1/Kubernetes/info: Returns details about the Kubernetes cluster and its nodes, including resource information such as CPU, memory, and storage capacity.

3.3.2. kubecompss

kubecompss is a command-line interface (CLI) implemented in Go that simplifies the deployment and management of COMPSs applications in Kubernetes single and multiclusters. It provides a set of commands to configure, deploy, and monitor COMPSs runtimes and applications within Kubernetes clusters, automating the creation and management of the required Kubernetes resources.

By abstracting the underlying Kubernetes configuration, *kubecompss* facilitates the execution of COMPSs workloads in containerized environments and supports their integration within the EXTRACT Software Platform. This approach allows COMPSs applications to be deployed consistently across different Kubernetes clusters that form part of the compute continuum.

3.3.2.1. Installation

In order to install kubecompss, you need to install Go.

1. Download the TAR

```
wget https://go.dev/dl/go1.26.1.linux-amd64.tar.gz
```

2. Remove any previous Go installation by deleting the /usr/local/go folder (if it exists), then extract the archive you just downloaded into /usr/local, creating a fresh Go tree in /usr/local/go:

```
rm -rf /usr/local/go && tar -C /usr/local -xzf go1.26.1.linux-amd64.tar.gz
```

3. Add /usr/local/go/bin to the PATH environment variable.

```
export PATH=$PATH:/usr/local/go/bin
```

4. Verify that you've installed Go by opening a command prompt and typing the following command:

```
go version
```

The steps to install the kubecompss CLI are:

1. Clone the kubecompss git repository

```
git clone https://gitlab.bsc.es/extract/extract-sa/kubecompss.git
```

2. Compile the CLI:

```
go build -o kubecompss
```

3.3.2.2. Execution

The kubecompss CLI is used in the EXTRACT project to deploy the PER application in the compute continuum. The command to execute it is:

```
./kubecompss run multicluster \
    --workers=1 \
    --image-name=albabsc/per:compss-3.3.3 \
    --context-dir=/root/multicluster_per/src \
    --cpus=4 \
    --gpus=4 \
    --memory=4 \
    --local-cluster=hpc \
    --remote-workers=1 \
    --remote-cluster=edge \
    --multicluster-tool=linkerd \
    -- -d -g -t main_compss.py -o /data/results
```

The image specified is the one created using the steps described in Section 3.1.1 for COMPSs applications. The image below shows a *kubecompss* execution of the PER application in the *hpc* and *edge* Kubernetes clusters.

```
acanete@workstation:~/kubecompss$ ./kubecompss run multicluster \
    --workers=1 \
    --image-name=albabsc/per:compss-3.3.3 \
    --context-dir=/root/multicluster_per/src \
    --cpus=4 \
    --gpus=4 \
    --memory=4 \
    --local-cluster=hpc \
    --remote-workers=1 \
    --remote-cluster=edge \
    --multicluster-tool=linkerd \
    -- -d -g -t main_compss.py -o /data/results
2026/03/12 15:41:31 Execution summary ----------------
2026/03/12 15:41:31 Workflow ID:              compss-per-5d6ad9
2026/03/12 15:41:31 Local cluster name:       hpc
2026/03/12 15:41:31 Number of workers:        1
2026/03/12 15:41:31 Pod CPU units:            4
2026/03/12 15:41:31 Pod GPU units:            4
2026/03/12 15:41:31 Pod memory:               4 GB
2026/03/12 15:41:31 Image name:               albabsc/per:compss-3.3.3
2026/03/12 15:41:31 Context directory:        /root/multicluster_per/src
2026/03/12 15:41:31 Elastic Workers:          0
2026/03/12 15:41:31 Namespace                 default
2026/03/12 15:41:31 Priority:
2026/03/12 15:41:31 Remote cluster name:      edge
2026/03/12 15:41:31 Number of remote workers: 1
2026/03/12 15:41:31 ---------------------------------
2026/03/12 15:41:31 [Deployer] Creating RBAC resources
2026/03/12 15:41:31 [Deployer] Deploying remote workers
2026/03/12 15:41:31 [Deployer] Executing application in Kubernetes cluster
2026/03/12 15:41:32 [Deployer] Waiting for the application to start...
```

Figure 3: kubecompss CLI execution

3.3.3. Helm Charts

COMPSs is deployed in the TASKA use case using Helm Charts. Helm Charts arepackaging formats for Kubernetes applications, bundling YAML manifests, templates, and configurations into a single, versioned unit to simplify deployment and management. They enable complex application deployment via a single *helm install* command, allowing developers to avoid repetitive copy-pasting of Kubernetes configurations.

The Helm deployment instantiates the COMPSs execution environment in Kubernetes, including the master component, one or more worker components, the required service account and RBAC rules, the internal services used for communication, and the configuration objects required at runtime.

3.3.3.1. Installation

The master Pod runs two containers: one hosting the MQTT broker and another running the COMPSs master process. The worker pods use the same application image, which includes both COMPSs and Lithops, and expose the SSH and COMPSs communication ports required by the execution model.

The parameters required to start the workflow are defined in the Helm chart configuration file *values.yaml*. During deployment, Helm injects these values into the Kubernetes manifests and passes them to the COMPSs master container as startup parameters. These parameters include the namespace, the number of workers, the CPU and memory allocation, the container image to be used, the application directory inside the container, and the workflow execution arguments. This configuration allows the COMPSs runtime to coordinate the execution of the TASKA-C workflow across the deployed worker containers.

In addition, the deployment mounts the Lithops and Kubernetes configuration files inside both master and worker containers through configuration volumes. Environment variables are defined so that the runtime components can locate the configuration files during execution. The Lithops configuration contains the parameters required to connect to the external storage backend (OVH Object Storage) and to specify the container image used by the TASKA-C pipeline. The Kubernetes configuration allows Lithops to interact with the cluster, enabling it to dynamically launch additional Lithops worker pods when required.

Both configuration files are mounted inside the containers as volumes and made available at predefined locations. This approach allows the runtime components to access the required configuration without embedding environment-specific parameters directly into the container image.

A relevant feature of the deployment is that the Kubernetes and Lithops configuration files can be provided externally without modifying the container images. This makes it possible to adapt the deployment to different environments by simply updating the configuration files. To enable this functionality, the following directory must be created within the Helm application directory structure and he user must place the following configuration files in that directory:

```
mkdir –p helm-app/files/
helm-app/files/kube_config.yaml
helm-app/files/lithops_config.yaml
```

3.3.3.2. Execution

To deploy the application in the Kubernetes cluster, the Helm chart must be installed from the directory created using the following command:

```
cd helm-app/files/
helm install <deployment_name> .
```

3.4. Compute Continuum Abstraction Layer

This section provides detailed instructions on how to install the necessary software to deploy a Kubernetes cluster. The deployment uses Ansible playbooks to automate installation and configuration tasks.

3.4.1. Kubernetes

3.4.1.1. Installation

Each partner must define their own inventory file according to their infrastructure. A template is provided in the inventories folder. For more details on inventory configuration, refer to Ansible Inventory Documentation.

Roles overview

The automation is organized into roles, each performing a specific task:

- clean-network: Removes CNI network interfaces
- configure-containerd: Configures containerd with systemd enabled
- configure-node: Prepares network settings for Kubernetes
- install-containerd: Installs containerd runtime
- install-kubernetes: Installs kubeadm, kubelet, and kubectl
- install-cni: Installs CNI binaries
- install-calico / install-flannel: Installs CNI plugins for networking
- kubeadm-init: Bootstraps the Kubernetes cluster
- kubeadm-join: Adds worker nodes to the cluster
- kubeadm-reset: Resets cluster configuration
- delete-kubernetes / delete-cni: Removes Kubernetes and CNI components

Playbooks

Playbooks combine roles and configuration parameters. There are two main categories:

Installation Playbooks

- install-software: Installs Kubernetes dependencies and container runtime
- delete-software: Removes all installed components

Deployment Playbooks

- create-cluster: Creates a Kubernetes cluster (requires master_ip and pod_subnet)
- join-cluster: Adds worker nodes to the cluster
- delete-cluster: Deletes the cluster

Installation Steps

1. Run the following command to install Kubernetes and dependencies on all nodes in the inventory. Set `$USER` to your machine`s user:

```
ansible-playbook -i inventories/edge1.yaml playbooks/install-software.yaml -u $USER --ask-become-pass
```

2. Create the Kubernetes cluster, configure master_ip according to the master IP of your own cluster.

```
ansible-playbook -i inventories/edge1.yaml playbooks/create-cluster.yaml --extra-vars "master_ip=IP pod_subnet=10.244.0.0/16" -u $USER --ask-become-pass
```

3. Add the worker nodes specified in the inventory:

```
ansible-playbook -i inventories/edge1.yaml playbooks/join-cluster.yaml -u $USER --ask-become-pass
```

If it is necessary, the commands to delete the cluster and uninstall the software are provided too.

4. To delete the cluster set in the inventory:

```
ansible-playbook -i inventories/edge1.yaml playbooks/delete-cluster.yaml -u $USER --ask-become-pass
```

5. Uninstall the software previously installed in the inventory.

```
ansible-playbook -i inventories/edge1.yaml playbooks/delete-software.yaml -u $USER --ask-become-pass
```

3.4.1.2. Execution

```
acanete@workstation:~$ kubectl get nodes --context=hpc
NAME           STATUS   ROLES           AGE    VERSION
nano1          Ready    control-plane   511d   v1.29.5
workstation    Ready    <none>          511d   v1.29.5
workstation3   Ready    <none>          93d    v1.29.5
```

Figure 4: HPC cluster

```
acanete@workstation:~$ kubectl get nodes --context=edge
NAME    STATUS   ROLES           AGE    VERSION
agx14   Ready    <none>          135d   v1.29.5
agx15   Ready    <none>          135d   v1.29.5
rpi42   Ready    control-plane   135d   v1.29.5
```

Figure 5: edge cluster

3.4.2. Kubernetes multicluster

Kubernetes multicluster provides the infrastructure abstraction that enables the EXTRACT Software Platform to operate across multiple distributed computing environments. It allows workloads, services, and resources to be deployed and managed consistently across several Kubernetes clusters, supporting execution across the compute continuum. In this context, service connectivity between clusters can be implemented using service mirroring or multicluster networking tools; within the project, Skupper and Linkerd have been tested, although any equivalent service mirroring solution can be used.

3.4.2.1. Installation

MetalLB

MetalLB is a load-balancer implementation for Kubernetes clusters running on bare-metal infrastructure. Since bare-metal Kubernetes deployments do not have access to the cloud provider load balancers available in managed environments (e.g., AWS ELB or GCP Load Balancer), MetalLB provides the functionality required to expose Kubernetes services of type LoadBalancer.

The steps to install MetalLB are:

1. Install the MetalLB manifest (CRDs)

```
kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.15.2/config/manifests/metallb-native.yaml
```

2. Create an *IPAddressPool* and *L2Advertisement* with dynamic IPs in that cluster.

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
    name: my-ip-pool
    namespace: metallb-system
spec:
    addresses:
        - 192.168.50.200-192.168.50.205
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
    name: my-l2-advertisement
    namespace: metallb-system
spec: {}
```

```
kubectl apply –f metallb-config.yaml
```

Linkerd

Linkerd is an open-source service mesh designed for Kubernetes that provides secure and observable communication between services running in a cluster. It works by injecting a lightweight proxy alongside each service instance (a sidecar), which intercepts and manages the network traffic between services. In multi-cluster environments, Linkerd can also be configured to support cross-cluster service discovery and communication, allowing services running in different Kubernetes clusters to interact as if they were part of the same service mesh. This capability enables distributed deployments while maintaining consistent networking, security, and monitoring across clusters.

The steps to install Linkerd are:

1. Download Linkerd CLI

```
curl --proto '=https' --tlsv1.2 -sSfL https://run.linkerd.io/install-edge | sh

export PATH=$HOME/.linkerd2/bin:$PATH

linkerd check --pre
```

2. Install the step CLI to create certs

```
export ARCH=[[ arm64 | amd64 ]]

wget https://dl.smallstep.com/gh-release/cli/gh-release-header/v0.28.2/step-cli_0.28.2-1_$ARCH.deb

sudo dpkg -i step-cli_0.28.2-1_$ARCH.deb
```

3. Create the certs

```
mkdir -p /tmp/linkerd-certs/

cd /tmp/linkerd-certs/

step certificate create root.linkerd.cluster.local root.crt root.key --profile root-ca --no-password --insecure
```

4. Using this root key and certificate, we can create a certificate authority (CA) that will be used to sign the certificates for the Linkerd control plane and data plane proxies.

```
step certificate create identity.linkerd.cluster.local issuer.crt issuer.key --profile intermediate-ca --not-after 8760h --no-password --insecure --ca root.crt --ca-key root.key
```

5. Install Gateway API in the Kubernetes clusters.

```
for ctx in edge hpc; do

    kubectl --context=${ctx} apply -f https://github.com/kubernetes-sigs/gateway-api/releases/download/v1.4.0/standard-install.yaml

done
```

6. Install the Custom Resource Definitions in the Kubernetes clusters.

```
for ctx in edge hpc; do
    linkerd install --crds  --context=${ctx} | tee >(kubectl --context=${ctx} apply -f -)
done
```

7. Install linkerd using the created certs in the Kubernetes clusters.

```
for ctx in edge hpc; do
    linkerd install \
    --context=${ctx} \
    --identity-trust-anchors-file /home/acanete/linkerd-certs/root.crt \
    --identity-issuer-certificate-file /home/acanete/linkerd-certs/issuer.crt \
    --identity-issuer-key-file /home/acanete/linkerd-certs/issuer.key \
    --set proxyInit.iptablesMode=legacy \
    | tee \
    >(kubectl --context=${ctx} apply -f -)
done
```

8. Install the linkerd multicluster extension on both clusters

```
controllers:
- link:
    ref:
      name: edge
```

```
linkerd --context=hpc multicluster install -f values-edge.yaml | kubectl --context=hpc apply -f -
```

```
linkerd --context=edge multicluster install | kubectl --context=edge apply -f -
```

9. Link the clusters

```
linkerd --context=edge multicluster link-gen --cluster-name edge | kubectl --context=hpc apply -f -
```

10. Bi-directional link

```
controllers:
```

```
- link:
    ref:
      name: hpc
```

```
linkerd --context=edge multicluster install -f values-hpc.yaml | kubectl --context=edge apply -f -
```

11. Link the clusters bi-directionally

```
linkerd --context=hpc multicluster link-gen --cluster-name hpc | kubectl --context=edge apply -f -
```

Skupper

Skupper is an open-source networking tool that enables secure communication between services running in different Kubernetes clusters, cloud environments, or edge locations. It creates a virtual application network that connects services across these environments without requiring direct network connectivity or complex changes to existing infrastructure. Skupper operates by deploying routers inside Kubernetes clusters that establish secure connections using mutual TLS. These routers expose selected services to the virtual network and make them accessible from other connected clusters through service mirroring. From the perspective of applications, the mirrored services appear as local services within the cluster.

The steps to install Skupper are:

1. Install Skupper CLI

```
curl https://skupper.io/install.sh | sh
```

2. Install Skupper controller and CRDs in the clusters

```
for ctx in edge hpc; do
    kubectl --context=${ctx} apply -f https://skupper.io/v2/install.yaml
done
```

3. Create namespace. A *Skupper site* is created in whatever Kubernetes namespace your kubectl context is currently using (normally `default`). If you don't want Skupper to be installed in the current namespace, you have to create the one you'll use beforehand.

```
for ctx in edge hpc; do
    kubectl --context=${ctx} create ns skupper
done
```

4. Create sites. A *Skupper site* is a place where components of your application are running. Sites are linked to form application networks.

```
for ctx in edge hpc; do
    skupper --context=${ctx} --namespace=skupper site create ${ctx}
done
```

5. Linking sites. Link access has to be enabled on the site where you want to issue the token. To link sites, you have to create a token on one site and redeem that token on the other site. **The link direction is not significant**, and is typically determined by ease of connectivity. For example, if edge is behind a firewall, linking from hpc to edge is the easiest option.

```
skupper --context=hpc --namespace=skupper site update --enable-link-access
```

Create token

```
skupper --context=hpc --namespace=skupper token issue skupper.token
```

Link edge and hpc with the token created

```
skupper --context=edge --namespace=skupper token redeem skupper.token
```

3.4.2.2. Execution

Linkerd

The figures below show the pods deployed in the *linkerd* and *linkerd-multicluster* namespaces of the *hpc* and *edge* Kubernetes clusters. These namespaces contain the core components of the Linkerd service mesh and the additional controllers required to enable communication between multiple Kubernetes clusters. The running pods indicate that the Linkerd control plane and the multi-cluster extension have been successfully deployed and are operational.

```
acanete@workstation:~$ kubectl --context=hpc -n linkerd get pods
NAME                                     READY   STATUS    RESTARTS   AGE
linkerd-destination-6d649d5854-4sltp     4/4     Running   0          8d
linkerd-identity-6c77d6cc6-vhhxz         2/2     Running   0          7d23h
linkerd-proxy-injector-9bbc7b76d-p84d2   2/2     Running   0          7d23h
acanete@workstation:~$ kubectl --context=hpc -n linkerd-multicluster get pods
NAME                                     READY   STATUS    RESTARTS        AGE
controller-edge-5f49cdd579-mmw5c         2/2     Running   2 (4h22m ago)   24h
linkerd-gateway-675c48c975-bmfqr         2/2     Running   0               8d
linkerd-local-service-mirror-5cb96655b4-bn9xq   2/2     Running   0          31s
```

Figure 6: Linkerd running in the HPC cluster

```
acanete@workstation:~$ kubectl --context=edge -n linkerd get pods
NAME                                     READY   STATUS    RESTARTS   AGE
linkerd-destination-65695d9f47-bphrf     4/4     Running   0          8d
linkerd-identity-6c77d6cc6-slsv4         2/2     Running   0          8d
linkerd-proxy-injector-6df8cc7987-zmvk5  2/2     Running   0          8d
acanete@workstation:~$ kubectl --context=edge -n linkerd-multicluster get pods
NAME                                      READY   STATUS    RESTARTS      AGE
controller-hpc-757fdc4c6f-nzmlb           2/2     Running   3 (12h ago)   8d
linkerd-gateway-675c48c975-dz78f          2/2     Running   0             8d
linkerd-local-service-mirror-5cb96655b4-sd2h6   2/2     Running   0             8d
```

Figure 7: Linkerd running in the edge cluster

Skupper

The figure below shows the pods deployed in the skupper namespace of the *hpc* and *edge* Kubernetes cluster. This namespace contains the components responsible for establishing the Skupper virtual application network, including the routers and service controllers that enable communication between services running in different clusters. The running pods indicate that the Skupper infrastructure required for cross-cluster connectivity has been successfully deployed and is operational.

```
acanete@workstation:~$ kubectl --context=hpc -n skupper get pods
NAME                                  READY   STATUS    RESTARTS         AGE
skupper-controller-6dc9c9bd9b-cqjx5   1/1     Running   4 (3y11d ago)    135d
acanete@workstation:~$ kubectl --context=edge -n skupper get pods
NAME                                  READY   STATUS    RESTARTS      AGE
skupper-controller-6dc9c9bd9b-rh26k   1/1     Running   4 (56y ago)   135d
```

Figure 8: Skupper running in the HPC and edge clusters

3.4.3. Security

3.4.3.1. Trusted Platform Module (TPM) security module

3.4.3.1.1. Installation

Overview

The Trusted Platform Module (TPM) 2.0 is a hardware-based security feature defined by the Trusted Computing Group (TCG) that protects system integrity and data. It can be implemented as a chip or firmware on a motherboard and supports secure boot, disk encryption, and more.

Key Features

- Cryptography: Secure key generation, storage, hashing, signing, encryption/decryption, and support for modern algorithms like RSA and ECC.
- Secure Storage: Safely stores sensitive keys, certificates, and measurements.
- Attestation: Verifies system state to external parties using PCRs.
- Random Number Generation: Hardware-based RNG for cryptographic operations.

- Platform Integrity: Measures and records hardware/software integrity during boot and runtime.

Demo setup

- TPM 2.0 simulator (mssim) running in a Docker container.
- Ansible to automate TPM initialization, key sealing, and API container setup.
- PCRs 0, 1, 2 to measure system state.
- Sealed objects to protect API keys.
- Flask API to unseal the key and use it.

The flow

- Launch TPM simulator in a container.
- Initialize TPM and create a primary key.
- Generate a PCR policy covering PCRs 0, 1, 2.
- Seal an API key (from an encrypted Ansible Vault) with this PCR policy.
- Load the sealed object in TPM.
- Use an API endpoint to unseal the key and call API, verifying integrity via PCRs.

Platform Configuration Registers (PCRs) Overview

- PCRs are internal registers in the TPM that store measurements (hashes) of system components or configuration states.
- Once a PCR is extended with a value, its previous state cannot be recovered, making it tamper-evident.
- PCRs are commonly used in sealing secrets: a secret can be sealed to specific PCR values, and it will only unseal if the PCRs match the expected state.

PCR 0, 1 and 2

- These PCRs are part of the TPM specification for platform integrity, but their exact use can vary depending on the system or simulator:
- PCR 0: BIOS / Boot firmware. Hash of the BIOS/UEFI or firmware code. Ensures the system booted with an expected firmware.
- PCR 1: Bootloader / MBR / Boot config. Hash of the bootloader, GRUB, or other boot components. Ensures bootloader hasn't been tampered with.
- PCR 2: Optionally OS loader or early OS environment. Hash of the OS kernel, initrd, or early boot scripts. Ensures OS boot integrity.
- These PCRs are often measured automatically during a real TPM boot.
- In the demo, PCRs 0, 1, and 2 simulate the integrity measurements of BIOS/boot firmware, bootloader, and kernel. This ensures that the TPM-bound API key can only be unsealed if the simulated environment matches the expected startup state.

Prerequisites

- ansible + ansible galaxy community.docker
- docker

Build the API

```
docker compose build
```

3.4.3.1.2. Execution

Create the encrypted Ansible vault

```
ansible-vault create vault.yaml
```

- give a passcode
- add valid api_key entry $ api_key: key_here
- in the demo, we use the service with a valid API key.

Run the playbook

```
ansible-plabook playbook.yaml --ask-vault-pass
```

- API and tpm container are created

```
❯ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                                    NAMES
5a8b321ce77a   tpm-demo-api                   "python app.py"          29 minutes ago   Up 29 minutes   0.0.0.0:5000->5000/tcp                   api
2b3e97e86357   tpmdev/tpm2-runtime:latest     "tpm_server -port 23…"   2 hours ago      Up 29 minutes   0.0.0.0:2321-2322->2321-2322/tcp         tpm-sim
```

Making a query with key

```
curl -X POST -F "cve=CVE-2021-4287" http://localhost:5000/search-with-key
```

Making a query without sealed key

```
curl -X POST -F "cve=CVE-2021-4287" http://localhost:5000/search-no-key
```

```
❯ curl -X POST -F "cve=CVE-2021-4287" http://localhost:5000/search-no-key
{"result":{"detail":"Not authenticated"}}
```

TPM tampering demo

- In this example, we tampered with the kernel PCR after creation

```
❯ docker exec api tpm2_pcrextend 2:sha256=bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

- Thus, we received a unsealing error from the API

```
❯ curl -X POST -F "cve=CVE-2021-4287" http://localhost:5000/search-with-key
{"error":"Bad Request","message":"TPM error.","status":400}
```

- The TPM reports the following error

```
172.21.0.1 - - [13/Mar/2026 11:29:42] "POST /search-no-key HTTP/1.1" 200 -
name: 000b95ef938084837722912cd3e48c12a25b26105915581b3c7369da810a1b652d29
91079d01724b3a85b14c8ea3e0fd7f3c1fde166727c1da1cdc0ae329f549da38
WARNING:esys:src/tss2-esys/api/Esys_Unseal.c:295:Esys_Unseal_Finish() Received TPM Error
ERROR:esys:src/tss2-esys/api/Esys_Unseal.c:98:Esys_Unseal() Esys Finish ErrorCode (0x0000099d)
ERROR: Esys_Unseal(0x99D) - tpm:session(1):a policy check failed
ERROR: Unable to run tpm2_unseal
[2026-03-13 12:14:23,635] ERROR in app: Exception on /search-with-key [POST]
```

4. Conclusion

This deliverable concludes the development cycle of the EXTRACT Software Platform by providing the final, consolidated release of the platform and the complete installation instructions for all its components. The installation guidance is intended to be sufficient for reproducible deployments across the target environments, covering prerequisites, configuration, deployment steps, and post-installation validation.

The provided instructions describe how to install and configure each platform component, while also documenting how components integrate. With these installation instructions, adopters can set up the full EXTRACT Software Platform from a clean environment, verify correct operation using the included validation steps, and perform routine operational tasks such as upgrades, backup/restore, and basic troubleshooting. As a result, D4.5 supports the platform's final handover by ensuring that all components can be installed consistently, operated reliably, and maintained over time, enabling the EXTRACT platform to be deployed by both technical teams and operational administrators in real-world settings.

5. Acronyms and Abbreviations

API - Application Programming Interface

AWS - Amazon Web Services

CCS - Computing Continuum Site

CNI - Container Network Interface

CPU - Central Processing Unit

CRD - Custom Resource Definition

ELB - Elastic Load Balancing

EU - European Union

GPU - Graphics Processing Unit

GUI - Graphical User Interface

HPC - High-Performance Computing

HTTP - Hypertext Transfer Protocol

JWT - JSON Web Token

MQTT - Message Queuing Telemetry Transport

MS - Measurement Set

PCR - Platform Configuration Register

RBAC - Role-Based Access Control

REST - Representational State Transfer

RNG - Random Number Generator

SaaS - Software as a Service

SSH - Secure Shell

SWIM NSM - SWIM Network State Monitoring

TLS - Transport Layer Security

TPM - Trusted Platform Module

TCG - Trusted Computing Group

UEFI - Unified Extensible Firmware Interface

WP - Work Package

YAML - YAML Ain't Markup Language

6. References

[1] Linkerd. linkerd multicluster CLI reference. (linkerd.io)

[2] MetalLB. Installation (bare metal load-balancer for Kubernetes). (metallb.io)

[3] Skupper. Skupper v2 (installation and overview). (skupper.io)

[5] Grafana Labs Documentation. Prometheus data source. (grafana.com)

[6] AWS S3 API. (https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html)