6GSNS

Co-funded by the European Union

SAFE-6G

SAFE-6G

A Smart and Adaptive Framework for Enhancing Trust in 6G Networks

Deliverable D3.5: MLOps for User-centric 6G System with differential privacy

Date: 30/06/2025

Version: V1.0

DISCLAIMER

Grant Agreement	101139031
Document number	D3.5
Document title	MLOps for User-centric 6G System with differential privacy
Lead Beneficiary	BULL
Editor(s)	Joaquín Cáceres Galán (BULL) Alejandro Fornés (UPV)
Author(s)	Joaquín Cáceres (BULL) Jose Luis Carcel (BULL) Stelios Erotokritou (8BELLS) Christos Betzelos (UNIWA) Francisco Mahedero (UPV) Spyridon Georgoulas (NCSRD) Dimitrios Roidis (NCSRD) Harilaos Koumaras (NCSRD)
Dissemination level	Public
Contractual date of delivery	30/06/2026
Status	Final
File name	SAFE-6G_D3.5_V1.0.pdf

Revision History

Version	
V0.1	TOC proposal, guidelines and templates
V0.2	First round of contributions for MLOps and Differential Privacy
V0.3	Content refinement and edition
V1.0	Final version after Internal review (UPV, INF), ready for submission

GLOSSARY

Abbreviations/Acronym	Description
AI	Artificial Intelligence
API	Application Programming Interface
CD	Continuous Delivery
CPU	Central Processing Unit
CI	Continuous Integration
DAG	Directed Acyclic Graph
DP	Differential Privacy
GDP	Global Differential Privacy
GPU	Graphics Processing Unit
GUI	Graphical User Interface
LDP	Local Differential Privacy
LSTM	Long Short-Term Memory
ML	Machine Learning
MLOps	Machine Learning Operations
PDE	Pipeline Development Platform
POP	Pipeline Orchestration Platform
RAM	Random Access Memory
REST	Representational State Transfer
SDK	Software Development Kit

EXECUTIVE SUMMARY

This document belongs to the second round of WP3 deliverables. In contrast to the first round where one single document contained the report of all WP tasks, now each of them has produced their own one. Specifically, D3.5: MLOps for User-centric 6G System with differential privacy, reports the final outcomes (mostly software) of Task 3.5.

The deliverable provides technical details of different aspects related with the integration of the MLOps framework within the 6G system architecture including the design, interfaces, models and libraries. Additionally, reports the final implementation of the differential privacy module.

KEYWORDS

MLOps, Differential Privacy, Machine Learning, Kubeflow, Model Serving

List of Figures

List of Tables

1 INTRODUCTION

Figure 1 highlights in green the building blocks of the SAFE-6G architecture implemented in WP3. The final implementation of their components is reported in the concurrent deliverables from the Work Package, being this one focused on the **MLOps platform** and the **Differential Privacy module** (stressed in yellow). The MLOps platform supports the preparation, training and deployment of ML models for the rest of the components of the SAFE-6G framework, namely those from the Chatbot, the Cognitive Coordinator and the AI agents of the Trust Functions. This deliverable offers a high-level overview of the technical implementation of their components, summarising their role within the SAFE-6G architecture, main final features, design choices changes and key implementation resources to facilitate their replication.

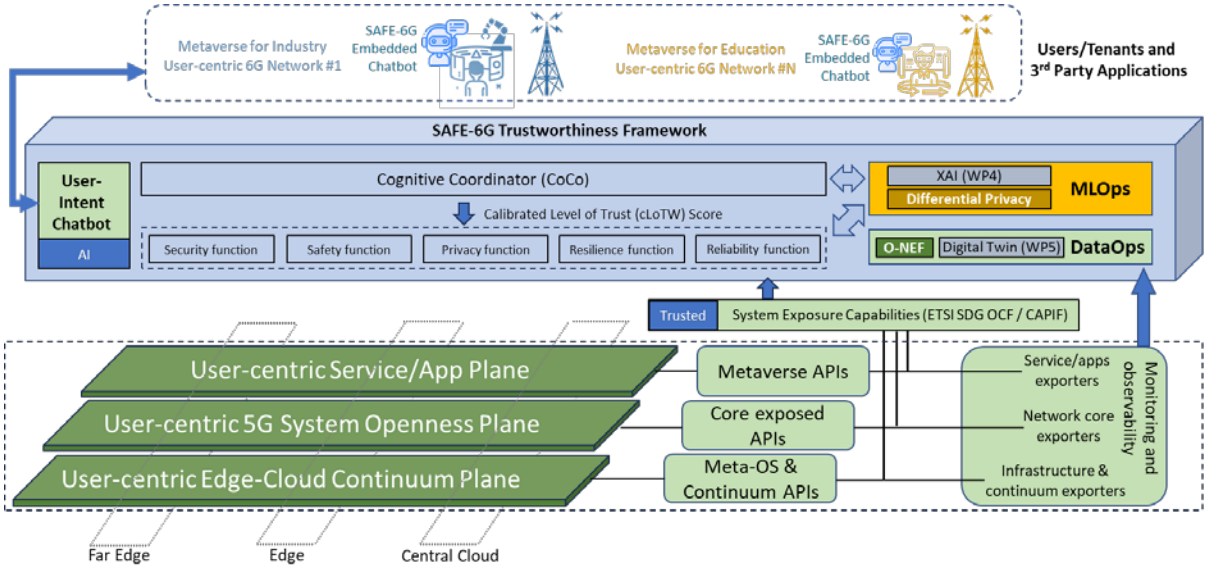

Figure 1: Building blocks and components of the SAFE-6G architecture

1.1 THE RATIONALE BEHIND THE STRUCTURE

The structure of this deliverable is as follows. Section 2 introduces the MLOps platform and the main components and modules that compose it. In particular, Section 2.1 describes the final implementation of the core MLOps platform, highlighting the main components, interfaces, models and libraries used in its design. Section 2.2 focuses on the development of the Differential Privacy module, explaining its architecture, operational behaviour and the different capabilities it provides within SAFE-6G.

Section 3 presents the operational integration and validation activities carried out for both the MLOps platform and the Differential Privacy module, demonstrating their deployment, integration and usage within the SAFE-6G ecosystem. Finally, Section 4 concludes the document and summarizes the main outcomes of this deliverable.

2 MLOps Platform Implementation

This section provides an overview of the functional blocks and main components of the SAFE-6G MLOps platform. The architecture remains consistent with the one described in D3.1, Section 8.1 (Overview and Main Features), with no new components introduced.

Within the SAFE-6G architecture, the core MLOps platform developed by BULL acts as the enabling layer that manages and supports the lifecycle of AI models across the edge-cloud continuum. The solution has been designed as a modular and cloud-native platform deployed on top of Kubernetes, facilitating the integration of AI capabilities into the SAFE-6G ecosystem while ensuring controlled and traceable model management through systematic versioning and automated deployment processes. The platform is intended to be used by AI/ML model developers within SAFE-6G, primarily those developing Trust Functions, as well as components such as the Chatbot or the Cognitive Coordinator.

In addition to the core MLOps capabilities, the platform was extended through the integration of the Differential Privacy module developed by 8BELLS, which introduces privacy-preserving functionalities for AI/ML workflows by protecting sensitive data during model training and inference processes.

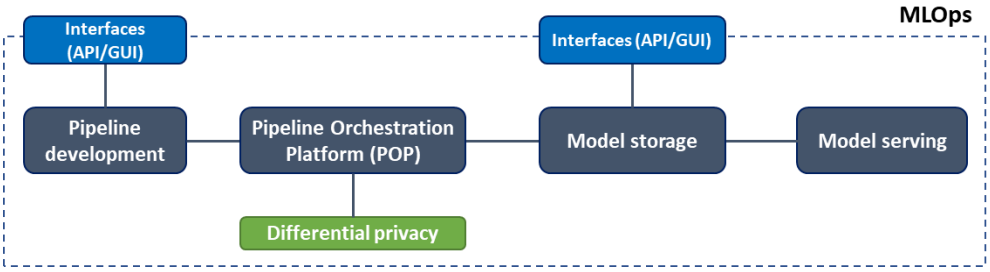

Figure 2: High-level architecture of the SAFE-6G MLOps platform (as introduced in D3.1)

As reported in D3.1 and illustrated in Figure 2, the SAFE-6G MLOps platform is structured around a set of functional blocks that collectively enable the development, training, storage and deployment of AI models across the project:

- **Pipeline Development Environment (PDE)**, providing interactive capabilities for data exploration, model development and pipeline configuration.
- **Pipeline Orchestration Platform (POP)**, enabling automated execution, scheduling and monitoring of ML workflows through containerized tasks deployed on Kubernetes.
- **Model Storage**, acting as a central registry for trained models and associated artifacts.
- **Model Serving**, enabling scalable deployment of trained models as microservices for real-time inference.
- **Differential Privacy (DP) Module**, supporting privacy-preserving ML by applying Local Differential Privacy techniques during data pre-processing.

The SAFE-6G MLOps platform builds upon an existing MLOps platform previously developed by BULL and available as part of the project background technologies. Consequently, the work performed within SAFE-6G has not focused on developing the platform from scratch, but rather on deploying, integrating and extending the existing platform to satisfy the project requirements. In particular, the baseline components were deployed and configured within the SAFE-6G Kubernetes infrastructure and integrated with the project ecosystem to enable interoperability with the different platform

components and Trust Functions. Furthermore, additional assets were developed to facilitate platform adoption by project partners, including reusable templates, pre-configured libraries and usage guidelines aimed at reducing the learning curve associated with MLOps workflows. The contributions to the SAFE-6G MLOps platform can be grouped as follows:

- PDE, POP and Model Storage: These components are based on the background MLOps platform developed by BULL. Within SAFE-6G, they were deployed and configured in the project infrastructure, integrated with the overall platform architecture, and complemented with reusable templates, supporting libraries and user guidelines to simplify their adoption and usage by project partners.
- Model Serving: This component was developed within SAFE-6G to enable the operational deployment of AI/ML models. Particular emphasis was placed on its integration with Meta-Ops, allowing trained models to be packaged and distributed as Docker images and Helm charts for automated deployment and scalable inference execution within Kubernetes environments.
- Differential Privacy Module: This component extends the baseline MLOps platform with privacy-preserving capabilities by applying differential privacy mechanisms during data processing workflows, enabling AI/ML model development while protecting sensitive information.

2.1 CORE MLOPS PLATFORM

This section focuses specifically on the final implementation of the core MLOps platform developed by BULL, providing a detailed description of the components that implement the capabilities introduced in the previous section. The MLOps components have been implemented following the requirements defined in Deliverable D2.1, in particular:

- **REQ-INFRA-RES-NF-M-02:** Support of Cloud Native technologies.
- **REQ-COM-DES-F-M-04**: Following Cloud Native paradigm.
- **REQ-COM-ETH-NF-M-05:** Conformance with the AI act.
- **REQ-COM-DES-F-R-09**: Interface to MLOps framework to train and execute ML models.
- **REQ-COM-DES-F-M-12**: Interoperability Requirement.
- **REQ-COM-RES-NF-M-15:** Provisioning of processing resources.
- **REQ-COM-DES-NF-O-16:** GitOps Monitoring Requirement.
- **REQ-COM-DES-F-M-19**: Access to performance metrics from 5G/6G core infrastructure.
- **REQ-COM-DES-F-M-20**: Access to performance metrics from the cloud continuum.
- **REQ-COM-DES-F-M-21**: Access to performance metrics from user application.
- **REQ-COM-DES-F-R-24:** Fault Tolerance.
- **REQ-COM-RES-NF-M-26:** Data persistence.
- **REQ-MLOPS-DEV-F-M-03**: AI/ML model and dataset provided by the user.
- **REQ-MLOPS-INF-F-M-07**: Inference connectivity.
- **REQ-MLOPS-MLLIB-F-M-10:** Large database to store ML models.

Based on these requirements, D3.1 provided an initial release of the MLOps platform implemented between M1 and M16 (Rel-A), extended during second execution period based on feedback received from project partners (M17-M30, Rel-B).

- Rel-A: **Deployment and configuration of an interactive platform for data exploration and model development**, including the creation of training pipelines that can be executed on demand.
- Rel-A: **Deployment of a storage component** to manage and store both trained models and datasets.
- Rel-A: **Implementation of a complete CI/CD-based deployment workflow**, including model containerisation as Docker images, model versioning for full traceability and Helm charts for simplified deployment in Kubernetes environments.
- Rel-A: **Deployment of the MLOps platform in third-party infrastructures**, including installation in GPU-enabled Kubernetes clusters to meet the computational needs of the project partners.
- Rel-B: **Development and documentation of a unified Helm** chart to provide a common deployment mechanism for all partners when deploying their models. This Helm chart also integrates all the required configurations to ensure compatibility with the Meta-OS (aeriOS).
- Rel-B: **Update of the GitLab CI/CD pipelines of the Model Serving component to complete the integration with the Meta-OS**. These pipelines now include direct calls to the Meta-OS API, enabling the automated deployment of models in the production environment.
- Rel-B: **Continuous integration support** has also been provided to all partners for the onboarding of their models into the MLOps platform, assisting them throughout the full lifecycle, including model development, training pipeline definition, and deployment.

A migration to Keycloak[1] as a common authentication solution was considered to centralise access management. However, since the MLOps platform is primarily used by project developers rather than end users, it was decided to maintain a dedicated authentication system, Dex[2], with developer accounts managed independently from the rest of the SAFE-6G components.

Based on these updates, the following sub-sections focus on describing in detail the final implementation of the components that compose the MLOps platform, including their main building blocks and the capabilities they provide.

2.1.1 Pipeline development Environment

The first component of the platform is the Pipeline Development Environment (PDE). This component provides an interactive workspace for data exploration, model development and initial pipeline configuration. It allows data scientists and developers to experiment with different model architectures, preprocess datasets and validate training workflows before their integration into automated pipelines. It has been built on top of **Kubeflow[3] Notebooks**, which enables the deployment of web-based development environments on Kubernetes. The platform provides three selectable interfaces (JupyterLab, Visual Studio Code and RStudio), giving users flexibility to work with the tool that best matches their preferred workflow, programming language and data science stack. This approach allows both notebook-based experimentation and full IDE-based development within the same cloud-native environment.

[1] Keycloak documentation: https://www.keycloak.org/guides
[2] Dex documentation: https://dexidp.io/docs/
[3] Kubeflow documentation: https://www.kubeflow.org/docs/

The authentication mechanism to access this environment is based on **Dex**, which role is to ensure that only authorized users can access the development workspace. After authentication, users can create, configure and manage their instances directly from the Kubeflow interface.

When creating a notebook instance, users can tailor computational resources to the needs of their experiments. This includes selecting the CPU allocation (number of cores and memory), optionally requesting GPU acceleration, and choosing from a set of container images that include pre-installed machine learning frameworks and libraries (e.g., TensorFlow, PyTorch or custom environments). In addition, users can define the amount of persistent storage required for their experiments. This storage is provisioned in the Kubernetes cluster as a **Persistent Volume**, guaranteeing data persistence across sessions and enabling the preservation of experiment history and intermediate results. This flexibility allows the development environment to be adapted to the specific requirements of each model and experimentation workflow. Figure 3 shows the resource configuration interface available to users, including the selection of computing resources, storage capacity and container images.

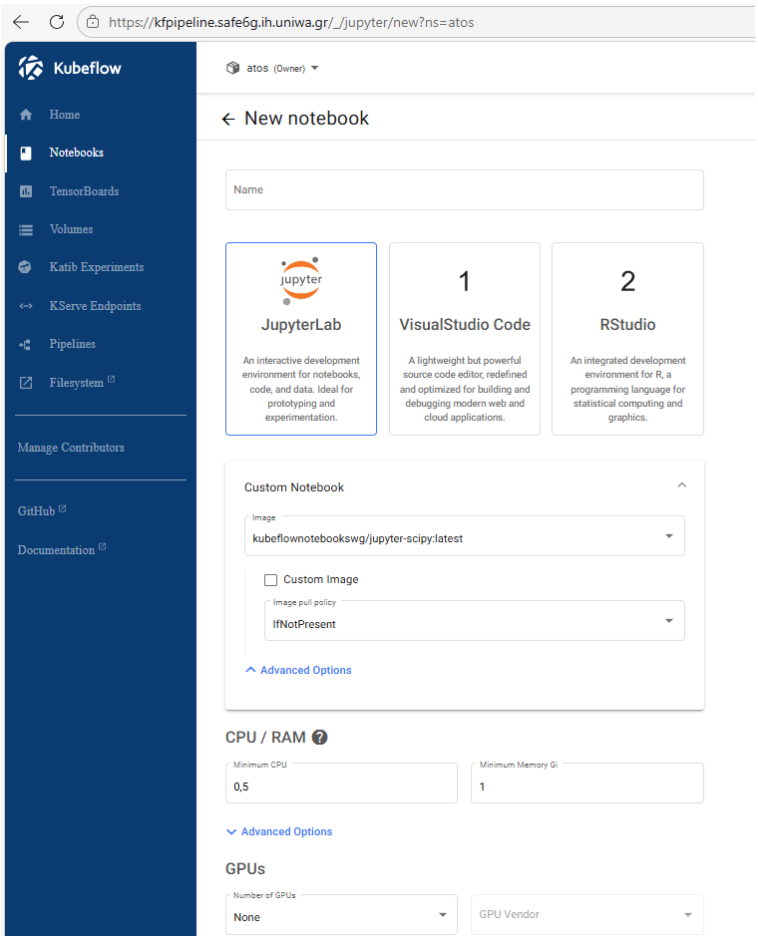

Figure 3: Resource configuration view in Kubeflow pipeline GUI

Once the notebook instance is deployed, users can access a fully functional web-based development workspace where they can interactively explore datasets, develop and test models, and prepare training pipelines prior to their automated execution within the MLOps platform. Figure 4 presents an example of a running notebook environment within Kubeflow.

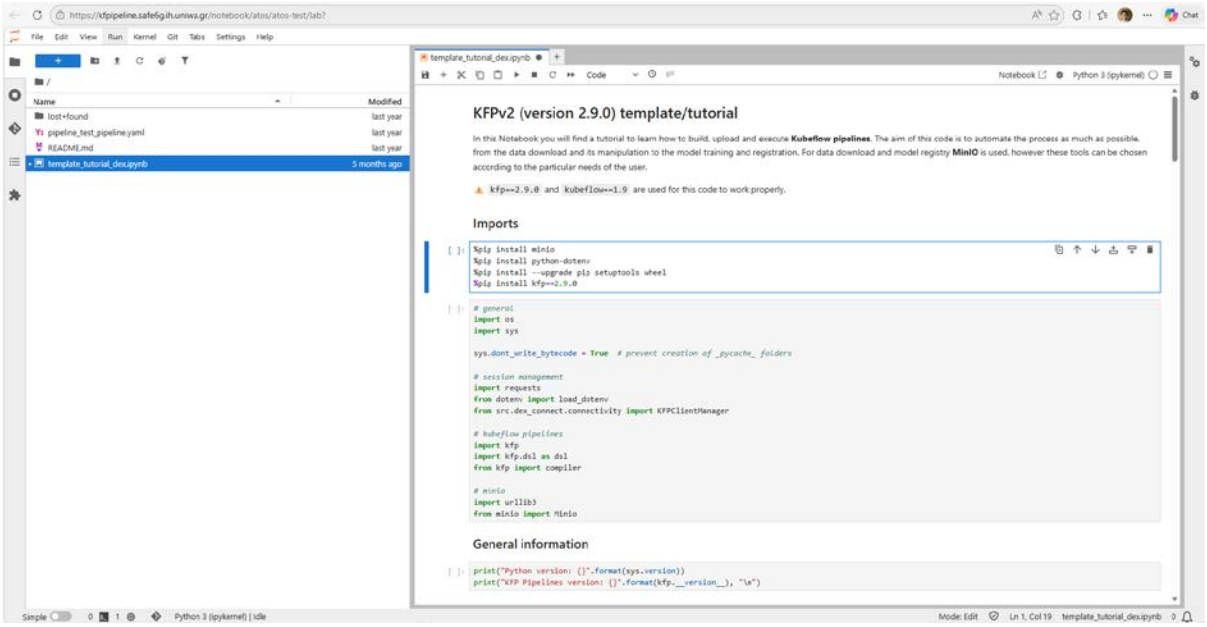

Figure 4: An example of a running notebook within Kubeflow

To complement the description of the component, Table 1 summarizes the main technical and operational information of the Pipeline Development components.

Category	*Status*
Version	V2.0.0-rc.1
Documentation	Official User Guide: https://www.kubeflow.org/docs/components/notebooks
Source code	https://github.com/kubeflow/notebooks
Docker images	A complete list of the container images available for this component can be found in https://www.kubeflow.org/docs/components/notebooks/container-images/ These images are broadly divided into two groups. The first group includes the base images used to deploy the development environments themselves (Code Server, JupyterLab and RStudio). The second group extends them with commonly used ML and data science packages, providing ready-to-use environments for real-world experimentation and model development.
Demo	YouTube link: https://www.youtube.com/watch?v=-mfUbtJZvwU The demo showcases the Pipeline Development component starting from minute 1:10, including the creation and configuration of a Kubeflow notebook, resource selection (CPU/GPU/storage), and the launch of the development environment for model experimentation.
Deployment requirements	Minimum resources: • Kubernetes cluster • ≥ 16 GB RAM recommended for platform services • ≥ 4 vCPUs (minimum control plane + worker node) • GPU nodes optional but recommended for AI workloads Other Requirements: • Persistent storage provisioner (for Persistent Volumes) • Ingress controller for web access

	• HTTPS access recommended
Dependencies	• Kubernetes cluster • Kubeflow core components • Dex authentication service (for user access) • Container registry access
License	Apache 2.0

Table 1: Pipeline Development Environment component final release summary

2.1.2 Pipeline Orchestration Platform

The Pipeline Orchestration Platform builds upon the experimentation phase enabled by the Pipeline Development Environment. While the previous component focuses on interactive research, testing and prototyping of models, this component is designed to transform those experiments into stable and repeatable training pipelines. Once a satisfactory model and training approach has been identified, the training procedure is converted into automated pipelines, where the main variation between executions is the input data.

The module is implemented using **Kubeflow Pipelines** and provides a web-based interface to manage the full lifecycle of ML workflows. Through this interface, users can browse the catalogue of available pipelines, manage experiments and monitor individual executions. Pipelines represent reusable training workflows composed of multiple steps, while experiments act as logical containers used to group related runs. Each run corresponds to a specific execution of a pipeline with a given configuration. The interface provides visibility of execution status, logs, metrics and generated artefacts, enabling users to monitor training progress and compare results across different runs.

Figure 5 shows the POP dashboard, where users can browse the catalogue of available pipelines. Pipelines are organised into public and private categories, allowing users to either share their workflows with other platform users or keep them restricted for individual use, depending on collaboration and privacy requirements. From this interface, users can inspect previously uploaded pipelines, as well as access the functionality to upload and register new pipelines within the system.

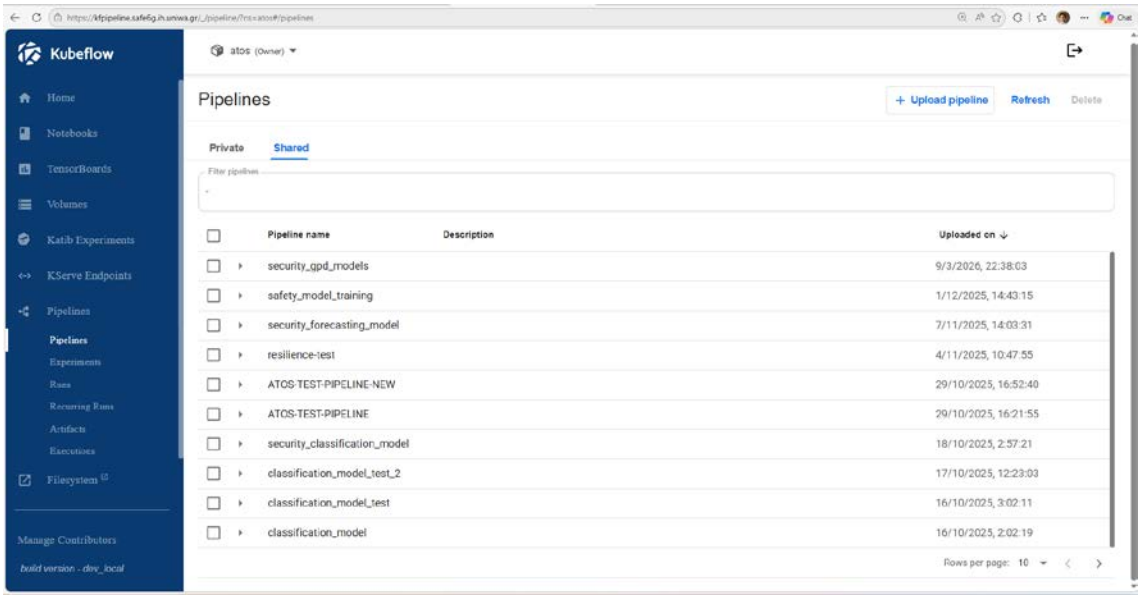

Figure 5: Catalogue of available pipelines within the Kubeflow GUI

The platform also provides a graphical representation of pipelines as Directed Acyclic Graphs (DAGs), where each node corresponds to a containerised step executed within a Kubernetes pod. When inspecting a previously uploaded pipeline, users can visualise its structure in DAG format, enabling a clear understanding of the execution flow, data dependencies and step sequencing within the pipeline. Figure 6 shows an example of DAG representation for a pipeline, highlighting how individual components are connected and executed in order. From this interface, users can also trigger a new run, create a new experiment, or upload a new version of the pipeline in case it has been updated.

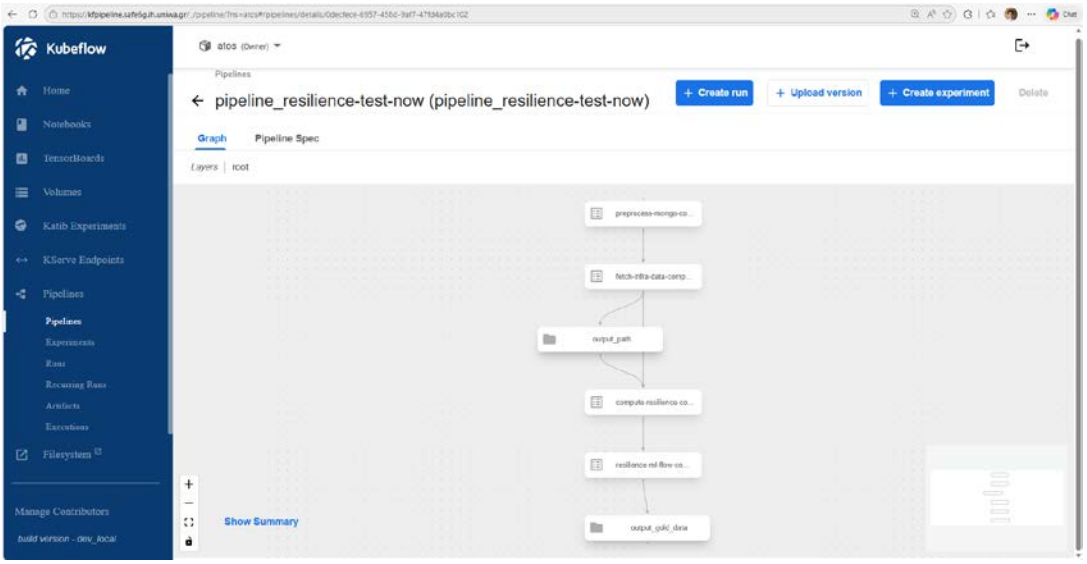

Figure 6: Kubeflow pipeline DAG

Pipeline construction using Kubeflow Pipelines SDK

The definition and management of pipelines within the Pipeline Orchestration Platform is carried out programmatically using the **Kubeflow Pipelines SDK**, which is the most used through Python. This SDK enables two key capabilities.

- Deployment and execution of pipelines in a Kubernetes-based environment.
- Operational control of pipelines, including remote execution, monitoring of runs, inspection of results, and querying of existing experiments and pipeline definitions.

The first step for constructing a pipeline is to define its individual components, or steps, that make up the full workflow. Each component is implemented as an independent Python function and specified using the `@component` decorator provided by the SDK. The required libraries, dependencies, and environment variables are explicitly declared for each component. Input and output parameters are also included, establishing clear interfaces that enable data flow between pipeline stages.

For instance, Figure 7 shows the implementation of the `download_minio_data` component, which is responsible for connecting to a MinIO object storage instance and retrieving the data required for model development. This component defines both the configuration parameters required to access the storage system and the output artefacts generated, which are passed downstream in the pipeline.

```python

@dsl.component(packages_to_install=["minio"], base_image="python:3.12-bullseye")
def download_minio_data(bucket: str, dataset: str, dataset_file: dsl.Output[dsl.Dataset]):
    import logging
    import os
    import urllib3
    from pathlib import Path
    from minio import Minio

    logging.basicConfig(level=logging.INFO)
    MINIO_USER = os.getenv("MINIO_ROOT_USER")
    MINIO_PASSWORD = os.getenv("MINIO_ROOT_PASSWORD")
    MINIO_URL = os.getenv("MINIO_OPEN_URL")
    MINIO_BUCKET = bucket

    def getMinioClient(
        minio_user=MINIO_USER,
        minio_pass=MINIO_PASSWORD,
        minio_url=MINIO_URL,
        minio_secure=True,
        minio_region="es",
    ):

        _http = urllib3.PoolManager(
            timeout=10,
            maxsize=10,
            retries=urllib3.Retry(
                total=3, backoff_factor=0.2, status_forcelist=[500, 502, 503, 504]
            ),
        )
        minioClient = Minio(
            minio_url,
            access_key=minio_user,
            secret_key=minio_pass,
            secure=minio_secure,
            http_client=_http,
            region=minio_region,
        )
        return minioClient

    minioClient = getMinioClient()
    Path(dataset_file.path).parent.mkdir(parents=True, exist_ok=True)
    logging.info("Downloading from bucket " + MINIO_BUCKET + " dataset " + dataset)
    minioClient.fget_object(MINIO_BUCKET, dataset, dataset_file.path)

```

Figure 7: Example code showing the pipeline component responsible for downloading data from MinIO

Once all components have been defined, they are composed into a complete pipeline by explicitly connecting the outputs of one step to the inputs of the next. This orchestration logic defines the overall structure of the pipeline, as illustrated in Figure 8, which shows the Python code used to assemble the pipeline by importing and linking the previously defined components. Through this approach, Kubeflow ensures that each step is executed only when its dependencies are satisfied, and that data flows consistently through well-defined artefact exchanges.

```python
@dsl.pipeline(
    name=name_pipeline,
    description="Pipeline template to test Kubeflow pipeline creation and run",
)
def test_pipeline(bucket: str, dataset_waterLevel: str, dataset_precipitations: str):

    # Download data from MinIO: provide MinIO URL, USR and PSW via env variables.
    download_waterLevel_task = download_minio_data(
        bucket=bucket, dataset=dataset_waterLevel
    )
    download_waterLevel_task.set_env_variable(
        "MINIO_ROOT_USER", os.getenv("MINIO_MLOPS_USER")
    )
    download_waterLevel_task.set_caching_options(True)

    download_precipitations_task = download_minio_data(
        bucket=bucket, dataset=dataset_precipitations
    )
    download_precipitations_task.set_env_variable(
        "MINIO_ROOT_USER", os.getenv("MINIO_MLOPS_USER")
    )

    preprocess_task = preprocess_data(
        waterLevel_data=download_waterLevel_task.output,
        waterLevel_columns=["datetime", "waterLevel"],
        precipitations_data=download_precipitations_task.output,
        precipitations_columns=["datetime", "precipitations"],
    )

    divide_data_task = divide_data(preprocessed_data=preprocess_task.output)

    train_model_task = train_model(
        trainX=divide_data_task.outputs["trainX"],
        trainY=divide_data_task.outputs["trainY"],
        testX=divide_data_task.outputs["testX"],
        testY=divide_data_task.outputs["testY"],
    )
```

Figure 8: Code snippet where the different steps of the pipeline are grouped

Rather than extending the core functionality of the PDE and POP components, effort has primarily focused on improving their usability and accessibility for project partners. To this end, a reusable pipeline template has been developed to simplify the creation of ML workflows and promote standardized development practices across the project. The template includes pre-implemented components covering the most common stages of the machine learning lifecycle, including data ingestion from MinIO, dataset splitting, preprocessing operations and model training. Together with the supporting libraries and usage guidelines provided, these assets significantly reduce the effort required to onboard new users and accelerate the development of AI/ML pipelines within SAFE-6G. The relevant information of this component has been summarized in the following table.

It is also worth commenting that both the PDE and POP modules have been connected to the **DataOps'** API of the SAFE-6G framework, enabling the MLOps platform to access the data generated across the entire ecosystem for model development and training tasks.

Category	*Status*
Version	KFP SDK: V2.9.0. Kubeflow Pipeline: v2.0.0
Documentation	Official User Guide: https://www.kubeflow.org/docs/components/pipelines/ KFP SDK API Reference: https://kubeflow-pipelines.readthedocs.io/en/stable/index.html Kubeflow pipeline creation and maintenance template: https://gitlab.com/safe-6g/integration-and-validation/kubeflow-pipelines-template
Source code	https://github.com/kubeflow/manifests
Demo	YouTube link: https://www.youtube.com/watch?v=-mfUbtJZvwU The demo showcases the POP starting from minute 2:00. It demonstrates how users access Kubeflow Pipelines, explore the pipelines catalogue, create and configure experiments, and launch pipeline runs. The video also illustrates how training pipelines are monitored through the web interface, including the visualization of the pipeline DAG, inspection of step logs, execution status, and generated artefacts.
Deployment requirements	Minimum resources: • Kubernetes cluster • ≥ 8–16 GB RAM • ≥ 4 vCPUs for control plane and base workload execution • Additional worker node capacity depending on pipeline parallelism and workload intensity • GPU nodes optional but recommended for training or inference-heavy pipeline steps
Dependencies	• Kubernetes cluster • Kubeflow core components • Dex authentication service (for user access) • Container registry access
License	Apache 2.0

Table 2: Pipeline Orchestration Platform component final release summary

2.1.3 Model Storage

The Model Storage component has been designed to address two essential needs within the AI/ML lifecycle: the persistent storage of datasets used for training, and the storage of trained models as well as intermediate artefacts generated during the experimentation and training processes. This component is based on **MinIO**[4], a high-performance, S3-compatible object storage system widely adopted in cloud-native and MLOps environments. MinIO provides scalable and secure storage for unstructured data such as datasets, trained models, logs and experiment artefacts. Its S3 compatibility enables seamless integration with modern data processing and machine learning tools. MinIO is particularly well suited for MLOps platforms because it provides:

- S3-compatible APIs that simplify integration with pipelines and training jobs
- High performance for large dataset transfers
- Fine-grained access control and multi-tenant storage
- Easy deployment within Kubernetes environments

[4] MinIO documentation: https://docs.min.io/

The deployed MinIO instance has been configured to support a multi-tenant approach aligned with SAFE-6G collaboration requirements. Each user is assigned a dedicated bucket, enabling them to manage their own datasets and trained models independently. Users can control whether their stored data is shared with other project partners or kept private according to privacy policies and data governance requirements. This approach ensures both collaboration and data sovereignty within the platform.

Integration within MLOps workflow

The Model Storage component acts as the central storage layer connecting the different SAFE-6G MLOps components:

- From the **PDE**, users access the DataOps services to retrieve raw data, create curated datasets, and store them in Model Storage for future reuse.
- From the **POP**, pipelines retrieve datasets from Model Storage to train models. The resulting trained models and artefacts are then stored again in Model Storage, ready for deployment and reuse.

Users can interact with Model Storage through two complementary mechanisms:

- **Graphical User Interface (GUI).** The web interface (Figure 9) provides a visual overview of available buckets, datasets, and stored models. It is mainly used for browsing, validation, and manual management of stored artefacts.
- **API access.** MinIO exposes an S3-compatible API that enables programmatic upload and download of artefacts. This API-based access is essential for automation and integration with the rest of the MLOps component, particularly pipelines and training workflows.

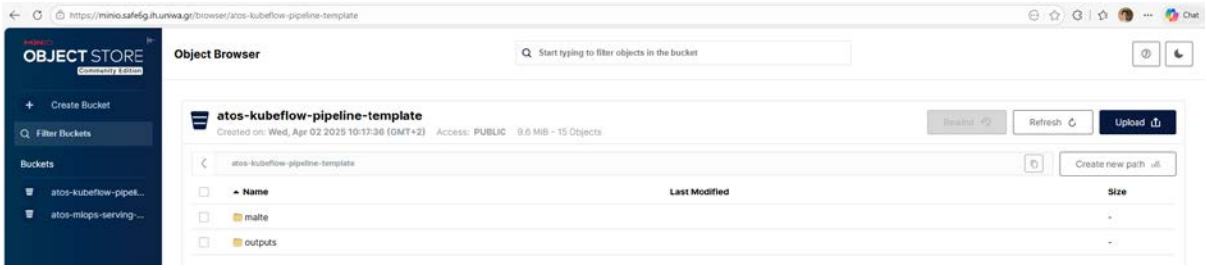

Figure 9: MinIO graphical User Interface

The main technical features, access interfaces and deployment requirements of the Model Storage component are summarized in Table 3.

Category	*Status*
Version	RELEASE.2025-09-07T16-13-09Z
Documentation	MinIO Python Client SDK (https://docs.min.io/enterprise/aistor-object-store/developers/sdk/python/)
Docker image	https://hub.docker.com/r/minio/minio
Source code	https://github.com/minio/minio
Open API Swagger	S3-compatible API (AWS S3 specification)
Demo	YouTube link: https://www.youtube.com/watch?v=-mfUbtJZvwU

	The demo showcases the Model Storage component starting from minute 3:00. It demonstrates how users access the MinIO web interface, browse buckets, and manage stored datasets and trained models.
Deployment requirements	Minimum requirements: • ≥ 4 vCPUs for basic MinIO deployment • ≥ 8 GB RAM for small-scale deployments • Minimum 50–100 GB of storage capacity (depends on usage)
License	Apache 2.0

Table 3: Model Storage component final release summary

2.1.4 Model Serving

The last native component of the MLOps platform is the Model Serving. It is responsible for deploying the AI/ML models trained in the previous stages and making them available for inference in cloud-native environments. This component represents one of the main developments carried out within SAFE-6G and constitutes a significant enhancement of the baseline platform capabilities. Considerable effort has been devoted to designing and implementing a deployment component that bridges the gap between model development and operational execution, enabling seamless integration with the SAFE-6G ecosystem and, in particular, with the **Meta-OS** (via its orchestration API).

The component has been designed to automate the deployment process as much as possible. Trained models are encapsulated into Docker images and Helm charts that can be deployed on Kubernetes clusters, significantly reducing manual intervention and allowing developers to focus primarily on model development rather than operational tasks. This approach facilitates the transition from experimentation to production-ready deployments, while ensuring consistency, portability and scalability across different execution environments.

A reusable repository template has been created to standardize the deployment of AI/ML models across SAFE-6G components and functions. This template serves as the baseline for creating a dedicated repository for each SAFE-6G function or component that requires ML inference capabilities. The template is built around three core building blocks:

1. **KServe-based model API**
2. **Helm chart for Kubernetes deployment**
3. **GitLab CI/CD automation**

Together, these elements provide a fully automated path from trained model artefacts to production-ready inference services. The complete CI/CD workflow is described in Deliverable D3.1 (Section 8.3). In this section, the focus is on the technical implementation of the component.

KServe-based inference service

The inference service is implemented using **KServe**[5], an open-source framework designed to deploy and serve AI/ML models on Kubernetes. KServe provides a standardised way to expose ML models through REST APIs, enabling real-time inference while abstracting the complexity of container orchestration and scaling. It supports multiple ML frameworks such as TensorFlow, PyTorch or Scikit-learn, allowing SAFE-6G developers to deploy models without framework restrictions.

[5] KServe documentation: https://kserve.github.io/website/docs/intro

To integrate a model into the serving platform, developers implement a Python script which imports the KServe library together with the required ML frameworks and defines a custom class inheriting from the base `Model` class provided by KServe. Three main methods must be implemented:

- **Load:** In charge of download the trained model weights from the Model Storage component and loads the model into memory.
- **Preprocess:** It prepares incoming request data before inference step.
- **Inference:** It performs the model inference and formats the response returned by the endpoint.

This structure standardizes how models are served across the project while maintaining flexibility for different ML frameworks and use cases. For local development and testing, the template includes a Docker Compose configuration along a Docker Image that allows developers to run the inference of API locally before deployment. Dependency management is handled using Poetry to ensure reproducible environments.

Helm chart for Kubernetes deployment

A reusable Helm chart has been developed to deploy model inference services into Kubernetes clusters. This chart is shared across all SAFE-6G trust functions and components, ensuring a consistent deployment strategy. Each model uses the same Helm chart but with customised configuration through the `values.yaml` file. The chart enables:

- Deployment of the model container image from a container registry
- Selection of the target Kubernetes namespace
- Automatic creation of Kubernetes Deployment and Service resources
- Exposure of the inference API through a NGINX Ingress

GitLab CI/CD automation

The third pillar of the component is the automated CI/CD pipeline implemented in GitLab. This pipeline automates the full lifecycle of model deployment, including:

- Retrieval of trained model weights from Model Storage.
- Packaging the inference service into a versioned docker image and upload it to a registry.
- Execution of integration and unit tests.
- Deployment to staging or production environments.

The full CI/CD workflow is described in D3.1 (Section 8.3). In summary, the process starts with the retrieval of the trained model weights from the Model Storage, which are then made available to KServe together with the inference script in order to build the REST inference API. This service is packaged into a Docker image, which is stored in a container registry after passing both unit and integration tests to validate the correct behaviour of the API. Once the image is stored in the registry, the deployment stage is triggered. If the commit is made to the develop branch, the model is deployed to a development Kubernetes cluster using ArgoCD. In contrast, if the commit is made to the main branch and tagged with a specific version (e.g., v1.1.0), the model is deployed to the production environment through the Meta-OS. This versioning strategy also provides traceability for the models deployed by each component or trust function, allowing partners to identify the exact deployed version and, if necessary, redeploy previous model versions in a controlled and reproducible manner.

The main technical characteristics, interfaces, templates and deployment requirements of the Model Serving component are summarized in Table 4.

Category	*Status*
Version	Model Template version 1.0.0
Documentation	KServe documentation: https://kserve.github.io/website/
Docker image	Generated per-model by the CI/CD pipeline and stored in the project container registry. The Dockerfile used can be found in the Model Serving template repository
Source code	Model Serving template: https://gitlab.com/safe-6g/integration-and-validation/model-serving-example Helm chart code and documentation: https://gitlab.com/safe-6g/integration-and-validation/model-serving-helm
Open API Swagger	Kserve API endpoint official documentation: https://kserve.github.io/website/docs/concepts/architecture/data-plane/v1-protocol?_highlight=endponts#api-endpoints
Demo	YouTube link: https://www.youtube.com/watch?v=-mfUbtJZvwU The demo shows from minute 4:00 how trained models are integrated into the MLOps pipelines and later deployed as inference services within the SAFE-6G platform.
Deployment requirements	Requirements depend on the complexity and size of the model to be deployed. As a minimum baseline, a Kubernetes cluster with at least 4 CPU cores and 8 GB RAM per deployed model service is recommended. For GPU-based or large deep learning models, GPU-enabled nodes and higher memory allocations may be required.
Dependencies	Container registry to store Docker images, GitLab CI/CD runners, access to Model Storage for retrieving model weights
License	Apache 2.0

Table 4: Model Serving component final release summary

2.2 Differential Privacy

Since the design of the Differential Privacy (DP) module was not included in D3.1, the information provided in this deliverable with higher depth in comparison with other WP3 components. Its design and implementation have been guided by the following project-level requirements from D2.1:

- **REQ-COM-ETH-NF-M-14:** Compliance with legal requirements on data protection.
- **REQ-MLOPS-TRAIN-F-R-09:** Anonymization of datasets.
- **REQ-MLOPS-DP-NF-M-12:** Accuracy in datasets regardless the level of noise introduced by Differential privacy.

With this implementation, the MLOps platform is extended with privacy-preserving capabilities, which is able to protect sensitive data during model training and when inference occurs. Considering the above requirements, the implemented capabilities are presented in this section, as a mechanism to protect datasets used in AI/ML pipelines, while also maintaining accurate (enough) model behaviour. As will be described, the module supports Local Differential Privacy (integrated as a pre-processing step in the MLOps training pipeline), and Global Differential Privacy (used for the SAFE-6G Chatbot intent classification model).

The DP module provides a privacy-preserving technique, used in two different ways during the ML training pipeline:

- Following a **Local Differential Privacy (LDP)** approach, during the data pre-processing stage of a ML training pipeline, calculated noise is added to training data before model training begins. This reflects the case when there is no central trusted entity between data sources and where model training/inferencing will take place (or there are no secure links connecting the two together). By **applying noise to data at the source**, this protects data used to train Trust Function AI models, which does not leak any private or sensitive information about data subjects or the infrastructure itself. With the use of LDP, resulting models can be shared and deployed without exposing the underlying training distribution. The LDP module has been integrated within the MLOps pipeline, acting as a pre-processing layer between the data ingestion and the model training step. It has been coded to be model-agnostic and task-agnostic, supporting both classification and regression workflows, having also been validated across the datasets of multiple SAFE-6G Trust Functions.
- Following a **Global Differential Privacy (GDP)** approach, which is used in the SAFE-6G AI-powered Chatbot, during deep learning training and the gradient computation stage of a pipeline, noise is added to the gradients of model parameters (instead of input data). This reflects the case where a trusted central training environment exists, but the trained model, when queried – and thus when it provides a prediction, must not reveal information about any training data which were used. The specific GDP approach was chosen for the Chatbot, due to its textual input data, where it was found that LDP did not produce models of adequate accuracy.

2.2.1 Local Differential Privacy with the Laplace Mechanism

The Laplace Mechanism is used for the LDP case – which is a well-established way of achieving ε-differential privacy. Given a dataset input training feature vector X, Laplace noise is added to each feature independently:

$$X\_noisy = X + Lap(0, 1/\varepsilon)$$

where ε is the *privacy budget parameter.* This is a tradeoff which needs to be considered between privacy and utility - with a lower ε value (e.g. ε = 0.5) corresponds to stronger privacy protections which lead to greater changes to training data and consequently affect the accuracy of ML models. On the contrary, higher values of ε (e.g. ε = 10) cause much smaller alterations to the data and this leads to models with accuracy equating to model training on clear (non-noisy) datasets.

The primary module supports a list of ε values (which can of course be altered). This allows the full privacy-utility curve to be evaluated in a single experimental run. The following epsilon range has been used in the SAFE-6G experiments carried out for all the Trust Functions:

- ["No noise", 10, 7.5, 6, 5, 4, 3, 2, 1.8, 1.6, 1.4, 1.2, 1.0, 0.8, 0.5]
  - The "No noise" run is used as a baseline, to ascertain the upper bound of model performance without any privacy preserving noise addition.

2.2.1.1 Local Differential Privacy Training Pipeline

The LDP module can integrated into the MLOps training pipeline with the following steps:

1. **Data ingestion**: The dataset is loaded and the target variable is separated from the input training features. The task type (binary classification or multi-class classification or regression) is automatically detected based on the cardinality of the target variable.
2. **Train/test split**: The dataset is split into training and test sets using an 80 to 20 ratio, to maintain original class proportions to have a representative split (stratification).
3. **Feature scaling**: A StandardScaler is fitted on the clean training split and applied to both training and test sets. The fitted scaler is saved for use for future inferencing.
4. **Noise injection**: For each ε-value in the list, Laplace noise is independently added to the data, producing a separate noisy dataset for each ε-value.
5. **Model training**: For each epsilon-noisy dataset, model training takes place and models are evaluated on the noisy test set.
6. **Model persistence**: The best model for each ε-value is serialised and saved for later use and inferencing.
7. **Model training**: For each epsilon-noisy dataset, model training a GridSearchCV is performed over an expanded hyperparameter grid (described below), using 5-fold cross-validation. The best-performing model configuration is selected and evaluated on the noisy test set.

2.2.1.2 Supported Models

The LDP module is task-agnostic and has been integrated with the following scikit-learn estimators:

- **Classification tasks**
  - Gaussian Naïve Bayes.
  - Logistic Regression.
  - Support Vector Machine.
  - Random Forest Classifier.
- **Regression tasks**
  - Linear Regression.
  - Support Vector Regressor.
  - Random Forest Regressor.

An example of dataset runs and their results when trained using the earlier described pipeline are presented in Figure 10 and Figure 11. From the above graphs, it is important to note the importance of selecting the right ML Classification or Regression model, and the right trade-off between privacy and model accuracy. After testing them with the datasets of some Trust Functions, Random Forest models were the best choice – selected after applying an expanded hyperparameter search grid for identifying the best model configuration under each privacy level. These were chosen to provide an expansive beneficial search while still allowing for computational running times to be reasonable – i.e., hyperparameters which would provide a possible measurable accuracy performance (given a literature review) were only included (see Table 5).

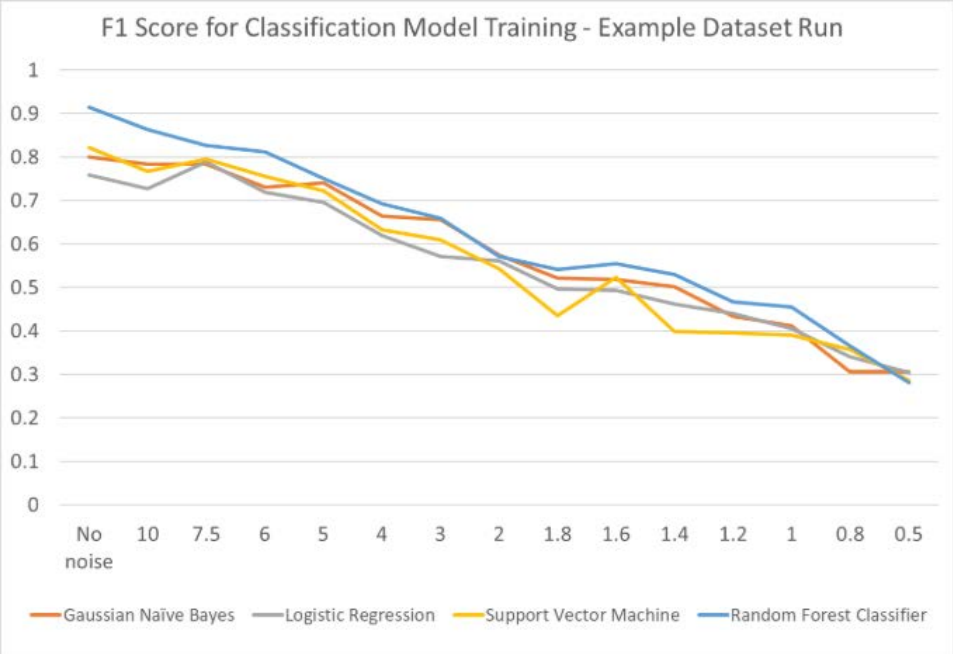

Figure 10: Dataset Run on Local Differential Privacy Pipeline, and its results – Classification Model Training Example

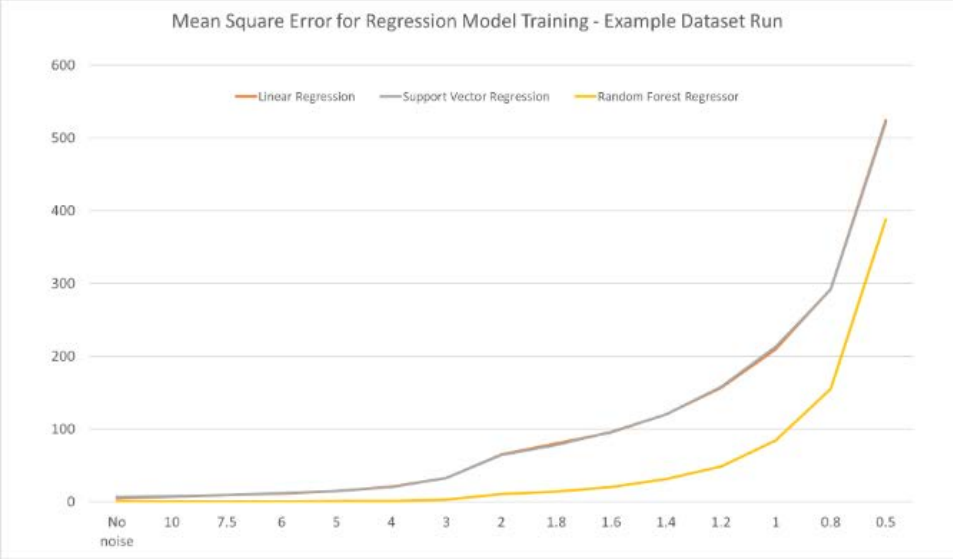

Figure 11: Dataset Run on Local Differential Privacy Pipeline, and its results – Regressor Model Training Example

Hyperparameter	***Values searched***
n_estimators	100, 200
max_depth	5, 10, 20
min_samples_split	2, 10
min_samples_leaf	1, 4
max_features	sqrt, log2
bootstrap	TRUE

Table 5: Random Forest hyper-parameters for model training

Model selection in this case was decided using F1 score for classification tasks and Mean Squared Error (MSE) for regression tasks.

2.2.1.3 Inference Workflow

It should be highlighted that the noisification of data must be replicated at inference time, as models are trained on data that have been altered with an ε-level of Laplace noise at a specific ε, with the consequence that the trained model's decision boundaries are calibrated to that noise distribution. Data for inferencing will thus need to be noisified before being passed through the model – otherwise for the model they would be out-of-distribution. The required inference workflow is as follows:

- Load the frozen scaler.pkl and apply scaler.transform() to the new input features.
- Add Laplace noise at the same ε value used during training:
  X_noisy = X_scaled + Lap(0, 1/ε).
- Pass the noisy, scaled input to the loaded trained model for prediction.

Following these steps will ensure consistency with the model training distribution, and that local differential privacy guarantees, assumption and constraints are considered on inference data.

2.2.2 Global Differential Privacy for the SAFE-6G Chatbot

Global Differential Privacy (GDP) was used mainly for the privacy preservation training of the Chatbot's intent classification model. In this case, DP is applied at the training level of the ML algorithm, with noise added to gradient updates during the backpropagation process. This ensures that training data cannot be recovered from the published model weights, irrespective of the number of queries.

The two different approaches to Differential Privacy which have been used are important and very useful in practical scenarios. LDP protects data during transit and is suitable for scenarios without a trusted central entity. On the other hand, GDP protects the training process itself and is appropriate when a trusted training environment exists but the final trained model is widely shared or deployed externally. Both scenarios are present within SAFE-6G. Below are provided the specific details on how this was achieved for the case of the Chatbot.

2.2.2.1 Differentially Private Stochastic Gradient Descent

Global Differential Privacy guarantees are achieved using Opacus[6], a PyTorch-native library developed by Meta AI for differentially private deep learning. Opacus implements the Differentially Private Stochastic Gradient Descent (DP-SGD) algorithm, which modifies the standard SGD training loop in the following two important ways at each training step:

- **Per-sample gradient clipping:** Before aggregation, the gradient of each individual training sample is clipped to a maximum set value. This bounding step is important when it comes to privacy preservation, as it limits the maximum influence any single training sample can exert on the model update. Clipping is essential in order to protect the privacy of data.
- **Gaussian noise addition:** After clipping, Gaussian noise is added to gradients. The noise magnitude is automatically computed by Opacus based on the target privacy budget, ensuring the specified ε guarantee is met by the end of training.

Once training is complete, the trained PyTorch linear classification head, saved as a state dictionary for deployment.

[6] Opcaus documentation: https://opacus.ai/

It should be noted that a GDP implementation using Opacus - which works on numerical datasets, has also been made available on the project's GitLab repository as open-source code.

2.2.3 Differential Privacy Summary

Category	*Details*
Version	V1.0
Documentation	https://gitlab.com/safe-6g/development/differentialprivacyimplementation
Source code	https://gitlab.com/safe-6g/development/differentialprivacyimplementation
Docker images	Local Differential Privacy: https://hub.docker.com/r/safe6g/local-differential-privacy-implementation Global Differential Privacy with Opacus: https://hub.docker.com/r/safe6g/opacus-neural-network-global-differential-privacy-implementation
Helm Chart	https://gitlab.com/safe-6g/development/differentialprivacyimplementation/-/tree/dp-initial/helm?ref_type=heads
Open API Swagger	NA (no REST API)
Demo	NA
Deployment requirements	CPU environment sufficient for LDP pipeline; CUDA-compatible GPU recommended for GDP/Chatbot training
Dependencies	scikit-learn, NumPy, pandas, joblib (LDP), PyTorch, Opacus, sentence-transformers, scikit-learn (GDP)
License	MIT License

Table 6: Differential Privacy module final release summary

Category	*Local Differential Privacy (LDP)*	*Global Differential Privacy (GDP)*
Key inputs	CSV dataset with labelled features and target column; list of ε values; task type (auto-detected)	Labelled intent CSV (text, intent columns); privacy parameters (ε, δ, gradient clipping norm)
Key outputs	Per-ε trained model .pkl files; fitted scaler.pkl; F1/Accuracy or MSE/MAE metrics per ε value	dp_head_model.pth (trained DP classifier head)
Privacy mechanism	Laplace mechanism; noise scale = 1/ε applied to scaled input features	DP-SGD via Opacus; per-sample gradient clipping (L2 norm = 1.0) + Gaussian noise on gradients; RDP accountant for live ε tracking

Table 7: Local vs Global Differential Privacy

2.3 Integration of the Differential Privacy module in the MLOps platform

As described in previous sections, the Differential Privacy module consists of a Python library that enables the protection of sensitive data by applying privacy-preserving mechanisms during the model training stage. This process is not straightforward, since several parameters must be properly adjusted in order to preserve the original accuracy and behaviour of the trained model while guaranteeing data privacy. Considering these requirements and taking advantage of the flexibility of the SAFE-6G MLOps

platform, which allows the incorporation of external libraries into the modules, the integration of the Differential Privacy module was performed in a direct and seamless way.

Within the Pipeline Development Environment, developers are able to load the Differential Privacy library into development environments such as Jupyter Notebooks, where datasets can be accessed and experimentation tasks can be carried out. During this stage, developers can test different configurations over the Differential Privacy module and tune the corresponding parameters in order to identify the most suitable setup for their specific use case. Once the optimal configuration has been identified, developers can move to the POP module to create automated training pipelines that incorporate dedicated steps for applying differential privacy to the data before model training.

In addition, it must be considered that the same data treatment applied during the training phase must also be reproduced during the inference stage in order to ensure consistency between training and operational execution conditions. For this reason, when the model is prepared for deployment within the Model Serving component, the Differential Privacy library is loaded again and integrated into the inference stage, so some functions of it can be used to enable the incoming data to undergo the same privacy-preserving processing as in the training process. This approach guarantees that privacy protection is consistently maintained throughout the complete AI/ML lifecycle, from model development and training to deployment and runtime inference.

3 ADOPTION AND VALIDATION OF THE MLOPS PLATFORM

This section presents the operationalization, adoption and validation of the MLOps platform, enabling the training and deployment of AI/ML models with additional privacy-preserving capabilities applied to the managed data. Section 3.1 focuses on the utilization of both modules by the rest of the SAFE-6G trust functions and components, while Section 3.2 describes the validation process carried out for the whole platform.

3.1 ADOPTION OF THE PLATFORM

The utilization of the MLOps platform by other modules of the SAFE-6G framework, such as the Chatbot, the Trust Functions or the Cognitive Coordinator, must be understood from two different perspectives. The first one relates to the development and production deployment of AI/ML models. Thanks to the on-premise deployment of the MLOps platform within the project Kubernetes clusters, together with the generation of the previously described templates, project partners were able to access the platform and use it to onboard, develop and deploy their own models. Table 8 groups the different public endpoints where the platform has been deployed. The first two entries correspond to the Kubeflow framework, which includes the PDE and POP modules. Two different endpoints are provided because they correspond to two independent deployments. In the first case, the Kubernetes cluster only provides CPU resources, while the second deployment enables the usage of GPU resources available on the project servers. The remaining endpoint corresponds to the Model Storage component.

Module	***Endpoint***
Kubeflow + CPU	https://kfpipeline.{{uniwa_domain}}
Kubeflow + GPU	https://kubeflow.{{uniwa_domain}}
Model Storage	https://minio.{{uniwa_domain}}

Table 8: MLOps Platform endpoints

As a result of the platform adoption by the SAFE-6G partners, the models listed in the following table were successfully deployed. It is worth to highlight that the Privacy function is not included in the table since its operation does not require the use of AI/ML models.

Component/ Function	***Model API REST***
Cognitive Coordinator	https://model.develop.cognitive.coordinator.{{uniwa_domain}}
Chatbot	https://model.develop.chatbot.{{uniwa_domain}}
Resilience Function	https://model.develop.resilience.function.{{uniwa_domain}}
Reliability Function	https://model.develop.reliability.function.{{uniwa_domain}}
Safety Function	https://model.develop.safety.function.{{uniwa_domain}}
Security Function	https://model.develop.security.function.{{uniwa_domain}}

Table 9: AI/ML Model API REST endpoints exposed through Model Serving.

The second perspective concerns the usage of the deployed models by the partners within their corresponding trust functions and components. This utilization is possible because, as described in the Model Serving section, the models are deployed encapsulated within REST APIs that enable inference

execution. Thus, developers of the Trust Function, Chatbot and Cognitive Coordinator can leverage the deployed models by performing inference requests through the corresponding endpoints. In the same way, the deployed models can also be consumed by other SAFE-6G framework modules, such as XAI, enabling interoperability between the different components of the ecosystem.

3.2 VALIDATION OF THE PLATFORM

3.2.1 MLOPS

In this section, the objective is to demonstrate the capability of the MLOps platform to support the end-to-end development lifecycle of AI/ML models within the SAFE-6G ecosystem. Once the platform was fully deployed and operational in the project Kubernetes cluster, a simple classical ML model was developed using all the modules to validate its performance. The development of this model relied on the templates and workflows developed within the SAFE-6G project to facilitate the adoption of the MLOps platform by project partners. These assets, which were specifically designed to simplify pipeline creation and standardise development practices across the consortium, enabled the validation of the platform components in an integrated and realistic scenario. This exercise verifies that the proposed MLOps platform effectively supports the complete workflow from data preparation and training to deployment.

The validation process has been divided into two parts. The first one focuses on the creation and execution of a pipeline, while the second one addresses the deployment of the trained model.

Creation and execution of the training pipeline

To validate the capabilities of the MLOps platform, a complete training pipeline was created and executed using the templates and modules previously described. The first step consisted of creating a development environment that allowed iterative experimentation with the data, testing of models and implementation of the final training pipeline. This environment was provisioned through the **PDE** by deploying a Jupyter Notebook instance configured with **1 vCPU and 8 GB of RAM**, using the image *kubeflownotebookswg/jupyter-scipy:latest*, which provides sufficient resources for the validation scenario.

Within this environment, the pipeline template was loaded. The template includes the Kubeflow Pipelines component library, the notebook used to orchestrate the pipeline definition, and the required interactions with the Kubeflow Pipelines API to register pipelines, create experiments and launch runs.

The objective of the pipeline is to train a Deep Learning model, specifically an LSTM neural network. The pipeline was constructed using the reusable components provided in the template, resulting in a workflow composed of four different stages:

- **Stage 1: Raw data ingestion**. The *download data from MinIO* component was used to retrieve the datasets required for training. To simulate the ingestion of data from multiple sources, the component was executed twice, each time retrieving data from a different directory within a dedicated bucket created in the Model Storage.

- **Stage 2: Data preprocessing**. The retrieved raw datasets were processed using the *preprocessing* component included in the template. This step prepares the data and transforms it into a format suitable for training.
- **Stage 3: Dataset generation**. The processed data was split into training and testing subsets, creating the artifacts required for the training stage.
- **Stage 4: Model training.** The final step uses the *training* component, where the LSTM architecture and training procedure are implemented. The resulting trained model weights are stored in the Model Storage.

Once the pipeline was implemented, it was uploaded to the POP, where an experiment was created and executed as a pipeline run. The execution produced the DAG shown in Figure 12 and generated the trained model artefacts within the Model Storage. This setup enables future retraining by simply updating the datasets and re-executing the pipeline.

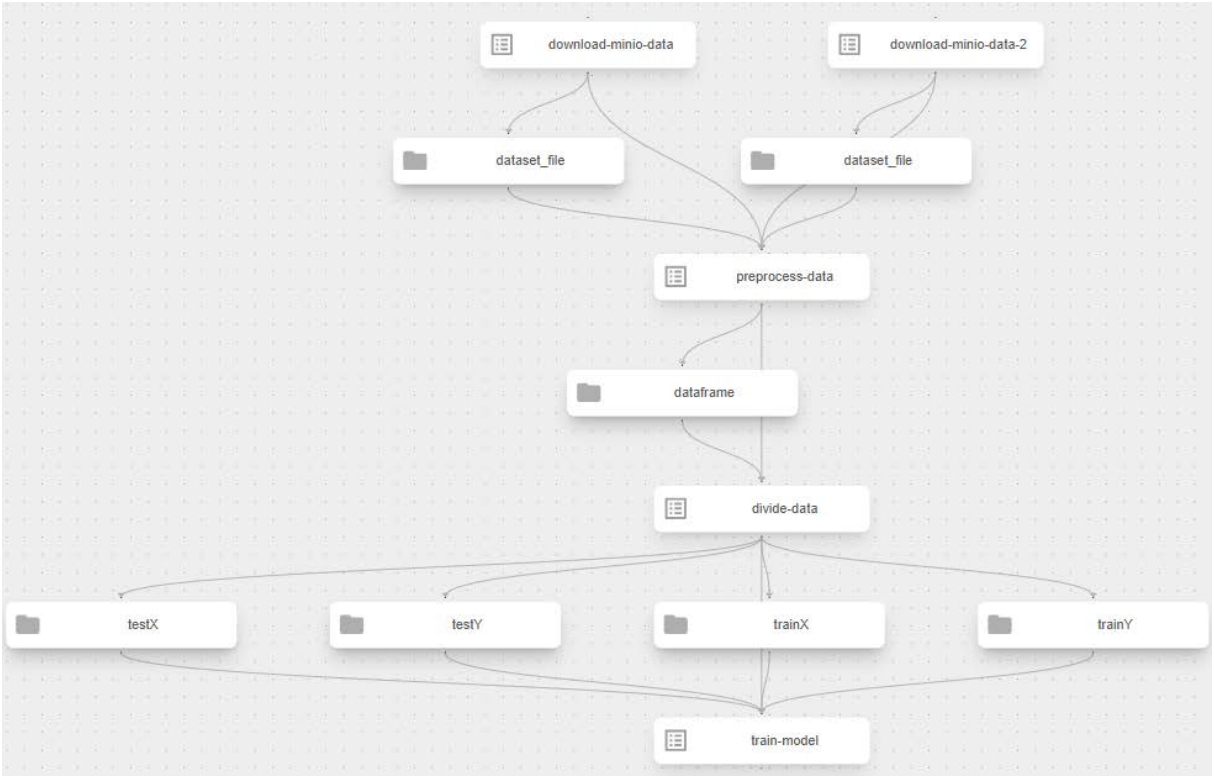

Figure 12: Kubeflow Pipeline DAG generated from the pipeline template

Deployment of the trained model

Following the successful training and storage of the model artefacts, the Model Serving template is used to perform the operational validation of this module. A new repository is instantiated using the template, which provides all the required files and structure. The following steps are then carried out to configure and deploy the inference service.

1. **REST API configuration:** the first step consists of configuring KServe, the framework responsible for exposing the model as a REST API and acting as the entry point for inference requests. Three main functions are implemented:

- `load`, which defines how the model weights are retrieved and loaded.
- `preprocess`, which specifies the transformations applied to incoming request data.
- `predict`, which handles the inference logic by passing the processed input to the model and returning the generated output.

2. **Integration test configuration:** integration tests are defined to validate the correct behaviour of the REST API during the deployment process. These tests ensure that the service correctly handles input data, including format validation, required fields and response structure, preventing incorrect deployments.
3. **Local execution validation:** to verify the correct implementation of the service, a local execution environment is set up using Docker Compose. The same Dockerfile used in later deployment stages is executed locally, allowing developers to validate the full-service behaviour and ensure that the API works as expected before deployment.
4. **Deployment to the development environment:** the deployment process begins with the configuration of the required environment variables in GitLab CI/CD, as specified in the template. These include registry credentials, model identifiers, MinIO access credentials and model storage paths, among others. Once configured, a commit to the `develop` branch triggers the CI/CD pipeline. This pipeline builds the Docker image, executes the integration tests, pushes the image to the container registry, and deploys the service to the Kubernetes development cluster using the Helm chart and ArgoCD. Figure 13 shows the different stages of the pipeline completed after executing the deployment of the template. The resulting deployed model is exposed through a dedicated domain configured for the project: model.develop.mlops.example.{{uniwa_domain}}, which is validated through REST API calls to KServe, as illustrated in Figure 14.
5. **Deployment to the production environment:** the production deployment workflow mirrors the development process, with two main differences. First, the commit must be made to the `main` branch and associated with a version tag (e.g. v1.1.0). Second, instead of ArgoCD, the deployment to the production cluster is managed through the Meta-OS using API-driven deployment calls integrated within the CI/CD pipeline.

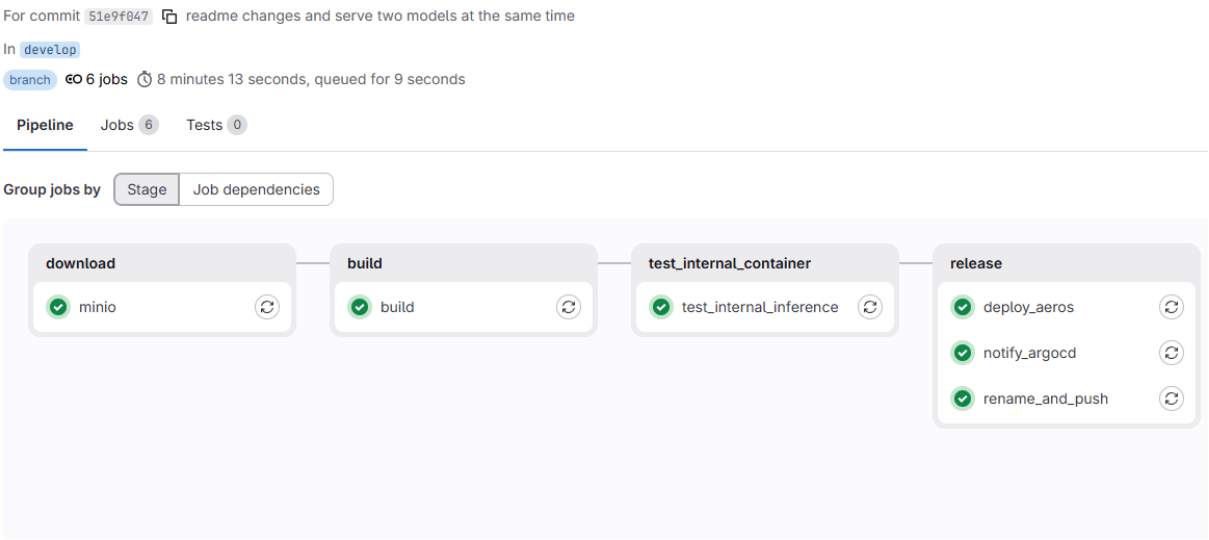

Figure 13: Model Serving CI/CD

```
# Health check
curl -X GET "https://model.develop.mlops.example.{{uniwa_domain}}/"

# Result:
# {"status":"alive"}

echo ""

# List available models
curl -X GET "https://model.develop.mlops.example.{{uniwa_domain}}/v1/models"

# Result:
# {"models":["mlops-model-example","mlops-model-example-2"]}

echo ""

# Inference request
HOST="https://model.develop.mlops.example.{{uniwa_domain}}"
MODEL_NAME="mlops-model-example"

curl -X POST \
  "$HOST/v1/models/$MODEL_NAME:predict" \
  -H "Content-Type: application/json" \
  -d '{
    "instances": [
      [6.8, 2.8, 4.8, 1.4],
      [1, 1, 1, 1]
    ]
  }'

# Result:
# {"prediction":"[1 0]"}
```

Figure 14: REST API calls to the deployed example model endpoint

3.2.2 DIFFERENTIAL PRIVACY

In this section, the operational validation of the Differential Privacy module are presented, demonstrating the correct execution for both the LDP pipeline and the GDP implementation for the SAFE-6G Chatbot. The objective of this validation process is to confirm that the module, when applied to SAFE-6G datasets operates as expected, produces structured ML model artefacts and exhibits the expected privacy-utility trade-off behaviour across a range of epsilon values. The LDP pipeline was run and validated using project's datasets and covering both classification and regression task types. The pipeline went through all its stages: data ingestion, stratified train/test splitting, StandardScaler fitting on clean training data, Laplace noise injection across the configured epsilon range, model training with 5-fold cross-validation GridSearchCV, and per-epsilon model serialisation.

The following datasets were used during validation:

- **INQBIT datasets:** decider_dataset.csv, gatekeeper_dataset.csv, and predictor_dataset.csv, covering binary classification and regression tasks related to the Safety Trust Function.
- **TID datasets:** final_data_synthetic_phone.csv and final_data_synthetic_xr_glasses.csv, covering regression tasks related to network telemetry from XR-assisted use cases and specifically the Resilience Trust Function.

For each dataset, the code automatically detected the task type based on the cardinality of the target variable, selected the appropriate scoring metric (F1 for classification, MSE for regression), and trained all configured models across the following epsilon list:

["No noise", 10, 7.5, 6, 5, 4, 3, 2, 1.8, 1.6, 1.4, 1.2, 1.0, 0.8, 0.5]

The results confirm the expected privacy-utility degradation curve. Model performance at high epsilon values ($\varepsilon \geq 5$) remained close to the "No noise" case. On the other hand, performance progressively decreased at lower epsilon values (especially $\varepsilon \leq 1.0$). This is expected behaviour, as Laplace noise added to the dataset dominates the training signal. This behaviour is consistent with the theoretical properties of Differential Privacy and the Laplace mechanism and confirms that the implementation is operating correctly. Figure 10 and Figure 11 from Section 2.2 illustrate representative results from a classification and a regression runs respectively, visually demonstrating this trade-off.

Importance of model and epsilon selection

The validation results confirm that the choice of ML model and the selection of ε-value are both very important decisions. Despite this, it is important to remark that it was aimed for ε-values with which privacy could be obtained – aiming for $\varepsilon \leq 3.0$. For both INQBIT and TID Trust Function datasets, Random Forest models achieved the highest F1 scores and lowest MSE values across the epsilon range. Separate experiments were carried out, using an expanded hyperparameter search grid applied specifically to these models across a lower set of a select ε-values.

Model artefact validation

Upon completion of each pipeline run, the following were confirmed to be correctly serialised and saved for each epsilon value:

- saved_models/{model_name}/model_eps_{ε}.pkl – the best-performing trained model for that epsilon level.
- saved_models/scaler/scaler.pkl – the StandardScaler fitted exclusively on clean training data, shared across all epsilon runs.

These artefacts were later verified to be loadable and functional for inference, using the inference workflow described in Section 2.2, confirming end-to-end operational correctness from training to deployment.

3.2.2.1 GLOBAL DP VALIDATION - SAFE-6G CHATBOT PREDICT INTENT CLASSIFIER

The GDP implementation was validated by training the Chatbot's intent classification model under DP-SGD guarantees using the Opacus library. The training was executed on the predict_intent_dataset_final.csv dataset, containing labelled user utterances which are mapped to intent categories relevant to the SAFE-6G context.

The Opacus PrivacyEngine was used upon the model, optimiser and data loader, with training taking place across a number of epochs. At the end of each epoch, the accumulated privacy expenditure was checked and confirmed to remain within the target budget throughout training. The per-epoch output confirmed correct operation of both the per-sample gradient clipping and the Gaussian noise injection steps, with training loss decreasing steadily and validation accuracy reaching an acceptable level for intent classification. After several runs and experiments, it was found that the most accurate model was achieve with parameter values of - LEARNING_RATE = 0.4 and EPOCHS = 15. Upon completion of training, the following two outputs were confirmed to be correctly generated and saved:

- **dp_head_model.pth** - the trained PyTorch linear classification head, verified to load correctly as a state dictionary for later inference purposes.
- **metadata.json** - the training provenance record, detailing privacy parameters (ε, δ, MAX_GRAD_NORM), encoder name, batch size, epoch count, random seed, and the complete list of intent classes. This provides an auditable record of the privacy-preserving training process.

The validation confirmed that GDP via DP-SGD is the appropriate privacy mechanism for the Chatbot use case. As noted in Section 2.2, applying LDP produced models which were of poor and random accuracy levels, most likely due to the added Laplace noise overwhelming the semantic signal. The GDP approach on the other hand, preserves embedding quality and applies privacy protection at the gradient level.

4 CONCLUSION

This document reported the outcomes of T3.4, related to the final release of the software assets developed, tailored or updated of the MLOps platform. In particular, it described the final implementation of its core components and the Differential Privacy module, as well as their integration within the SAFE-6G ecosystem. The document also presented the operational integration activities carried out with the different trust functions, components and infrastructure services, together with the validation activities performed in a real Kubernetes-based environment.

The results demonstrate the maturity and readiness of the developed software, providing to the SAFE-6G framework a complete platform for the development, deployment and operation of AI/ML models while incorporating privacy-preserving capabilities. The work carried out in SAFE-6G has built upon the MLOps platform provided by BULL, extending its capabilities and validating its applicability within a realistic multi-partner environment. The activities performed in SAFE-6G have enabled the delivery of a complete end-to-end MLOps solution covering the full AI/ML lifecycle, from model development and training to the automated deployment and operation of models packaged as Docker images and Helm charts in Kubernetes environments. In addition, reusable templates, supporting libraries and user guidelines have been developed to simplify platform adoption, reduce the learning curve for project partners and promote a consistent approach to AI/ML development across the consortium. Furthermore, the successful onboarding of partner models and their deployment through the provided MLOps workflows validate the practical applicability of the platform within the project.

It should be noted that the validation activities presented in this document focus on the operational verification of the developed components. A comprehensive assessment of the overall SAFE-6G platform, including the evaluation against project requirements, KPIs, and test cases, will be carried out in the integration and validation Work Package (WP5) and reported in the corresponding deliverables.