6GSNS

Co-funded by the European Union

SAFE-6G

SAFE-6G

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

Deliverable D5.1: Integration activities and Digital Twin of SAFE-6G pilot (Intermediate)

Date: 30/01/2026

Version: v1.0

DISCLAIMER

Grant Agreement	101139031
Document number	D5.1
Document title	Deliverable D5.1: Integration activities and Digital Twin of SAFE-6G pilot (Intermediate)
Lead Beneficiary	CMC
Editor(s)	Jose Costa-Requena (CMC)
Author(s)	Jose Costa-Requena (CMC) Panagiotis Karkazis (UNIWA) Dimitrios Uzunidis (UNIWA) Christos Betzelos (UNIWA) Stamatia Drampalou (UNIWA) Anastasios Vetsos (UNIWA) Rodrigo Partearroyo (TID) Chen Mu Wang Lin (TID) Dimitrios Roidis (NCSRD) Spyridon Georgoulas (NCSRD) Harilaos Koumaras (NCSRD) Panagiotis Pavlidis (NCSRD) Alejandro Fornés (UPV) Theocharis Saoulidis (eBOS) Hassan Yeganeh (eBOS) Marios Sophocleous (eBOS) Stelios Erotokritou (8BELLS) Kushal Mehta (IQB) Vagelis Anagnostopoulos (INF) Georgios Koumaras (INF) Panagiotis Koumaras (INF) Charles Bailly (IMM) Guillaume Hébert (KEY) Joaquín Cáceres (EVIDEN) Apostolos Garos (SHG) Victoria Katsarou (SHG)
Dissemination level	Public
Contractual date of delivery	31/01/2026
Status	Final
File name	SAFE-6G_D5.1_v1.0.pdf

Revision History

Version	
V0.1	TOC proposal, guidelines and template
V0.2	First Contribution of all the partners
V0.3	Online contributions (second round), with iterative editor feedback.
V0.4	Online contributions (third round), with iterative editor feedback.
V0.5	First review comments performed by UNIWA and NCSRD
V0.6	Final draft after the comments from the second review performed by the Technical Steering Committee. Final review performed by the Editor.
V1.0	Final version following the Quality check

GLOSSARY

Abbreviations/Acronym	Description
3GPP	3rd Generation Partnership Project
5G	5th Generation of Mobile Networks
5G SA	5G Standalone
5G-EIR	5G Equipment Identity Register
5QI	5G Quality Indicator
6G	6th Generation of Mobile Networks
AF	Application Function
AI	Artificial Intelligence
aLoTw	Achievable LoTw
AMF	Access And Mobility Management Function
API	Application Programming Interface
ARP	Allocation And Retention Policy
AUSF	Authentication Server Function
B5G	Beyond 5G
BERT	Bidirectional Encoder Representations From Transformers
CAPIF	Common API Framework
CD	Continuous Deployment
CI	Continuous Integration
cLoTw	Calibrated Level Of Trustworthiness
CNCF	Cloud Native Computing Foundation
CNI	Container Network Interface
CoCo	Cognitive Coordinator
CPU	Central Processing Unit
DAST	Dynamic Application Security Testing
DataOps	Data Operations
dB	decibel
dBm	decibel-milliwatts
DevSecOps	Development, Security And Operations
DID	Defense In Depth
DNS	Domain Name System
DT	Digital Twin
E2E	End-To-End
FDD	Frequency Division Duplexing
gNB	Next Generation Node B
GPU	Graphics Processing Unit
HARQ	Hybrid Automatic Repeat Request
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol
Hz	Hertz
ID	Identification Document
IMSI	International Mobile Subscriber Identity
IT	Information Technology
JSON	Javascript Object Notation
JWT	Json Web Token
K8s	Kubernetes

KMS	Key Management Service
KPI	Key Performance Indicator
KVI	Key Value Indicator
LLM	Large Language Model
LoTw	Level Of Trustworthiness
LTE	Long-Term Evolution
MB	Message Broker
MCS	Modulation and Coding Scheme
Meta-OS	Meta-Operating System
MIMO	Multiple-Input And Multiple-Output
ML	Machine Learning
MLOps	Machine Learning Operations
nApp	Network Application
NDT	Network Digital Twin
NEF	Network Exposure Function
NF	Network Function
NFS	Network File System
nLoTw	Non-Calibrated Level Of Trustworthiness
NR	New Radio
NSN	Network Service Nodes
NWDAF	Network Data Analytics Function
OAI	Open Air Interface
OAS	Openapi Specification
OIDC	Openid Connect
OS	Operating System
OSD	Object Storage Devices
PaaS	Platform-As-A-Service
PHY	Physical Layer
Pub/Sub	Publish–Subscribe
QoS	Quality Of Service
RAN	Radio Access Network
RAT	Radio Access Technology
REST	Representational State Transfer
RF	Radio Frequency
RRC	Radio Resource Control
RSRP	Reference Signal Received Power
SAST	Static Application Security Testing
SCM	Source Code Management
SDP	Software-Defined Perimeter
SDPCF	Service/Data Plane Control Function
SDPGW	Service/Data Plane Gateway
SINR	Signal-to-Interference-plus-Noise Ratio
SMF	Session Management Function
SNMP	Simple Network Management Protocol
S-NSSAI	Single Network Slice Selection Assistance Information
SQL	Structured Query Language
SSC	Session And Service Continuity
SSI	Self-Sovereign Identity
SSL	Secure Sockets Layer
SVM	Support Vector Machines

TCP	Transmission Control Protocol
TDD	Time Division Duplexing
TF	Trust Function
TLS	Transport Layer Security
UC	Use Case
UDM	Unified Data Management
UE	User Equipment
UPF	User Plane Function
USN	User Service Nodes
UT	Unit Test
vApp	Vertical Application
VC	Verifiable Credential
VCS	Version Control System
VGF	VPN Gateway Function
VM	Virtual Machine
VPN	Virtual Private Network
VR	Virtual Reality
W	Watt
WP	Work Package
XAI	Explainable Artificial Intelligence
XR	Extended Reality

EXECUTIVE SUMMARY

This document is the first deliverable of WP5 - D5.1: Integration activities and Digital Twin of SAFE-6G pilot. This deliverable reports on the intermediate status of the integrated SAFE-6G platform and the Digital Twin component. This deliverable sets the foundation for the continuous integration of components developed in the project to deliver an E2E system.

Specifically, D5.1 describes the technologies, the modules and tools that will be integrated as part of the SAFE-6G platform to provide the expected trustworthy-related features. In particular, this document describes the infrastructure that constitutes the distributed computational and networking continuum of SAFE-6G, the integration status of all SAFE-6G components, including the GitOps automations for continuous integration, testing, and deployment, as well as the definition of the Digital Twin component. Furthermore, it provides detailed sequence diagrams that illustrate the data flow among all SAFE-6G components from WP4 and the DataOps and MLOps frameworks delivered by WP3.

KEYWORDS

Continuous integration, Continuous development, GitOps, DevSecOps, Computational and network continuum, Chatbot, Cognitive Coordinator, Network Digital Twin, Trust Functions.

TABLE OF CONTENTS

List of FIGURES

List of TABLES

1 INTRODUCTION

The SAFE-6G project is dedicated to fostering an End-to-End (E2E) cognitive trustworthiness framework for user-centric distributed 6G networks, emphasizing the edge-cloud continuum as a key enabler of their future evolution. As 6G network complexity increases, the trustworthiness of user-centric applications can no longer be taken for granted. SAFE-6G designed the architecture (WP2) of an end-to-end cognitive trustworthiness framework for user-centric 6G networks across the edge-cloud continuum, addressing safety, security, privacy, resilience, and reliability to enable trusted services driven by user intent. In parallel, WP3 and WP4 develop AI-assisted functions and a Cognitive Coordinator (CoCo) that compute Levels of Trustworthiness (LoTw) using AI/ML techniques, covering services lifecycle from deployment to decommitment in user service nodes (USN) and network service nodes (NSN). Following the SAFE-6G architecture, WP5's scope is to successfully integrate all the components, deliver the SAFE-6G platform, and coordinate the validation process using the two metaverse-based use cases. This deliverable provides a detailed report of the outputs generated from activities conducted up to M25 within Tasks T5.1 and T5.2 and delivers the intermediate versions of the SAFE-6G platform alongside the DT component.

1.1 OBJECTIVE OF THIS DELIVERABLE

The primary objective of this deliverable is to describe the process followed by the consortium for integrating all modules and components that constitute the SAFE-6G framework, including the CoCo, the Trust Functions (TFs), the edge–cloud continuum infrastructure, and the distributed core functions. It also aims to present the integration of the respective Artificial Intelligence (AI) agents and the components related to the Metaverse pilots. Furthermore, the deliverable indicates how SAFE-6G will leverage the Stream C SNS 6GSANDBOX platform, as well as the UPV and UNIWA infrastructures, which already provide computing resources for hosting experimental use cases that serve as the baseline for the integration activities in WP5. Another key objective is the design of the Digital Twin (DT) over the SAFE-6G ecosystem's components and network to both generate reliable synthetic datasets for training AI/ML models when real-world datasets are not available, and also perform an analysis about the performance of the physical and network layers before a user request is served by the other SAFE-6G components.

Overall, this document serves as a fundamental pillar for the ongoing integration activities of the project's building blocks, as it enumerates their features and provides the specifications required for their integration. It describes the current status of the various modules of the SAFE-6G architecture and reports on the integration activities already performed among partners. Additionally, it outlines the process for testing and validating each component at both the unit and system levels. At the time of this deliverable's submission, intermediate versions of all involved building blocks and components have been fully implemented and integrated. Although development and integration activities continue, the SAFE-6G platform currently consists of the following:

- The edge-cloud continuum, which apart from computing and network infrastructure, comprises the Meta-OS tailored to the needs of SAFE-6G, the Cloud Native technologies (e.g.,

Container Network Interface (CNI) plugin, service mesh, etc.) that supports it as well as the Application programming interfaces (APIs) exposed to interact with it.

- The main trustworthiness modules of SAFE-6G, comprising the CoCo and the five TFs that address Safety, Security, Privacy, Resilience, and Reliability.
- The B5G core network based on open5GS and Cumucore 5G Core, which consists of the evolved components of the network core and its exposed capabilities (APIs).
- The metaverse-based application, which includes all relevant hardware, software, and APIs to implement and manage the project use cases.
- The Common API Framework (CAPIF), which unifies and exposes the APIs from the previous planes (i.e., continuum, 5G core and application) using a trusted, secure framework.
- The user-intent chatbot, which will act as the main interface for the users of the platform.
- The MLOps, which enable the development of ML pipelines (from data pre-processing for models training and inference processes). Among its components, one can find the Differential privacy module, which is analysed in detail in this deliverable and the Explainable artificial intelligence (xAI) component, which falls outside the scope of this document.
- The DataOps which along with the deployed exporters, gathers and stores all relevant (real and simulated) data needed to support the training processes managed by the MLOps.

The project is following a thorough development and integration time plan (*Section 3.3 Integration Plan*) to steadily implement new continuum, core and use case functionalities that will be integrated into 6G-SANDBOX platform.

The following sections will report the integration process of each building block, focusing on the efforts carried for the integration.

1.2 Relation to other documents

This deliverable builds upon the foundational work presented in prior deliverables: D2.2, which describes the overall architecture of the SAFE-6G platform; D2.3, which reports the definition of metaverse-based use cases; D3.1, which details the user-centric distributed B5G core over the edge-cloud continuum with MLOps integration; and D4.1, which defines the Cognitive Coordinator, AI agents, and user-centric functions.

For the purposes of this deliverable, only the components directly related to the integration activities of WP5 are described in detail. The underlying infrastructure elements developed and extended in WP3, such as aerOS, the MLOps framework, the testbed, and the B5G core, are considered the foundational technical layer of the SAFE-6G ecosystem and are therefore not re-analyzed here, as their full specifications are already covered in their respective deliverables. The focus of this document is on how the remaining modules and functionalities are built upon and integrated with this baseline infrastructure, ensuring that the content remains aligned with the scope of WP5.

2 DevSecOps practices and tools

SAFE-6G adopts Development, Security and Operations (DevSecOps) practices to support platform integration activities and manage upcoming platform releases. In simple terms, DevSecOps extends the DevOps methodology by embedding security considerations throughout the entire DevOps cycle. DevOps itself is a set of practices aligned with Agile software development that integrates software development and Information Technology (IT) operations, with the goal of accelerating the system development lifecycle and enabling continuous delivery of high-quality software. DevSecOps enhances this approach by incorporating security by design, meaning that infrastructure and application security are integrated into the workflow from the outset rather than added later. DevSecOps emphasizes the automated integration of security measures at every stage of the software lifecycle, from initial design and integration to testing, deployment, and final delivery. It ensures that security becomes a fundamental component of Agile DevOps methods, including Continuous Integration (CI), continuous delivery, and collaborative development.

SAFE-6G adopts DevSecOps best practices by incorporating threat modeling and security requirements from initial design phases, embedding automated Static Application Security Testing (SAST)/ Dynamic Application Security Testing (DAST) scans into CI/ Continuous Deployment (CD) pipelines starting from the first code commit to enable early vulnerability detection. As detailed in subsequent sections, the project integrates specific tools that apply vertical tests across all components through predefined CI/CD pipelines, while allowing developers to extend these with additional, development-specific tests.

SAST, often referred to as “white box testing,” enables developers to identify security vulnerabilities directly within the application’s source code early in the software development lifecycle. SAST examines code without executing it, ensuring adherence to coding guidelines and standards while helping detect issues at an early stage. Integrating a static analyser into the CI/CD pipeline further reduces the likelihood of defects progressing to later phases of development.

DAST, or “black box testing,” focuses on identifying security weaknesses in a running application. Conducted later in the development lifecycle, DAST requires a fully built and tested system, and the tester works without insight into the underlying source code, technologies, or frameworks. Essentially, DAST assesses the security of deployed software by supplying potentially malicious inputs to uncover vulnerabilities, particularly those associated with the application’s operational environment.

2.1 Continuous Integration and Continuous Delivery

CI is a development practice focused on maintaining a consistently functioning system by integrating small, incremental changes into the mainline on a frequent basis, typically at least once per day. This process relies on automation tools and extensive automated testing, enabling teams to collaborate on shared codebases while improving transparency and overall system quality. As a developer-centric practice, CI generally assumes the use of test driven development and continual refactoring. By writing unit tests (UTs) before code and ensuring these tests always pass locally, developers help maintain a stable working environment. A detailed description of the predefined tests for each component is provided in Section 4 *Integration status of SAFE-6G components*. CD refers to the automated release

of new versions of a system into the production environment. Building on the CI process, once the system meets predefined maturity or quality criteria, CD manages the automated rollout of updated versions with minimal service interruption. The description of the automated CI/CD pipelines and the SAFE-6G computational continuum is provided in Section 3 *SAFE-6G Integration methodology* . Together, CI and CD form a pipeline that delivers new developments and ensures that an updated, operational version of the system is consistently available within the targeted environment. In SAFE-6G, a CI/CD environment has already been set up in GitLab. Figure 1 provides a high-level view of GitLab's CI/CD pipeline.

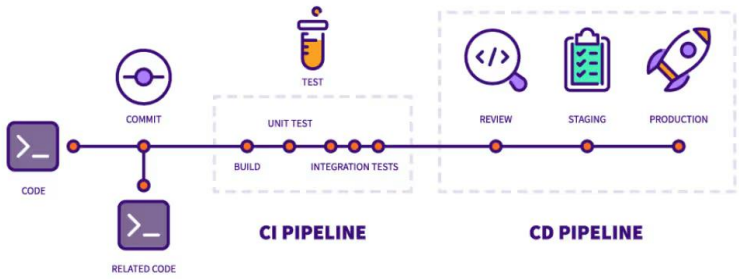


Figure 1 High level overview of CI/CD pipeline

Figure 2 illustrates the main steps of GitLab's CI/CD process in more detail, which can be summarized as follows:

- The developer implements a software component intended to extend or integrate with an existing module.
- Corresponding unit tests are created to validate the correctness and consistency of the module's outputs.
- The developer commits and pushes the new code from the local repository to a designated development branch in the remote GitLab repository, as defined within the CI infrastructure.
- A merge request is submitted.
- The project's CI/CD pipeline is automatically triggered, and the CI platform executes the predefined unit tests.
    - If unit testing succeeds, the code proceeds to integrated system testing. If integrated system testing also succeeds, the merge request is approved, and the new code becomes part of the master branch.
    - If testing fails at any stage, the merge request is rejected. In such cases, the developer must revise the implementation or update the unit tests to meet the required criteria.
- Once merged, the source code management (SCM) platform transitions to the CD phase, deploying the corresponding package.

- After successful deployment, the updated code becomes available in the operational infrastructure and enters user testing or production use.

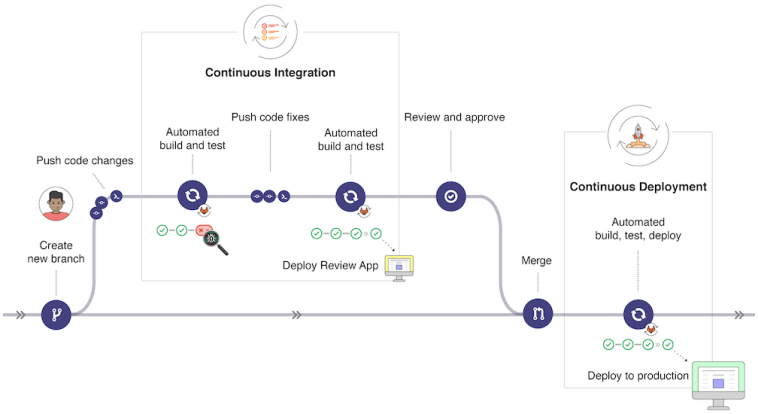


Figure 2 CI/CD pipeline process

2.2 CONTAINERIZATION

Containerization refers to an Operating System (OS) -level virtualization approach used for deploying and running distributed applications. It provides a lightweight alternative to Virtual Machines (VMs) by encapsulating an application and its dependencies within an isolated container environment. The term “container” is inspired by logistics, where goods are packaged in standardized units for efficient transport.

Docker [1] is the most widely adopted containerization technology. It offers a suite of Platform-as-a-Service (PaaS) tools that leverage OS-level virtualization to package and deliver software. A Docker container image is a lightweight, standalone executable bundle that contains everything required to run an application such as its code, runtime, system tools, libraries, and configuration settings [2] Complementing Docker, Kubernetes (K8s) [3] is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. It has become the de facto standard for orchestrating large-scale container environments, including those built with Docker.

Aligned with the cloud-native approach and the modular architectural principles described above, SAFE-6G adopts Docker as its primary containerization framework and strongly encourages project developers to package their technical components as Docker images. These containerized applications will ultimately be deployed within the SAFE-6G integration and testing infrastructure, which uses K8s as its orchestration platform. Further details on the specifications of the SAFE-6G integration and testing cluster are provided in the following sections.

2.3 SAFE-6G GITOPS

SCM refers to the systematic tracking of changes made to a codebase. Maintaining a detailed history of code modifications allows programmers, developers, and testers to work with accurate, up-to-date versions of the software and helps prevent or resolve conflicts when merging contributions from multiple sources. SCM is often referred to as a Version Control System (VCS). Among available VCS tools, GIT is the most widely used, which is an open-source, distributed system designed for tracking

changes across sets of files. SAFE-6G implements a GitOps that enables secure, auditable workflows with protected branches, merge requests, and drift detection, aligning with DevSecOps practices for 6G platform reliability across distributed consortium teams. Platforms such as GitLab [3] and GitHub [4] build on Git by providing repository management along with additional capabilities including code reviews, issue tracking, and documentation through wikis. These solutions function as comprehensive, stand-alone web platforms that support the entire software development lifecycle.

For the SAFE-6G project, the GitLab CI/CD framework has been set up and organized in a Gitlab group. The official GitLab group is entitled "SAFE-6G" and is accessible publicly at https://gitlab.com/safe-6g. The group hosts the source code that is related to each thematic entity-specific development as dictated by the SAFE-6G architecture. Each thematic entity is organized as a subgroup (i.e. code development, ML models development/training/serving, Use-case development, automations, documentation, etc.) of the SAFE-6G GitLab group, as illustrated in Figure 3.

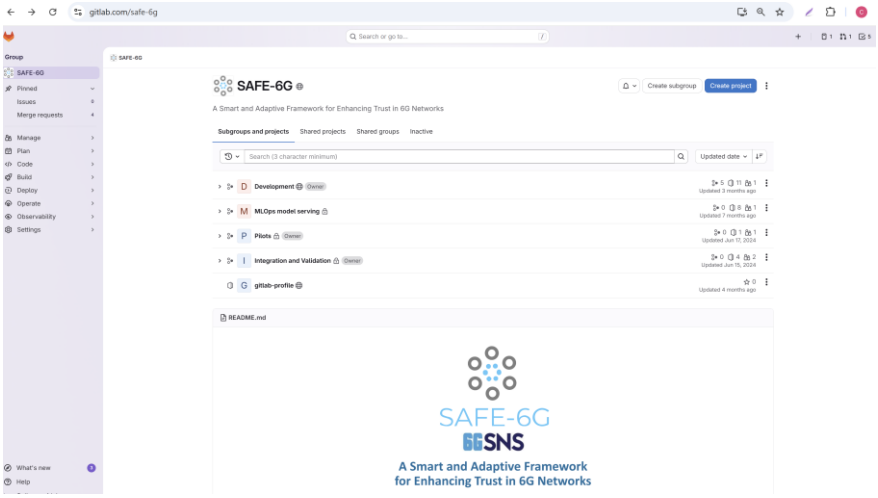


Figure 3 GitLab group of SAFE-6G

Within the GitLab platform, SAFE-6G has configured CI/CD pipelines that enables the automation for the integration and deployment of the project's technical outcomes. The CI/CD pipeline is driven by specific GitLab Runner instances that have been deployed and integrated within the UNIWA, UPV and NCSRD K8s clusters. The Figure 4 depicts the GitLab runners of SAFE-6G and their configuration. More information on the specification of the UNIWA cluster is presented in the following sections.

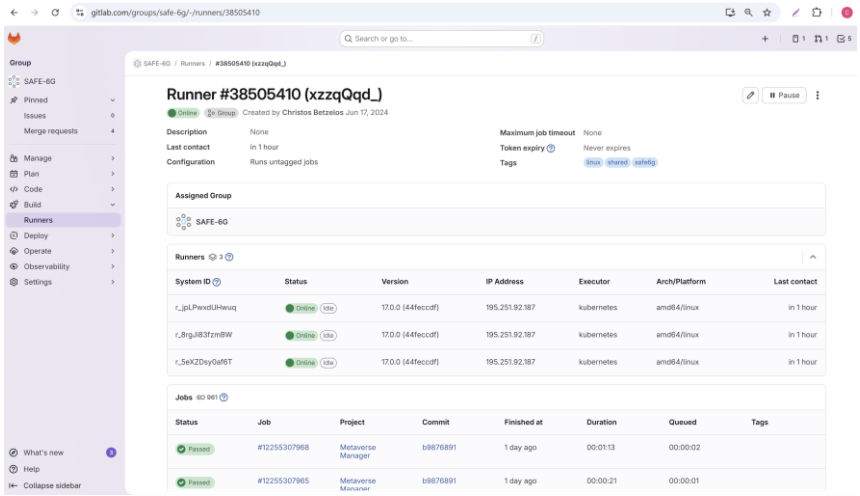


Figure 4 Configurations of GitLab runners

In addition, SAFE-6G, adhering to the cloud native approach, provides automated delivery procedures. More specifically, SAFE-6G docker images are built based on the uploaded source code and in parallel the images are uploaded to the SAFE-6G public docker repository, that has been established for the needs of the project. The SAFE-6G profile in the DockerHub repository is available at https://hub.docker.com/u/safe6g. Figure 5 provides an indicative snapshot of this profile.

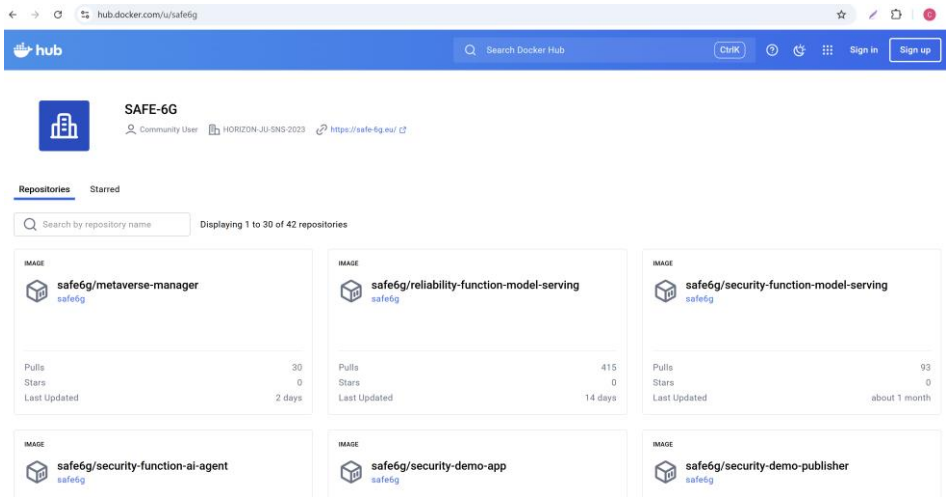


Figure 5 DockerHub repository of SAFE-6G

The issue tracking capabilities of the GitLab web platform can be also utilized for the purposes of the SAFE-6G project. The next Figure 6 illustrates the relevant issue tracking webpage that can represent open and closed issues introduced by the project's technical teams.

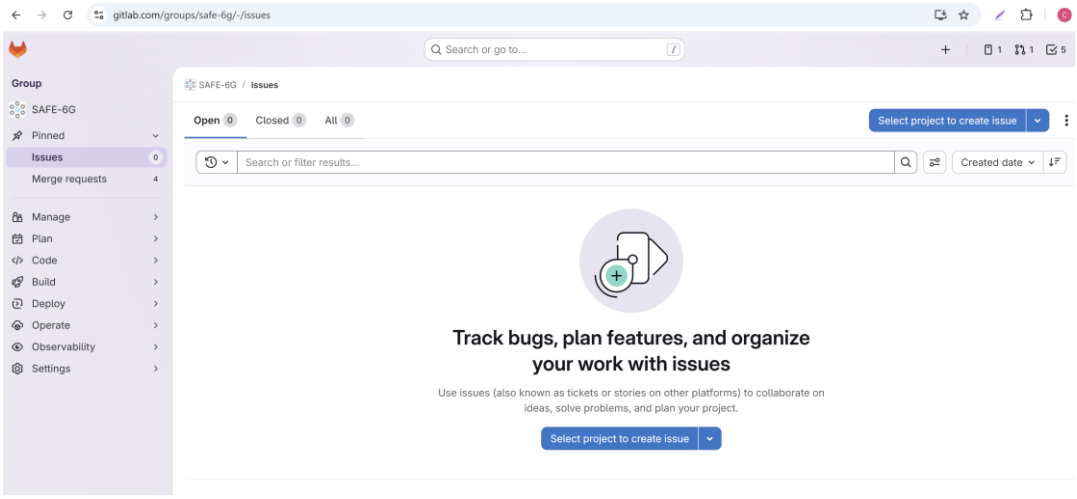


Figure 6 Issue tracking webpage

2.3.1 CI/CD PIPELINES

SAFE-6G leverages GitLab CI for the CI phase and Argo CD for the CD phase within its DevSecOps cycle. Argo CD, a declarative GitOps tool for Kubernetes, automates application deployments and lifecycle management, ensuring they remain fully auditable, transparent, and straightforward to comprehend.

The structure of the GitLab CI process is specified through one or more pipelines. Pipelines are the top-level component of continuous integration, delivery, and deployment. Pipelines comprise:

- Jobs, which define what to do. For example, jobs that compile or test code.
- Stages, which define when to run the jobs. For example, stages that run tests after stages that compile the code.

Jobs are executed by runners. If all jobs in a stage succeed, the pipeline moves on to the next stage. If any job in a stage fails, the next stage is not (usually) executed, and the pipeline ends early.

To ensure software quality, security, and reproducibility within the SAFE-6G project, the following CI/CD pipeline, depicted in Figure 7, is adopted as a standard initial template for all software deliverables.

Figure 7 CI/CD Template Pipeline

The pipeline is structured into six sequential stages:

- The ***lint*** stage performs static code quality checks (e.g., formatting, style rules, and simple defect detection). This stage provides rapid feedback to developers and prevents avoidable issues (such as syntax errors, unused imports, or style violations) from propagating downstream. The pipeline proceeds only if linting completes successfully.

- The ***sast*** stage integrates SAST using GitLab's SAST capabilities and complementary analysers. The stage is expected to pass, and findings can be configured as blocking (fail the pipeline) or non-blocking (raise warnings) depending on specific component policy and criticality. In the provided template, two SAST analysers are executed:
  Bandit (bandit-sast) [5]: Focused on Python security issues (e.g., unsafe use of subprocess, weak cryptographic practices, injection risks).
  Semgrep (semgrep-sast) [6]: Rule-based static analysis with broad coverage, suitable for detecting security anti-patterns and software-specific policy violations.

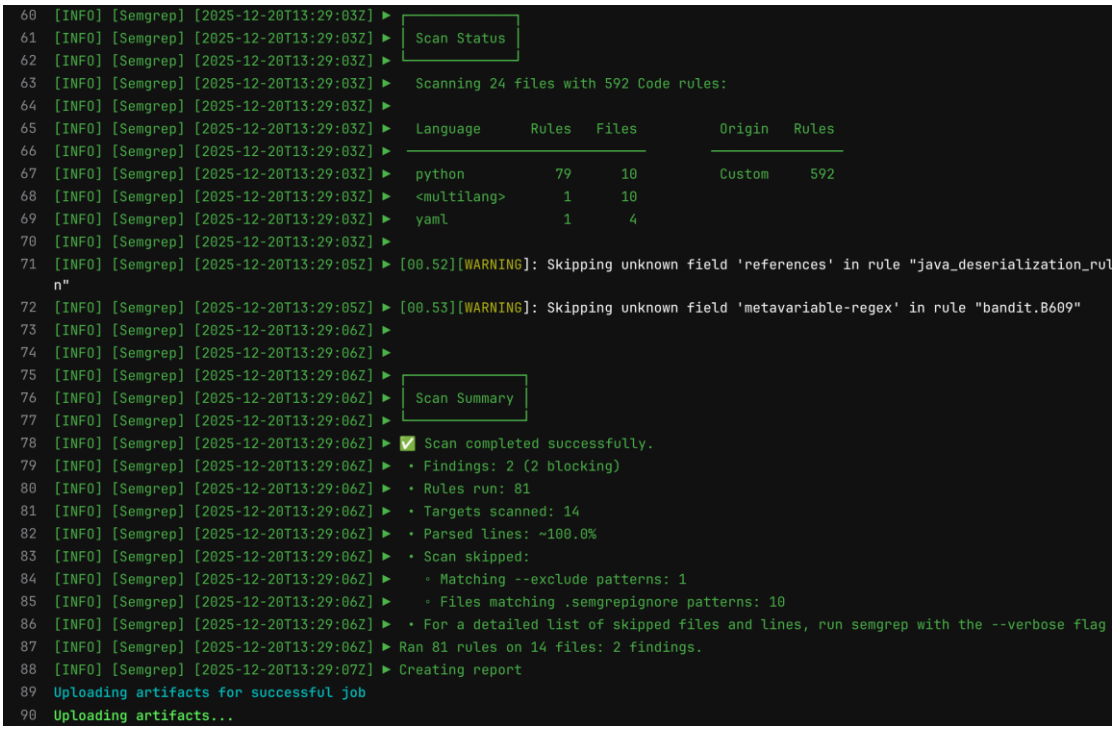


Figure 8 Semgrep SAST analyser within the GitLab SAST stage for the reliability TF

The Figure 8 depicts the execution output of the Semgrep SAST analyser within the GitLab SAST stage for the Reliability TF. The scan completed successfully, analysing 24 files using 592 custom security rules across multiple languages (primarily Python, with additional YAML and multilang rules). The scan finished without errors and produced 2 findings, both marked as blocking, indicating that these issues meet the configured policy threshold to fail the pipeline. The summary confirms that 81 rules were executed on 14 targets, with all relevant lines successfully parsed. Warnings related to unsupported rule fields were logged but did not affect scan completion. The SAST job also uploads scan results as GitLab artefacts, ensuring findings are retained for review, reporting, and audit purposes.

- The ***unit_test*** stage executes automated unit tests, designed to confirm that the smallest testable parts of the software behave as intended. Unit tests provide fast and deterministic feedback and are an established mechanism for preventing regressions. Failing unit tests stop the pipeline, preventing unverified code from being built or deployed. A detailed description

of the unit tests per SAFE-6G component is provided in Section 4 *Integration status of SAFE-6G components*.

- The ***integration_test*** stage executes integration tests (ITs) covering interactions among multiple components (e.g., service-to-service calls, database connections, API contracts, or message broker interactions). This stage targets defects that are not observable in unit tests, such as configuration mismatches, interface incompatibilities, and runtime integration issues. The pipeline continues only if integration tests succeed, ensuring that build artefacts are derived from a system that is validated at the integration level. A detailed description of the integration tests per SAFE-6G component is provided in Section 4 *Integration status of SAFE-6G components*4.
- The ***build*** stage generates delivery artefacts (e.g., container images, packages, or binaries) using a standardised build process. The build stage also include version tagging aligned with release procedures and generation of metadata useful for audits and reporting.
- The ***deploy*** stage performs deployment of the built artefact to the SAFE-6G platform using the Argo CD. Deployment occurs only after successful completion of all previous stages.

More details about the SAFE-6G integration and validation platform are described in the following sections.

2.3.2 CI/CD Tutorial

To support developers in aligning with SAFE-6G's CI/CD approach, a simple demonstration application is provided as a tutorial, illustrating the basic steps of the CI/CD process. This demo app (the Dockerfile of which is illustrate in Figure 9) is available in GitLab under “Integration and Validation” sub-group (Figure 10) and includes a simple Python Flask application with a My Structured Query Language (MySQL) database, that uses GitLab CI/CD and Argo CD for the DevOps cycle (URL: https://demo-app.safe6g.ih.uniwa.gr) The purpose of this demo application is to serve as a practical tutorial for SAFE-6G developers, streamlining the deployment process of all system components. Next, a detailed walkthrough is provided of the key steps required to deploy a microservice within the SAFE-6G continuum, including code development, container image creation, testing, image publication on Docker Hub, and service deployment to one of the available K8s clusters in the SAFE-6G infrastructure.

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r /app/requirements.txt
COPY . .
CMD ["python", "/app/app.py"]
```

Figure 9 Dockerfile of the demo App

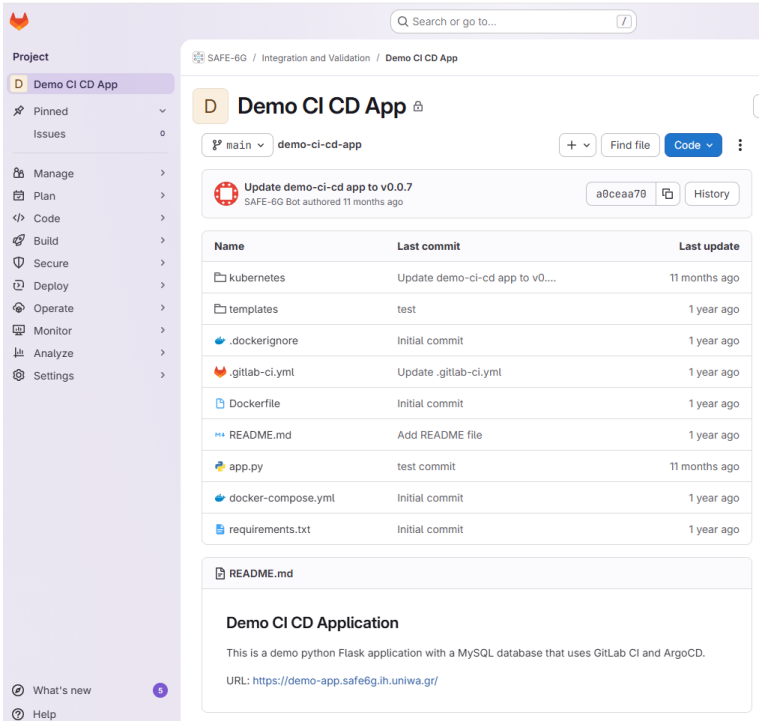


Figure 10 CI/CD demonstration App

2.3.3 GitLab CI Pipeline

The pipeline description is based on the *(.gitlab-ci.yml)* file located at the repository root (Figure 11), which defines:

- The structure and the order of jobs that the runner will execute.
- The decisions the runner will make when specific conditions are encountered.

For this tutorial, the pipeline is configured with three basic stages:

- The **test stage** for testing execution (i.e. unit tests and integration test).
- The **build stage** for code compilation, docker image building and is pushing it to docker registry.
- The **deploy stage** for deployment of the new image to the K8s platform. Semantic versioning has been used for our application, and the pipeline is configured accordingly.

Moreover, three predefined group variables that can be used from any project of the SAFE-6G group regarding the DockerHub authentication and Argo CD authentication:

- $DOCKER_USER
- $DOCKER_PASSWORD
- $SSH_PRIVATE_KEY

```
default:
  image: docker:26.0.0
  services:
    - docker:26.0.0-dind
```

```
variables:
  DOCKER_IMAGE: "$DOCKER_USER/$CI_PROJECT_NAME"
  DEPLOY_FILE: kubernetes/web-app.yaml

stages:
  - test
  - build
  - deploy

test:
  stage: test
  script:
    - echo "Running tests..."
  rules:
    - if: '$CI_COMMIT_AUTHOR =~ /SAFE-6G Bot <bot@safe6g.eu>/'
      when: never
    - if: '$CI_COMMIT_REF_NAME == "main"'
      when: always
    - if: '$CI_COMMIT_TAG'
      when: always
    - when: never
build:
  stage: build
  script:
    - docker login -u $DOCKER_USER -p $DOCKER_PASSWORD
    - docker build -t $DOCKER_IMAGE:latest .
    - docker push $DOCKER_IMAGE:latest
  rules:
    - if: '$CI_COMMIT_AUTHOR =~ /SAFE-6G Bot <bot@safe6g.eu>/'
      when: never
    - if: '$CI_COMMIT_REF_NAME == "main"'
      when: always
    - if: '$CI_COMMIT_TAG'
      when: never
    - when: never

build-tags:
  stage: build
  script:
    - docker login -u $DOCKER_USER -p $DOCKER_PASSWORD
    - docker build -t $DOCKER_IMAGE:$CI_COMMIT_TAG .
    - docker push $DOCKER_IMAGE:$CI_COMMIT_TAG
  rules:
    - if: '$CI_COMMIT_AUTHOR =~ /SAFE-6G Bot <bot@safe6g.eu>/'
      when: never
    - if: '$CI_COMMIT_REF_NAME == "main"'
      when: never
    - if: '$CI COMMIT TAG'
      when: always
    - when: never
deploy:
  stage: deploy
  image: ubuntu
  before_script:
    - 'command -v ssh-agent >/dev/null || ( apt-get update -y && apt-get install openssh-
client git -y )'
    - eval $(ssh-agent -s)
    - chmod 400 "$SSH_PRIVATE_KEY"
    - ssh-add "$SSH_PRIVATE_KEY"
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - git config --global user.email "bot@safe6g.eu"
    - git config --global user.name "SAFE-6G Bot"
    - '[[ -f /.dockerenv ]] && echo -e "Host *\n\tStrictHostKeyChecking no\n\n" >>
~/.ssh/config'
  script:
    - git clone git@gitlab.com:safe-6g/integration-and-validation/demo-ci-cd-app.git
    - cd demo-ci-cd-app
    - sed -i "s&\(${DOCKER_IMAGE}\):v[0-999]\+\.[0-999]\+\.[0-999]\+&\1:${CI_COMMIT_TAG}&g"
"${DEPLOY_FILE}"
    - git add .
    - git commit -m "Update demo-ci-cd app to ${CI_COMMIT_TAG}"
    - git push origin main
```

```
rules:
  - if: '$CI_COMMIT_AUTHOR =~ /SAFE-6G Bot <bot@safe6g.eu>/'
    when: never
  - if: '$CI_COMMIT_REF_NAME == "main"'
    when: never
  - if: '$CI_COMMIT_TAG'
    when: always
  - when: never
```

Figure 11 .gitlab-ci.yml

When the user pushes his code to main branch, only the first two stages are triggered as shown in next Figure 12. The build job also pushes the image (safe6g/demo-ci-cd-app) to DockerHub with a tag latest.

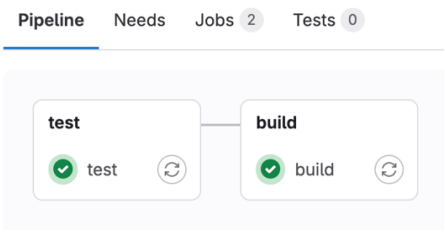


Figure 12 Demo CI/CD App

Upon creating a new tag, all pipeline stages are triggered, as shown in Figure 13 below. The build job pushes the image (safe6g/demo-ci-cd-app) to DockerHub with the tag $CI_COMMIT_TAG, which is the tag name, while the deploy job updates the Kubernetes manifests accordingly. The K8s manifest configuration has been specified on the *kubernetes* directory at the root of the repository. In the *web-app.yaml* file we have the flask application manifests with the (Domain Name System) DNS and (Transport Layer Security) TLS configuration using a Deployment, a Service and an Ingress object. In the *mysql-helm-chart.yaml* file we have the configuration of the MySQL Helm release.

More information about this step will be presented on the next subsection.

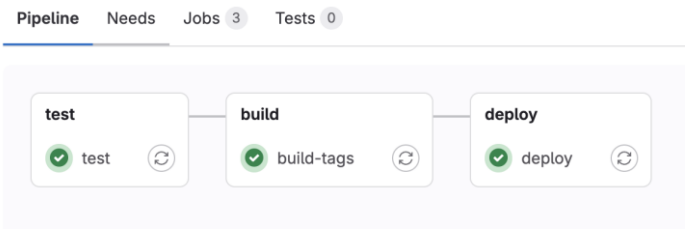


Figure 13 Deployment to DockerHub

More details of a job can be viewed by selecting the job name as depicted in Figure 14.

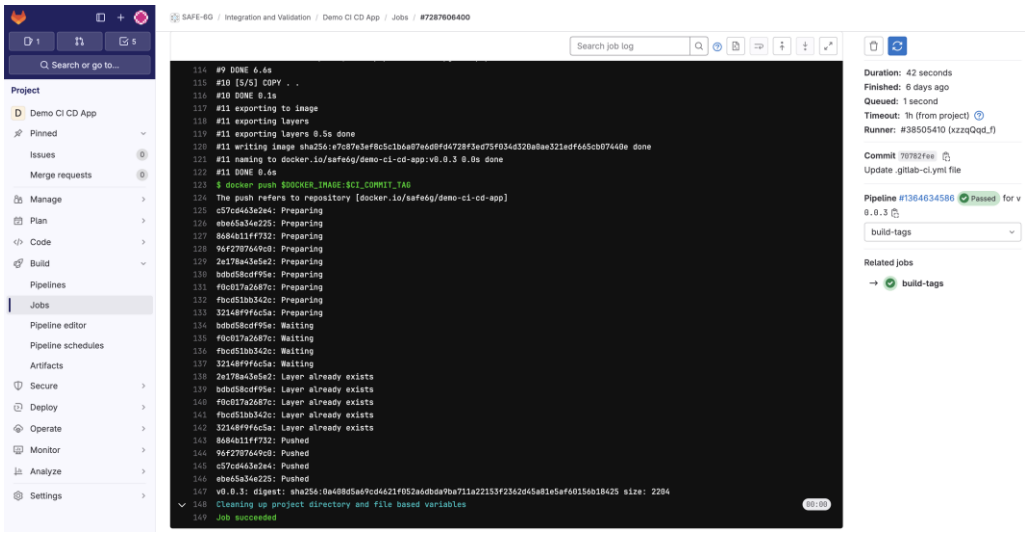


Figure 14 Execution output of the build job

2.3.4 CONTINUOUS DEVELOPMENT CONFIGURATION

This subsection presents the configuration process, based on the Argo CD framework, that is necessary to implement the CD within the SAFE-6G GitOps approach. The goal is to deploy and automatically update the K8s manifests continuously, and the Helm releases stored in the repository. So, to deploy the demo application to the SAFE-6G Kubernetes cluster, the steps below should be followed to connect the GitLab repository with the Argo CD:

1. Through the SAFE-6G Argo CD Page, open **Settings** > **Repositories page** and click on the *Connect Repo button*.

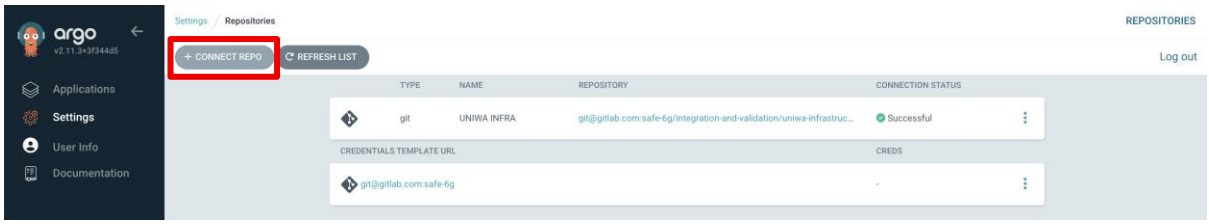


Figure 15 Argo CD: Repositories

2. Provide a *Name*, select **safe6g** *Project* and in the *Repository URL* provide the repository clone with ssh URL.

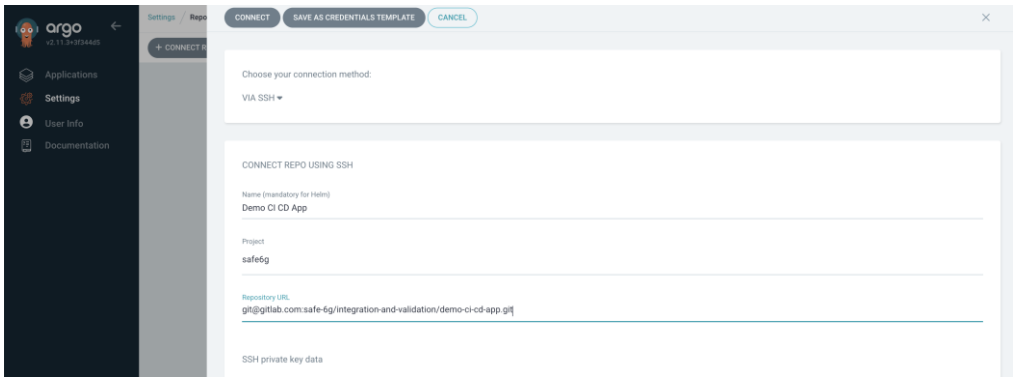


Figure 16 Argo CD: *+ Connect Repo* configuration

3. Click *Connect* and wait until the *Connection Status* to be **Successful**.

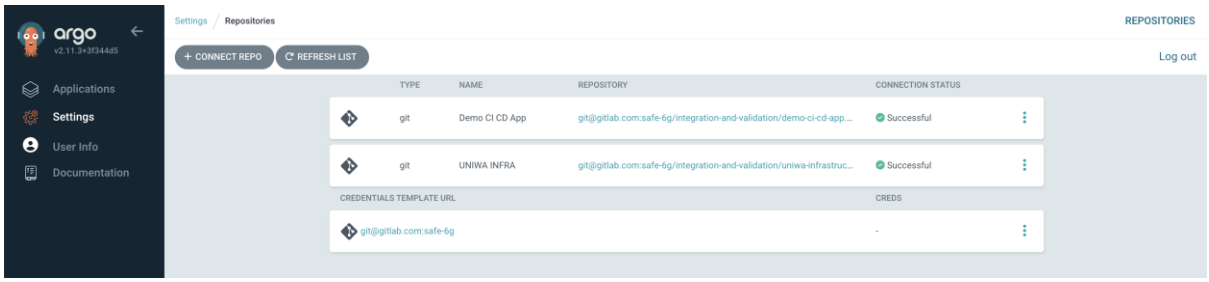


Figure 17 Argo CD: Repository connection successfully

4. After repository connection, create an Argo Application that uses this repository. Navigate to **Applications** and click **New App**. On the General fields, provide an *Application Name*, select the **safe6g** as *Project Name* and the **Automatic** *Sync Policy* with **Self Heal**.

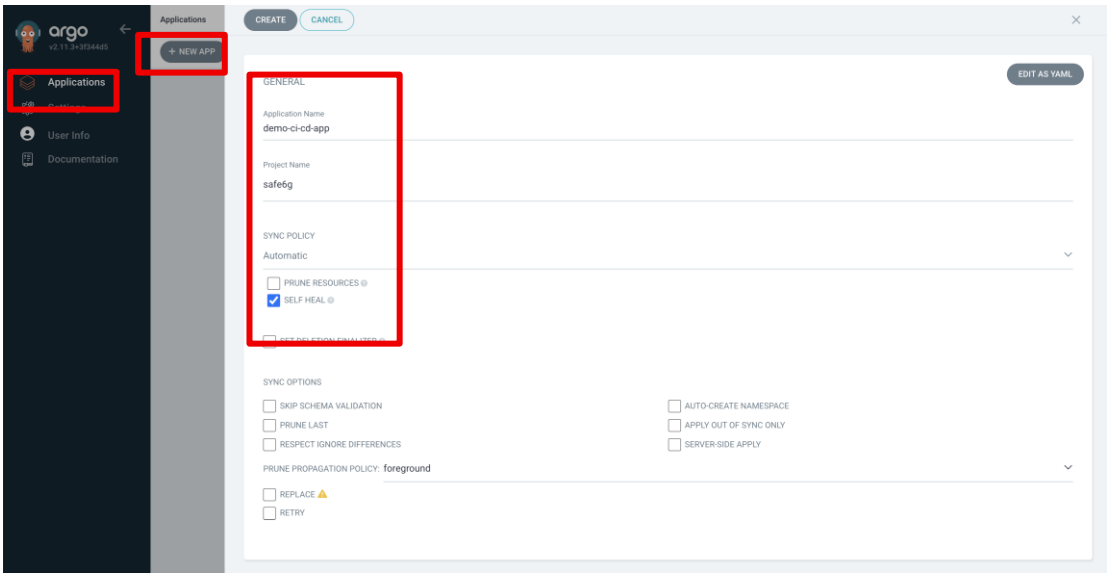


Figure 18 Argo CD: Application

5. Then select the repository in the *Repository URL* field and in the *path* field select the directory that contains the K8s manifests. In this case for the Demo Application is the **kubernetes** directory name. In the *Cluster URL,* select the predefined one and, in the namespace, provide the **safe6g**. Click on the *Directory Recurse* check box in case there are nested directories inside the main kubernetes directory.

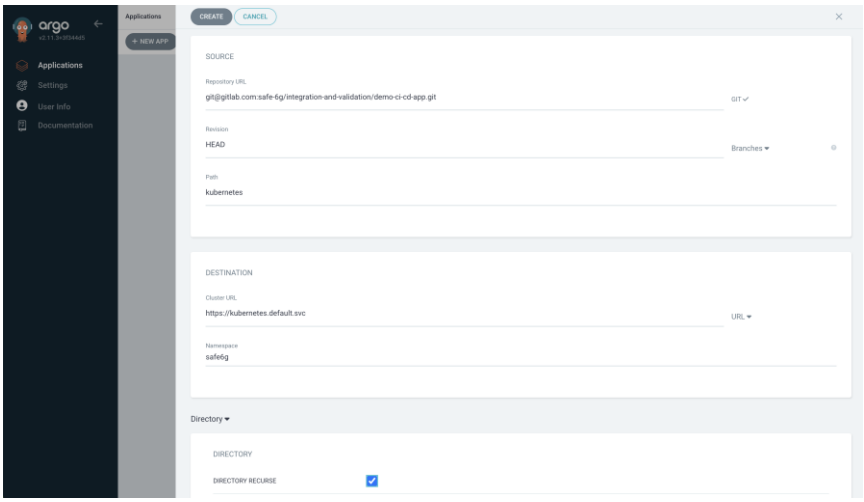


Figure 19 Argo CD: Repository configuration

6. Click on the *Create* button and wait until the manifests been deployed to the K8s cluster.

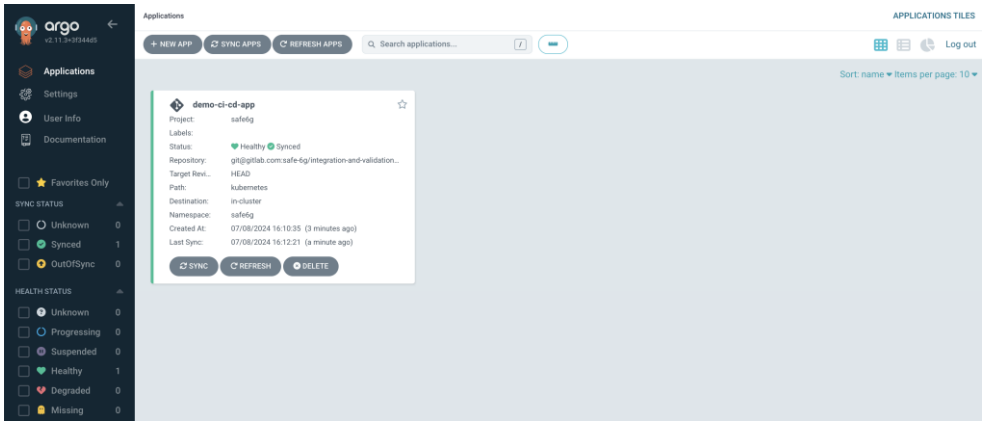


Figure 20 Argo CD: K8s manifests deployment

7. By clicking on the application, Argo CD displays all the K8s resources deployed with their status.

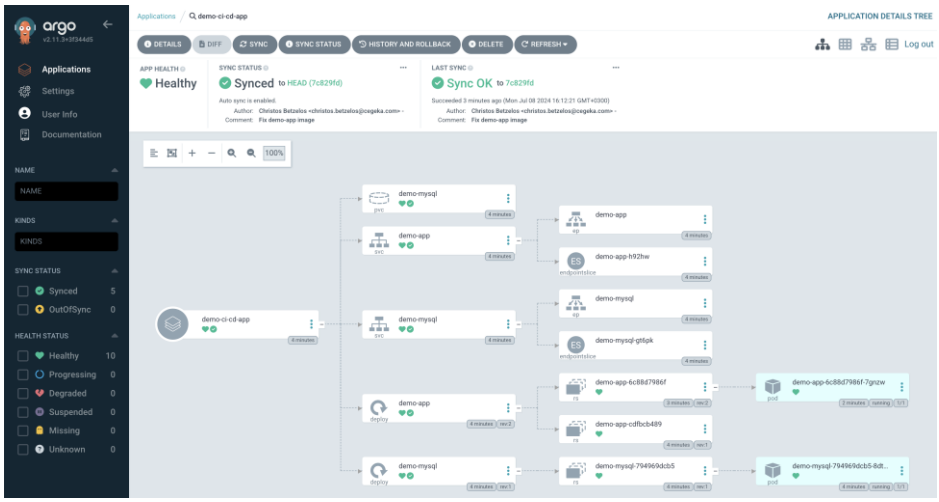


Figure 21 Argo CD: Status application

8. Click on the **Pod**, in case you want to see **logs**, **events**, or to take a **terminal** shell.

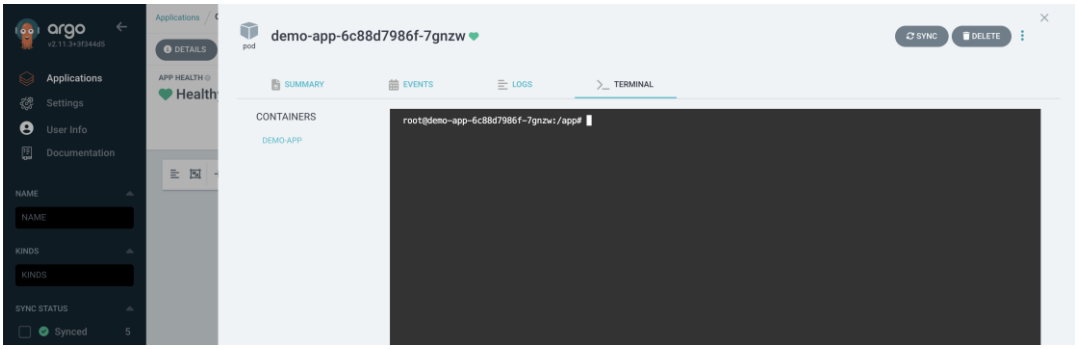


Figure 22 Argo CD: Terminal application

9. To deploy a **Helm release**, add a declarative Argo CD Application manifest on the *kubernetes* directory, like the one depicted in Figure 23 for the Demo App's MySQL. For more information regarding Argo Applications for Helm, the official Argo CD documentation can be followed.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: demo-mysql
  namespace: argocd
spec:
  destination:
    namespace: safe6g
    server: 'https://kubernetes.default.svc'
  project: safe6g
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
      allowEmpty: true
    syncOptions:
      - CreateNamespace=false
  source:
    chart: mysql
    repoURL: https://charts.bitnami.com/bitnami
    targetRevision: 11.1.9
```

```
    helm:
      releaseName: demo-mysql
      values: |
        auth:
          rootPassword: rootpassword
          database: test_db
        primary:
          persistence:
            size: 1Gi
```

Figure 23 ArgoCD Helm Chart

10. In this case, there are two applications on the Argo CD page. One for the plain K8s manifests and one for the Helm release.

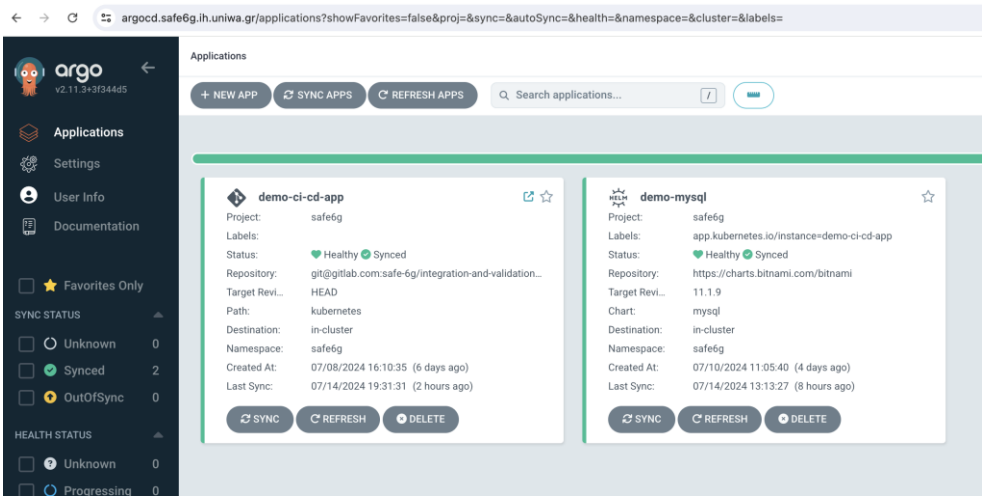


Figure 24 Argo CD: Application's K8s manifests and Helm release

3 SAFE-6G INTEGRATION METHODOLOGY

This section provides an in-depth analysis of the implementation of the continuum infrastructure and the integration plan of the SAFE-6G. Given the complexity and interdependence of software development, integration, and testing processes, these activities can become highly challenging without a systematic approach from the outset, particularly when development teams collaborate remotely. To ensure efficiency and consistency, a clear and comprehensive integration plan must be established and adopted by all technical partners of the project. This plan is presented in *Section 3.3 Integration Plan* and is structured around the activities and outputs of WP2, WP3, and WP4, ensuring that all technical results are integrated into the SAFE-6G platform without delays and that the ongoing development of one module does not negatively impact the others.

3.1 THE SAFE-6G CONTINUUM

In line with the integration strategy defined in WP5, the infrastructure environment hosts consolidated versions of the SAFE-6G building blocks developed in WP3 and WP4, including the edge-cloud continuum, the 5G core network, and the supporting MLOps/DataOps frameworks. This environment serves as the reference platform for the validation and demonstration of SAFE-6G platform. All SAFE-6G technical components produced within the project have been deployed, integrated, and tested on a distributed environment spanning three interconnected domains, namely NCSRD (6G-SANDBOX Athens platform [7]), UNIWA, and UPV. This multi-domain setup constitutes the primary experimentation and validation infrastructure of SAFE-6G, enabling realistic E2E evaluations across heterogeneous administrative and technological sectors. In particular, the 6G-SANDBOX Athens platform at NCSRD acts as the production reference site, providing a realistic B5G experimentation environment that supports the execution of E2E scenarios combining compute, network, and application planes across the edge-cloud continuum. The UNIWA and UPV domains complement this setup by contributing additional compute, High-performance computing (HPC), AI, and orchestration capabilities, collectively forming a federated development and validation environment aligned with the continuum and Meta-OS principles. The SAFE-6G components are deployed following cloud-native principles, leveraging containerized services and K8s-based orchestration extended through the aerOS-enabled edge-cloud continuum [8] enabling multi-domain experimentation and federation of the three heterogeneous computational nodes enabling the validation of SAFE-6G mechanisms under realistic multi-stakeholder and multi-domain conditions, which are essential for future 6G deployments. In Figure 25 Management of SAFE-6G Continuum domains via aerOS we illustrate the SAFE-6G continuum domains.

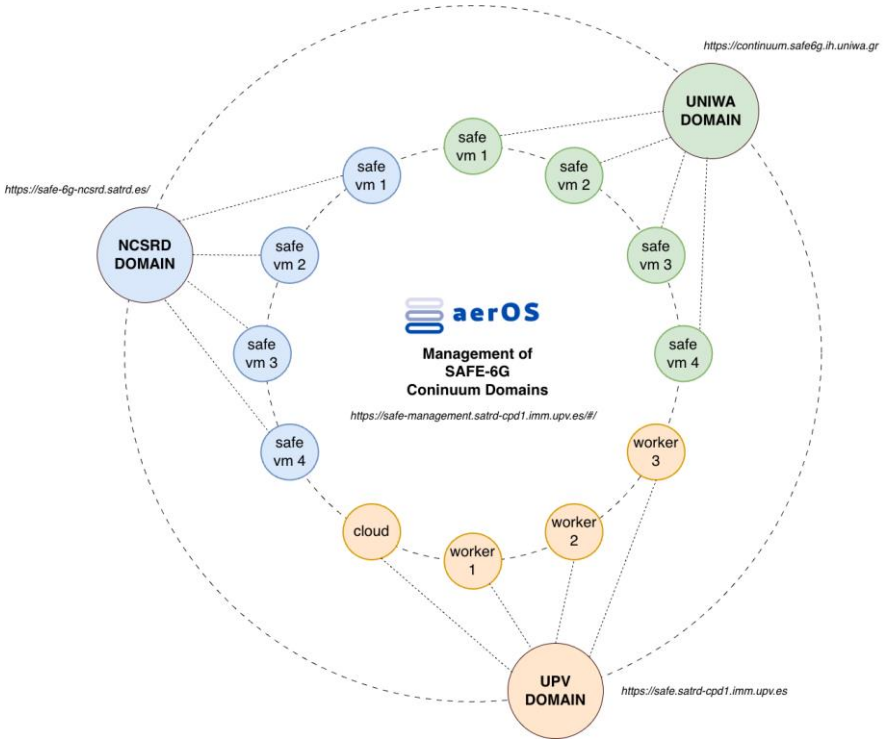


Figure 25 Management of SAFE-6G Continuum domains via aerOS

3.1.1 UPV DOMAIN

The UPV domain serves as the development environment, enabling partners to implement software designs from scratch, execute initial tests, and perform preliminary integrations. While each consortium partner maintains their own computing premises for early development activities, UPV has provided a dedicated set of servers to facilitate platform component development and initial validations. This infrastructure is composed of two K8s clusters, deployed on top of Ubuntu VMs managed by a Proxmox hypervisor [9] , which serve the following purposes:

- Test, validate and adapt the technologies for the computing continuum where more than one cluster will be involved (primarily for multi-cluster orchestration, communication, logging and monitoring).
- General development and initial validation purposes.

The detailed configuration of these clusters is presented in the following Table 1 and Table 2:

Node ID	Node Type	CPU Cores	Memory	Local Storage
worker-1	Worker	4	8 GB	64 GB
worker-2	Worker	4	16 GB	128 GB
worker-3	Worker	4	8 GB	90 GB
cloud	Control Plane	6	32 GB	104 GB

Table 1 UPV Main cluster configuration

Node ID	Node Type	CPU Cores	Memory	Local Storage

dev-worker1	Worker	4	16 GB	100 GB
dev-worker2	Worker	4	32 GB	100 GB
master-dev	Control Plane	8	64 GB	100 GB

Table 2 UPV Main-dev cluster configuration

The infrastructure involves a set of Networking and Cloud Native technologies to facilitate their operation, as seen in Figure 26.

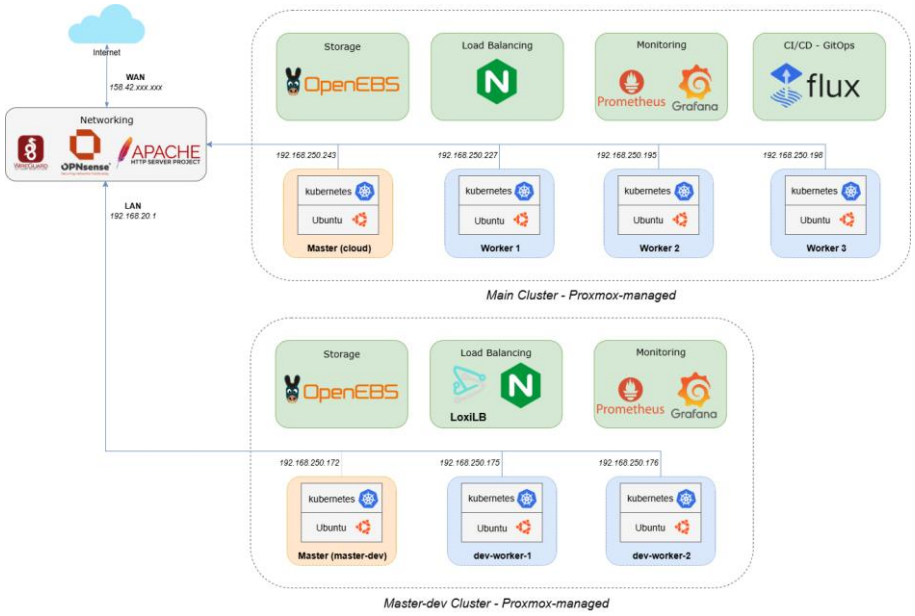


Figure 26 Integration and Testing Environment in UPV premises

The infrastructure relies on Wireguard [10] , OPNsense [11] and Apache server [12] as reverse proxy to facilitate networking from external networks, exposing services and facilitate users' access. Both K8s clusters include OpenEBS [13] for managing the storage of the workloads. While at this moment each node has assigned its own storage, they can be configured to share the same space (configuring an NFS [14]), if needed in case of host corruption or failure – not critical for development environment. Cert manager [15] is being used for certificates' management, with Let's Encrypt [16] as Certificate Authority. Both of them use Prometheus [17] and Grafana [18] for monitoring, alerting, and visualization. In the upcoming months, Loki [19] will be deployed for logs management.

Some differences can be appreciated for other aspects. For instance, the main server makes use of NGINX as Ingress controller [20] acting as both reverse proxy and load balancer, whereas master-dev also leverages LoxiLB [21] for balancing purposes. This is due to LoxiLB being a load balancer more tailored to K8s and 5G ecosystems, as it has some interfaces dedicated to the core, yet some compatibility issues are being studied with the CNI networking plugin implemented, such as Cilium [22] , before moving to the main cluster. Besides, the latter also hosts FluxCD [23] to enable automated deployments and upgrades. It has been chosen as it is the one used by aerOS' developers, the main technology studied, used and being tested in this cluster.

3.1.2 UNIWA DOMAIN

The UNIWA domain serves as integration infrastructure for ensuring the coherent integration of all SAFE-6G components and TFs developed by consortium partners into a unified platform. Within the project, UNIWA provides access to its AI Innovation Hub [24] , which provides cloud computational resources and access to a powerful HPC infrastructure for system testing and training of ML models. The infrastructure of the UNIWA AI Innovation Hub is designed to support multi-user access, easy maintenance, and the ability to add hardware expansion, to adapt to current and future needs. The researchers are able to connect to the infrastructure easily and in a secure manner so that while sending their data for further processing and analysis, their privacy and integrity is ensured. SAFE-6G utilize two K8s clusters designed for different purposes:

- The main cluster acts as the primary testing and integration environment based on VMs over OpenStack.
- The secondary cluster focuses on the development of AI applications, dedicated to the MLOps framework and other components requiring GPU workloads, suitable to support Metaverse use-cases.

The two systems are interconnected, and they can operate collaboratively to host complex workloads that need both CPU and GPU resources. The detailed configuration of the main and secondary clusters is presented in Table 3 and Table 4 respectively.

Node ID	Node Type	CPU Cores	Memory	Local Storage	Ceph Storage
safe-vm1	Worker	8	32 GB	160 GB	150 GB
safe-vm2	Worker	8	32 GB	160 GB	150 GB
safe-vm3	Worker	8	32 GB	160 GB	150 GB
safe-vm4	Control Plane	4	12 GB	100 GB	-

Table 3 UNIWA Main Cluster Configuration

Node ID	Node Type	CPU Cores	Memory	Local Storage	GPU
dgx001	Worker	256	1 TB	3.5 TB	8 × A100-40GB
dgx002	Worker	256	1 TB	3.5 TB	8 × A100-40GB
head1	Control Plane	96	250 GB	450 GB	-

Table 4 UNIWA HPC Cluster Configuration

The current architecture with the most important integrated CNCF applications is presented in Figure 27.

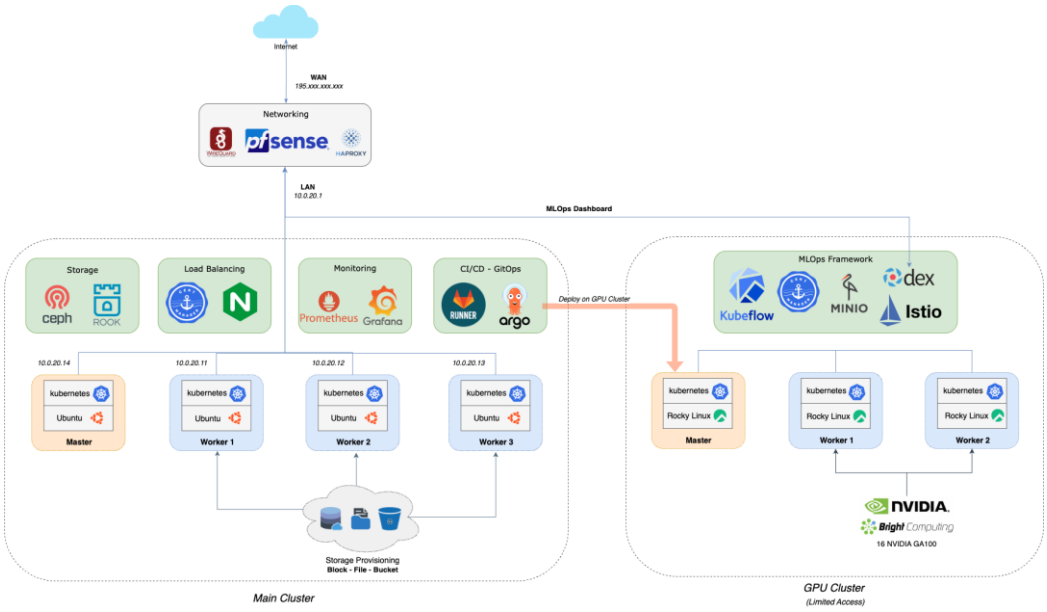


Figure 27 Integration and Testing Environment in UNIWA premises

The main K8s cluster, is built on top of OpenStack [25] Ubuntu [26] OS nodes, provides the core infrastructure services. Currently it consists of four nodes, but it can scale up easily without the redesign of the architecture. The deployment followed the hyper converged architecture paradigm, with specific node profiles for different roles in the cloud. It includes Rook [27] and Ceph [28] for providing reliable and scalable distributed block, file and object storage. Data are stored and replicated across a pool of OSDs to ensure high availability and resilience. Each worker node includes a dedicated partition configured as an OSD, allowing data to be replicated across multiple hosts. This setup helps prevent data loss in the event of a host failure by ensuring that redundant copies of the data remain accessible.

Following the same approach as in UPV, an NGINX as Ingress controller [20] has been deployed as a reverse proxy and load balancer, while cert-manager [15] simplifies the process of obtaining and renewing SSL certificates, using the Let's Encrypt [16] Certificate Authority. Prometheus [17] and Grafana [18] are used for monitoring, alerting, and visualization. The main cluster also hosts the CI/CD and GitOps services through self-hosted GitLab Runners [4] and ArgoCD [29] , enabling automated deployments, as analysed in *Section 2 DevSecOps practices and tools*.

The GPU K8s cluster, running on Rocky Linux OS [30] , is designed for high-performance ML workloads. It consists of two NVIDIA DGX A100 [31] Systems. The DGX A100 is the universal system for all AI infrastructure and workloads, built on NVIDIA A100 Tensor Core GPU and backed by over a decade of AI innovation at NVIDIA. It is a single platform that brings together training, inference, and analytics into a consolidated system with optimized software that creates the foundational building block for AI infrastructure. The job scheduling on the cluster is done by the Bright Cluster Manager [32] , which provides more efficient use of resources and more productive workers. Every NVIDIA DGX A100 server features eight NVIDIA A100 Tensor Core GPUs, which deliver unmatched acceleration, and is fully optimized for NVIDIA CUDA-X software [33] and the E2E NVIDIA data center solution stack. This environment hosts the SAFE-6G MLOps framework including Kubeflow [34] , MinIO [35] Dex [36] and

Istio [37] allowing scalable ML model training, workflows, and serverless inferencing. Networking is handled centrally through a pfSense [38] firewall/router instance which also includes a WireGuard [10] Virtual Private Network (VPN) server, and a HAProxy [39] reverse-proxy for secure remote connection and load balancing respectively.

The current exposed endpoints are described in Table 5.

Public Endpoints	Services
https://argocd.safe6g.ih.uniwa.gr	ArgoCD Dashboard (CI/CD)
https://ceph.safe6g.ih.uniwa.gr	Ceph Object Storage API
https://ceph.safe6g.ih.uniwa.gr/dashboard/	Ceph Dashboard
https://minio.safe6g.ih.uniwa.gr	MinIO Console
https://minio-api.safe6g.ih.uniwa.gr	MinIO API
https://kfpipeline.safe6g.ih.uniwa.gr	Kubeflow Dashboard (main cluster)
https://kubeflow.safe6g.ih.uniwa.gr	Kubeflow Dashboard (sec cluster)
https://model.develop.<TF>.example.safe6g.ih.uniwa.gr	ML Model Serving for each TF

Table 5 Endpoints

3.1.3 NCSRD DOMAIN

The NCSRD domain, also referred to as the experimentation testbed, constitutes the final integration and validation layer of the SAFE-6G framework, along with the interconnected UPV and UNIWA domains, where all developed components are deployed under conditions closely resembling operational B5G infrastructures. In contrast to development and pre-integration environments, this environment is designed to support stable, E2E experimentation, cross-domain validation, and performance assessment of the SAFE-6G architecture under realistic deployment constraints.

The SNS 6G-SANDBOX 5G/6G Athens platform [7] is an advanced large-scale B5G SA experimental facility. At the NCSR Demokritos site, the platform provides fully operational B5G Standalone (SA) infrastructure supporting programmable network functions, open service exposure, and AI-driven experimentation across radio access and core network domains. At the core of the platform's continuum lies aerOS's unified execution layer, which abstracts heterogeneous computing and networking resources into a single cohesive resource pool. This layer is supported by a unified network and compute fabric, enabling the aerOS double-layer orchestration model. In this model, an AI-enabled decision layer optimizes service placement based on a global view of the continuum, while the enforcement layer ensures reliable execution on the selected resources. This architecture ensures decisions are based on comprehensive resource awareness, with seamless deployment across domains. The continuum is inherently scalable, allowing for easy integration of new domains and resources as needed, further enhancing its orchestration capabilities.

The platform hosts four distinct 5G SA networks which brings together the most advanced and widely recognized technologies:

i) The first network leverages Amarisoft 5G RAN with flexible 5G SA Core options, including Amarisoft and Open5GS. These cores offer varying levels of openness (e.g., NEF, NWDAF, CAPIF) and support programmable experimentation. The platform also integrates

software RAN solutions such as UERANSIM (monolithic and containerized) and srsRAN [40] (Next Generation Node B (gNB) and UE via ZeroMQ).

ii) The second network is an OpenAirInterface (OAI) OAIBOX [41] , integrating a complete 5G Core with an O-RAN Split 8 architecture, enabling flexible, open-source experimentation. The OAIBOX serves as a versatile testbed for RAN–core integration in research scenarios.

iii) The third network is provided by INF and is a portable 5G system based on RPi, featuring a containerized 5G Core for lightweight and flexible deployments. This compact solution enables rapid setup of portable 5G networks and supports integration with a variety of external or standalone RAN units, making it suitable for constrained, temporary, or remote environments.

iv) The fourth network is based on the Cumucore 5G Standalone Core and RAN, providing a fully compliant 5G SA implementation.

Cumucore: CMC provided a fully virtualized, software-defined 4G/5G mobile core network specifically designed for private and non-public networks, enabling secure, reliable, and scalable connectivity for industrial, media, and enterprise applications. Its technology supports 3GPP-compliant private networks with features.

Open5GS [40] : Open5GS is an open-source 5Gcore network and a highly suitable option for the Athens platform due to its support for distributed network function (NF) deployments, which aligns with the evolution toward the distributed 6G architecture. One of the key advantages of using Open5GS is the ability to deploy UPF in different locations within the testbed, such as at the edge site and the core site, and associate them with different network slices (e.g., S-NSSAI). Overall, this approach allows for greater flexibility and enables the support of multiple user planes in three dimensions, including network slicing, traffic steering, and Application Function (AF) traffic influence.

Amarisoft [41] 5G SA Core: The Amarisoft 5G Core network solution provides essential NFs for the operation of a 5G network, Access and Mobility Management Function (AMF), Authentication Server Function (AUSF), Session Management Function (SMF), User plane Function (UPF), UDM (Unified Data Management) and 5G-EIR (5G Equipment Identity Register) all integrated within the same software component.

Amarisoft [41] 5G RAN: The Amarisoft 5G New Radio (NR) can operate in FDD/TDD frequency bands below 6 GHz with up to 50 MHz of bandwidth. It supports various subcarrier spacing options for both data and synchronization signals, and can operate in MIMO configurations up to 4x4 in downlink. The aforementioned MIMO layers can be also complimented either in one 5G cell with 50MHz bandwidth and 2x2 MIMO configuration, or three 5G cells with 20MHz bandwidth and 2x2 MIMO configuration each.

This setup enables realistic E2E experimentation, interoperability testing, and validation of advanced 5G features under near-production conditions, complementing the more experimental and lightweight network deployments of the platform. The SAFE-6G experimental environment has been

extended through the aerOS-enabled edge-cloud continuum, enabling multi-domain experimentation and federation of heterogeneous infrastructures. Through this continuum, the laboratory environment interconnects multiple experimental sites, enabling the validation of SAFE-6G mechanisms under realistic multi-stakeholder and multi-domain conditions, which are essential for future 6G deployments. So, containerized 5G NFs, such as Open5GS UPF, benefit from aerOS's orchestration, allowing dynamic deployment and migration based on application-specific or user-centric policies.

Elaborating on the technical details as mentioned, the platform provides extensive RAN and data monitoring capabilities: i) Amarisoft 5G RAN offers flexible configurations for sub-6 GHz bands, supporting bandwidths up to 50 MHz, multiple subcarrier spacings, and MIMO up to 4×4. ii) OAI OAIBOX RAN implements a flexible, open-source 5G RAN architecture based on 3GPP standards, supporting Split 8 functional split for integration with external components. It supports both NSA and SA modes, operates in sub-6 GHz bands with up to 100 MHz bandwidth, 2×2 MIMO, and provides 250 mW (24 dBm) per port output power with up to 32 simultaneous connections. The OAIBOX allows fine-grained protocol stack configuration, making it highly suitable for research-driven, experimental scenarios. iii) Data Collection & Monitoring: A monitoring framework based on Prometheus includes a main server with a time-series database, an alert manager for alarms, Node Exporters for resource metrics, and Grafana for real-time visualization.

Finally, NCSRD contributes to SAFE-6G infrastructure with a suite of open-source implementations that advance network programmability and service exposure in 5G/B5G environments. These include a Monitoring Event API for the NEF, fully aligned with 3GPP specifications, enabling location reporting services for User Equipment (UE) via the 5G Core and delivering this information to Application Functions (AF) on demand in a cloud-native, easily integrable. In the context of CAMARA project, NCSRD has developed an open-source Device Location Retrieval API, which retrieves and renders network-detected device locations from a 5G System (Open5GS) to a Network Application (nApp), including a demonstrator that visualises the fetched location on a map. Complementing these, the O-NEF emulator provides a flexible, open-source platform for simulating NEF behaviour, supporting experimentation with service exposure, data collection, and application integration without requiring full production infrastructure. Together, these tools form a robust, standards-compliant foundation for developing, validating, and integrating a number of different applications within the SAFE-6G ecosystem. NCSRD hosts a robust data center within the Athens platform, providing 10 high-performance servers each equipped with 256 GB RAM, 2×12-core CPUs, and a combined 2.4 TB storage capacity. The environment leverages OpenNebula and Proxmox for efficient virtualization and cloud orchestration, enabling scalable, secure, and flexible experimental deployments for open callers.

In Figure 28, we pictorially describe the 6G-SANDBOX Athens platform.

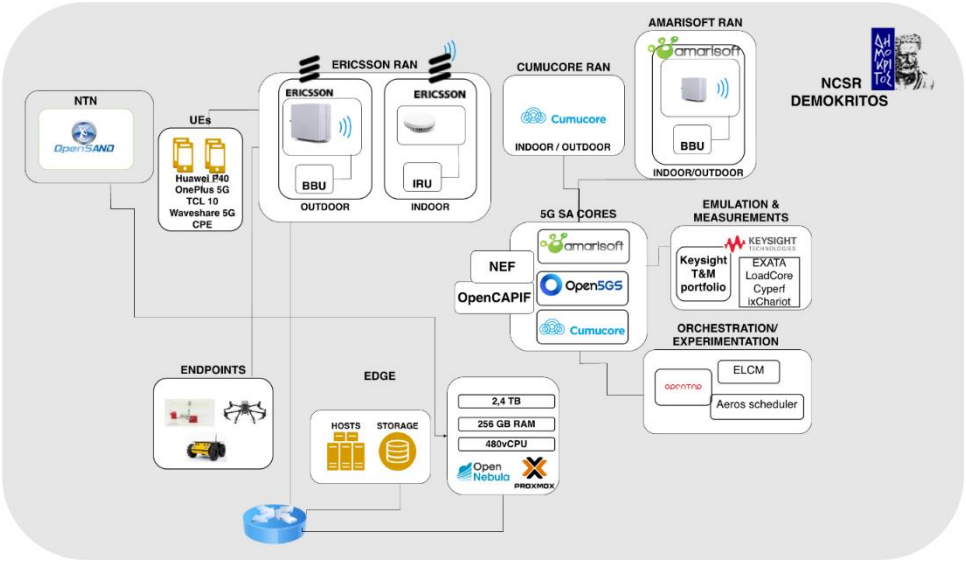


Figure 28 6G-SANDBOX Athens platform

The detailed technical characteristics and capabilities of this virtualized environment (e.g., compute, storage, networking, and acceleration resources) are detailed below:

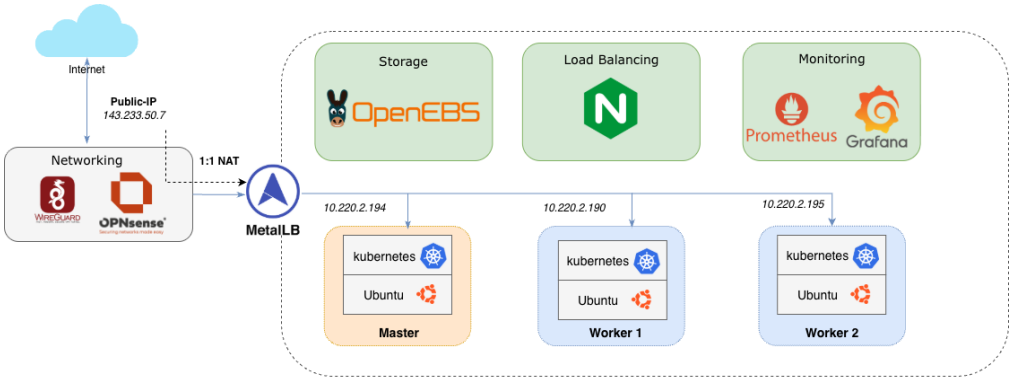


Figure 29 Deployment environment in NCSRD premises.

Node ID	Node Type	CPU Cores	Memory	Local Storage
ncsrd-worker	worker	3	13 GB	57 GB
ncsrd-worker3	worker	3	13 GB	57 GB
ubuntu	Control-plane	3	13 GB	57 GB

Table 6 NCSRD cluster configuration

3.2 SAFE-6G Software Architecture and Integration Guidelines

The technical foundation of the SAFE-6G platform is built upon its core components and TFs which can be viewed as a set of multiple modules that follow specific functional design principles defined by the SAFE-6G architecture. They are developed by distributed and diverse software teams, and once implemented, they must interoperate to deliver the functionality envisioned for SAFE-6G. For this reason, SAFE-6G adopted a modular architecture which reflected to the development activities. In SAFE-6G each module behaves as an independent, self-contained building block, exposing its

functionality through a set of well-defined and well-documented interfaces. This approach allows each team to focus solely on the external interfaces provided by other modules, without needing to understand their internal implementation. In effect, modular design promotes a) the decomposition of the system into independent modules, with minimal interdependencies and b) the clear separation within each module between the communication interfaces and the actual implementation of code. Furthermore, the modular design allows SAFE-6G to fully harness cloud computing principles by integrating cloud-native technologies with DevSecOps practices and modern automation tools, thereby streamlining the integration process and making it significantly faster and more efficient.

3.2.1 INTERFACES AND DATA MODELS

Beyond modular design practices, which align naturally with cloud-native technologies and automation to support the seamless integration of SAFE-6G, the well-defined component interfaces and data models play an equally critical role. Data models specify the structure and format of the information exchanged between components via their interfaces. They serve as visual and conceptual representations of data elements and relationships among them. As such, a data model defines not only the types or classes of data used within the system but also how these data elements relate to one another. Doing so promotes the open exchange and sharing of structured information among SAFE-6G stakeholders and enhances interoperability across the components and functions developed within the project.

Regarding interfaces, SAFE-6G adopted standardized approaches for the inter-module communication, such as RESTful APIs. REST is a stateless client-server architectural style based on a set of principles that govern how resources are defined and accessed. Communication between client and server is performed over the Hypertext Transfer Protocol (HTTP), and RESTful APIs use standard HTTP methods (POST, GET, PUT, DELETE) to support the creation, retrieval, updating, and deletion of resources. Through RESTful architectural paradigm SAFE-6G a) implements a lightweight and language independent communication b) supports scalability and flexibility and c) simplifies the integration process with existing and future modules and platforms. In addition to the REST architectural style, SAFE-6G developers are also encouraged to adopt the publish–subscribe (Pub/Sub) messaging pattern. Pub/Sub is an architectural design pattern that provides a structured framework for exchanging messages between publishers and subscribers. In this pattern, a message broker mediates communication, receiving messages from publishers, and distributing them to all interested subscribers. Pub/Sub communication approach makes SAFE-6G capable to support a) asynchronous message exchange, meaning that once a request is issued, the sending application is not blocked while waiting for a response. This reduces the risk of performance degradation or time-outs during long-running data exchanges and b) scalability and flexibility by adding or removing subscribers, without complex programming efforts. Furthermore, the decoupling between publishers and subscribers allows individual components to scale independently based on workload demands and enables the seamless integration of new services or consumers at runtime. This flexibility is particularly important in the distributed 6G infrastructures, where system components may be dynamically instantiated, updated, or decommissioned without disrupting ongoing operations.

Each component of the SAFE-6G platform employs either the RESTful or Pub/Sub communication approach, or a combination of both, depending on its specific interaction requirements. The following

Table 7 presents the communication methods established among the core modules of the SAFE-6G platform.

SAFE-6G Module	Communication Interfaces
Cognitive Coordinator	RESTful API
Chatbot	RESTful API
DataOps	RESTful API
MLOps	RESTful API
Safety Function	Pub/Sub
Security Function	Pub/Sub
Privacy Function	Pub/Sub
Resilience Function	Pub/Sub
Reliability Function	Pub/Sub
5G Core network	RESTful API
Edge Cloud Continuum	RESTful API
Metaverse App manager	RESTful API
Network Digital Twin	RESTful API
XAI	RESTful API

Table 7 SAFE-6G Communication Interfaces

SAFE-6G employs a cluster of Apache Kafka [44] brokers that receive messages from producers, assign them sequential offsets, and persist them in topic partitions on disk before subsequently serving them to consumers on demand. In this architecture, SAFE-6G is capable to support multiple partitions across various topics, replicate data to additional brokers to achieve fault tolerance, and use leader/follower roles so that exactly one broker per partition handles all reads and writes while followers remain synchronized and can seamlessly take over in case of a failure. Collectively, the brokers provide a distributed, horizontally scalable, and highly available streaming backbone, where any broker can serve as a bootstrap entry point to expose cluster metadata and direct clients to the appropriate partitions. The defined topic names and their corresponding message types are provided in Annex 1. However, for deployments in environments that require simpler operations or have limited resources, the same functionalities may be provided by Redpanda [45] .

3.2.2 DOCUMENTATION

Proper documentation in SAFE-6G is considered a fundamental enabler of effective development and integration activities. It is viewed as a vertical requirement that spans the entire project and must be met with every development and integration step. Beginning with source code documentation, clear and well-structured comments significantly improve the comprehensibility of software, whether it is revisited by the original author after some time or by another developer altogether. Such documentation also facilitates future updates or corrections, helping developers understand the logic of the code and avoid misconceptions or errors that might otherwise arise. Overall, it constitutes the best practice that all project contributors are encouraged to follow.

API documentation is also essential, especially for integration purposes. It allows consumers of a component to understand how to invoke and interact with the provided components. As discussed

earlier, the OpenAPI Specification (OAS) offers a standardized, language-agnostic interface description for RESTful APIs, enabling developers to explore a service's capabilities without needing access to its source code, additional documentation, or network inspection. Within SAFE-6G, developers use online documentation tools such as Swagger to comprehensively document all APIs (Figure 30), This documentation is continuously updated and hosted on GitLab Pages.

It is also important to emphasize that the SAFE-6G platform is intended to be released as Open Source. Consequently, the adoption of standardized practices and high-quality documentation is critical. All SAFE-6G components and TFs must therefore be accompanied by comprehensive documentation that clearly describes each component's objectives, functionality, API specifications, deployment details, and example-based usage instructions. This ensures that the project's technical outputs are reusable, maintainable, and accessible to the broader community.

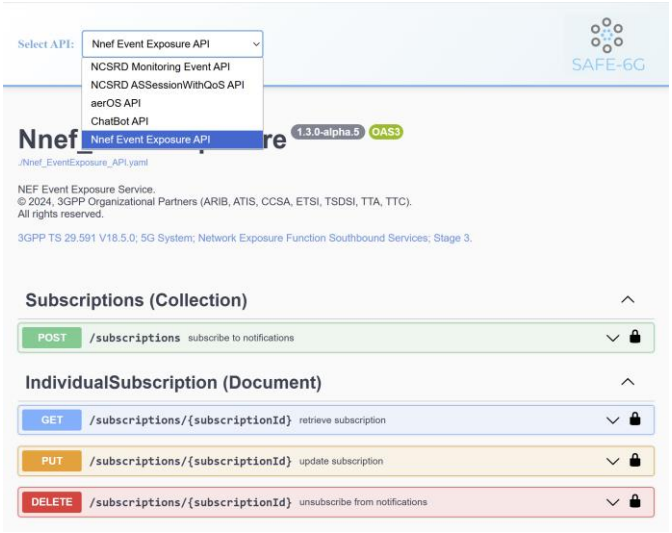


Figure 30 SAFE-6G RESTful API documentation OpenAPI Specification

3.3 Integration Plan

SAFE-6G follows a structured integration plan phased approach across five key milestones (M18, M22, M24, M28, M30), ensuring progressive development of technical work packages WP3 and WP4, alongside comprehensive testing and validation of the modular architecture established in WP2. The adopted timeline coordinates distributed development teams through a linear yet iterative workflow, emphasizing DevSecOps practices, CI/CD automation, and testing procedures. Each milestone serves as a quality checkpoint, leading to the final version of SAFE-6G platform (v1.0), while ensuring compliance with SAFE-6G's use-case requirements. Figure 31 summarizes the critical milestones of the integration plan, along with the corresponding actions planned for each one of them.

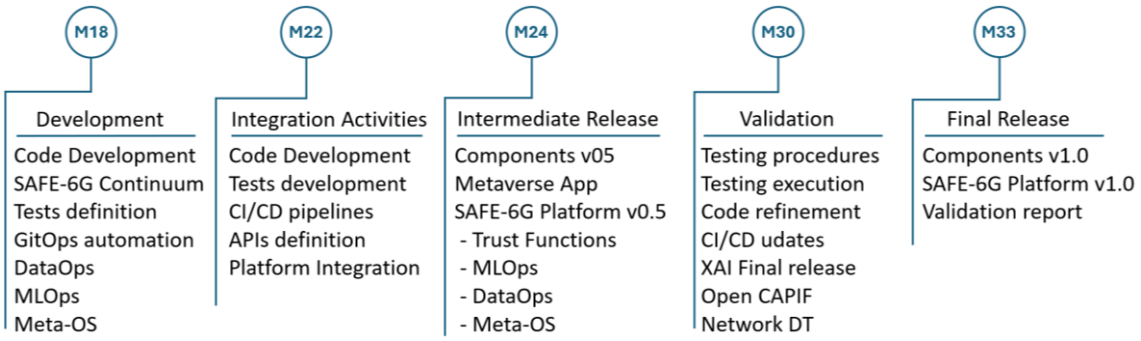


Figure 31 SAFE-6G platform integration plan

M18 (Code Development) establishes the core development toolchain (GitOps workflows) to support WP3 and WP4 activities, while delivering the initial integrated versions of DevOps, MLOps, and meta-OS frameworks. This phase also defines the RESTful APIs and Pub/Sub interfaces, alongside module-specific testing procedures.

M22 (Integration Activities) advances the platform through CI/CD pipeline implementation, full platform integration, and SAFE-6G metaverse application development. DataOps and MLOps frameworks reach full integration maturity, while 5G network connectivity at NCSRD premises is finalized, providing a fully integrated SAFE-6G continuum prototype ready for early inter-component communication and data flows across K8s clusters hosted at UPV, UNIWA, and NCSRD nodes.

M24 (Intermediate V0.5) delivers stable intermediate versions (V0.5) of individual components integrated within the SAFE-6G platform. This phase delivers mature version of SAFE-6G modules, TFs, and Metaverse application, ready for comprehensive testing, with particular emphasis on interface compliance and data model interoperability between TFs according to the use-case scenarios.

M30 (Validation) implements comprehensive testing procedures aligned with WP2-defined KPIs/KVIs. Testing outcomes drive code refactoring across SAFE-6G modules, TFs, Metaverse application, and DT, while DevOps/MLOps deliver the production-ready final versions of ML models. This phase produces detailed validation reports covering E2E functionality, platform performance, and use-case validation.

M33 (Final V1.0) delivers the final version (1.0) of the SAFE-6G platform and the Metaverse application. At the end of this phase, the platform will achieve full maturity through performance benchmarking, validation, and use-case demonstrations, enabling operational deployment across the SAFE-6G continuum.

The adopted integration plan enables parallel development across distributed teams while strictly enforcing modular design principles by emphasizing interface/code implementation separation, standardized communication protocols, and containerized microservices. Continuous DevOps/MLOps integration ensures zero-downtime updates through rolling deployments and automated validation gates, guaranteeing platform reliability and scalability for production 6G deployments.

3.4 SAFE-6G SEQUENCE DIAGRAM

The first integrated version (v0.5) of the SAFE-6G platform incorporates all core system modules, including the Chatbot, Cognitive Coordinator, Pub/Sub message broker, TFs, DataOps, and MLOps, deployed across the SAFE-6G continuum (cloud/edge/5G network). This section outlines the standardized sequence flow that SAFE-6G employs for deploying TFs, structured across five distinct phases. These phases illustrate the collection and transformation of user intents into calibrated trust values, and how these values engage the appropriate Trust Framework to ensure system trustworthiness throughout the SAFE-6G ecosystem. The presented flowchart illustrates the generic approach to interactions between SAFE-6G platform components, with a focus on TF deployment in response to user intents. However, certain TFs follow a slightly different approach tailored to their specific requirements. Detailed sequence diagrams for each TF are provided in their respective sections.

Phase 1: User intents collection

- The process begins when the **Tenant/User** submits intents to the SAFE-6G system through the **Chatbot**.
- The chatbot forwards these intents to the **Cognitive Coordinator**.
- The **LoTw Calculator** computes the non-calibrated level of trustworthiness (nLoTw) and queries the **User Registry** for information related to the active services and users.
- The LoTw Calculator informs the Conflict Resolution component with the nLoTw values of all Trust Frameworks to detect potential contradictory actions, then triggers TF deployment via the message broker.

Phase 2: TF deployment

- The **AI Agent** of the corresponding TF consumes the nLoTw score from the **Message Broker**.
- **AI Agent** acquires a token from **CAPIF** to ensure secure access to the aerOS API.
- The **AI Agent** requests deployment of the appropriate flavors of vertical Application **(vApp)** and **nApp(s)** via **aerOS**.
- The **AI Agent** maps the score to the appropriate trustworthiness level (low, medium, high).
- The **vApp** retrieves the appropriate AI/ML model from the model serving service of the MLOps.
- The **nApp** acquires a token from CAPIF to authorize API interactions with the 5G Core and DataOps framework.

Phase 3: TF Interfacing with Different Planes

- The **nApps** collect data via the DataOps framework from three distinct data planes, based on the needs of the deployed vApp flavor:
  - The **Edge-Cloud Continuum**, for infrastructure-level telemetry.
  - The **5G Core**, for network-level metrics and session data.
  - The application plane's **Metaverse Manager**, for application-level metrics and contextual information**.**

Phase 4: Data Processing and Analysis

- Collected data is processed internally and forwarded to the **vApps**.

- The **vApps** perform data analysis and generate events, which are then sent back to the AI Agent for integration with the trustworthiness logic.

Phase 5: Decision execution and feedback

- Based on the analysis outputs, the **AI Agent** determines the appropriate optimization actions.
- Proposed actions are published via the **Message Broker** to the **Cognitive Coordinator**.
- The **Conflict Resolution** module consumes these actions, validates them against the **Knowledge base** and confirms or rejects them.
- Approved actions are then executed by the AI Agent across multiple planes:
  - **aerOS** for resource orchestration over the cloud/edge continuum.
  - **Metaverse Manager** for application level optimisation.
  - **5G Core** for network level optimization.
- Finally, the **Message Broker** delivers the outcome to the **Chatbot**, which reports the calibrated LoTw (cLoTw) and explanatory context back to the user.

In Figure 32, we illustrate the referred sequence flow of SAFE-6G platform.

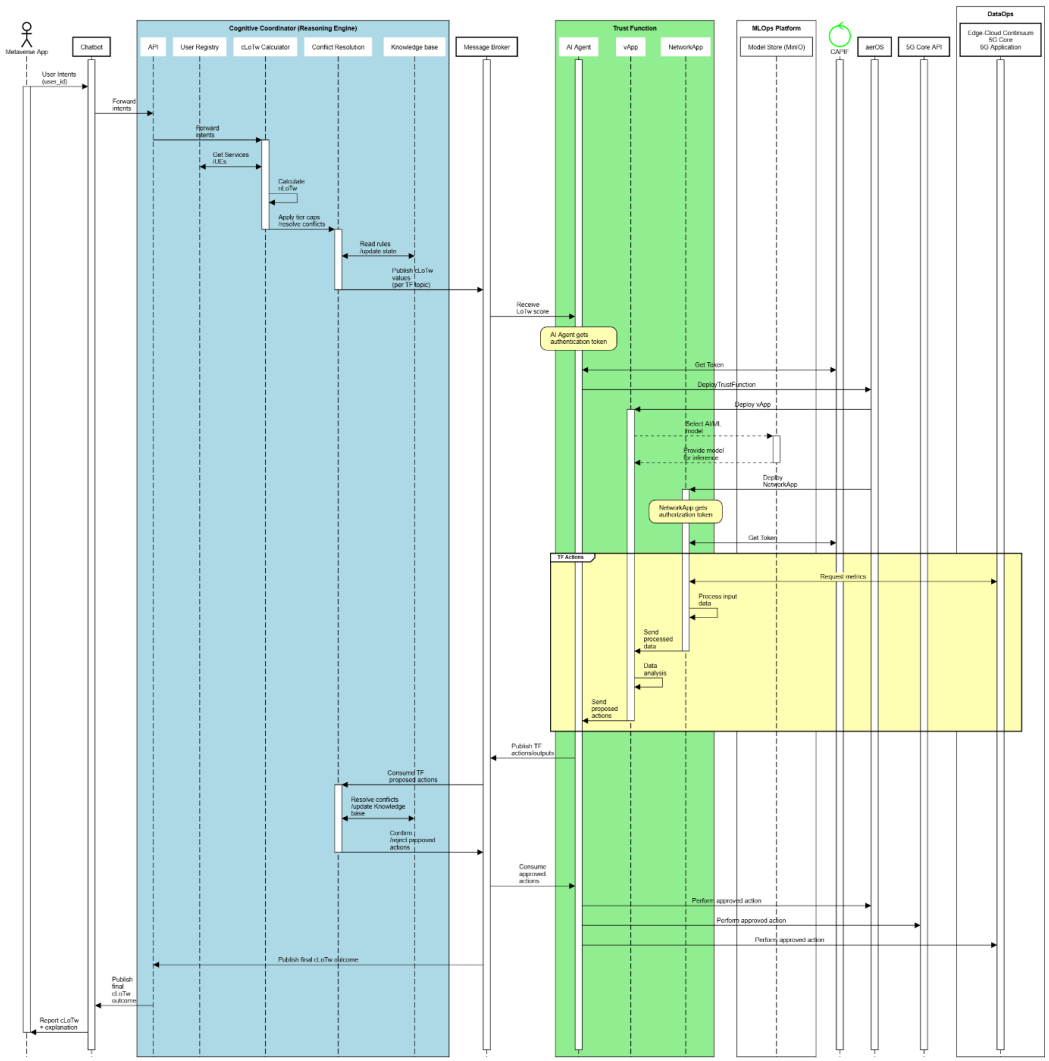


Figure 32 SAFE-6G sequence diagram

4 INTEGRATION STATUS OF SAFE-6G COMPONENTS

This section presents the current integration status of the SAFE-6G building blocks, focusing on how the individual components developed within WP3 and WP4 are progressively combined into a coherent, E2E system in the context of WP5. The objective is to document the level of maturity, interaction readiness and deployment status of each component as they are instantiated within the SAFE-6G laboratory environment. Figure 33 provides a consolidated view of the SAFE-6G system architecture as described in WP2 and WP3, illustrating the relationships between the main building blocks and their placement across the distributed experimentation environment. For more details on the operation and role of each component, the reader is referred to Deliverables 2.2 and 4.1. The next sub-sections present the integration of each major component in the SAFE-6G ecosystem along with the corresponding unit and integration tests.

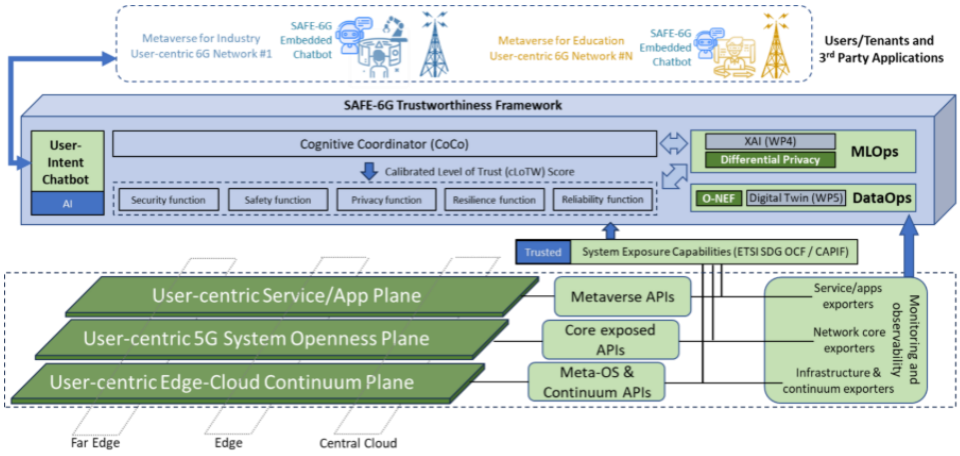


Figure 33 SAFE-6G system architecture

4.1 CHATBOT

4.1.1 OVERVIEW

The chatbot API component provides a REST-style interface that enables users to interact conversationally with the SAFE-6G virtual assistant. Through natural-language input, the users can express their intent and define requirements as a prosumer within the human-centric 6G environment. The chatbot focuses on dialogue management and intent understanding: it extracts and classifies user requests across the five trust-related functions (Safety, Security, Privacy, Resilience and Reliability) and keeps the conversation flowing to gather any missing information. Instead of computing trust-levels itself, the API delivers the structured intent and TF data to the CoCo, which is responsible for deriving the actual trust-levels. In this way, the chatbot supports an adaptive, user-centric 6G service workflow by bridging the conversational interface with the CoCo. In addition to the API, a web-based chatbot application was developed. This web application implements the same conversational logic and processing pipeline as the chatbot API and serves as an interactive front-end for end users. It was also used as the primary vehicle for unit testing and validation of the chatbot functionality, ensuring consistency between the API-level behaviour and the user-facing interface.

4.1.2 ARCHITECTURE

The chatbot architecture is built around two core AI components exposed through the chatbot API: an Large Language Model (LLM)-based dialogue engine (Llama 3 orchestrated via LangChain) and a dedicated text-classification model that either maps user requests to five predefined TFs or designates them as irrelevant. When users interact with the system, their messages are sent to the chatbot API, where LangChain handles prompt construction, conversation memory and LLM orchestration to generate fluent responses from Llama 3. In parallel with answering the user, every incoming request is passed to the classifier, which tags the message with one or more of the five TFs.

Across the whole dialogue, the system collects and accumulates this classification information per user request. When the conversation between the user and the chatbot is finished, the set of classified requests is forwarded to the CoCo component. CoCo then uses these trust-function labels to compute the cLoTw for the user's requests along each of the five trust dimensions. In this way, the overall architecture cleanly separates concerns as follows. The Llama 3 (via LangChain) focuses on natural-language understanding and response generation, and the classifier focuses on trust-function labelling. The web application operates as a thin client on top of the chatbot API, reusing the same backend logic while enabling interactive use and systematic unit testing. In Figure 34, we illustrate the communication between Metaverse App-Chatbot-CoCo.

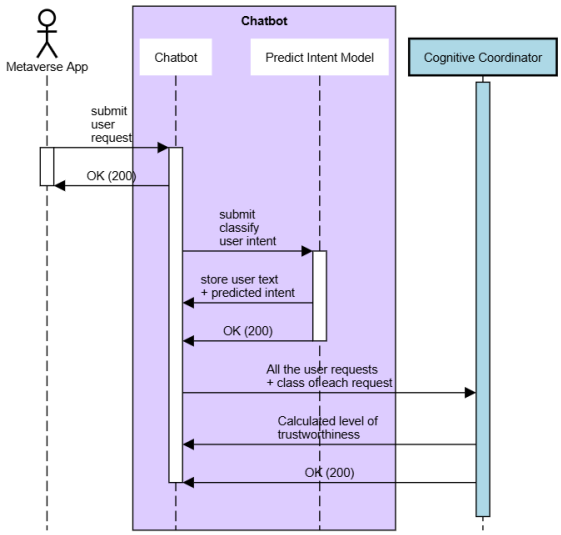


Figure 34 Chatbot Sequence Diagram

4.1.3 INTEGRATION WITH OTHER COMPONENTS

Component	Protocol (API/Kafka)	Action (HTTP REST/ Publish/Subscribe)	Details
CoCo	API	POST (submit)	Post the user request + predicted intent in order to receive the calculated cLoTw
CoCo	API	POST (calculate)	Trigger the trustworthiness computation once data has been submitted

CoCo	API	GET (reset)	Get request in order to clear the user requests from CoCo

Table 8 Chatbot's interaction with CoCo

Chatbot API <-> CoCo API

The chatbot sends a JavaScript Object Notation (JSON) payload via a POST request that includes, at minimum, a userId, a timestamp and the intents extracted by the TFs. It then issues a second POST request to the calculation endpoint, where CoCo computes the cLoTw and returns the result. CoCo also exposes a reset endpoint to clear any stored requests and predicted intents.

4.1.4 Unit Tests

testObjective:	To validate that the Chatbot API correctly interprets user inputs and generates relevant, coherent and context-aware responses across a variety of dialogue scenarios.
Components	a. Chatbot API
Requirements	1.Chatbot should generate relevant text to user request.
Features to be tested	1. Understanding of user request. 2. Relevance of generated response to the user query. 3. Handling of unclear/invalid/out-of-scope questions. 4. Context handling across multiple turns in a conversation. 5. Response time of the Chatbot API.
Test Steps	1.Test a list of sample user requests. 2. Send each user request to the Chatbot API and capture the response returned across with response time. 3. Verify that the response text is relevant to the request.

Verification Checklist for Chatbot-UT1:

Step	Description	Yes	No	Comments
1	The generated responses are relevant, accurate and free of inappropriate or unsafe content.	☒		
2	For all requests (including unclear/invalid ones), the chatbot handles them gracefully and returns a response within the target time.	☒		

Execution Output:

In Chatbot-UT1 (Chatbot API dialogue test), we evaluated both the performance and the quality of the Chatbot API's conversational behaviour by sending a set of representative user prompts, capturing the API outputs and validating them against the expected behaviour. As shown in the first execution output (Figure 35), response-time performance was measured directly from the API logs for repeated POST /api/chat calls, confirming that the service consistently returned successful responses (HTTP 200) with response latencies at the order of a few seconds.

```
==================================== test session starts =====================================
ollama-service|[GIN] 2025/12/18 - 09:30:03 |200| 3.121500203s | 172.18.0.3 | POST "/api/chat"
ollama-service|[GIN] 2025/12/18 - 09:30:26 |200| 3.204577364s | 172.18.0.3 | POST "/api/chat"
ollama-service|[GIN] 2025/12/18 - 09:31:03 |200| 3.863127733s | 172.18.0.3 | POST "/api/chat"
ollama-service|[GIN] 2025/12/18 - 09:31:29 |200| 2.846863921s | 172.18.0.3 | POST "/api/chat"
ollama-service|[GIN] 2025/12/18 - 09:31:44 |200| 3.037687859s | 172.18.0.3 | POST "/api/chat"
ollama-service|[GIN] 2025/12/18 - 09:34:15 |200| 3.798451575s | 172.18.0.3 | POST "/api/chat"
==================================== test session ends ======================================
```

Figure 35 Response times

In parallel, the second execution output (Figure 36) demonstrates the qualitative evaluation of response relevance and dialogue handling, **showing a sample of the user requests used in the evaluation**, including how the assistant reacts to a valid help request, an unclear/invalid input, and a more specific requirement.

```
ollama-service  | time=2026-01-20T08:29:14.166Z level=INFO source=routes.go:1426 msg="entering low vram mode" "total
vram"="5.9 GiB" threshold="20.0 GiB"
llama-chat      | INFO:     Started server process [1]
llama-chat      | INFO:     Waiting for application startup.
llama-chat      | INFO:     Application startup complete.
llama-chat      | INFO:     Uvicorn running on http://0.0.0.0:5050 (Press CTRL+C to quit)
(venv) ubuntu@ubuntu ollama-chatbot % python3 chatbot.py
chatbot         | INFO:     Using API base URL: http://localhost:5050
You: I need help
SAFE-6G assistant: What's going on? Do you have a specific problem or question you'd like help with? I'm here to
listen and assist if I can.
You: hdfgsrfsd
SAFE-6G assistant: It looks like you might be typing something that doesn't make sense. Would you like to try again,
or is there something else on your mind that you'd like to talk about? I'm here to help!
You: I want to ensure stable connection for the application I am building.
SAFE-6G assistant: That's a great goal! Having a stable connection can make a big difference in the overall user
experience of your application. What kind of application is it and which parts must always be available to users?
```

Figure 36 Prompts in Chatbot (UI)

Chatbot-UT2: Predict intent model testing

Objective	To validate that the Predict Intent model correctly classifies relevant user requests into the appropriate one of the five defined TFs.
Components	a. Predict intent model
Requirements	1.Relevant user request should be classified into the 5 TFs.
Features to be tested	1.Classification of user requests.
Test Steps	1.Test a list of sample user requests. 2.Send each user request to the Chatbot API and capture the classification results. 3. Verify that the classified function is the right one.

Verification Checklist for Chatbot-UT2:

Step	Description	Yes	No	Comments
1	The predict intent model predicted the right classes to the predefined user requests.	☒		

Execution Output:

In Chatbot-UT2 (Intent prediction validation), we evaluated the Predict Intent model by providing a predefined set of representative user requests and comparing the model's predicted class labels against the expected categories. Some example requests and their corresponding predicted intent

labels are shown in the Figure 37, where inputs related to topics such as Safety, Reliability, Privacy, Security and Resilience are correctly mapped to their respective predefined classes.


```
{
  "data": [
    {
      "label": "Safety",
      "text": "I would like to ensure the safety of my application."
    },
    {
      "label": "Reliability",
      "text": "I am developing an application for machinery control in a factory and I need
it to have stable internet connection."
    },
    {
      "label": "Privacy",
      "text": "It will handle sensitive data such as employees information."
    },
    {
      "label": "Resilience",
      "text": "How do you currently balance QoS distribution across various traffic types in
your network?"
    },
    {
      "label": "Security",
      "text": "Yes I use encryption for the data"
    }
  ]
}
```


Figure 37 Intent prediction validation (JSON)

4.1.5 Integration Test

Chatbot-IT1: Checking Chatbot API <-> CoCo connectivity

Objective	
Components	a. CoCo b. SAFE-6G chatbot
Requirements	Once the conversation with chatbot is finished, the user requests across with the predicted intent must be sent to the CoCo.
Features to be tested	Chatbot <-> CoCo
Test Steps	1. Deploy and run chatbot. 2. Submit some requests to chatbot. 3. When the conversation finishes, submit the data to CoCo. 4. Call Calculate endpoint for calculation of cLoTw and get the calculated cLoTw.

Verification Checklist for Chatbot-IT1:

Step	Description	Yes	No	Comments
1	Deploy and run the chatbot.	☒		Chatbot deployed and runs normally.
2	Submit some requests to the chatbot.	☒		Chatbot generates responses relevant to the user requests and saves the users requests and the predicted intent of each request.
3	Submit the conversation to CoCo.	☒		HTTP request code status is 200. Chatbot API sents to CoCo API the user requests and the predicted intent of each request.

4	Call Calculate endpoint for calculation of cLoTW.	☒		HTTP request code status is 200. The chatbot gets the calculated cLoTw from the CoCo API.

Execution Output:

The Chatbot-IT1 validates the communication between the chatbot and the CoCo by invoking the submit, calculate and reset endpoints. The Figure 38 shows the terminal execution of these API calls, each returning HTTP 200, confirming successful data submission, LoTw calculation and state reset.

```
curl -sS -X POST http://coco:8080/submit \
  -H "Content-Type: application/json" \
  -d @submit_payload.json \
  -w "\nHTTP %{http_code}\n"
Data sent successfully
HTTP 200

curl -sS -X POST http://coco:8080/calculate \
  -H "Content-Type: application/json" \
  -d @calculate_payload.json \
  -w "\nHTTP %{http_code}\n"
Calculation request sent successfully
HTTP 200

curl -sS -X GET http://coco:8080/reset \
-w "\nHTTP %{http_code}\n"
Reset complete
HTTP 200
```

Figure 38 CoCo Integration tests (terminal)

4.1.6 Next Steps

During the remaining development phase, we will test alternative LLMs to validate generation performance and overall suitability against our requirements. In parallel, we will update the API as needed to enable E2E support for the target use-case scenarios, including any required changes to request/response formats, validation and error handling. Additional testing will also be conducted to confirm functional stability, performance and integration readiness.

4.2 Cognitive Coordinator

4.2.1 Overview

The CoCo is the core intent-to-trust translation engine of the SAFE-6G framework. Its primary role is to transform natural-language user requests, received via the chatbot, into trust scores per TF, so as the TFs to perform calculations and based on them make actions in the different planes, aiming to meet the LoTw requested by the user. In order to achieve that, the CoCo leverages its internal components for trustworthiness predictions, calibration of scores, user information storage and resolution of any conflicts of actions should they emerge.

In parallel, as illustrated on diagram shown in Figure 39, in order to perform actions in the different planes, information needs to be spread inwards and outwards CoCo, thus four main integration points are being introduced. In the first, CoCo integrates with the Chatbot to get the user intents and responds back with the final outcome to inform the user about the achieved level of trustworthiness.

Second, on the user information side, CoCo is able to query Open CAPIF for additional user-related characteristics such as serviceID or active UEs which prove to be vital for the rest of the flow and are utilised by the TFs. Third, since the final outcome towards the user should be explained and understandable, CoCo establishes an API centered integration with the xAI module to gather explanations specific to the user's flow. Lastly but not least, CoCo communicates with the TFs via a kafka-based Message Broker in order to publish the predicted trustworthiness scores and align on feasible deployable trustworthiness.

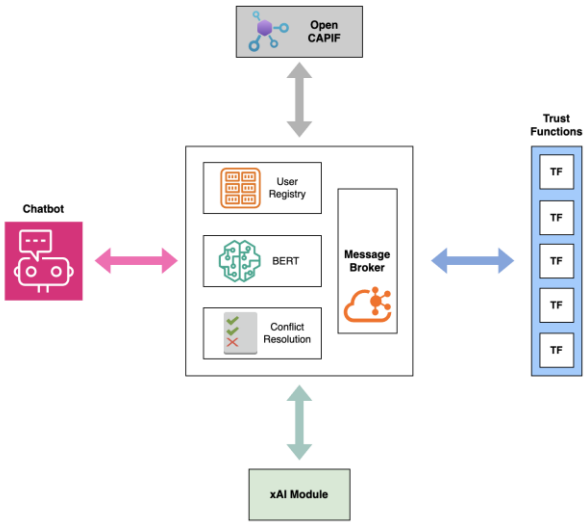


Figure 39 CoCo's four main integration points in the SAFE-6G framework.

4.2.2 ARCHITECTURE

In a deeper explanatory dive, CoCo is implemented as a single coordination pipeline that starts from user intents and ends in confirmations for deployable, trust-aware actions over the continuum. The user expresses its needs via the chatbot, which tokenizes the request, maps them to the relevant TFs (security, safety, resilience, reliability, privacy), and sends these intents as a JSON encoded payload to CoCo through the CoCo exposed API in the form of the example below:

```
{
  "data": [
    { "label": "Privacy", "text": "I need strong privacy over my data." },
    { "label": "Resilience", "text": "I could use some resilience over my connection as
well" }
  ]
}
```

Figure 40 A typical intents payload sent from chatbot to CoCo

At this point, it is worth mentioning that if the request regards a new user, CoCo will query the Open CAPIF to gather user-specific information such as serviceID, which is vital for the rest of the flow and the communication with the TFs. Upon retrieval, this information is stored in the user registry, a CoCo component dedicated to user information storage.

Following the received intents, CoCo runs a Bidirectional Encoder Representations from Transformers (BERT)-based with 5 regression heads inference step to predict trustworthiness scores per TF (which reflect the perceived importance of each trust dimension-privacy, security, safety, resilience, reliability- to the user) and based on how many intents fall under each dimension, derives weights that are used to normalize these scores into a normalized weighted trust vector, nLoTw, containing one score per TF (e.g. [76.1, 23.9, 0.0, 0.0, 0.0]). This vector is passed to the Reasoning Engine, whose conflict resolution logic enforces user-tier limits (privileged/non-privileged user) by capping the maximum allowed score per TF, in what is considered a first round of calibration. Worth noting here, is that a privileged or non-privileged user may reflect the tenant's rights inside an organization and thus whether he/she has accessibility to high levels of trustworthiness, which consequently require specific and sometimes resource intensive deployments. Those deployments dependent on trustworthiness scores are categorized in tiers called flavors (e.g. low, medium, high). So, CoCo's aim here is to identify the user's tier accessibility and cap the scores according to the respective system limits. Those capped scores are communicated to the TFs with dedicated topics via the kafka-based Message Broker. At this point each TF consumes its respective topic, evaluates the feasibility of the received nLoTw value based on its local AI agent, operational constraints, available resources and flavor compatibility, and reports back to CoCo with the maximum feasible deployable flavor (e.g. low, medium, high) for its TF. In this context, flavors reflect deployment actions (e.g high privacy could be translated into tweaking the service in a state that prevents too much logging).

The feedback from the TFs is consumed by the Message Broker (CoCo's kafka consumer) and once all proposed by the TFs actions are collected, the Reasoning Engine performs Conflict Resolution and final calibration, ensuring cross-TF compatibility (ensuring that no conflict occurs between the suggested actions) and producing the final deployable trust vector, cLoTw. This final cLoTw scores are then distributed to the TF AI agents, by utilizing the Message Broker once again, which consumes the dedicated topics, map the scores to the appropriate, finalized flavor and finally execute deployment actions over the continuum, the 5G core and the application. Finally, the overall outcome (success/failure, errors) is published back to CoCo and is passed to the xAI module, which generates human-understandable explanations and returns them to CoCo which provides them back to the chatbot via the CoCo API so the user can see not only what was decided, but also why. In Figure 41, we illustrate the communication the CoCo with the referred components.

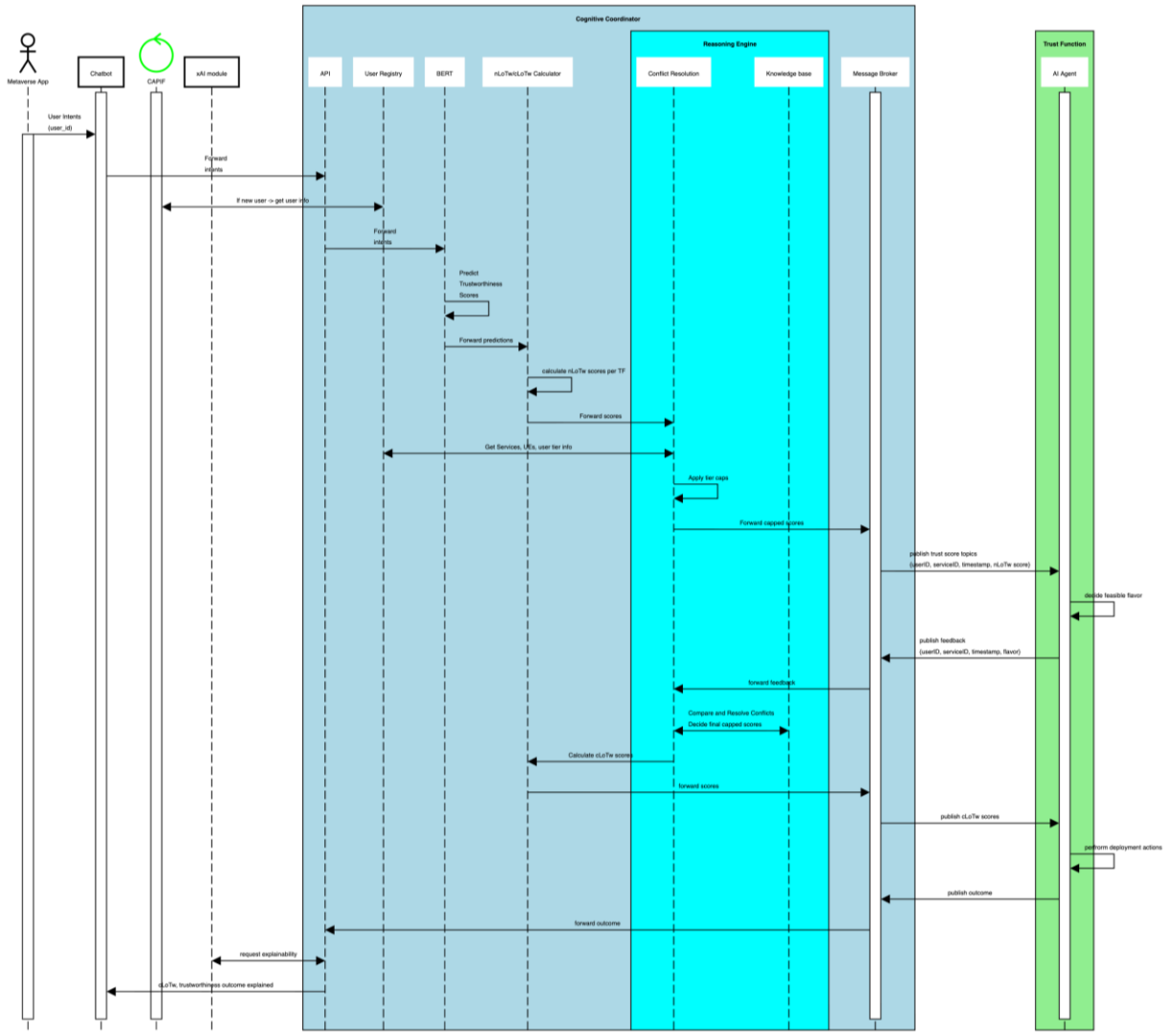


Figure 41 CoCo integrated with the rest of the components in the SAFE-6G framework.

4.2.3 Integration with other Components

The CoCo sits at the center of a minimal but well-defined integration loop:

Component	Protocol (API/Kafka)	Action (Read/Publish/Subscribe)	Details
Chatbot	API	POST	Receive intents/ Report Trustworthiness Outcome
6G Planes	Open CAPIF		Query user-related info
xAI	API	POST	User info and BERT's predictions
TFs	Kafka	Publish /Subscribe	JSON structured messages in dedicated topics

Table 9 CoCo communication with SAFE-6G components

- **Chatbot ↔ CoCo API**

CoCo exposes an API endpoint implemented with FastAPI framework that is invoked by the chatbot. The chatbot sends a JSON-formatted payload containing at least userID, timestamp and the extracted intents per TF by utilising the POST request called. CoCo processes this payload to generate nLoTw and proceeds with the rest of the flow. As soon as the final cLoTw scores have been produced and the corresponding deployment actions have been decided, the final outcome is reported back to the chatbot by utilising the same API.

- **CoCo ↔ Message Broker/TFs**
  CoCo integrates with the TFs via the Kafka Message Broker. Once nLoTw/cLoTw is computed, CoCo publishes per-TF messages to dedicated Kafka topics by utilising its kafka producer. Each TF subscribes to its topic, consumes the non-calibrated score, determines updated feasible score and sends feedback back to CoCo through predefined Kafka channels or endpoints. More details and integration specifics about the Message Broker are provided in its dedicated section.
- **CoCo ↔ xAI Module**
  CoCo will integrate with the xAI module by providing the final trust decisions (cLoTw), along with the trained regression model used for trust-score prediction. Although the exact interface, the time when this Deliverable is written, is still under definition, the xAI module will use this information to generate human-readable explanations that are passed back to CoCo and finally to the chatbot alongside the raw outcomes.
- **CoCo ↔ SAFE-6G Planes via Open CAPIF**
  CoCo also integrates with the three SAFE-6G planes through APIs exposed from Open CAPIF. Through these northbound APIs, CoCo can obtain network-and subscription-related user information (e.g. identifiers such as international mobile subscriber identity (IMSI), serviceID and other context attributes), which is stored in the user registry and is used for user-tier enforcement, contextual reasoning and alignment of trust decisions with the actual network state and capabilities.

4.2.4 Unit Tests

Unit tests for the CoCo validate the internal logic of the intent-to-trustworthiness translation pipeline in complete isolation from external systems (Kafka, CAPIF, TF agents, xAI). All external interfaces are replaced by mocks allowing fast and deterministic validation of CoCo's behavior on well-controlled inputs. These tests focus on correctness of trustworthiness-score prediction and normalization into nLoTw, user-tier enforcement and conflict-resolution rules that conclude to calculate cLoTw.

CoCo-UT1: TF Trust Prediction and nLoTw Creation

Objective	To verify correct trust-score prediction and nLoTw generation per TF
Internal Components	a. CoCo API b. BERT model c. nLoTw construction logic
Requirements	1. CoCo shall invoke the model once per request 2. BERT's classification head should assign intents per TFs 2. BERT's regression heads should assign one score per TF in [0,100]
Features to be tested	1. Model invocation and output mapping

	2. nLoTw vector population
Test Steps	1. Create input payload with intents per TF 2. Mock a post request towards CoCo 3. Inspect the received intents and the calculated nLoTw

Verification Checklist for CoCo-UT1:

Step	Description	Yes	No	Comments
1	Regression head called exactly once per TF	☒	☐	
2	nLoTw scores are the expected ones per TF	☒	☐	
3	nLoTw for not present TFs is 0	☒	☐	
4	All nLoTw scores are within [0, 100]	☒	☐	

Execution Output:

```
========================================= test session starts ==========================================
coco-api|2026-01-12 10:22:05-coco.app.services.lotw_service - INFO - Calculating nLr 2 items...
coco-api|2026-01-12 10:22:05 - coco.app.services.lotw_service - INFO - Grouped data by label:
[(<TrustFunction.PRIVACY: 'Privacy'>, 1), (<TrustFunction.RESILIENCE: 'Resilience'>, 1)]
coco-api|2026-01-12 10:22:07 - coco.app.services.lotw_service - INFO - Predicted score for class
TrustFunction.PRIVACY: 60.27 (text: I need strong privacy over my data....)
coco-api|2026-01-12 10:22:07 - coco.app.services.lotw_service - INFO - Predicted score for class
TrustFunction.RESILIENCE: 65.91 (text: I could use some resilience over my connection as ...)
coco-api|2026-01-12 10:22:07 - coco.app.services.lotw_service - INFO - nLoTw result: {<TrustFunction.PRIVACY:
'Privacy'>: 47.76891189301457, <TrustFunction.RESILIENCE: 'Resilience'>: 52.23108810698543}
========================================== test session ends ===========================================
```

Figure 42 CoCo-UT1 Output

CoCo-UT2: Multi-TF Weighting and Normalization

Objective	To verify that CoCo correctly derives weights from the number of intents per TF and normalizes scores.
Internal Components	a. Intent counting per TF b. Weight computation c. Aggregation into nLoTw
Requirements	1. Weights shall be proportional to the number of intents and calculated according to the specified mathematical formula 2. nLoTw shall be correctly normalized according to its mathematical formula
Features to be tested	1. Weight calculation 2. Normalization and edge cases (e.g. zero intents)
Test Steps	1. Construct payload: 3 security intents, 1 privacy intent, 0 for others 2. Mock a post request towards CoCo 3. Inspect derived weights and nLoTw

Verification Checklist for CoCo-UT2:

Step	Description	Yes	No	Comments
1	Weights for received TFs intents are as expected	☒	☐	
2	TFs with zero intents do not cause division by zero	☒	☐	
3	nLoTw values follow expected aggregation formula	☒	☐	
4	Normalization constraint is satisfied	☒	☐	

Execution Output:

```
======================================== test session starts ========================================
coco-api|2026-01-12 10:22:07 - coco.app.services.lotw_service - INFO - Computing W_TF values...
coco-api|2026-01-12 10:22:07 - coco.app.services.lotw_service - INFO - W_TF values:
{<TrustFunction.PRIVACY: 'Privacy'>: 0.5, <TrustFunction.RESILIENCE: 'Resilience'>: 0.5}
========================================= test session ends =========================================
```

Figure 43 CoCo-UT2 Output

4.2.5 Integration Tests

On the other hand, integration tests validate CoCo's interaction with other SAFE-6G components along the full intent-to-deployment loop, while still mocking the non-required external infrastructure per test. Tests include HTTP request for CoCo's FastAPI to verify integration with the chatbot and CoCo's integration while the integration with the xAI module and the 6G planes via Open CAPIF, has not been established yet and the related tests will be provided in future deliverables.

CoCo-IT1: Chatbot to CoCo via API

Objective	To validate that CoCo receives the intents and the trustworthiness process is triggered
SAFE-6G Components	a. CoCo REST API b. Intent processing and model invocation
Requirements	1. API shall accept chatbot payloads and CoCo should parse them correctly and trigger the nLoTw process
Features to be tested	1. API behavior and schema 2. Flow is triggered upon receiving intents
Test Steps	1. Call CoCo API with intent payload 2. Inspect the received intents, predictions, weights and nLoTw scores

Verification Checklist for CoCo-IT1:

Step	Description	Yes	No	Comments
1	HTTP response status is 200	☒	☐	
2	Intents are parsed correctly	☒	☐	
3	nLoTw calculation is triggered	☒	☐	

Execution Output:

```
======================================== test session starts ========================================
{"message": "Data received successfully", "count": 2}
========================================= test session ended ========================================
```

Figure 44 CoCo-IT1 output

4.2.6 Next Steps

The next steps will focus mostly on CoCo-xAI integration aspects to support a production-ready explanation layer within CoCo. This includes the selection of the best fitting xAI methods for CoCo's model (BERT + regression heads), interface definition, standardizing the to-be-exchanged payloads and ensuring on both unit and integration level, that cLoTw, model metadata, TF feedback, and reasoning traces are systematically provided to the xAI module. From this point on, CoCo will need to consume and parse the explainable AI outcome correctly, thus more unit and integration tests will be created and performed towards this direction. In parallel, integration with OpenCAPIF is in progress and the related unit and integration tests will be implemented and performed.

Additionally, to align with the existing unit and integration tests, as more scenarios emerge, the next phase will introduce a broader and deeper test suite that captures more edge cases, additional reasoning scenarios, and extended validation of the CoCo pipeline. Such scenarios include handling of malformed payloads and verification of produced scores and weights. To conclude, the next steps are as follows:

- Focus on CoCo-xAI module integration. Establish the interface, implement and perform both unit and integration tests to verify that the integration and the exchanged payloads are concrete (4 tests).
- Implementation and execution of both unit and integration tests between CoCo and Open CAPIF. Focus on the requests/responses and the payloads (4 tests).
- Provide more tests that capture and evaluate performance in corner cases.

4.3 Message Broker

4.3.1 Overview

The Message Broker is the event-distribution backbone of the SAFE-6G system. Its primary role is to publish trustworthiness-score updates generated by the CoCo and deliver them to the appropriate TFs via dedicated topics with structured messages, as well as receive feedback from the TFs, parse them and hand them over to CoCo's Reasoning Engine for further utilization. In essence, the Message Broker is the component within CoCo that ensures communication between CoCo and the TFs in order to achieve a horizontal scalability across the system and transmit the requested level of trustworthiness.

4.3.2 Architecture

The Message Broker architecture, as an internal part of CoCo, is built around a lightweight Kafka deployment backed by a Zookeeper instance for metadata and cluster coordination and is composed of a producer and a consumer component. In a more explanatory manner, as soon as the CoCo produces nLoTw scores, it triggers a Kafka producer responsible for encoding trustworthiness-score messages in JSON format and publishing them to function-specific topics (e.g., privacy_trust_score, reliability_trust_score). Each message contains a structured payload including user identifiers and trustworthiness scores. Worth noting is that this Kafka producer is also responsible for: a) publishing valid and not incompatible or erroneous trustworthiness scores and b) ensuring the correct message structure. As soon as TFs consume their dedicated topics, they utilize their AI agents to decide feasible deployment flavors related to the received scores. Those flavors along with other related info, such as timestamp and userID are published back to CoCo's Message Broker as feedback topics.

On the Kafka consumer side of Message Broker, the time each TF publishes its own feedback towards the CoCo, the task is to consume those topics independently, one by one, until all five are consumed and feed their content back to the Conflict Resolution Manager for the final calibration to take place. After the Conflict Resolution Manager, all cross-TF conflicts are resolved (e.g. high privacy and high security could potentially mean conflicting deployments) and a cap is produced for each one of the TFs trustworthiness scores. Those capped scores are provided to the LoTw calculator and are utilized by the cLoTw formula to produce the final cLoTw scores per TF.

As soon as the calibration procedure is complete, a new topic distribution takes place. The Message Broker publishes the final calibrated scores towards the TFs, scores are mapped to deployment flavors and upon deployment completion, the outcome is pushed back to CoCo thanks to the Kafka consumer being utilized once again.

In essence, Message Broker handles the back-and-forth communication between CoCo and the TFs until the trustworthiness scores are calibrated into feasible, deployable scores, as seen in Figure 45.

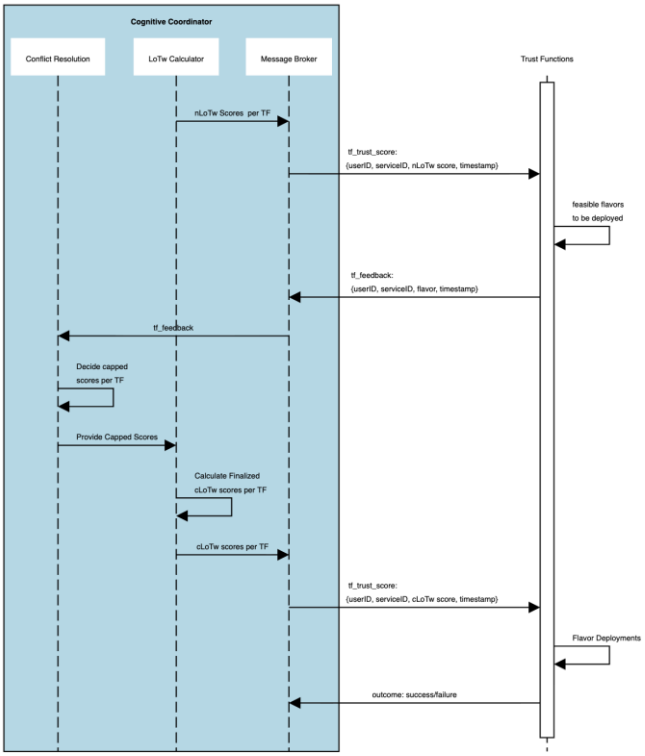


Figure 45 Message Broker operation

4.3.3 Integration with other Components

Component	Protocol (API/Kafka)	Action (Read/Publish/Subscribe)	Details
TFs	Kafka	Publish	JSON encoded message containing scores and user/system related info
TFs	Kafka	Subscribe	JSON encoded message containing the TFs' feedback on proposed actions

Table 10 Message Broker's interaction with the Trust Functions

The Trust Score Calculation component of CoCo is integrated with the TFs through this Kafka-based messaging workflow. Once trust scores are computed, they are distributed to the appropriate TFs, which consume these messages and trigger their respective AI agents. The resulting outcomes (updated, feasible trustworthiness scores/ respective actions) are then fed back to CoCo via dedicated feedback channels, enabling continuous state updates and transparent reporting to the chatbot.

- CoCo's Trust Score Calculation Component calls the publish_trust_scores() function, which serializes trust scores and publishes them to Kafka using the configured broker address (bootstrap servers). Messages are keyed by TF and optionally by userID to support ordered processing.
- TFs subscribe to their respective trust-function topics. Upon message reception, each TF parses the payload and initiates its decision logic via its AI agent, typically involving calls to orchestration services to apply or update feasible vApp/nApp flavors.
- After executing the required action, each TF reports the outcome back to CoCo through a dedicated feedback channel—another Kafka topic. This response includes proposed and feasible trustworthiness flavors/actions, success/failure statuses, error details (if any), and contextual metadata (timestamp, tenant, applied flavor). CoCo's Kafka consumer subscribes and uses this feedback to trigger the Reasoning Engine, resolve any conflicts and proceed with the final deployments.

Publisher	Consumer	Topic Name	Message Schema
Message Broker	TFs	<TF_name>_trust_topic	{"service_id", "user_id", "timestamp", "nlotw"}
TFs	Message Broker	<TF_name>_report	{"service_id", "user_id", "timestamp", "flavor"}

Table 11 Template for Kafka's topic

4.3.4 Unit Tests

Unit tests validate the behavior of the message broker in complete isolation from external systems. By mocking the Kafka producer, these tests verify that the correct topics, payloads and invocation patterns are used when publishing trust scores. They ensure that edge cases, such as malformed inputs or invalid structures, are handled gracefully, without triggering unintended Kafka operations. Unit tests therefore provide fast, deterministic validation of the broker's internal logic without requiring Kafka to be running.

MB-UT1: Publish Valid Trustworthiness Scores

Objective	To verify that the broker publishes correctly structured trustworthiness-score messages to the correct Kafka topics.
Components	a. Kafka producer wrapper b. Topic selection logic c. Trust-score serialization (JSON encoder)
Requirements	1. Trust scores shall be published to TF-specific topics. 2. Messages shall be JSON-encoded with mandatory fields (userID, serviceID, TF_name, score, timestamp)
Features to be tested	1. Topic name resolution per TF. 2. JSON payload structure.
Test Steps	1. Instantiate Message Broker with a mock Kafka producer 2. Call publish_trust_scores() with a valid score mapped to TFs 3. Capture calls on the mock producer(topic name, message) 4. Inspect captured records for topics and payload fields

Verification Checklist for MB-UT1:

Step	Description	Yes	No	Comments
1	TF scores published to <TF name>_trust_topic per TF	☒	☐	
2	Each message payload contains userID, serviceID, timestamp, TF name, score	☒	☐	
3	JSON payloads are syntactically valid	☒	☐	

Execution Output:

```
======================================== test session starts ========================================
coco-api|2026-01-12 10:22:07 - coco.app.services.kafka_producer - INFO - Published to topic
'privacy_trust_score': {"nlotw": {"Privacy": 47.76891189301457, "Resilience": 52.23108810698543}}
coco-api|2026-01-12 10:22:07 - coco.app.services.kafka_producer - INFO - Published to topic
'resilience_trust_score': {"nlotw": {"Privacy": 47.76891189301457, "Resilience": 52.23108810698543}}
========================================= MB UT1 ended ==========================================
```

Figure 46 MB-IT1 Output

4.3.5 Integration Tests

Integration tests validate the interaction between the Message Broker and the TFs. By selectively mocking only external systems such as Kafka while leaving internal logic intact, integration tests ensure that data flows correctly across components.

MB-IT1: Message Broker → TFs

Objective	To validate that Message Broker successfully publishes topics towards the TFs
Components	a. CoCo REST API b. Trust-score calculation service c. Message Broker
Requirements	1. API requests resulting in valid nLoTw shall trigger broker publication 2. Exactly one message per TF is produced 3. All topics are successfully published
Features to be tested	E2E path from CoCo API call to score calculation to Kafka publish call and topic consumption
Test Steps	1. Run CoCo API in test mode 2. Send a mocked but valid chatbot request (JSON with intents for all TFs) 3. After response, inspect producer calls

Verification Checklist for MB-IT1:

Step	Description	Yes	No	Comments
1	HTTP response status is 200	☒	☐	
2	Broker publish_trust_scores() called exactly once	☒	☐	
3	One produced record per TF topic	☒	☐	

Execution Output:

```
======================================== test session starts ========================================
coco-api|2026-01-12 10:22:07 - coco.app.services.kafka_producer - INFO - Published to topic
'privacy_trust_score': {"nlotw": {"Privacy": 47.76891189301457, "Resilience": 52.23108810698543}}
coco-api|2026-01-12 10:22:07 - coco.app.services.kafka_producer - INFO - Published to topic
'resilience_trust_score': {"nlotw": {"Privacy": 47.76891189301457, "Resilience": 52.23108810698543}}
========================================= test session ended ========================================
```

Figure 47 MB-IT1 Output

MB-IT2: TFs → Message Broker

Objective	To validate that Message Broker successfully consumes feedback topics from the TFs
Components	a. Mocked TF feedback b. Message Broker
Requirements	1. Mocked feedback topic from a TF 2. Topic payload 3. Message Broker consumes the topics
Features to be tested	Kafka consumer of Message Broker is able to consume and parse the topics from the TFs
Test Steps	1. Establish a session with kafka consumer 2. Mock a feedback topic 3. Inspect consumer logs

Verification Checklist for MB-IT2:

Step	Description	Yes	No	Comments
1	Session is initiated	☒	☐	
2	Topic is received	☒	☐	
3	Topic's message is consumed	☒	☐	
4	Interaction is logged	☒	☐	

Execution Output:

```
======================================== test session starts ========================================
kafka-1|[2026-01-12 13:18:07,641] INFO Created log for partition privacy_report-0 in
/var/lib/kafka/data/privacy_report-0 with properties {} (kafka.log.LogManager)
% docker exec -it updated-cognitive-coordinator-kafka-1 kafka-console-producer \
--bootstrap-server kafka:9092 \
--topic privacy_report \
--property parse.key=true \
--property key.separator=":"
>user-1:{"serviceID":"a1b2c3d4","userID":e5f6g7h8,"timestamp":"2026-01-12T10:30:00Z", "flavor": "medium"}
========================================= test session ended ========================================
```

Figure 48 MB-IT2 output

4.3.6 Next Steps

The next phase of development for the Message Broker will focus on expanding and strengthening its validation within the intent-to-trustworthiness pipeline, ensuring reliable, deterministic and scalable communication between CoCo and the TFs. Additional tests will be introduced to cover multi-topic publication behavior, TF-specific topic isolation, message ordering guarantees and fault scenarios, such as delayed TF responses, partial feedback, or broker-level congestion due to multitenancy.

4.4 Differential Privacy

4.4.1 Overview

This section describes the Differential Privacy component within the SAFE-6G MLOps framework. In distributed ML workflows, maintaining data privacy is necessary for some applications. This is especially the case for scenarios where datasets contain sensitive information, for example, when this data originates from the Edge-Cloud continuum or Metaverse pilots.

Differential Privacy is a mathematical privacy-preserving technology designed to protect individual data points while still allowing for ML models to be trained and for data analytics utility to be achieved. To implement this within SAFE-6G, the Differential Privacy component mathematically injects noise into data before it is used by ML algorithms. This ensures that sensitive user-related information cannot be leaked from models used. The module uses standard Python libraries to implement privacy-preserving versions of classifiers, specifically, Gaussian Naïve Bayes, Logistic Regression, Support Vector Machines (SVM) and Random Forest Classifiers.

4.4.2 Architecture

The architecture of the Differential Privacy component is developed using a modular Python-based engine which follows the workflow of normalization, noise injection and model optimization, as described below:

- Normalisation and Noise Injection Layer: This mechanism uses the Laplace distribution centered at 0 with a scale parameter b = Δf/ε, where Δf is the sensitivity of the function (set to 1.0) and ε (epsilon) is the privacy budget. The component dynamically adjusts the ε, looping over a range of ε values - from 2.0 down to 0.2 – with a 0.2 decrement each time. This is done, for us to better analyse and assess the trade-off between privacy preservation, model utility, the accuracy of models, and the best model and ε-value to consider.
- Model Optimisation: The component includes an automated hyperparameter tuning mechanism (using GridSearchCV), which optimises classifiers specifically for noise-injected data. This is needed, to minimize the degradation in accuracy associated with privacy preservation (as more noise is added to data).
- Model Serialization: To support the MLOps lifecycle, the component automatically serialises the best-performing models into Python pickle (.pkl) files and logs performance metrics (F1 Score, Accuracy). This can be used for other SAFE-6G components to asynchronously load and utilize privacy-preserving models.

A sequence diagram of the above process, and how this can fit into the SAFE-6G architecture is presented in the Figure 49.

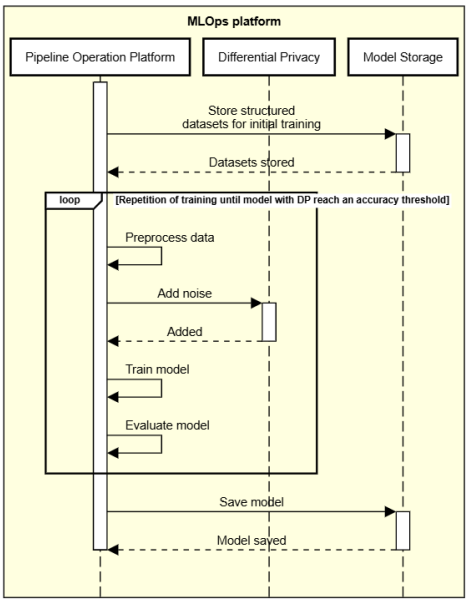


Figure 49 Pipeline operation with Differential Privacy

4.4.3 Integration with other Components

The Differential Privacy component can be integrated within the SAFE-6G architecture, and act as a model factory. It can expose interfaces, which can be used to generate "safe" versions of ML models and deployed to protect the privacy of sensitive data. An example workflow is described below:

1. Request: A SAFE-6G component requests a model for a specific dataset with a defined privacy budget (for example ε = 1.0).
2. Processing: The Differential Privacy component normalizes the data, injects Laplace noise and optimizes models upon this dataset, serializing the best performing model.
3. Delivery: The serialized model is saved in a shared model repository. Components can then load this model, and perform inference on new data (such as live traffic). In this way, we ensure that training data remain private.

The following interface can be exposed for integration:

- prepare_training_data: Will require a CSV dataset, scale and add noise to the data.
- fit_model: Will train and optimise models on the dataset, and serialise the best performing model given performance metrics.

In Figure 50, the following sequence diagram can be used to describe the integration flow:

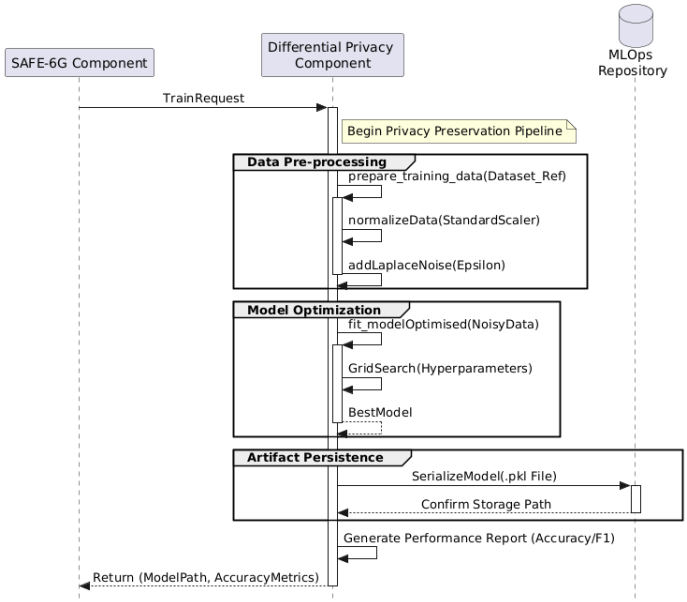


Figure 50 Differential Privacy Model Integration Flow

Given this workflow, we will be exploring the components with which Differential Privacy can be incorporated. The Reliability and Security TFs, and the Cognitive Coordinator are possible components, as they deal with sensitive network telemetry. Work will also be carried out to explore the use of Differential Privacy with the Chatbot component – if this is something that can be of benefit.

4.4.4 Unit Tests

Unit testing for the Differential Privacy component should focus on verifying the correctness of the model generation pipeline. This is required to ensure that privacy guarantees are met and that the created models can be used in the broader SAFE-6G architecture.

We define the following **Model Integrity Unit Test** Definition.

Differential Privacy-UT1: Model Integrity Unit Test

Objective	To ensure that the Differential Privacy component produces valid, readable model artifacts after training on noisy data.
Components	Model optimization engine, serialisation process.
Requirements	The Differential Privacy component must successfully generate and save a model file that can be loaded and used for inference, within another environment/component.
Features to be tested	Hyperparameter tuning, file serialization (serialisation to .pkl file).
Test Steps	1. Initiate the training pipeline with a standard configuration. 2. Verify that a .pkl file is created in the expected output directory. 3. Attempt to load the .pkl file using a standard library. 4. Pass input data (dummy data or another dataset entry) into the loaded model. 5. **Pass Condition**: The model returns a valid prediction (not an error) for the input provided.

Verification Checklist for Differential Privacy-UT1:

Step	Description	Yes	No	Comments
1	**Artifact Generation**: Pipeline successfully creates a .pkl file for each optimized model type.	☒	☐	Verified in local output directory.
2	**Model Functionality**: Loaded model accepts input data and returns a valid prediction or score.	☒	☐	Confirms model is active and not a corrupted object.
3	**Privacy Compliance**: Output data variance confirms that Laplace noise was successfully applied.	☒	☐	Mathematical verification of DP guarantees.

The above verifications can be carried out by inspection.

4.4.5 Integration Test

The integration testing strategy for the Differential Privacy component should validate the seamless flow of data and models between the Differential Privacy engine and the wider SAFE-6G MLOps infrastructure. The tests focus on the "Model Factory" workflow, to validate that the component can ingest data, process it under privacy constraints, and produce and deliver a usable model for other SAFE-6G components.

We define the following integration test.

Differential Privacy-IT1

Objective	To validate the E2E workflow of a privacy-preserving model, from use of a dataset to model availability in the MLOps repository.
Components	DP component, model storage, data layer.
Requirements	The process should autonomously transition through the data preparation, noise injection, model optimisation phase and model serialization phase.
Features to be tested	Cross-component dataset transmission and use, automated model optimisation, model storage persistence.
Test Steps	1. Provide the Differential Privacy component with a SAFE-6G dataset. 2. Monitor the workflow as it executes normalization, noise addition and hyperparameter model optimization. 3. Verify the successful creation and export of a serialised model. 4. Acceptance Criteria: The test is successful if other components can locate, load and execute a prediction using the generated model.

Verification Checklist for Differential Privacy-IT1:

Step	Description	Yes	No	Comments
1	**Data pipeline execution**: Successful dataset ingestion, normalization and Laplace noise injection.	☒	☐	
2	**Autonomous phase transition**: Automated transition from data preparation to model optimization.	☒	☐	
3	**Storage persistence**: Serialisation of the optimized model to the repository which is verified.	☒	☐	
4	**Cross-component loading**: Model can be located and loaded by other components.	☒	☐	

5	**Inference verification**: Loaded model generates valid predictions when used.	☒	☐	

The above verifications can be carried out by inspection.

4.4.6 Next Steps

While the main Differential Privacy component functionality has been implemented and tested, the next steps are for the actual integration process to occur, and for the described integration tests to be validated through comprehensive real-word test scenarios.

4.5 Core openness and programmability

4.5.1 Overview

Openness and programmability constitute core design principles of the SAFE-6G architecture, enabling flexible integration, orchestration, and control across heterogeneous infrastructure, network, and application domains. Building upon the architectural foundations defined in Deliverable D3.1, SAFE-6G adopts a unified API-driven approach that exposes capabilities from three distinct yet tightly coupled planes: the 6G core network plane, the edge–cloud continuum plane, and the application plane. This approach prioritizes open standards, open-source components, and programmable service interfaces to ensure extensibility, interoperability, and long-term sustainability. This approach prioritizes open standards, open-source components, and programmable service interfaces to ensure extensibility, interoperability.

At the **6G Core network plane**, SAFE-6G leverages both open-source and commercial 5G/6G core implementations (Cumucore Core, Open5GS), combining the transparency and extensibility of community-driven platforms with the advanced capabilities of commercial ones. These platforms expose programmable network capabilities through standardized and extended APIs (e.g., CNC and NEF-related interfaces), as well as standardized service-based interfaces and OpenAPI-described REST services. This enables programmability, policy enforcement, real-time network state monitoring, and seamless integration with higher-level orchestration and trust management components. Open service exposure is further reinforced through OpenCAPIF, enabling controlled, secure, and standardized access to network capabilities by external applications and trust orchestration modules.

At the **edge–cloud continuum plane**, SAFE-6G relies on the aerOS Meta-OS to provide abstraction, orchestration, and monitoring of distributed compute and network resources across federated domains. The continuum APIs enables service lifecycle management, context sharing, and infrastructure observability across edge and cloud environments, supporting dynamic placement and adaptation of all the SAFE-6G components.

At the **application plane**, SAFE-6G integrates APIs exposed directly by the Metaverse applications defined in the project use-cases. These APIs enable SAFE-6G components to interact with the Metaverse applications for session establishment, control of collaborative XR environments and exchange of contextual and application-level information. Through these interfaces, the SAFE-6G framework supports user-centric configuration, dynamic adjustment of trustworthiness levels, and

coordination with the underlying network and edge–cloud continuum, as required by the Metaverse use-cases and the chatbot virtual assistant.

To ensure coherence, security, and interoperability across all three planes, SAFE-6G adopts the **3GPP CAPIF** as a unifying exposure and consumption layer. All APIs from the core, continuum, and application planes are registered, published, and accessed through OpenCAPIF [46] , providing a trusted and standardized mechanism for API discovery, authentication, authorization, and monitoring. This design enables higher-level components, such as the CoCo and the TFs, to interact with the underlying system in a consistent and controlled manner.

4.5.2 ARCHITECTURE

The core openness and programmability section of the project consists of 3GPP native APIs, OpenCAPIF implementation for the CAPIF and a Metaverse Manager. Some of the 3GPP native APIs that are supported from the architecture are the aerOS APIs, the NEF Monitoring Event API as well as the AsSessionWithQoS and the Analytics Exposure APIs among with the IMM's APIs for the Metaverse Manager. All the APIs have a common exposure to external applications through OpenCAPIF, which provides authentication and authorization mechanisms between external applications and provider applications (3GPP APIs, Metaverse Manager API).

In Figure 51, a sequence diagram illustrates the onboarding process from a provider or invoker application into the CAPIF. The Aeros APIs, NEF Monitoring Event and AsSessionWithQoS as well as the Analytics Exposure APIs and Metaverse Manager APIs acting as provider applications.

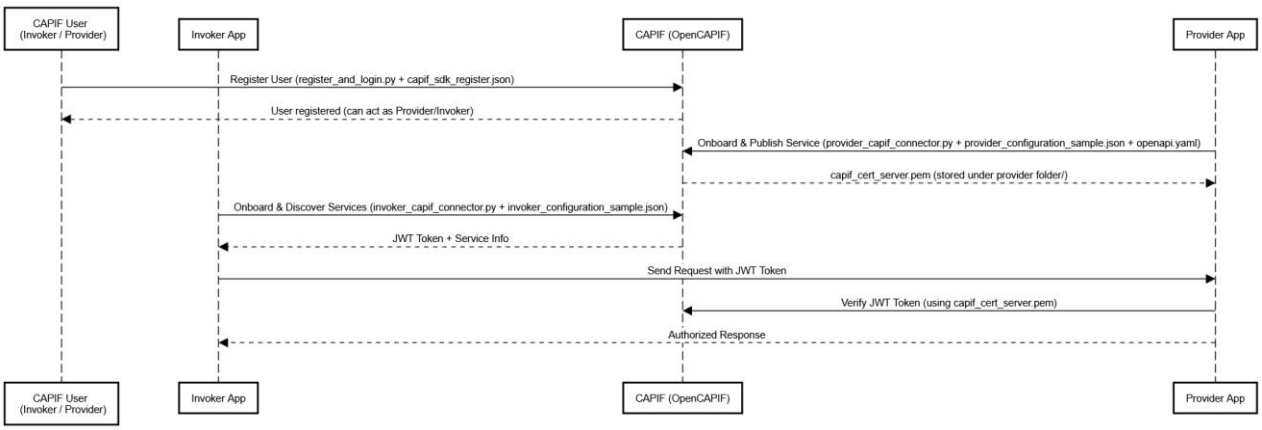


Figure 51 CAPIF registration, onboarding and authorization flow

The procedure is as follows:

1. User registration is performed with the openCAPIF so as to have a CAPIF user that can act either as a Provider or Invoker.
2. After the user registration into CAPIF, the onboarding and publishing of the provider application takes place. In that step, CAPIF sends back to the provider application the capif_cert_server.pem after publishing is completed while provider app should use it to authorize the incoming requests from external applications.

3. As a next step, the onboarding of the invoker should be done alongside the discovery of the published services through CAPIF. By finishing the discovery of the published APIs from the CAPIF, a JSON Web Token (JWT) token is sent back to the invoker to use it on future requests to be authorized from provider applications like any 3GPP native API or Metaverse Manager.
4. Lastly, an invoker (external application) performs a HTTP request with the JWT token that retrieved earlier integrated in its HTTP Request Headers to be authorized successfully from provider

4.5.3 Integration with other Components

For better illustration of the CAPIF workflow with the APIs exposed through CAPIF, the following example that describes the exposure of the IMM Metaverse API is considered. The principles of the CAPIF workflow are then applied to any of the APIs that are integrated with the openCAPIF.

To that extent, the first step of the CAPIF workflow is to create a new CAPIF user that can act as an invoker or provider in the scope of CAPIF. After the creation of such a user, the onboarding process into the CAPIF is performed with respect to that CAPIF user.

User Creation and Registration into CAPIF

With regards to the Figure 52, the outcome of the execution script register_and_login.py with configured values in capif_sdk_register.json can be verified by the following document in CAPIF's MongoDB register database as a JSON Format. As can be seen, a username as safe-6g-imm has been created for that user. This username is configurable in the capif_sdk_register.json file.


```
{
    _id: ObjectId('692872afe544c92cef0eccd5'),
    username: 'safe-6g-imm',
    password: '$2b$12$H/ZWdpDpaoWSFCaczlBYE.3VbCnrFjxibFCGBxdQggksOArEz1kk2',
    description: 'description',
    email: 'csr_email_address@dimokritos.gr',
    enterprise: 'csr_organization',
    country: 'csr_locality',
    purpose: 'test SDK user',
    phone_number: null,
    company_web: null,
    uuid: 'e6002375-025c-591f-b448-d845813d528f',
    onboarding_date: ISODate('2025-11-27T15:47:58.840Z')
}
```

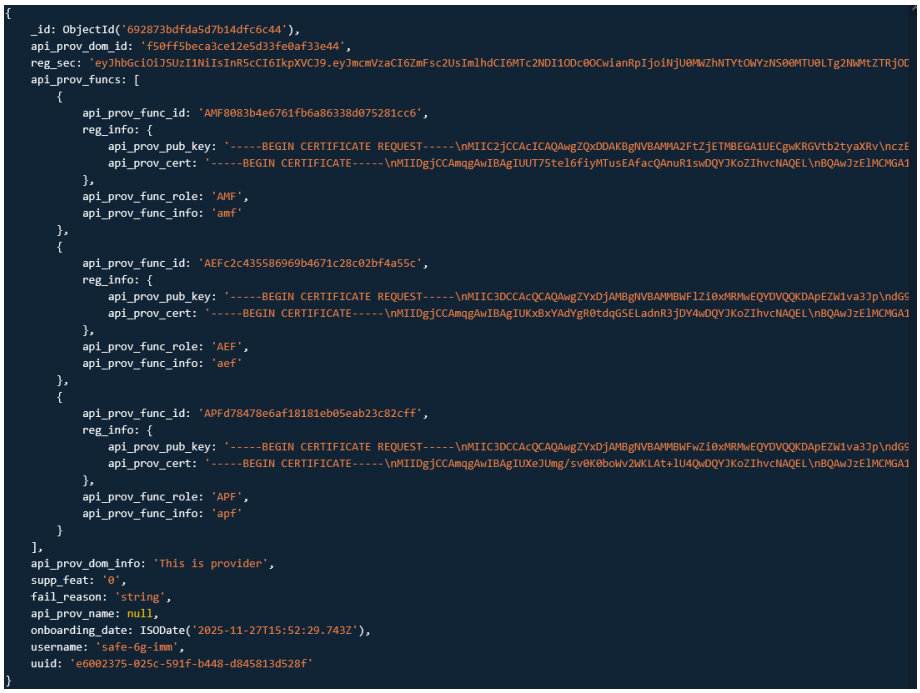

Figure 52 CAPIF User Registration in CAPIF's MongoDB Register Database

Provider onboarding and publishing into CAPIF

After a user has been created, it is time for onboarding the provider and publishing its API into CAPIF to be able to be discoverable from invoker applications that will request it later as well as to be able to authorize the incoming requests with regards to capif_cert_server.pem certificate that will receive from CAPIF after the publishing process has finished.

For this step, the openAPI schema yaml of the provider API is required alongside with the provider_config_sample.json which includes configuration options mainly for connection to the CAPIF, certificate generation, provider folder option for storing created certificates. The outcome of the execution script with the aforementioned configuration files can be verified by the following document in CAPIF's MongoDB database as a JSON Format. As it can be seen in Figure 53, the

onboarding and publishing process has been created the required certificates that is used for a provider to communicate with the CAPIF for information retrieval like generation of the capif_cert_server.pem, that will use later for authorization of the incoming requests.



```
{
    _id: ObjectId('692873bdfda5d7b14dfc6c44'),
    api_prov_dom_id: 'f50ff5beca3ce12e5d33fe0af33e44',
    reg_sec: 'eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTc2NDI1ODc0OCwianRpIjoiNjU0MWZhNTYtOWYzNS00MTU0LTg2NWMtZTRjOC
    api_prov_funcs: [
        {
            api_prov_func_id: 'AMF8083b4e6761fb6a86338d075281cc6',
            reg_info: {
                api_prov_pub_key: '-----BEGIN CERTIFICATE REQUEST-----\nMIIC2jCCAcICAQAwgZQxDDAKBgNVBAMMA2FtZjETMBEGA1UECgwKRGVtb2tyaXRv\nczE
                api_prov_cert: '-----BEGIN CERTIFICATE-----\nMIIDgjCCAmqgAwIBAgIUUT75tel6fiyMTusEAfacQAnuR1swDQYJKoZIhvcNAQEL\nBQAwJzElMCMGA1
            },
            api_prov_func_role: 'AMF',
            api_prov_func_info: 'amf'
        },
        {
            api_prov_func_id: 'AEFc2c435586969b4671c28c02bf4a55c',
            reg_info: {
                api_prov_pub_key: '-----BEGIN CERTIFICATE REQUEST-----\nMIIC3DCCAcQCAQAwgZYxDjAMBgNVBAMMBWFlZi0xMRMwEQYDVQQKDApEZW1va3Jp\ndGS
                api_prov_cert: '-----BEGIN CERTIFICATE-----\nMIIDgjCCAmqgAwIBAgIUKxBxYAdYgR0tdqGSELadnR3jDY4wDQYJKoZIhvcNAQEL\nBQAwJzElMCMGA1
            },
            api_prov_func_role: 'AEF',
            api_prov_func_info: 'aef'
        },
        {
            api_prov_func_id: 'APFd78478e6af18181eb05eab23c82cff',
            reg_info: {
                api_prov_pub_key: '-----BEGIN CERTIFICATE REQUEST-----\nMIIC3DCCAcQCAQAwgZYxDjAMBgNVBAMMBWFwZi0xMRMwEQYDVQQKDApEZW1va3Jp\ndGS
                api_prov_cert: '-----BEGIN CERTIFICATE-----\nMIIDgjCCAmqgAwIBAgIUXeJUmg/sv0K0boWv2WKLAt+lU4QwDQYJKoZIhvcNAQEL\nBQAwJzElMCMGA1
            },
            api_prov_func_role: 'APF',
            api_prov_func_info: 'apf'
        }
    ],
    api_prov_dom_info: 'This is provider',
    supp_feat: '0',
    fail_reason: 'string',
    api_prov_name: null,
    onboarding_date: ISODate('2025-11-27T15:52:29.743Z'),
    username: 'safe-6g-imm',
    uuid: 'e6002375-025c-591f-b448-d845813d528f'
}
```


Figure 53 Successful Provider's onboarding document inside CAPIF's MongoDB database

OpenCAPIF constitutes the common API exposure and consumption layer within the SAFE-6G framework, enabling uniform, secure, and decoupled access to capabilities provided by the core network, the edge–cloud continuum, and the application plane. By acting as a mediation layer between API providers and API consumers, OpenCAPIF supports interoperability across heterogeneous domains while preserving access control, authentication, and monitoring mechanisms, in line with the SAFE-6G reference architecture.

From an integration perspective, a SAFE-6G component (e.g., the CoCo or a TF) interacts with OpenCAPIF to retrieve information or invoke actions exposed by another component through the following process. First, the component queries OpenCAPIF to discover the relevant API corresponding to the required functionality (e.g., network state, infrastructure metrics, or application-level context). Once the API is identified, the component performs an authorized request via OpenCAPIF, which brokers the interaction and forwards the request to the appropriate API provider. The response is then returned through OpenCAPIF to the requesting component, enabling it to consume the retrieved information in a consistent and technology-agnostic manner

This interaction pattern allows SAFE-6G components to access operational, contextual, and application-specific information without requiring direct coupling to the underlying implementation of the API providers. As a result, OpenCAPIF facilitates modular integration, simplifies cross-domain interactions, and supports the scalable deployment of trust-aware functionalities across the SAFE-6G ecosystem.

4.5.4 Next Steps

The next steps focus on extending and keep validating the use of OpenCAPIF across all the SAFE-6G technical components that require access to network, infrastructure, and application-level information. This includes systematically integrating all TFs and supporting subsystems, and performing additional E2E testing of CAPIF-based API discovery, authorization, and invocation workflows. The objective is to verify secure and reliable information exchange through OpenCAPIF, ensure correct enforcement of access control and authorization mechanisms, and confirm that all components can interact in a decoupled and interoperable manner.

4.6 Safety Trust Function

4.6.1 Overview

The User-Centric Safety Function is a core enabler of the SAFE-6G architecture, providing intelligent, adaptive, and fine-grained protection mechanisms that safeguard users, services, and network assets from unintentional harm, misconfigurations, and unsafe operational states. Operating as a Software-Defined Perimeter (SDP) stack, the Safety Function introduces a trust-aware layer of dynamic control within the 6G Packet Core, ensuring that only authenticated, authorized, and contextually validated entities gain access to network microservices and data paths. At its core, the Safety Function integrates data collection, preprocessing, AI-driven safety inference, and run-time network enforcement. It operates in close coordination with existing 5G/6G core NFs that reflect the live state of the network. Its AI agent consumes information by those functions to perform real-time inference and derive safety scores to determine the necessary mitigation or enforcement actions.

4.6.2 Architecture

The internal architecture of the Safety Function is modular, multi-layered, and designed to integrate seamlessly with the SAFE-6G CoCo, network core functions, and aerOS orchestration environment. It is composed of three tightly coupled layers named AI Agent, vApp and nApp. Together, they form a continuous pipeline spanning from data ingestion, decision-making to ultimately network policy enforcement. The high-level sequence diagram for the Safety TF is depicted in Figure 54.

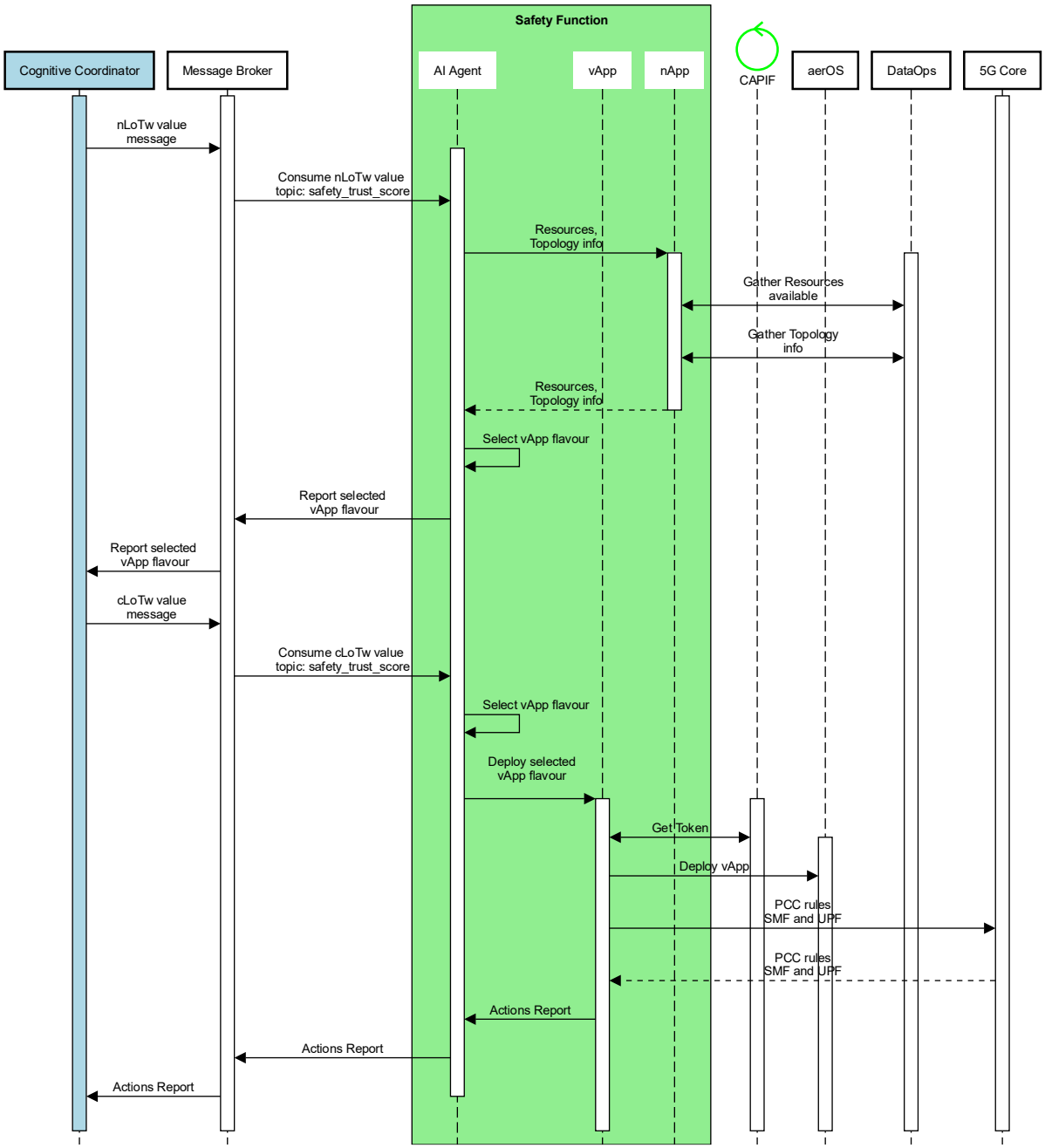


Figure 54 Safety Trust Function Sequence Diagram

At the top of the architecture, the AI layer hosts the Local AI Agent, which serves as the central decision-making entity. The agent receives the nLoTw value, derived from the user's request, and queries the nApp layer to obtain current network and infrastructure conditions. By combining the nLoTw value with real-time network metrics and predefined safety thresholds, the AI agent determines the appropriate actions to take. It then reports its assessment to the CoCo. After resolving any conflicts between the TFs, the CoCo provides a cLoTw to the Safety Function, which interprets and executes the actions required to provide the desired safety level. These actions may include instantiating additional vApps, deploying new NFs, or adjusting security and policy configurations across the user's network perimeter.

The nApp layer consists of two specialized nApps that provide the AI agent with comprehensive network awareness. These two nApps are reusable software components that provide up-to-date information on network topology and available resources, which the AI layer leverages to make more informed decisions. The first, the Topology Manager, analyzes available network paths and computes a weighted cost based on metrics such as jitter, packet loss, and hop count, enabling the selection of the safest and most efficient routing option that will be used to define the perimeter. The second, the Resource Manager, assesses the current infrastructure load by modeling real-time CPU, memory, and disk utilization to achieve the optimal resource allocation to ensure SDP operation. It evaluates whether incoming requests at different demand levels can be accommodated without violating safety or performance constraints. Together, these nApps supply the AI layer with both network-level and resource-level insights required for informed, safety-driven decisions.

The vApp Layer implements the core SDP logic and contains the three main nApps that constitute the safety perimeter named Service/Data Plane Control Function (SDPCF), Service/Data Plane Gateway (SDPGW) and VPN Gateway Function (VGF). Their purpose is to:

- Authenticate the gateway and UE, manage PDU session control, and interface with policy components (SDPCF, an enhanced SMF).
- Establish a safe and isolated forwarding plane between the UE and target applications (SDPGW, an enhanced UPF).
- Configure and manage the E2E encrypted tunnels per user and per application flow (VGF, a VPN Gateway Function).

These functions are instantiated based on the AI agent's trust-based decisions, ensuring that only authorised microservices are added to the perimeter. This layer enforces microservice-level access control, encryption, and traffic isolation, resulting in a robust, adaptive safety perimeter.

4.6.3 Integration with other Components

Source	Destination	Action/Method	Details
CoCo	Safety TF	Consumer/kafka	User request + UE info Topic: safety_trust_score
Safety TF	DataOps	Invoker/API	Information about network resources and topology
Safety TF	Core	Invoker/API	Enforce Safety Policies to the network.
Safety TF	CoCo	Producer/kafka	Topic: safety_report

Table 12 Safety TF communication with SAFE-6G components

4.6.4 Unit Tests

This section describes the unit tests implemented for the Safety Function, focusing on the verification of individual software components and internal logic in isolation. The objective of these tests is to ensure that each unit behaves as expected under controlled conditions, using mocked dependencies where appropriate. Unit tests provide early validation of functionality and serve as the foundation for higher-level integration and system testing. The components below, presented in similar manner as _parse_request, _lookup_amf_context, _safe_validate_token, and _send_response, represent internal implementation details of the Safety TF component. These elements are part of the internal program logic and are used to structure and organize processing features. As such, they are not

represented as separate components in the sequence diagrams, which focus instead on higher-level architectural interactions between the AI agent layer, nApp layer and vApp layer.

Safety-UT1: Kafka Processor

Objective: To verify that path generation and persistence work correctly and deterministically.	
Components	a. KafkaProcessor.start b. _parse_request, _lookup_amf_context, _safe_validate_token, _send_response c. Mock consumer, mock producer, mock path selector, mock FLCM, mock token validator
Requirements	1. Valid Kafka messages trigger the full processing flow 2. Response produced with correct action and updated context
Features to be tested	1. Full message processing loop 2. Token validation 3. AMF context lookup 4. Path selection and response sending

Test Steps / Cases

Case	Action	Expected Result
1	Kafka message with SUPI/MSISDN/IMEI/USR/LOT/Target_App_ID	Response produced with action_performed=True, AMF context updated, path selected

Verification Checklist for Safety-UT1:

Step	Description	Yes	No	Comments
1	Provide mocked Kafka message with all required fields	☒	☐	
2	Execute KafkaProcessor.start loop once	☒	☐	
3	Verify _parse_request correctly extracts fields	☒	☐	
4	Verify _lookup_amf_context runs and updates context	☒	☐	
5	Verify _safe_validate_token executes and validates token	☒	☐	
6	Verify _send_response produces correct response	☒	☐	
7	Check action_performed=True in response	☒	☐	
8	Confirm selected path matches expected outcome	☒	☐	

Execution Output:
Safety-UT1 was executed with all required dependencies configured as defined in the test setup. The test verified the complete processing loop of the Kafka processor, including message parsing, AMF context lookup, token validation, path selection, and response production. All assertions were satisfied, demonstrating that the processor correctly handles valid Kafka messages and executes the full workflow. Figure 55 illustrates the test execution results, including relevant logs and assertions that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_kafka_processor.py::test_happy_path_triggers_all_steps
====================================== test session starts ======================================

--- live log call ---
PASSED
tests/test_kafka_processor.py::test_safe_validate_token_handles_system_exit
--- live log call ---
```

```
ERROR safety_af.kafka_processor - validate_token triggered SystemExit: missing env
PASSED
tests/test_kafka_processor.py::test_safe_validate_token_rejects_falsy_token
--- live log call ---
ERROR safety_af.kafka_processor - validate_token returned empty/falsey token.
PASSED
======================================== 3 passed in 7ms ========================================
```

Figure 55 Safety-UT1 output

Safety-UT2: Kafka Processor – Token Validation

Objective: To ensure _safe_validate_token handles errors and falsy tokens gracefully without crashing the processor.	
Components	a. KafkaProcessor._safe_validate_token b. Mock validate_token
Requirements	1. SystemExit exceptions must be caught and logged 2. Falsy tokens must be rejected and logged 3. _safe_validate_token must return None on invalid tokens
Features to be tested	1. Exception handling for SystemExit 2. Rejection of empty or falsy tokens 3. Logging of errors

Test Steps / Cases

Case	Action	Expected Result
1	validate_token raises SystemExit("missing env")	_safe_validate_token returns None, error logged
2	validate_token returns empty string	_safe_validate_token returns None, error logged

Verification Checklist for Safety-UT2:

Step	Description	Yes	No	Comments
1	Patch validate_token to raise SystemExit	☒	☐	
2	Call _safe_validate_token	☒	☐	
3	Verify _safe_validate_token returns None	☒	☐	
4	Check that error is logged for SystemExit	☒	☐	
5	Patch validate_token to return empty string	☒	☐	
6	Call _safe_validate_token	☒	☐	
7	Verify _safe_validate_token returns None	☒	☐	
8	Check that error is logged for falsy token	☒	☐	

Execution Output:

Safety-UT2 was executed with all the required dependencies configured as defined in the test setup. The test passed successfully, and all assertions were satisfied, demonstrating correct behaviour of the tested functionality. Figure 56 illustrates the test execution results, including relevant logs and assertions that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_kafka_processor.py::test_safe_validate_token_handles_system_exit
============================= test session starts ==============================
--- live log call ---
ERROR safety_af.kafka_processor - validate_token triggered SystemExit: missing env
PASSED
tests/test_kafka_processor.py::test_safe_validate_token_rejects_falsy_token
--- live log call ---
ERROR safety_af.kafka_processor - validate_token returned empty/falsey token.
PASSED
============================== 2 passed in 2ms ===============================
```

Figure 56 Safety-UT2 output

Safety-UT3: AI Model Inference

Objective: To verify that the AI model endpoint correctly handles different input scenarios, including valid input, invalid formats, and missing required fields.	
Components	a. KSERVE model endpoint b. make_inference_request() function c. Response handler / logging
Requirements	1. Valid input (HTTP 200 with inference result) 2. Invalid input format (HTTP 400 with error) 3. Missing required fields (HTTP 400 with error)
Features to be tested	1. Input validation 2. Model inference processing 3. Error reporting

Test Steps / Cases

Case	Input	Expected Result
1	Valid payload with all required fields	Status code = 200, response contains valid inference
2	Invalid data format ({"instances": "invalid_data"})	Status code = 400, error returned
3	Missing required field ("instances" missing)	Status code = 400, error returned

Verification Checklist for Safety-UT3:

Step	Description	Yes	No	Comments
1	Valid payload returns HTTP 200	☒	☐	
2	Response for valid payload contains expected inference result	☒	☐	
3	Invalid data format returns HTTP 400	☒	☐	
4	Missing required field returns HTTP 400	☒	☐	
5	Error messages/logs are correctly generated	☒	☐	

Execution Output:

Safety-UT3 was executed with all required dependencies configured as defined in the test setup. The test verified that the AI model serving endpoint responds correctly to inference requests, including handling of valid, invalid, and missing data inputs. All assertions were satisfied, demonstrating that the model produces expected outputs and handles errors gracefully. Figure 57Figure illustrates the test execution results, including relevant logs and responses that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_integration.py::test_valid_data
========================================= test session starts ==========================================
--- live log call ---
INFO root:kserve_request.py:9 Kserve endpoint: http://localhost:8080/v1/models/safety-model:predict
INFO root:test_integration.py:21 Test test_integration was successful: {"predictions":[0,1]}
PASSED                                                                                        [ 33%]
--- live log call ---
INFO root:kserve_request.py:9 Kserve endpoint: http://localhost:8080/v1/models/safety-model:predict
INFO root:test_integration.py:27 status_code: 400, response.text:{"error":"Expected \"instances\" to
be a list"}
INFO root:test_integration.py:29 Test test_integration_invalid_data was successful
PASSED                                                                                        [ 66%]
--- live log call ---
INFO root:kserve_request.py:9 Kserve endpoint: http://localhost:8080/v1/models/safety-model:predict
INFO root:test_integration.py:40 status_code: 400, response.text:{"detail":"Invalid input: Missing
required fields: ['PathCost', 'ResourceUsage', 'CurrentLoad']"}
INFO root:test_integration.py:42 Test test_integration_missing_field was successful
PASSED                                                                                        [100%]
========================================== 3 passed in 0.22s ===========================================
```

Figure 57 Safety-UT3 output

Safety-UT4: Repository – Cluster and Candidate Cluster Persistence

Objective: To verify that clusters and candidate clusters are persisted and retrievable correctly			
Components	a. Repository SQLite tables: clusters, candidate_clusters b. Dataclasses: Cluster, CandidateCluster, Resources c. Helper: utcnow		
Requirements	1. Upsert clusters and candidate clusters correctly 2. Persisted rows retrievable with correct cluster_id and cost		
Features to be tested	1. Repository upsert 2. Listing clusters and candidate clusters		
Test Steps / Cases			
Case	**Action**	**Expected Result**	
1	Cluster(id="c1"), CandidateCluster(cost=0.42, Resources(CPU/mem/storage))	One row returned with same cluster_id, cost in [0,1]	

Verification Checklist for Safety-UT4:

Step	Description	Yes	No	Comments
1	Upsert cluster into repository	☒	☐	
2	Upsert candidate cluster into repository	☒	☐	
3	List candidate clusters and verify cluster_id	☒	☐	
4	Verify cost in [0,1]	☒	☐	
5	Ensure SQLite table contains exactly one row	☒	☐	

Execution Output:

Safety-UT4 was executed with all required dependencies configured as defined in the test setup. The test verified that clusters and candidate clusters are correctly persisted and retrieved from the repository, including validation of stored fields such as cluster IDs and candidate costs. All assertions were satisfied, demonstrating that the repository layer behaves as expected under standard operations. Figure 58 illustrates the test execution results, including relevant logs and database entries that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_candidate_paths_service.py::test_repository_upsert_and_list
====================================== test session starts ======================================
--- live log call ---
PASSED
========================================== 1 passed in 32ms ==========================================
```

Figure 58 Safety-UT4 output

Safety-UT5: Cost Model – Environmental Variable

Objective: Verify that environment-based region preference affects cost computation correctly.		
Components	a. CostModel.cost function b. Environment variable lookup (e.g., PREFERRED_REGION)	
Requirements	1. Clusters in the preferred region must have lower computed cost 2. Clusters in other regions should have higher cost	
Features to be tested	1. Cost computation 2. Region-based preference logic	
Test Steps / Cases		
Case	**Action**	**Expected Result**

1	PREFERRED_REGION=eu-west-1, two clusters (one matching region, one non-matching) with same resources	Cost of preferred-region cluster < cost of non-preferred cluster

Verification Checklist for Safety-UT5:

Step	Description	Yes	No	Comments
1	Set environment variable PREFERRED_REGION=eu-west-1	☒	☐	
2	Compute cost for cluster in preferred region	☒	☐	
3	Compute cost for cluster in non-preferred region	☒	☐	
4	Verify preferred-region cluster cost < non-preferred	☒	☐	
5	Check that computed costs are within expected range [0,1]	☒	☐	

Execution Output:

Safety-UT5 was executed with all required dependencies configured as defined in the test setup. The test verified that the cost model correctly adjusts candidate cluster costs based on the configured environment variable (e.g., PREFERRED_REGION). All assertions were satisfied, demonstrating that the cost computation reflects environmental preferences as expected. Figure 59 illustrates the test execution results, including relevant logs and computed cost values that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_candidate_paths_service.py::test_cost_model_components_env_preference
====================================== test session starts ======================================
--- live log call ---
PASSED
======================================= 1 passed in 1ms =======================================
```

Figure 59 Safety-UT5 output

Safety-UT6: Candidate Path Service – Tick Persistence

Objective: To verify that a service tick stores candidate clusters and legacy paths correctly.		
Components	a. CandidatePathService._tick b. Repository.upsert_candidate_cluster c. Repository.upsert_paths_bulk d. Logging	
Requirements	1. Candidate clusters must be persisted with correct costs 2. Legacy paths must be stored correctly in the database	
Features to be tested	1. Tick persistence 2. Candidate cluster and path creation 3. Logging during persistence	
Test Steps / Cases		
Case	**Action**	**Expected Result**
1	Fake MANO with 2 clusters, dummy resource provider map, temp SQLite	2 candidate rows with different costs; 4 path rows saved

Verification Checklist for Safety-UT6:

Step	Description	Yes	No	Comments
1	Execute _tick with fake MANO clusters and dummy resource provider map	☒	☐	
2	Verify 2 candidate clusters persisted with different costs	☒	☐	

3	Verify 4 path rows saved in repository	☒	☐	
4	Check logging output for persistence steps	☒	☐	
5	Ensure no exceptions or errors during tick	☒	☐	

Execution Output:
Safety-UT6 was executed with all required dependencies configured as defined in the test setup. The test verified that a service tick correctly persists candidate clusters and legacy paths in the repository, including validation of the number of candidate rows and path entries. All assertions were satisfied, demonstrating that the tick operation behaves as expected and maintains data integrity. Figure 60 illustrates the test execution results, including relevant logs and repository entries that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_candidate_paths_service.py::test_service_tick_persists_candidates
====================================== test session starts ======================================
--- live log call ---
INFO candidate-paths - event=tick status=started
INFO candidate-paths - event=candidate_upsert cluster_id=c1 cost=0.233 cpu_free=28.00/32.00
mem_free=128849018880/137438953472 , storage_free=966367641600/1099511627776
INFO candidate-paths - event=candidate_upsert cluster_id=c2 cost=0.677 cpu_free=10.00/32.00
mem_free=42949672960/137438953472 , storage_free=214748364800/1099511627776
INFO candidate-paths - event=paths_upsert status=ok count=4
INFO candidate-paths - paths_link_cost:
INFO candidate-paths - RANK PATH(nodes) BW_MIN LAT_SUM(ms)JIT_MAX(ms)PL_MAX PATH_COST
INFO candidate-paths - 2 ran_cluster → eks-prod 10 41.0 15.1 0.075 0.273
INFO candidate-paths - 1 ran_cluster → onprem-a 4 65.8 8.8 0.061 0.602
INFO candidate-paths - 4 ran_cluster → eks-prod → onprem-a 6 138.8 15.1 0.075 0.650
INFO candidate-paths - 3 ran_cluster → onprem-a → eks-prod 4 163.5 11.8 0.061 0.720
INFO candidate-paths - event=paths_link_cost_dump status=ok path=legacy_paths.json count=4
INFO candidate-paths - event=tick status=finished count=2
PASSED
====================================== 1 passed in 49ms ======================================
```

Figure 60 Safety-UT6 output

Safety-UT7: CandidatePathService – Path Persistence

Objective: To verify that path generation and persistence work correctly and deterministically.		
Components	a. CandidatePathService._dump_paths_link_cost b. Repository.upsert_paths_bulk c. Environment flags (to enable legacy path build)	
Requirements	1. Paths must be generated and saved in SQLite with deterministic costs and hops 2. Re-running the path dump updates updated_at while keeping the same cost	
Features to be tested	1. Path generation logic 2. Persistence of paths in repository 3. Cost and timestamp correctness	
Test Steps / Cases		
Case	**Action**	**Expected Result**
1	Env flags enable legacy path build, fake MANO clusters	4 paths saved, hops=2 or 3, cost in [0,1], ISO timestamp parseable
2	Re-run with same seed, patched utcnow returning different timestamps	Same path costs, different updated_at values

Verification Checklist for Safety-UT7:

Step	Description	Yes	No	Comments
1	Enable legacy path build via environment flags	☒	☐	
2	Generate and dump paths with _dump_paths_link_cost	☒	☐	
3	Verify 4 paths saved in repository	☒	☐	
4	Check hops of each path = 2 or 3	☒	☐	
5	Verify cost of each path is within [0,1]	☒	☐	
6	Re-run path dump with patched utcnow	☒	☐	
7	Verify path costs remain the same, updated_at timestamps differ	☒	☐	

Execution Output:

Safety-UT7 was executed with all required dependencies configured as defined in the test setup. The test verified that candidate paths are generated, persisted, and sorted correctly in the repository, and that repeated executions update the updated_at timestamp while preserving path costs. All assertions were satisfied, demonstrating that the path persistence logic behaves deterministically and maintains data integrity. Figure 61 illustrates the test execution results, including relevant logs, generated paths, and repository entries that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_candidate_paths_service.py::test_paths_persist_and_sorted_dump
========================================= test session starts ==========================================
--- live log call ---
INFO candidate-paths - event=paths_upsert status=ok count=4
INFO candidate-paths - paths_link_cost:
INFO candidate-paths - RANK PATH (nodes) BW_MIN LAT_SUM(ms) JIT_MAX(ms) PL_MAX PATH_COST
INFO candidate-paths - 2 ran_cluster → eks-prod 10 41.0 15.1 0.075 0.273
INFO candidate-paths - 1 ran_cluster → onprem-a 4 65.8 8.8 0.061 0.602
INFO candidate-paths - 4 ran_cluster → eks-prod → onprem-a 6 138.8 15.1 0.075 0.650
INFO candidate-paths - 3 ran_cluster → onprem-a → eks-prod 4 163.5 11.8 0.061 0.720
PASSED
--- live log call ---
INFO candidate-paths - event=paths_upsert status=ok count=4
INFO candidate-paths - paths_link_cost:
INFO candidate-paths - RANK PATH (nodes) BW_MIN LAT_SUM(ms) JIT_MAX(ms) PL_MAX PATH_COST
INFO candidate-paths - 2 ran_cluster → eks-prod 10 41.0 15.1 0.075 0.273
INFO candidate-paths - 1 ran_cluster → onprem-a 4 65.8 8.8 0.061 0.602
INFO candidate-paths - 4 ran_cluster → eks-prod → onprem-a 6 138.8 15.1 0.075 0.650
INFO candidate-paths - 3 ran_cluster → onprem-a → eks-prod 4 163.5 11.8 0.061 0.720
INFO candidate-paths - event=paths_upsert status=ok count=4
INFO candidate-paths - paths_link_cost:
INFO candidate-paths - RANK PATH (nodes) BW_MIN LAT_SUM(ms) JIT_MAX(ms) PL_MAX PATH_COST
INFO candidate-paths - 2 ran_cluster → eks-prod 10 41.0 15.1 0.075 0.273
INFO candidate-paths - 1 ran_cluster → onprem-a 4 65.8 8.8 0.061 0.602
INFO candidate-paths - 4 ran_cluster → eks-prod → onprem-a 6 138.8 15.1 0.075 0.650
INFO candidate-paths - 3 ran_cluster → onprem-a → eks-prod 4 163.5 11.8 0.061 0.720
PASSED
========================================== 3 passed in 61ms ==========================================
```

Figure 61 Safety-UT7 output

Safety-UT8: Path Selector – Best Path Selection

Objective: To verify that the path selector chooses the best path based on AI LOS score and requested RAN cluster, and handles empty path lists.

Components	a. PathSelector.select_best_path b. _assign_roles
Requirements	1. Path with higher AI score should be preferred 2. Only paths matching requested RAN cluster should be selected 3. Handles empty DB gracefully
Features to be tested	1. AI score prioritization 2. RAN cluster filtering 3. Empty path handling

Test Steps / Cases

Case	Action	Expected Result
1	Two paths, AI scores {0.3, 0.8}, lot=0.1	Path with higher score chosen
2	Paths for ranA/ranB, requested RAN cluster = "ranB"	Selected path ranB->y, roles node1=ranB
3	No paths available	ok=False, reason=no_paths_available

Verification Checklist for Safety UT8:

Step	Description	Yes	No	Comments
1	Provide two paths with different AI scores	☒	☐	
2	Call select_best_path	☒	☐	
3	Verify path with higher AI score is selected	☒	☐	
4	Provide paths for multiple RAN clusters, request specific cluster	☒	☐	
5	Verify only paths for requested RAN cluster are selected	☒	☐	
6	Call select_best_path with no paths	☒	☐	
7	Verify output is ok=False and reason=no_paths_available	☒	☐	
8	Ensure roles are assigned correctly for chosen path	☒	☐	

Execution Output:

Safety-UT8 was executed with all required dependencies configured as defined in the test setup. The test verified that the path selector correctly chooses the best path based on AI scores and requested RAN clusters, including handling of multiple candidate paths and the case where no paths are available. All assertions were satisfied, demonstrating that the path selection logic behaves as expected under normal and edge-case scenarios. Figure 62 illustrates the test execution results, including relevant logs and selected path information that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_path_selector.py::test_select_best_path_prefers_highest_ai_score
======================================== test session starts ========================================
--- live log call ---
PASSED
tests/test_path_selector.py::test_select_best_path_filters_by_ran_cluster
--- live log call ---
PASSED
tests/test_path_selector.py::test_select_best_path_when_no_paths
--- live log call ---
PASSED
========================================= 3 passed in 1ms ==========================================
```

Figure 62 Safety-UT8 output

Safety-UT9: OSM Adapter – Cluster Normalization

Objective: To ensure that the raw OSM cluster list is correctly converted into Cluster objects.		
Components	a. OSMAdapter.list_clusters b. Patched validate_token c. Patched get_k8s_lst	
Requirements	1. Raw cluster data from OSM is transformed into Cluster dataclass objects 2. Cluster IDs and names match expected values	
Features to be tested	1. Token validation 2. Cluster object creation from OSM data 3. Correct mapping of UUIDs and names	
Test Steps / Cases		
Case	**Action**	**Expected Result**
1	Patched OSM API returning two UUID/name pairs	Two Cluster objects created with expected IDs and names

Verification Checklist for Safety-UT9:

Step	Description	Yes	No	Comments
1	Patch validate_token and get_k8s_lst	☒	☐	
2	Call OSMAdapter.list_clusters	☒	☐	
3	Verify two Cluster objects are returned	☒	☐	
4	Check that cluster IDs and names match expected values	☒	☐	
5	Ensure no exceptions or token errors during listing	☒	☐	

Execution Output:

Safety-UT9 was executed with all required dependencies configured as defined in the test setup. The test verified that the OSM Adapter correctly converts raw OSM cluster data into normalized Cluster objects, including validation of cluster IDs and names. All assertions were satisfied, demonstrating that the adapter transforms input data accurately and reliably. Figure 63 illustrates the test execution results, including relevant logs and generated Cluster objects that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_candidate_paths_service.py::test_osm_adapter_normalizes_clusters
========================================= test session starts ==========================================
--- live log call ---
12:22:05 INFO candidate-paths - event=mano_list_clusters status=ok count=2
PASSED
========================================== 1 passed in 1ms ===========================================
```

Figure 63 Safety-UT9 output

Safety-UT10: Legacy Discover – Candidate Paths Schema

Objective: To verify that the legacy discovery function returns candidate paths with the expected structure.	
Components	a. discover_candidate_paths b. _build_paths c. Cost and resource checker functions
Requirements	1. Returned paths must include bandwidth and per-node resource dictionaries 2. Paths must be keyed by path IDs 3. Structure must conform to expected schema

Features to be tested	1. Path structure correctness 2. Resource and bandwidth inclusion 3. Integration with _build_paths and validation functions	
Test Steps / Cases		
Case	**Action**	**Expected Result**
1	Patched token and k8s list, default hops and seed	Dict keyed by path IDs, each path contains bandwidth and per-node resource dictionaries

Verification Checklist for Safety-UT10:

Step	Description	Yes	No	Comments
1	Patch token validation and k8s cluster list	☒	☐	
2	Call discover_candidate_paths with defaults	☒	☐	
3	Verify returned dictionary is keyed by path IDs	☒	☐	
4	Verify each path contains bandwidth information	☒	☐	
5	Verify each path contains per-node resource dictionaries	☒	☐	
6	Ensure no exceptions occur during path discovery	☒	☐	

Execution Output:

Safety-UT10 was executed with all required dependencies configured as defined in the test setup. The test verified that the legacy discovery process produces candidate paths with the expected schema, including bandwidth information and per-node resource dictionaries. All assertions were satisfied, demonstrating that the discovery logic correctly generates paths and maintains the required data structure. Figure 64 illustrates the test execution results, including relevant logs and generated path structures that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_legacy_discover.py::test_discover_candidate_paths_schema
======================================== test session starts ========================================
--- live log call ---
PASSED
========================================= 1 passed in 4ms ==========================================
```

Figure 64 Safety-UT10 output

Safety-UT11: Legacy Discover – Deterministic Output

Objective: To verify that the legacy discovery produces deterministic paths when using a fixed seed.		
Components	a. discover_candidate_paths randomness handling b. RNG seed control	
Requirements	1. Fixed RNG seed should produce consistent, repeatable path outputs 2. Hash of JSON representation of paths should match expected value	
Features to be tested	1. Deterministic behavior under fixed seed 2. Path stability across runs	
Test Steps / Cases		
Case	**Action**	**Expected Result**
1	RNG seed = 123, known k8s list	Hash of JSON string matches expected; paths stable across runs

Verification Checklist for Safety-UT11:

Step	Description	Yes	No	Comments
1	Set RNG seed to fixed value (123)	☒	☐	
2	Call discover_candidate_paths with known k8s list	☒	☐	
3	Generate JSON representation of returned paths	☒	☐	
4	Compute hash of JSON and compare to expected	☒	☐	
5	Verify paths remain stable across multiple runs with same seed	☒	☐	
6	Ensure no exceptions occur during deterministic discovery	☒	☐	

Execution Output:

Safety-UT11 was executed with all required dependencies configured as defined in the test setup. The test verified that the legacy discovery process produces deterministic candidate paths when a fixed random seed is used, ensuring that path generation and cost values are consistent across multiple executions. All assertions were satisfied, demonstrating that the discovery logic behaves predictably and reliably under controlled conditions. Figure 65 illustrates the test execution results, including relevant logs and path outputs that confirm compliance with the expected outcomes described in the test steps.

```
tests/test_legacy_discover.py::test_deterministic_with_seed
======================================== test session starts ========================================
--- live log call ---
PASSED
========================================= 1 passed in 6ms ==========================================
```

Figure 65 Safety-UT11 output

4.6.5 Integration Tests

Safety-IT1: AI Model – CI/CD Deployment & KSERVE Inference

Objective: To validate that the AI model is successfully downloaded, built, internally tested, released, and served via KSERVE, ensuring it can respond correctly to inference requests.	
Components	a. MinIO bucket & endpoint (download stage) b. Docker build environment (build stage) c. Unit test framework for internal container testing (test_internal_container stage) d. CI/CD release mechanism (release stage) e. KSERVE model serving endpoint f. Test client for inference requests
Requirements	1. Model artifact can be downloaded from MinIO 2. Docker image builds successfully from model code 3. Internal unit tests pass inside the container 4. Model is deployed to KSERVE and endpoint is available 5. Model returns correct inference for valid requests, and appropriate errors for invalid/missing fields
Features to be tested	1. Artifact download from MinIO 2. Docker image build 3. Internal container unit tests (AI model validation) 4. Deployment via release stage 5. KSERVE endpoint availability 6. Inference correctness & error handling
Test Steps / Cases by Pipeline Stage	

Stage	Action	Expected Result
Download	Pull model artifact from MinIO bucket/endpoint	Artifact downloaded successfully, correct version
Build	Build Docker image using model code and Dockerfile	Docker image built successfully, no errors
Test Internal Container	Run internal unit tests for AI model (valid, invalid, missing field)	All unit tests pass inside container
Release	Deploy container to KSERVE via CI/CD release stage	Container deployed successfully, KSERVE endpoint is available
Inference Testing (1)	Send valid payload to KSERVE	HTTP 200, response contains valid inference
Inference Testing (2)	Send invalid payload (invalid_data)	HTTP 400, error returned
Inference Testing (3)	Send payload missing required fields	HTTP 400, error returned
Monitoring / Logging	Check deployment logs, endpoint readiness	Logs show no critical errors, endpoint responds within timeout

Verification Checklist for Safety-IT1: AI Model – CI/CD Deployment & Kserve Inference

Step	Description	Yes	No	Comments
1	Model artifact successfully downloaded from MinIO bucket	☒	☐	
2	Docker image built successfully from model code	☒	☐	
3	Internal unit tests for AI model pass inside the container	☒	☐	
4	Container successfully deployed to KSERVE via CI/CD release stage	☒	☐	
5	KSERVE endpoint becomes available and ready to serve requests	☒	☐	
6	Inference test – valid payload returns HTTP 200 and correct result	☒	☐	
7	Inference test – invalid data format returns HTTP 400	☒	☐	
8	Inference test – missing required field returns HTTP 400	☒	☐	
9	Deployment and inference logs contain no critical errors	☒	☐	
10	Endpoint responds within acceptable timeout	☒	☐	

Execution Output:
The integration test was executed using the project's existing CI/CD pipeline to validate the E2E deployment and serving of the AI model. The pipeline stages were triggered automatically and executed sequentially, including model artifact download, container image build, internal container testing, and release deployment. All pipeline stages completed successfully, resulting in a Docker image being built and deployed to the model serving platform using KSERVE. After deployment, the model endpoint became available and responded correctly to inference requests, confirming proper integration between the CI/CD pipeline, container image, and model serving infrastructure.

Figure 66 shows the successful execution of the CI/CD pipeline stages, demonstrating that the AI model was built, tested, and deployed without errors, and is ready to serve inference requests.

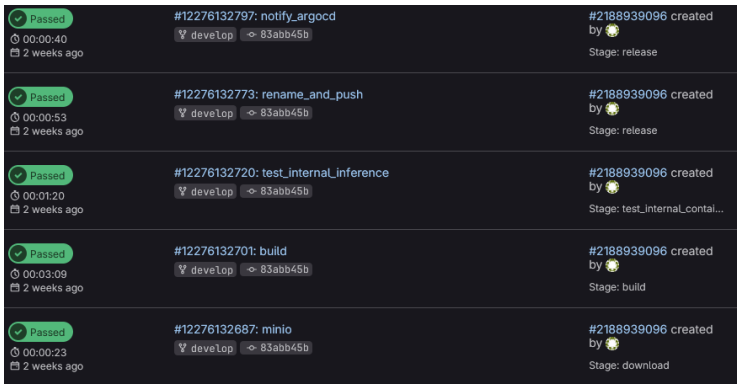


Figure 66 Safety-IT1 output

Safety-IT2: CoCo to SafetyTrustFunction Kafka Consumption

Objective: To verify that STF correctly consumes and parses messages published by CoCo on the Kafka topic.		
Components	a. CoCo Kafka Producer b. Kafka Broker c. SafetyTrustFunction Kafka Consumer d. SafetyTrustFunction Message Parser	
Requirements	1. CoCo must publish a valid message to the Kafka topic 2. SafetyTrustFunction must consume the message without errors 3. SafetyTrustFunction must correctly parse required fields	
Features to be tested	1. Kafka message consumption 2. Message parsing and validation	
Test Steps / Cases by Pipeline Stage		
Stage	**Action**	**Expected Result**
CoCo *to* Kafka	CoCo publishes a valid message to the topic	Message is available on Kafka topic
Kafka *to* SafetyF.	STF consumer polls Kafka	Message is consumed successfully
SafetyTrustFunction	STF parses message payload	Required fields are extracted correctly

Safety-IT3: SafetyTrustFunction to DataOps API Invocation

Objective: To verify that Safety TF correctly invokes the DataOps service API based on the received Kafka message and internal processing logic.		
Components	a. Safety TF Processing Logic b. DataOps Invoker / API c. Network communication layer	
Requirements	1. Safety TF must construct a valid API request for DataOps 2. DataOps API must be reachable from STF 3. Safety TF must handle the API response correctly	
Features to be tested	1. API request construction 2. Safety TF to DataOps service invocation 3. Response handling and error propagation	
Test Steps / Cases by Pipeline Stage		
Stage	**Action**	**Expected Result**

Safety TF Internal	Safety TF derives DataOps request from Kafka message	Request payload is correctly constructed
Safety TF *to* DataOps	Safety TF invokes DataOps API	API is called with correct parameters
DataOps *to* STF	DataOps returns a response	Response is received and processed by STF

Safety-IT4: SafetyTrustFunction to Core API Invocation

Objective: To verify that Safety TF correctly invokes the Core service API as part of its decision and orchestration flow.		
Components	a. Safety TF Processing Logic b. Core Invoker / API c. Network communication layer	
Requirements	1. Safety TF must generate a valid request for the Core API 2. Core API must be reachable and respond correctly 3. Safety TF must correctly process the Core API response	
Features to be tested	1. API request construction for Core 2. Safety TF to Core service invocation 3. Response handling and decision continuation	
Test Steps / Cases by Pipeline Stage		
Stage	**Action**	**Expected Result**
Safety TF Internal	Safety TF determines need to invoke Core API	Core API request is prepared
Safety TF *to* Core	Safety TF invokes Core API	Core API is called successfully
Core *to* Safety TF	Core returns response	Response is received and processed correctly

Safety-IT5: SafetyTrustFunction to CoCo Kafka Production

Objective: To verify that Safety TF publishes processed results back to CoCo via Kafka correctly, completing the E2E integration flow.		
Components	a. Safety TF Processing Logic b. Kafka Producer c. Kafka Broker d. CoCo Kafka Consumer	
Requirements	1. Safety TF must format the response message correctly 2. Kafka broker must accept the message from Safety TF 3. CoCo must be able to consume the message successfully	
Features to be tested	1. Kafka message production from Safety TF 2. Correct message formatting and payload 3. E2E flow completion	
Test Steps / Cases by Pipeline Stage		
Stage	**Action**	**Expected Result**
Safety TF Internal	Safety TF prepares response message	Message payload is correctly formatted
Safety TF *to* Kafka	Safety TF publishes message to Kafka	Kafka broker accepts the message

Kafka *to* CoCo	CoCo consumes the message	CoCo receives message with expected content

4.6.6 Next Steps

This section described the planned unit and integration tests for the Safety TF component, focusing on validating its interactions with upstream and downstream systems. The objective of these tests is to ensure that the Safety TF correctly consumes messages, invokes external services, processes responses, and publishes results as part of the overall system workflow. The integration tests are designed around the key communication flows involving Safety TF, including Kafka-based message exchange and API-based service invocations. These tests aim to verify interface correctness, data flow integrity, and basic error handling across system boundaries, rather than internal algorithmic behavior. The following integration tests will be implemented using test or staging environments, leveraging existing messaging infrastructure and service interfaces where available. Together, they provide confidence that the Safety TF operates correctly within the larger system context and can be safely integrated with other components.

4.7 Security Trust Function

4.7.1 Overview

The Security TF provides real-time enforcement of security decisions across the SAFE-6G system by leveraging dynamic trust levels, contextual information, and system metrics supplied by the CoCo. Its primary objective is to adapt security controls to live trust conditions and system behaviour, rather than relying solely on static identities or fixed rule sets. The Security Function operates as a distributed, AI-driven security enforcement pipeline tightly integrated into the SAFE-6G control plane. It enables hybrid identity and access management, combining OAuth2/OpenID Connect (OIDC) for low-latency, token-based authorization with Self-Sovereign Identity (SSI) and verifiable credentials (VCs) to support decentralized and privacy-preserving identity assurance. In addition, the function delivers traceable and trust-aware enforcement through security audit logs. AI-driven risk evaluation determines proportional security responses, while blockchain-backed logging ensures the integrity, traceability, and compliance of all critical security actions without exposing sensitive data. At the core of the function is the Local AI Agent that continuously listens for trust-score updates and consumes real-time system telemetry data. The agent is responsible for transforming infrastructure monitoring data and trust signals into actionable security decisions. Through real-time reasoning, it evaluates evolving system conditions and user-related trust intents to determine and trigger the most appropriate security enforcement response.

4.7.2 Architecture

The architecture of the Security TF is designed as a modular, layered, and event-driven control loop that enables adaptive security enforcement within the SAFE-6G system. It incorporates an interface and ingestion layer responsible for receiving trust scores, contextual intelligence, and system information from the CoCo and monitoring components. This layered design ensures adaptive, decentralized, and verifiable security enforcement, continuously aligning protection mechanisms with live trust conditions and evolving system behaviour.

The internal architecture of the Security TF comprises the following key elements:

- **Local AI Agent**: Responsible for interpreting trust scores, evaluating feasibility of the user's request under current system conditions, and selecting the appropriate security enforcement flavor. The agent implements event-based interfaces through a Kafka message broker, enabling scalable and decoupled interactions with the CoCo.
- **vApp**: Implementing application-level security enforcement such as authentication strengthening, credential handling, and access restrictions.
- **nApps**: Applying network-level security policies and optimizations accessed through OpenCAPIF, where approved security actions are deployed and enforced.
- **Identity Infrastructure**: Integrating OAuth2/OpenID Connect with SSI mechanisms on a Fabric blockchain, allowing for DID (Defense in Depth) and VCs operations.

Figure 67 illustrates the internal architecture and interaction points of the Security TF within the SAFE-6G ecosystem. The operational workflow of the Security TF is organized as a closed-loop process consisting of three main phases.

Phase 1 – Trust Score Reception and Feasibility Evaluation

The workflow begins when the CoCo publishes the nLoTw for the security dimension to the Message Broker. The Security TF consumes this value and initiates a feasibility assessment. To this end, the function authenticates through OpenCAPIF and gathers real-time network information and operational metrics from the underlying execution planes. Using this contextual information, the Local AI Agent evaluates whether the requested trust level can be supported under current conditions and determines the maximum achievable trust level.

Phase 2 – Trust Calibration and Action Approval

The achievable trust value is reported back to the CoCo through the Message Broker as feedback. Upon receiving feasibility feedback from the Security TF, the CoCo performs conflict resolution and final calibration across all TFs to ensure cross-function compatibility. If required, an updated cLoTw is produced and redistributed. Once the calibrated trust level is confirmed, the Security TF derives a set of proposed security actions, including the selection of enforcement flavours and deployment strategies. These proposed actions are published to the Message Broker and consumed by the CoCo, which validates them against global system policies and other TF actions. Approved actions are then returned to the Security TF for execution.

Phase 3 – Enforcement Deployment and Execution

Following approval, the Security TF deploys the selected vApp and nApp flavours through the execution planes exposed via OpenCAPIF. Deployment identifiers are reported back to the CoCo to ensure full traceability. During enforcement, the Security TF may request UE and session context from the 5G Core and associated systems. Depending on the selected enforcement flavour, thed function performs identity-related operations such as user registration, credential issuance, and tokenized action generation, leveraging OAuth2/OIDC and SSI mechanisms as appropriate. Finally, the approved security policies are applied and enforced across the planes, completing the trust-driven security loop.

This process ensures that security enforcement remains adaptive, proportional, and verifiable, continuously aligned with live trust conditions and system behaviour.

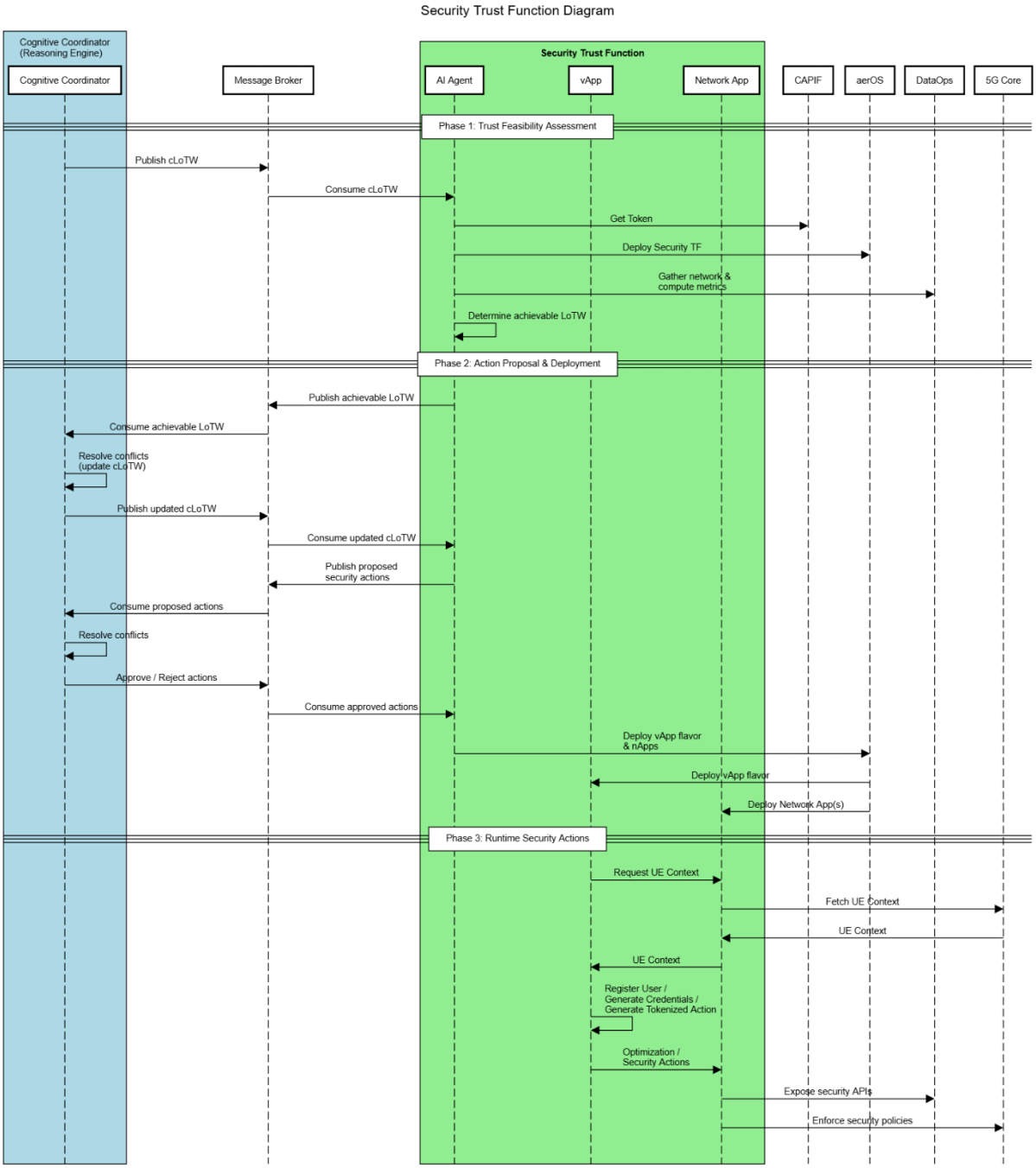


Figure 67 Security Trust Function sequence diagram

4.7.3 INTEGRATION WITH OTHER COMPONENTS

Component	Protocol (API/Kafka)	Action (Read/Publish/Subscribe)	Details

CoCo	Kafka	Subscribe	Consumes LoTw for the security dimension published by the CoCo on the Security TF topic.
CoCo	Kafka	Publish	Publishes achievable trust level (feasible LoTw) and proposed security actions back to the CoCo for conflict resolution and final calibration.
aerOS	REST API (via CAPIF)	Invoke	Receives deployment and orchestration requests for Security TF components, including vApp and nApp flavours, and executes approved security enforcement actions.
DataOps	REST API / Telemetry	Read	Provides operational and telemetry data used by the Security TF to evaluate feasibility and determine achievable trust levels.

Table 13 Security TF communication with the SAFE-6G components

4.7.4 UNIT TESTS

Security-UT1: vApp – HTTP – Route Mounting Verification

Objective	**To verify that the vApp HTTP server correctly mounts the expected route groups at startup, without exercising route logic or external dependencies. This test validates structural correctness of the vApp.**
Components	a. HTTP Server hosting vApp b. Route registration module(s) c. Application bootstrap (app)
Requirements	1. Application must instantiate successfully 2. Core route groups must be registered 3. No external services are required
Features to be tested	1. Application bootstrap 2. Route group mounting 3. Internal router state
Test Steps	1. Import the server application instance 2. Inspect registered routes 3. Assert presence of expected paths

Verification Checklist for Security-UT1:

Step	Description	Yes	No	Comments
1	App instantiated	☒	☐	
2	Health routes mounted	☒	☐	
3	DID routes mounted	☒	☐	
4	Key Management Service (KMS) routes mounted	☒	☐	

Execution Output:
This unit test validates the structural correctness of the vApp HTTP server during application startup. The test focuses exclusively on verifying that the application instance is successfully instantiated and that the expected route groups are correctly mounted and exposed by the internal router. The presence of the expected route prefixes (e.g., health, DID, and KMS routes) confirms that the routing configuration is correctly wired and that the vApp can be started in a valid operational state. This test does not validate functional correctness of the individual endpoints; such behavior is covered by separate unit or integration tests where applicable. By validating route mounting independently of runtime behavior, this test provides early detection of configuration or wiring errors that could otherwise prevent the vApp from starting correctly or exposing its required interfaces.

```
========================================= test session starts =========================================
✓ Low vApp – HTTP > General > [integration] should have mounted the Veramo agent [3.37ms]
✓ Low vApp – HTTP > DID Routes > [unit] should mount the /did routes [0.45ms]
✓ Low vApp – HTTP > KMS Routes > [unit] should mount the /kms routes [0.25ms]
========================================== test session ends ==========================================
```

Figure 68 Security-UT1 output

Security-UT2: vApp – SSI – Plugin Mounting Verification

Objective	To verify that the SSI agent mounts the required plugins at initialization time. This test validates agent capability exposure, not behaviour.
Components	a. SSI Agent b. Identity Plugin c. DID Resolver Plugin (Fabric) d. KMS Plugin
Requirements	1. Agent must initialize successfully 2. Required plugins must be registered 3. Plugin methods must be callable
Features to be tested	1. Plugin registration 2. Agent method exposure 3. Plugin wiring
Test Steps	4. Import initialized agent 5. Inspect available methods 6. Assert presence of plugin APIs

Verification Checklist for Security-UT2:

Step	Description	Yes	No	Comments
1	Agent initializes	☒	☐	
2	Identity plugin mounted	☒	☐	
3	Fabric DID resolver mounted	☒	☐	

Execution Output:
This unit test verifies that the SSI agent initializes correctly and exposes the required identity-related capabilities through its plugin system. The test focuses on structural and wiring correctness by ensuring that all mandatory plugins are registered and that their corresponding interfaces are accessible through the agent instance. The validation is performed by importing the initialized agent, inspecting the exposed methods, and asserting the availability of the expected plugin APIs. No

behavioral logic, protocol execution, or external interactions are exercised. Successful execution of this test confirms that the agent is correctly composed, that identity and resolver capabilities are available to higher-level components, and that the SSI layer can be safely relied upon as a foundational building block for subsequent trust and security operations.

```
========================================== test session starts ==========================================
✓ vApp – SSI  > Identity Plugin Tests > [unit] mounted the Identity plugin on Veramo [0.14ms]
✓ vApp – SSI > Identity Plugin Tests > [unit] returns the identity of the agent [0.26ms]
✓ vApp – SSI > Fabric Resolver Tests > [unit] mounted the custom Fabric resolver on Veramo [0.15ms]
✓ vApp – SSI > Fabric Resolver Tests > [unit] resolves a valid DID when gateway returns a document [1.52ms]
========================================== test session ends ==========================================
```

Figure 69 Security-UT2 output

Security-UT3: vApp – SSI – Database Initialization

Objective	**To verify that the SSI agent database layer initializes correctly under valid configuration and fails deterministically under invalid configuration. This test validates persistence layer correctness and defensive configuration handling, independent of higher-level agent logic.**
Components	a. SSI Agent b. Database Abstraction Layer c. SQLite (in-memory) Backend
Requirements	1. The agent must initialize an in-memory database successfully 2. Invalid database configuration must raise a deterministic error 3. No external database service is required
Features to be tested	1. Database bootstrap 2. Configuration validation 3. Error propagation
Test Steps	1. Initialize the SSI agent with valid in-memory SQLite configuration 2. Verify successful database creation 3. Initialize the agent with an invalid database configuration 4. Verify that initialization fails with an error

Verification Checklist for Security-UT3:

Step	**Description**	**Yes**	**No**	**Comments**
1	In-memory SQLite database initializes	☒	☐	
2	Agent starts with valid config	☒	☐	
3	Invalid config throws error	☒	☐	

Execution Output:

This unit test validates the correct initialization and defensive behavior of the SSI agent's persistence layer. The test ensures that the agent can successfully bootstrap its internal database under a valid configuration and that misconfigurations are detected and handled deterministically at initialization time. The validation is performed by instantiating the agent with both valid and invalid database configurations and observing the resulting behavior. No higher-level agent logic, identity operations, or external services are involved. Successful execution confirms that the persistence layer is correctly wired, that configuration errors are not silently ignored, and that the agent fails fast and predictably when invalid settings are provided.

```
========================================= test session starts ==========================================
✓ vApp – SSI > Database Tests > [unit] initializes an in-memory SQLite database successfully [101.05ms]
✓ vApp – SSI > Database Tests > [unit] throws an error for invalid configuration [1.01ms]
========================================== test session ends ===========================================
```

Figure 70 Security-UT3 output

Security-UT4: Local AI Agent – Configuration & Runtime Safety

Objective	**To verify that the Local AI Agent initializes correctly under a valid configuration and that its core internal components are properly wired and accessible at startup. This test validates structural readiness of the agent independently of inference execution, messaging infrastructure, or external integrations.**
Components	a. Local AI Agent b. Agent Configuration c. Message Handler d. Model Access
Requirements	1. The agent must initialize successfully with a valid configuration 2. Core internal components must be available after initialization 3. No external services are required
Features to be tested	1. Agent bootstrap 2. Configuration loading 3. Internal component wiring
Test Steps	1. Import the Local AI Agent initialization module 2. Initialize the agent with a valid configuration 3. Inspect internal state and component availability 4. Verify successful startup without invoking inference or messaging

Verification Checklist for Security-UT4:

Step	Description	Yes	No	Comments
1	Agent initializes successfully	☒	☐	
2	Configuration loaded correctly	☒	☐	
3	Core components wired	☒	☐	
4	No external dependencies required	☒	☐	

Execution Output:

This unit test validates the structural correctness of the Local AI Agent during initialization. The test ensures that the agent can be instantiated with a valid configuration and that its essential internal components are properly wired and accessible immediately after startup. The validation is performed in isolation, without triggering inference logic, message consumption, or external service interactions. Successful execution confirms that the Local AI Agent reaches a stable and operational baseline state, providing a reliable foundation for subsequent trust evaluation, inference processing, and action-triggering workflows.

```
==================================== test session starts ====================================
tests/unit/test_config.py::test_config_has_global_keys PASSED
tests/unit/test_config.py::test_env_is_str PASSED
tests/unit/test_config.py::test_port_is_int PASSED
tests/unit/test_config.py::test_secret_key_is_str PASSED
tests/unit/test_config.py::test_log_level_is_str PASSED
tests/unit/test_config.py::test_config_has_minio_keys PASSED
tests/unit/test_config.py::test_config_has_handler_keys PASSED
tests/unit/test_config.py::test_config_has_aeros_keys PASSED
==================================== test session ends ====================================
```

Figure 71 Security-UT4 output

Security-UT5: Local AI Agent – Model and Data Artifact Management

Objective	**To verify that the Local AI Agent correctly manages its internal model and data artifacts, including loading, caching, reuse, and controlled failure behavior. This test validates the correctness and determinism of the agent's model lifecycle handling, independent of inference execution or external storage systems.**
Components	a. Local AI Agent b. Model Registry c. Dataset Loader d. Internal Cache
Requirements	1. Model and dataset artifacts must be loadable under valid configuration 2. Previously loaded artifacts must be reused without redundant loading 3. Failures during artifact loading must be handled deterministically 4. No external storage service is required
Features to be tested	1. Model artifact loading 2. Dataset loading 3. In-memory caching 4. Error handling and fallback behavior
Test Steps	1. Initialize the Local AI Agent model management module 2. Load a valid model or dataset artifact 3. Verify that the artifact is stored and reused from internal state 4. Attempt to load an invalid or missing artifact 5. Verify that the error is handled gracefully

Verification Checklist for Security-UT5:

Step	Description	Yes	No	Comments
1	Model artifacts load successfully	☒	☐	
2	Dataset artifacts load successfully	☒	☐	
3	Artifact caching is applied	☒	☐	
4	Invalid artifact handled	☒	☐	

Execution Output:

This unit test validates the Local AI Agent's ability to manage model and dataset artifacts in a deterministic and fault-tolerant manner. The test ensures that artifacts can be successfully loaded when valid, that repeated access reuses cached instances to avoid redundant operations, and that loading failures are detected and handled predictably. The validation is performed without invoking inference logic or external storage services, focusing strictly on lifecycle management and internal

state consistency. Successful execution confirms that the agent's model and data handling mechanisms are robust and suitable for reliable operation within the trust evaluation pipeline.

```
========================================= test session starts ==========================================
tests/unit/test_models.py::test_fetch_model_file_loads_and_caches PASSED
tests/unit/test_models.py::test_fetch_model_file_uses_cache PASSED
tests/unit/test_models.py::test_fetch_model_file_failure PASSED
tests/unit/test_models.py::test_fetch_model_dataset_loads_and_caches PASSED
========================================== test session ends ===========================================
```

Figure 72 Security-UT5 output

Security-UT6: Local AI Agent – Messanging Input Handling and Validation

Objective	**To verify that the Local AI Agent safely ingests, validates, and processes incoming trust-related messages. This test validates defensive input handling, message parsing robustness, and safe behavior in the presence of malformed or incomplete inputs, independently of downstream inference or action-triggering logic**
Components	a. Local AI Agent b. Message Handler c. Input Parser and Validation logic d. Internal Request Dispatcher
Requirements	1. The agent must accept well-formed input messages 2. Malformed or invalid messages must be handled safely 3. No unhandled exceptions must propagate to the runtime 4. No external messaging infrastructure is required
Features to be tested	1. Message parsing 2. Input validation 3. Graceful error handling 4. Internal request routing
Test Steps	1. Initialize the Local AI Agent message handling component 2. Submit a valid trust-related message 3. Verify correct parsing and internal handling 4. Submit an invalid or malformed message 5. Verify that the agent handles the input safely without triggering unintended behavior

Verification Checklist for Security-UT6:

Step	Description	Yes	No	Comments
1	Valid message parsed successfully	☒	☐	
2	Missing or malformed input detected	☒	☐	
3	Invalid message handled safely	☒	☐	
4	No unintended side effects triggered	☒	☐	

Execution Output:

This unit test validates the robustness of the Local AI Agent's message handling logic by ensuring that incoming inputs are parsed and validated correctly and that malformed or incomplete messages do not lead to runtime failures or unintended state changes. The test is performed in isolation, without relying on external messaging systems or invoking inference or action-triggering logic. Successful execution confirms that the agent enforces defensive input handling and can safely operate in environments where message quality or structure cannot be guaranteed.

```
==================================== test session starts ====================================
tests/unit/test_kafka.py::test_handle_message_with_security_score PASSED
tests/unit/test_kafka.py::test_handle_message_without_security_score PASSED
tests/unit/test_kafka.py::test_handle_message_invalid_json PASSED
========================================= test session ends =================================
```

Figure 73 Security-UT6 output

Security-UT7: Local AI Agent – External Dependency Handling

Objective	To verify that the Local AI Agent safely integrates optional external dependencies and degrades gracefully when such dependencies are unavailable, disabled, or misconfigured. This test validates defensive design and fault isolation, ensuring that optional integrations do not compromise agent stability or core functionality.
Components	a. Local AI Agent b. External Minio Service Utility c. Internal Configuration
Requirements	1. Optional external dependencies must not be required for agent startup 2. Failures or unavailability of optional services must not cause agent crashes 3. Errors must be handled deterministically and logged appropriately 4. Core agent functionality must remain unaffected
Features to be tested	1. Conditional integration 2. External dependency invocation guards 3. Graceful failure handling 4. Isolation of optional services
Test Steps	1. Initialize the Local AI Agent with optional external integrations disabled 2. Verify successful agent startup and normal operation 3. Simulate failure or unavailability of an optional external dependency 4. Verify that the agent handles the failure without propagating errors or affecting core behavior

Verification Checklist for Security-UT7:

Step	Description	Yes	No	Comments
1	Agent starts without optional dependencies	☒	☐	
2	Optional integration guarded by configuration	☒	☐	
3	External failure handled gracefully	☒	☐	
4	Core agent behavior unaffected	☒	☐	

Execution Output:

This unit test validates the Local AI Agent's ability to operate reliably in environments where optional external services (e.g. Minio) may be unavailable, misconfigured, or intentionally disabled. The test ensures that such integrations are correctly guarded by configuration and that failures are handled in a controlled and isolated manner. The validation is performed without relying on external systems, focusing on error containment and resilience at the agent boundary. Successful execution confirms that optional dependencies do not introduce instability and that the agent maintains robust and predictable behavior under partial integration conditions

```
==================================== test session starts ====================================
tests/unit/test_minio.py::test_try_download_minio_disabled PASSED
tests/unit/test_minio.py::test_try_download_minio_success PASSED
tests/unit/test_minio.py::test_try_download_minio_s3_error PASSED
===================================== test session ends =====================================
```

Figure 74 Security-UT7 output

Security-UT8: vApp – HTTP – Core Initialization & Health Monitoring

Objective	To verify that the Low vApp HTTP server initializes correctly, mounts core dependencies and exposes health/status endpoints with proper error handling.
Components	a. HTTP Server b. SSI Agent (mounted dependency) c. HTTP Health Routes (/status)
Requirements	1. Server must start with Veramo agent mounted 2. Health endpoints must be exposed 3. Server must return meaningful status and errors
Features to be tested	1. Server bootstrap 2. Health route registration 3. Runtime health reporting 4. Error handling
Test Steps	1. Start HTTP server 2. Verify SSI agent is mounted 3. Call ***/status*** endpoint under normal and error conditions

Verification Checklist for Security-UT8:

Step	Description	Yes	No	Comments
1	SSI agent mounted on server init	☒	☐	
2	/status route registered	☒	☐	
3	Valid status response returned	☒	☐	
4	Unknown routes return errors	☒	☐	

Execution Output:

This unit test validates the correct initialization and runtime readiness of the vApp HTTP server by ensuring that core dependencies are mounted at startup and that health monitoring endpoints are exposed and operational. The test focuses on structural and runtime correctness by verifying that the server can start successfully with its required internal components attached, that the health/status route is properly registered, and that meaningful status information is returned during normal operation. Error handling is also exercised by accessing undefined routes to confirm that the server responds predictably to invalid requests. The test does not evaluate application logic or external integrations but confirms that the HTTP layer provides a reliable entry point and basic observability for higher-level services.

```
======================================== test session starts ========================================
✓ vApp - HTTP > General > [integration] should have mounted the Veramo agent [3.37ms]
✓ vApp - HTTP > Health Routes > [unit] should mount the GET /status route [12.58ms]
✓ vApp - HTTP > Health Routes > [integration] should respond with server status [41.41ms]
✓ vApp - HTTP > Health Routes > [integration] handles internal errors [3.83ms]
✓ vApp - HTTP > Health Routes > [e2e] should fetch a valid response from /status API  [16.20ms]
✓ vApp - HTTP > Health Routes > [e2e] should handle error from unknown API  [8.97ms]
========================================= test session ends =========================================
```

Figure 75 Security-UT8 output

Security-UT9: vApp – HTTP – DID Management API

Objective	To validate DID lifecycle management via HTTP server's REST API, including creation, resolution, aliasing, component access, and deletion.
Components	a) HTTP Server b) SSI DID Manager c) DID Resolver
Requirements	1. DID providers must be discoverable 2. DIDs must support aliases and components 3. Invalid inputs must be safely handled
Features to be tested	1. DID creation 2. DID resolution 3. Alias resolution 4. Component lookup 5. Error handling
Test Steps	1. Query DID providers 2. Create DID (valid/invalid cases) 3. Resolve DID and components 4. Delete DID

Verification Checklist for Security-UT9:

Step	Description	Yes	No	Comments
1	DID providers listed	☒	☐	
2	DID created successfully	☒	☐	
3	Alias-based resolution works	☒	☐	
4	Invalid DID handled correctly	☒	☐	

Execution Output:

This unit test validates the correctness and robustness of the vApp's HTTP-based DID management interface by exercising the full lifecycle of decentralized identifiers through the exposed REST API. The test verifies that DID providers are discoverable, that new DIDs can be created and resolved successfully, and that aliases and component references are handled consistently. Both valid and invalid input scenarios are evaluated to ensure that erroneous requests are detected and handled safely without affecting server stability. The test focuses on API-level behavior and error handling, without assessing cryptographic properties or external resolution mechanisms, and confirms that the HTTP layer provides reliable access to core DID management capabilities.

```
==================================== test session starts ====================================
✓ vApp – HTTP > DID Routes > [integration] should list available DID providers [11.65ms]
✓ vApp – HTTP > DID Routes > [integration] should list all DIDs [15.79ms]
✓ vApp – HTTP > DID Routes > [integration] should create a new DID [305.86ms]
✓ vApp – HTTP > DID Routes > [integration] should handle existing alias in DID creation [17.88ms]
✓ vApp – HTTP > DID Routes > [integration] should handle invalid provider in DID creation [13.25ms]
✓ vApp – HTTP > DID Routes > [integration] should resolve a DID [9.42ms]
✓ vApp – HTTP > DID Routes > [integration] should resolve a DID component [14.25ms]
✓ vApp – HTTP > DID Routes > [integration] should handle invalid DID for component lookup [0.94ms]
✓ vApp – HTTP > DID Routes > [integration] should handle invalid component of a DID [2.84ms]
✓ vApp – HTTP > DID Routes > [integration] should resolving DID using alias [20.46ms]
✓ vApp – HTTP > DID Routes > [integration] should handle resolving invalid DID [0.95ms]
✓ vApp – HTTP > DID Routes > [integration] should handle resolving an invalid DID alias [11.42ms]
✓ vApp – HTTP > DID Routes > [integration] should delete a DID [122.61ms]
✓ vApp – HTTP > DID Routes > [integration] should handle deleting an invalid DID [4.61ms]
==================================== test session ends ====================================
```

Figure 76 Security-UT9 output

Security-UT10: vApp – HTTP –KMS API

Objective	To ensure cryptographic key lifecycle operations are exposed correctly via HTTP server's KMS routes.
Components	a. HTTP Server b. SSI KMS c. Cryptographic Providers (Ed25519, X25519)
Requirements	1. Keys must be created, read, and deleted 2. Signing and encryption must function 3. Missing resources must fail gracefully
Features to be tested	1. Key generation 2. Signing (data & JWT) 3. Encryption / Decryption 4. Error handling
Test Steps	1. List KMS providers 2. Create keys 3. Sign payloads 4. Encrypt/decrypt data 5. Delete keys

Verification Checklist for Security-UT10:

Step	Description	Yes	No	Comments
1	Keys created successfully	☒	☐	
2	Payload signing works	☒	☐	
3	JWE encryption/decryption works	☒	☐	
4	Missing keys handled correctly	☒	☐	

Execution Output:

This unit test validates that the vApp's HTTP-based key management interface correctly exposes and supports cryptographic key lifecycle operations. The test verifies that keys can be created, accessed, and deleted through the exposed routes, and that core cryptographic operations such as signing and encryption/decryption execute successfully when valid inputs are provided. Error handling is also exercised by attempting operations on missing or invalid key references to ensure that failures are handled gracefully and do not impact server stability. The test focuses on API-level correctness and

robustness, without evaluating cryptographic strength or external interoperability, and confirms that the HTTP layer provides reliable access to essential key management capabilities required by higher-level security functions.

```
======================================== test session starts ========================================
✓ vApp – HTTP > KMS Routes > [integration] should list available KMS providers [13.88ms]
✓ vApp – HTTP > KMS Routes > [integration] should create a new Ed25519 key [40.43ms]
✓ vApp – HTTP > KMS Routes > [integration] should create an X25519 key for encryption [34.92ms]
✓ vApp – HTTP > KMS Routes > [integration] should read an existing key [5.70ms]
✓ vApp – HTTP > KMS Routes > [integration] should handle missing key [13.93ms]
✓ vApp – HTTP > KMS Routes > [integration] should sign a data payload [25.49ms]
✓ vApp – HTTP > KMS Routes > [integration] should sign a JWT payload [23.78ms]
✓ vApp – HTTP > KMS Routes > [integration] should encrypt data to JWE [36.58ms]
✓ vApp – HTTP > KMS Routes > [integration] should decrypt encrypted JWE [41.06ms]
✓ vApp – HTTP > KMS Routes > [integration] should delete a key [43.43ms]
✓ vApp – HTTP > KMS Routes > [integration] should handle deleting missing key [2.86ms]\
======================================== test session ends ========================================
```

Figure 77 Security-UT10 output

Security-UT11: vApp – SSI – DID Operations

Objective	To verify internal Veramo agent DID functionality independent of the REST layer.
Components	a. SSI Agent b. DID Manager c. DID Resolver
Requirements	1. DID providers must be accessible 2. Alias handling must be consistent 3. Import and deletion must work
Features to be tested	1. DID creation 2. Alias assignment 3. DID import 4. DID deletion
Test Steps	1. List DID providers 2. Create and resolve DID 3. Resolve by alias 4. Import DID with keys 5. Delete DID

Verification Checklist for Security-UT11:

Step	Description	Yes	No	Comments
1	DID created & resolved	☒	☐	
2	Alias resolution works	☒	☐	
3	DID import succeeds	☒	☐	
4	Non-existent DID fails safely	☒	☐	

Execution Output:

This unit test validates the correct operation of decentralized identifier management within the SSI agent, independently of any HTTP or external interfaces. The test exercises core DID lifecycle functionality by verifying that identifiers can be created, resolved, aliased, imported, and deleted through the agent's internal APIs. Both successful and failure scenarios are evaluated to ensure

consistent alias handling and safe behavior when non-existent identifiers are referenced. The test focuses on internal agent logic and state management, without involving REST-layer routing or external resolution services, and confirms that the SSI agent provides reliable and well-defined DID operations for use by higher-level components.

```
======================================== test session starts ========================================
✓ vApp – SSI > DID Tests > [integration] lists DID providers [0.46ms]
✓ vApp – SSI > DID Tests > [integration] lists all existing DIDs [41.39ms]
✓ vApp – SSI > DID Tests > [integration] creates and resolves a DID document [88.58ms]
✓ vApp – SSI > DID Tests > [integration] fails resolving a non-existent DID [0.48ms]
✓ vApp – SSI > DID Tests > [integration] creates a DID with alias and resolves it by alias [86.46ms]
✓ vApp – SSI > DID Tests > [integration] fails resolving unknown alias [11.22ms]
✓ vApp – SSI > DID Tests > [integration] resolves a DID component by fragment [57.78ms]
✓ vApp – SSI > DID Tests > [integration] sets a DID alias [100.57ms]
✓ vApp – SSI > DID Tests > [integration] deletes a DID [142.13ms]
✓ vApp – SSI  > DID Tests > [integration] imports a DID with keys [81.85ms]
======================================== test session ends ========================================
```

Figure 78 Security-UT11 output

Security-UT12: vApp – SSI – KMS Cryptographic Operations

Objective	To validate cryptographic operations directly through the SSI agent.
Components	a. SSI Agent b. KMS c. Crypto Providers
Requirements	1. Keys must be generated and deleted 2. Signing & encryption must work 3. Errors must be deterministic
Features to be tested	1. Signing 2. JWT creation 3. JWE encryption 4. Shared secret derivation
Test Steps	1. Create keys 2. Sign payloads 3. Encrypt/decrypt data 4. Generate shared secrets

Verification Checklist for Security-UT12:

Step	Description	Yes	No	Comments
1	Keys generated	☒	☐	
2	Data/JWT signing works	☒	☐	
3	Encryption/decryption works	☒	☐	
4	X25519 shared secret created	☒	☐	

Execution Output:

This unit test validates the correctness and reliability of cryptographic operations performed directly through the SSI agent's key management capabilities. The test exercises key lifecycle and cryptographic functionality by verifying that keys can be generated and removed, that data and token signing operations execute successfully, and that encryption, decryption, and shared secret derivation behave as expected. Both successful and failure scenarios are evaluated to ensure deterministic error handling and predictable behavior. The test is performed entirely through the agent's internal

interfaces, without relying on external services or HTTP routing, and confirms that the SSI agent provides a stable and secure cryptographic foundation for higher-level identity and trust operations.

```
======================================== test session starts ========================================
✓ vApp - SSI > KMS Tests > [integration] lists available KMS providers [0.50ms]
✓ vApp - SSI > KMS Tests > [integration] creates a new Ed25519 key [32.03ms]
✓ vApp - SSI > KMS Tests > [integration] fails with invalid key type [0.97ms]
✓ vApp - SSI > KMS Tests > [integration] reads an existing key [35.58ms]
✓ vApp - SSI  > KMS Tests > [integration] returns 404 for a missing key [9.84ms]
✓ vApp - SSI  > KMS Tests > [integration] deletes a key successfully [47.23ms]
✓ vApp - SSI  > KMS Tests > [integration] returns internal error for non-existing key [1.79ms]
✓ vApp - SSI  > KMS Tests > [integration] signs data using an Ed25519 key [38.31ms]
✓ vApp - SSI  > KMS Tests > [integration] signs a JWT payload [40.10ms]
✓ vApp - SSI  > KMS Tests > [integration] encrypts and decrypts a JWE successfully [54.23ms]
✓ vApp - SSI  > KMS Tests > [integration] creates a shared secret using X25519 [65.98ms]
========================================= test session ends =========================================
```

Figure 79 Security-UT12 output

Security-UT13: Local AI Agent – Inference & Decision Logic

Objective	To verify that the Local AI Agent correctly ingests metrics and trust inputs, executes inference, and produces deterministic trust decisions.
Components	a. Local AI Agent b. Inference pipeline c. Input Validation and Preprocessing d. Model and Artifact Manager
Requirements	1. The agent must accept structured input metrics 2. Inference must execute successfully using loaded models 3. The agent must return a deterministic trust result 4. Invalid or incomplete inputs must be handled safely 5. No external orchestration services are required
Features to be tested	1. Model loading and reuse 2. Input preprocessing and validation 3. Inference execution 4. Error handling during inference
Test Steps	1. Initialize the Local AI Agent with valid configuration 2. Load required model and preprocessing artifacts 3. Submit a valid input payload to the inference pipeline 4. Verify that a trust result is produced 5. Submit malformed or edge-case input and observe safe behavior

Verification Checklist for Security-UT13:

Step	Description	Yes	No	Comments
1	Model loads successfully	☒	☐	
2	Inference returns trust result	☒	☐	
3	Output schema is correct	☒	☐	
4	Invalid input handled gracefully	☒	☐	

Execution Output:
This integration test validates the Local AI Agent's ability to perform inference as a cohesive internal workflow, from input ingestion through preprocessing and model execution to trust decision output.

The test leverages real model artifacts and representative input data to exercise the full inference path while remaining isolated from external systems and deployment mechanisms. Both valid and adverse inputs are evaluated to ensure deterministic behavior and robust error handling. Successful execution confirms that the agent's inference and decision logic operate correctly as an integrated subsystem, providing reliable trust evaluations to downstream components.

```
========================================= test session starts ==========================================
tests/unit/test_classification.py::test_prediction_rows_schema PASSED
tests/unit/test_classification.py::test_make_prediction_valid PASSED
tests/unit/test_classification.py::test_make_prediction_invalid PASSED
tests/unit/test_classification.py::test_make_prediction_handles_outliers PASSED
tests/unit/test_classification.py::test_make_prediction_missing_artifacts_raises PASSED
tests/unit/test_classification.py::test_make_prediction_multiple_rows PASSED
========================================== test session ends ===========================================
```

Figure 80 Security-UT13 output

4.7.5 Integration Tests

Security-IT1: Local AI Agent – Trust Decisions & Action Triggering

Objective	**To verify that trust decisions produced by the Local AI Agent are evaluated deterministically against predefined thresholds and that the correct internal actions are triggered accordingly. This integration test validates E2E trust decision handling within the agent, excluding external deployment success or orchestration behavior.**
Components	a. Local AI Agent b. Trust Score Consumer c. Action Dispatcher (internal)
Requirements	1. Trust thresholds must be configurable 2. Trust decisions must be evaluated deterministically 3. Correct actions must be triggered for each trust level 4. No unintended actions must be triggered for low trust values 5. External orchestration systems are not required
Features to be tested	1. Trust threshold evaluation 2. Low / medium trust classification 3. Action triggering logic 4. Boundary condition handling
Test Steps	1. Initialize the Local AI Agent with a valid configuration 2. Submit a trust input producing a low trust decision 3. Verify that no action is triggered 4. Submit a trust input producing a medium or higher trust decision 5. Verify that the corresponding internal action is triggered 6. Repeat the test at the trust threshold boundary to verify determinism

Verification Checklist for Security-IT1:

Step	Description	Yes	No	Comments
1	Low trust detected correctly	☒	☐	
2	Medium trust detected correctly	☒	☐	
3	Correct action triggered for medium trust	☒	☐	
4	No action triggered for low trust	☒	☐	
5	Threshold boundary handled deterministically	☒		

Execution Output:
This integration test validates the Local AI Agent's ability to translate trust decisions into deterministic internal actions based on configurable thresholds. The test exercises the complete trust-handling workflow, from trust input ingestion through decision evaluation to action triggering, while remaining isolated from external orchestration systems. Both low and medium trust scenarios, including boundary conditions, are evaluated to ensure predictable behavior and to prevent unintended actions. Successful execution confirms that the agent enforces trust policies consistently and that its decision-to-action logic is reliable and suitable for integration within the broader security framework.

```
========================================= test session starts ==========================================
tests/unit/test_trust_handler.py::test_handle_request_triggers_medium_flavor PASSED
tests/unit/test_trust_handler.py::test_handle_request_low_trust_no_action PASSED
tests/unit/test_trust_handler.py::test_handle_request_threshold_boundary PASSED
========================================== test session ends ===========================================
```

Figure 81 Security-IT1 output

Security-IT2: Local AI Agent – Gitlab CI/CD Model Serving Pipeline

Objective	**Verify that the Local AI Agent's model serving pipeline is correctly integrated into the CI and continuous deployment workflow and that deployed models can be loaded, initialized, and queried successfully after automated deployment. This integration test validates E2E model serving readiness as part of the CI/CD process, rather than local agent behavior.**
Components	a. Model serving service b. Model loading and initialization logic c. Inference request handling pipeline d. CI/CD execution environment
Requirements	1. The model serving service must start successfully after deployment 2. The trained model artifact must be loaded correctly 3. Valid inference requests must be processed successfully 4. Invalid or malformed requests must be rejected deterministically 5. The test must execute automatically as part of the CI/CD pipeline
Features to be tested	1. Model artifact loading during deployment 2. Inference service availability 3. Request preprocessing and validation 4. Inference execution and response formatting 5. Error handling for invalid inputs
Test Steps	1. Model artifact loading during deployment 2. Inference service availability 3. Request preprocessing and validation 4. Inference execution and response formatting 5. Error handling for invalid inputs

Verification Checklist for Security-IT2:

Step	Description	Yes	No	Comments
1	CI/CD pipeline triggers model deployment	☒	☐	
2	Model loads successfully after deployment	☒	☐	
3	Model loads successfully after deployment	☒	☐	
4	Valid inference request succeeds	☒	☐	
5	Invalid input rejected deterministically	☒		

Execution Output:
This integration test validates the Local AI Agent's model serving pipeline as part of the automated CI/CD workflow. The test is executed automatically upon code changes to a designated branch and verifies that the deployed inference service correctly loads the trained model artifact, exposes an operational prediction endpoint, and handles inference requests as expected. Both valid and invalid input scenarios are exercised to confirm correct request preprocessing, inference execution, and deterministic error handling. The test does not assess model accuracy or deployment orchestration success beyond service availability. Successful execution confirms that the model serving pipeline is correctly integrated into the CI/CD process and that deployed models are ready for operational use.

```
========================================= test session starts ==========================================
tests/test_integration.py::test_valid_data  PASSED
tests/test_integration.py::test_invalid_data  PASSED
tests/test_integration.py::test_missing_field  PASSED
========================================== test session ends ===========================================
```

Figure 82 Security-IT2 output

4.7.6 Next Steps

To enhance the resilience, accuracy, and operational efficiency of the Security Function, the subsequent testing and validation activities are scheduled for the next development phase:

- Expansion of Unit Test Coverage:
  We will enhance the current unit test suite to comprehensively cover the internal decision logic of the Local AI Agent, focusing specifically on trust threshold evaluation, feature validation, and the deterministic mapping between inferred trust levels (Low / Medium / High) and corresponding enforcement actions.

- Integration and E2E Validation:

A unified E2E validation process will be established to assess the Security Function as a black-box component operating within the broader SAFE-6G system. The validation will span all relevant architectural layers and communication planes, covering interactions from service-level entry points at the vApp layer, through the Local AI Agent, and up to enforcement actions issued towards network-facing nApps.

The E2E tests will replicate a complete trust evaluation cycle, beginning with the injection of a security-related trust input (e.g., metrics, trust intent, or contextual signals) and finalizing with a concrete enforcement decision. Validation focuses on correct message propagation, API interactions, and consistency of side effects across components. This includes verifying that enforcement actions are selected according to the assessed trust level and that failure scenarios, such as unavailable AI Agent or nApp endpoints, are handled gracefully without breaking errors.

Integration is being considered across all relevant SAFE-6G communication planes, including interactions with core network and platform components via standardized interfaces (e.g., CAPIF) and automated deployment and lifecycle management through aerOS. While this deliverable emphasizes functional correctness and integration readiness of the Security Function, the defined E2E validation

framework explicitly supports progressive integration with the wider SAFE-6G ecosystem in subsequent phases and pilot deployments.

4.8 Privacy Trust Function

4.8.1 Overview

The Privacy Function is a component that enforces requested privacy levels in 6G networks through automated, AI-driven decision making and execution. The nLoTw is provided by the CoCo and is propagated with a unique execution identification document (ID) for E2E traceability. The AI Agent interprets this numeric value and maps it to a predefined privacy profile. Based on this profile, the AI Agent selects the appropriate vApp and requests its deployment. The vApp encapsulates the enforcement logic for the selected privacy level and is deployed in a fully containerized manner, supporting K8s environments. The vApp activates one or more nApps that translate high-level privacy intents into concrete network policies, such as Data Network Selection and Slice Config nApp applied through the 5G core. These components use the Configuration Manager to load their config and then apply their changes. All components communicate asynchronously over Kafka, using the execution ID to correlate actions and outcomes. The system continuously monitors enforcement results and can dynamically reconfigure policies or escalate actions if the requested privacy level is not maintained. The overall design is modular, extensible, and cloud-native, enabling scalable and adaptable privacy enforcement across current and future mobile networks.

4.8.2 Architecture

The internal operation of the Privacy Function and its interaction with other SAFE-6G components is shown in the sequence of events in Figure 83; the CoCo sends the nLoTw to the AI Agent over the event stream (Kafka). The AI Agent maps the numeric value to a semantic privacy tier (no/low/medium/high), selects the matching vApp and the set of required nApps. The chosen vApp then composes concrete privacy policies and triggers nApps to apply them by calling 5G core/control-plane APIs (e.g., DNN and slice configuration). Each component reports status and logs back over Kafka using a correlated execution_id; the AI Agent aggregates these results, verifies whether the applied policies meet the requested privacy level, and if necessary, issues corrective actions (reconfigure nApps, escalate to a stronger vApp, or roll back changes). Finally, the AI Agent returns the outcome and audit log to the CoCo for authorization and record-keeping.

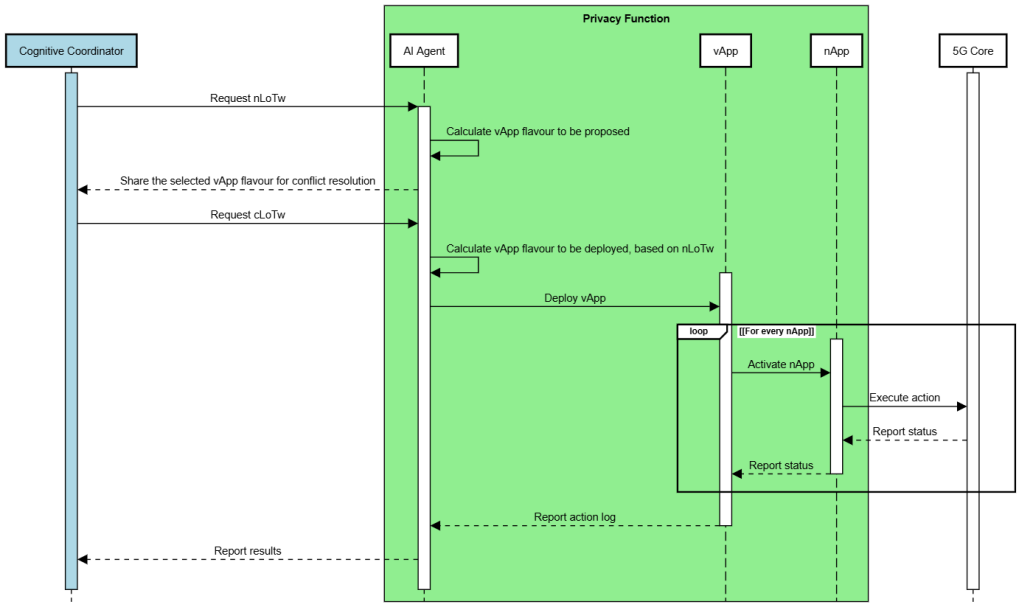


Figure 83 Privacy Trust Function Sequence Diagram

4.8.3 Integration with other Components

The Privacy TF interfaces with other components in 2 ways: 1. Through the message broker, 2. Through APIs.

Component	Protocol (API/Kafka)	Action (Read/Publish/Subscribe)	Details
CoCo	Kafka	Subscribe	Receives cLoTw from CoCo
CoCo	Kafka	Publish	Sends back executed report of executed actions and status
5G core	HTTP	Send	Sends request to the API to apply configuration changes

Table 14 Privacy TF communication with the SAFE-6G components

4.8.4 Unit Tests

Privacy-UT1: AI Agent Privacy Level Selection

Objective	To verify that the AI Agent correctly maps numerical privacy levels to the appropriate vApp type (Low, Medium, High).
Components	a. AI Agent Decision Logic
Requirements	Privacy levels 0-> No, 1-60 -> Low, 61-80 -> Medium, 81-100 -> High
Features to be tested	Privacy Level Decision Logic
Test Steps	1. Mock the Kafka Connector 2. Call handle_vapp_selection with privacy levels 60, 61, 80, and 81. 3. Verify the returned component name matches the expected vApp type.

Verification Checklist for Privacy -UT1:

Step	Description	Yes	No	Comments
1	Level 60 maps to "low-privacy-vapp",	☒	☐	

2	Level 61 maps to "medium-privacy-vapp"	☒	☐	
3	Level 81 maps to "high-privacy-vapp"	☒	☐	

Privacy-UT2: AI Agent Invalid Input Handling

Objective	To ensure that the AI Agent returns an error response when receiving invalid privacy level inputs.
Components	a. AI Agent Input Validation
Requirements	System must handle non-integer or out-of-bounds inputs gracefully.
Features to be tested	Input Validation
Test Steps	1. handle_vapp_selection with invalid inputs (-1, 101, "not-a-number"). 2. Check that the response dictionary contains an "error" key.

Verification Checklist for Privacy-UT2:

Step	Description	Yes	No	Comments
1	Negative numbers return error	☒	☐	
2	Numbers > 100 return error	☒	☐	
3	Non-numeric strings return error	☒	☐	

Privacy-UT3: AI Agent Execution Error

Objective	To verify that the execute_vapp method raises a ValueError if an unknown privacy string is passed internally
Components	a. AI Agent Execution Logic
Requirements	Internal methods should fail fast on invalid state
Features to be tested	Error Handling
Test Steps	1. Call execute_vapp directly with "bad-level". 2. Assert that ValueError is raised.

Verification Checklist for Privacy-UT3:

Step	Description	Yes	No	Comments
1	ValueError raised for invalid level	☒	☐	

Privacy-UT4: Results Message Validation

Objective	To test that the ResultsMessage Pydantic model accepts valid payloads and rejects extra fields.
Components	a. Messaging System (Pydantic Models)
Requirements	Strict schema validation for inter-service communication
Features to be tested	Schema Validation
Test Steps	1. Create a valid dictionary with success, log, execution_id, etc. 2. Validate it against ResultsMessage. 3. Add an extra field "unexpected" and verify ValidationError is raised.

Verification Checklist for Privacy-UT4:

Step	Description	Yes	No	Comments
1	Valid payload accepted	☒	☐	

2	Extra fields rejected	☒	☐	

Privacy-UT5: Results Message Type Enforcement

Objective	To ensure that ResultsMessage enforces specific allowed values for component_type.
Components	a. Messaging System
Requirements	component_type must be one of the allowed literals (e.g., "vapp").
Features to be tested	Enum/Literal Validation
Test Steps	1. Create a payload with component_type="other". 2. Attempt validation and expect ValidationError.

Verification Checklist for Privacy-UT5:

Step	Description	Yes	No	Comments
1	Invalid component_type rejected	☐	☒	

Privacy-UT6: vApp Selection Message Validation

Objective	To verify that VappSelectionRequestMessage correctly validates the privacy_level field.
Components	a. Messaging System
Requirements	privacy_level must be an integer.
Features to be tested	Type Validation
Test Steps	1. Validate payload {"privacy_level": 45}. 2. Validate payload {"privacy_level": "bad"} and expect failure.

Verification Checklist for Privacy-UT6:

Step	Description	Yes	No	Comments
1	Integer level accepted	☒	☐	
2	String level rejected	☒	☐	

Privacy-UT7: Action Message Defaults

Objective	To verify that ActionMessage automatically generates an execution_id if one is not provided
Components	a. Messaging System
Requirements	Every action must have a unique tracking ID.
Features to be tested	Default Field Generation
Test Steps	1. Create an ActionMessage without an execution_id. 2. Verify execution_id exists and is a non-empty string.

Verification Checklist for Privacy-UT7:

Step	Description	Yes	No	Comments
1	execution_id auto-generated	☒	☐	

Privacy-UT8: Settings Configuration

Objective	To ensure that the settings.toml file is loadable and contains critical Kafka topic configurations

Components	a. Configuration Manager
Requirements	Configuration file must exist and define topic names.
Features to be tested	Config Loading
Test Steps	1. Load settings.toml. 2. Check for presence of keys "dev.aiagent.requests" or "dev.containermanager.requests".

Verification Checklist for Privacy-UT8:

Step	Description	Yes	No	Comments
1	Critical topics present in config	X		

Privacy-UT9: Slice Payload Creation

Objective	To verify that the Slice Config nApp correctly constructs the JSON payload for the 5G Core API.
Components	a. Slice Config nApp
Requirements	Payload must match 5G Core API spec (nested structure).
Features to be tested	Payload Construction
Test Steps	1. Call create_slice_payload with specific parameters (throughput, max UEs). 2. Assert nested keys like NetworkSliceSubnet.RANSlice SubnetProfile.resourceSharingLevel match inputs.

Verification Checklist for Privacy-UT9:

Step	Description	Yes	No	Comments
1	Nested JSON structure correct	☒	☐	
2	Values mapped correctly	☒	☐	

Execution Output:

The unit tests are designed to validate the correctness, robustness, and safety of the Privacy TF's internal logic in isolation. They ensure that privacy-level interpretation, message validation, configuration handling, and payload construction behave deterministically and fail safely under invalid conditions. This reduces the risk of incorrect privacy enforcement and guarantees reliable behavior before components are composed into an end-to-end system.

At the foundation of this testing strategy is Privacy-UT1, which addresses privacy level selection by ensuring deterministic and correct mapping from numerical privacy levels to semantic privacy tiers and corresponding vApps. This test establishes that privacy classifications are interpreted consistently across all invocations, eliminating ambiguity in privacy tier assignment and creating a reliable baseline for all downstream operations. Building upon this foundation, Privacy-UT2 validates invalid input handling by verifying that malformed, non-numeric, or out-of-range inputs are rejected early, preventing undefined enforcement behavior. By catching invalid inputs at the boundary before they propagate further, this test mitigates the risk of downstream components operating on corrupted or nonsensical privacy directives. Once inputs are validated, Privacy-UT3 confirms that execution error handling fails fast when encountering invalid internal state, improving system debuggability and safety. Rather than allowing the system to propagate errors silently through multiple layers, this test

ensures that internal inconsistencies surface immediately and provide actionable diagnostic information for rapid issue resolution. With correct execution flow established, the system must ensure message integrity. Privacy-UT4 enforces strict schema validation for result messages, preventing schema drift and unexpected fields in inter-service communication. This validation step protects against subtle compatibility issues that arise when schemas diverge between producer and consumer, ensuring reliable integration with downstream systems. Complementing schema validation, Privacy-UT5 guarantees that only explicitly allowed component types are propagated, preserving semantic consistency across services. By enforcing type constraints at the message level, this test prevents inadvertent type confusion or unsupported component classes from traversing service boundaries, maintaining the integrity of the information flow. This type-safety discipline extends upstream as well. Privacy-UT6 verifies strong type enforcement for incoming privacy-level requests to avoid runtime casting or interpretation errors. This upstream validation complements the downstream type enforcement validated in Privacy-UT5, creating a defense-in-depth approach to type safety that protects the system at multiple boundary points. To enable observability throughout execution, Privacy-UT7 ensures automatic generation of execution identifiers, enabling end-to-end traceability without relying on external components. This self-contained identifier generation reduces coupling to upstream systems and guarantees that every execution instance can be tracked independently, facilitating debugging and audit trails. Before the system can execute reliably, Privacy-UT8 confirms that required Kafka topics and configuration parameters are present before runtime execution. This startup validation prevents silent failures caused by misconfigured deployment environments and ensures that all runtime dependencies are satisfied, catching environmental issues early rather than during operation. Finally, Privacy-UT9 validates correct construction of nested 5G Core API payloads to prevent misconfiguration at the network control plane. By verifying payload structure in isolation, this test catches integration defects before they propagate to network control plane APIs where debugging becomes significantly more complex, ensuring that the Privacy TF correctly translates privacy requirements into valid network configuration directives.

Below we provide the console output of the unit test suite's execution. It is important to note that even though we have defined 9 tests above, the output mentions 11 passed tests. That happens because one of the 9 defined ones is executed 3 times with different inputs for better coverage.

```
============================================= test session starts ==============================================
platform linux -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0 -- /home/ubuntu/safe6g/privacy-function/venv/bin/python
cachedir: .pytest_cache
rootdir: /home/ubuntu/safe6g/privacy-function
configfile: pytest.ini
plugins: mock-3.14.1
collected 11 items

tests/unit/test_ai_agent.py::test_handle_vapp_selection_thresholds_low_medium_high PASSED                [  9%]
tests/unit/test_ai_agent.py::test_handle_vapp_selection_invalid_input_returns_error[-1] PASSED           [ 18%]
tests/unit/test_ai_agent.py::test_handle_vapp_selection_invalid_input_returns_error[101] PASSED          [ 27%]
tests/unit/test_ai_agent.py::test_handle_vapp_selection_invalid_input_returns_error[not-a-number] PASSED [ 36%]
tests/unit/test_ai_agent.py::test_execute_vapp_invalid_privacy_level_raises PASSED                       [ 45%]
tests/unit/test_messages.py::test_results_message_accepts_valid_payload_and_rejects_extras PASSED        [ 54%]
tests/unit/test_messages.py::test_results_message_component_type_literal_enforced PASSED                 [ 63%]
tests/unit/test_messages.py::test_vapp_selection_request_and_response_validation PASSED                  [ 72%]
tests/unit/test_messages.py::test_action_message_defaults_and_validation PASSED                          [ 81%]
tests/unit/test_config.py::test_settings_contains_expected_topics PASSED                                 [ 90%]
tests/unit/napps/test_slice_config_nApp.py::test_create_slice_payload_structure PASSED                   [100%]
============================================== 11 passed in 0.37s ==============================================
```

Figure 84 Privacy Unit tests Outputs

4.8.5 Integration Tests

Privacy-IT1: Kafka Roundtrip

Objective	To test the E2E capability to create a topic, produce a message, and consume it using the project's infrastructure.
Components	a. Kafka Broker b. Kafka Producer c. Kafka Consumer d. Kafka
Requirements	Working Kafka environment defined in KAFKA_BOOTSTRAP_SERVERS.
Features to be tested	Messaging Infrastructure
Test Steps	1. Create a unique ephemeral topic. 2. Produce a VappSelectionRequestMessage (privacy_level=42). 3. Consume the message from the topic. 4. Validate the consumed message matches the sent model. 5. Validate the consumed message matches the sent model.

Verification Checklist for Privacy-IT1:

Step	Description	Yes	No	Comments
1	Topic created successfully	☒	☐	
2	Message produced without error	☒	☐	
3	Message consumed and validated	☒	☐	
4	Topic cleaned up	☒	☐	

Privacy-IT2: K8s Manager Live Pod Lifecycle

Objective	To verify that the K8s Manager can communicate with a live cluster to create, monitor, and delete a real Pod.
Components	a. K8s Manager b. K8s Cluster (Local or Remote)
Requirements	Active Kubeconfig and permissions to manage pods in the default namespace.
Features to be tested	Container Orchestration
Test Steps	1. Instantiate K8s Manager. 2. Request creation of a "busybox" pod with specific env vars. 3. Poll the cluster until the pod reaches "Running" or "Succeeded" state. 4. Delete the pod via the manager. 5. Verify the pod is removed from the cluster.

Verification Checklist for Privacy-IT2:

Step	Description	Yes	No	Comments
1	Manager connected to cluster	☒	☐	
2	Pod created and reached Running state	☒	☐	
3	Pod deleted successfully	☒	☐	

Execution Output:
The integration tests validate the correct interaction of the Privacy TF with its runtime infrastructure, including messaging and orchestration systems. They demonstrate that the system operates correctly in realistic environments and that individual components interoperate as intended when deployed.

Integration Test–Level Motivation

1. **Privacy-IT1 (Kafka Roundtrip)**
   Verifies reliable end-to-end message production, consumption, and validation in a live Kafka environment.
2. **Privacy-IT2 (K8s Manager Live Pod Lifecycle)**
   Confirms that the Container Manager can deploy, monitor, and clean up real workloads on an active Kubernetes cluster.

Below we have pasted the console output for the integration test suite's execution.

```
============================================================ test session starts =============================================
platform linux -- Python 3.8.10, pytest-8.3.5, pluggy-1.5.0 -- /home/ubuntu/safe6g/privacy-function/venv/bin/python
cachedir: .pytest_cache
rootdir: /home/ubuntu/safe6g/privacy-function
configfile: pytest.ini
plugins: mock-3.14.1

collected 2 items
tests/integration/test_kafka_roundtrip.py::test_kafka_produce_consume_roundtrip PASSED                                [ 50%]
tests/integration/test_kubernetes_manager_live.py::test_kubernetes_manager_create_run_delete_pod PASSED              [100%
============================================================ 2 passed in 43.98s ==============================================
```

Figure 85 Privacy Integration Test Outputs

4.8.6 Next Steps

To further ensure the robustness, reliability, and production-readiness of the Privacy Function, the following testing activities are planned for the immediate future:

1. Expansion of Unit Test Coverage:
   a. We will integrate the remaining suite of unit tests that are currently in a staging phase. This includes comprehensive coverage for the dnn_config_nApp, specifically focusing on quality of service (QoS) parameter validation and preemption capability logic.
   b. Additional test cases for the low_privacy_vapp will be added to verify behavior under high-load configuration requests and to ensure correct fallback mechanisms when resource sharing levels are at their maximum.
   c. We will also formalize tests for the ContainerManager's internal state handling to ensure it gracefully recovers from transient K8s API failures.
2. E2E Validation:

a) We will implement a full-cycle E2E test suite that treats the system as a black box. These tests will simulate a complete user journey: injecting a privacy_level request into the entry Kafka topic and polling the final output topic for the completion signal.

b) The E2E validation will verify not just the successful message flow, but also the side effects: ensuring that the correct "mock" API calls were made to the 5G Core (verifying the correct slice parameters were applied) and that the K8s pods were spun up and torn down in the correct order.

4.9 Resilience Trust Function

4.9.1 Overview

The Resilience TF in SAFE-6G represents the network's ability to automatically detect, adapt to, and address faults or attacks, thereby allowing for graceful degradation and rapid recovery. This approach relies on intent-based, AI-driven operations and uses a sophisticated intent ontology grounded in knowledge management and semantic modeling. The Resilience TF integrates several ML models for monitoring cellular network status, predicting QoS, and detecting anomalies, all within an orchestrated pipeline that uses rule-based actions. These actions may include prioritizing or rerouting network traffic, optimizing terminal subscription parameters such as 5G Quality Indicator (5QI), Allocation and Retention Policy (ARP), or Session and Service Continuity (SSC), and refining the allocation of NFs that support the terminal. Collectively, these techniques enable SAFE-6G to anticipate, absorb, and recover from faults or attacks while maintaining uninterrupted service.

4.9.2 Architecture

The Resilience TF is divided into three internal modules. The **Local AI Agent** is the cornerstone of the function. Its role is to interpret the level of trustworthiness request from the CoCo and combine it with real-time statistics and information read from the user-centric core and cloud-continuum DataOps to 1) run the AI models to predict what is the achievable level of resilience and 2) determine the Resilience Flavor to apply. It then activates the **vApp** which is in charge to enforce the Resilience Flavor. To do so, it runs a flavor specific AI Model that predicts which sets of rules are needed. Finally, the **nApp** is invoked to apply those rules.

The internal operation of the Resilience TF and its interaction with other SAFE-6G components is pictorially described in the sequence diagram of Figure 86. The typical Resilience TF mode of operation starts when it receives a message from the CoCo that includes the identifier of the UE whose Resilience must be guaranteed and the nLoTw to be enforced.

Firstly, the Local AI Agent retrieves the network statistics and information relevant for the UE, querying the User-Centric Core network and the DataOps APIs. With this information and the type of UE -smartphone or XR glasses- the achievable LoTw (aLoTw) AI predictor is run. The predicted value brings the flavor of Resilience to be applied -low, medium or high- and is fed back to the CoCo. There might be conflicts in those cases when the required level of Resilience cannot be guaranteed, for example if the XR glasses UE is connected in a location where the occupancy level of the user-centric network is so high that cannot bring the required nLoTw. The CoCo is in charge to resolve such conflicts.

Once the CoCo receives the feedback from all the five TFs and resolves the possible conflicts it might encounter, it provides the cLoTw and the flavor of Resilience that must be enforced by the Resilience TF. The Local AI uses this information to select and invoke the appropriate vApp. There are different vApps depending on the type of UE -smartphone or XR glasses- and the Resilience flavor -low, medium or high.

All the vApps share the same flow of operation but run different Rule Selection AI models, each one trained specifically for the type of UE and Resilience flavor, and generate the Rules needed to guarantee the level of resilience. Those Rules are classified into three groups:

- UE subscription optimization Rules. These rules optimize the subscription in the UDM NF to provide the required flavor of Resilience. It manages parameters like 5QI, ARP, SSC, etc.
- QoS optimization Rules, include rules to create sessions with specific levels of pre-agreed quality. It leverages the ASsessionWithQoS NEF features.
- UE-NF allocation optimization rules. These rules are used for those flavors that require a very high level of resilience. They modify the assignation of the UPF where the UE is anchored, to select a UPF that provides a better level of Resilience.

nApps are invoked by the vApps every time a set of Rules must be applied. The nApp is able to transform these Rules into the specific operations of the 6G Core that are required, including updates in the UDM, specific NEF calls or proprietary 6G Core API calls through the CMC API. nApp finally informs the CoCo about each and every executed operation, so that the knowledge base can be enlarged for future retraining processes.

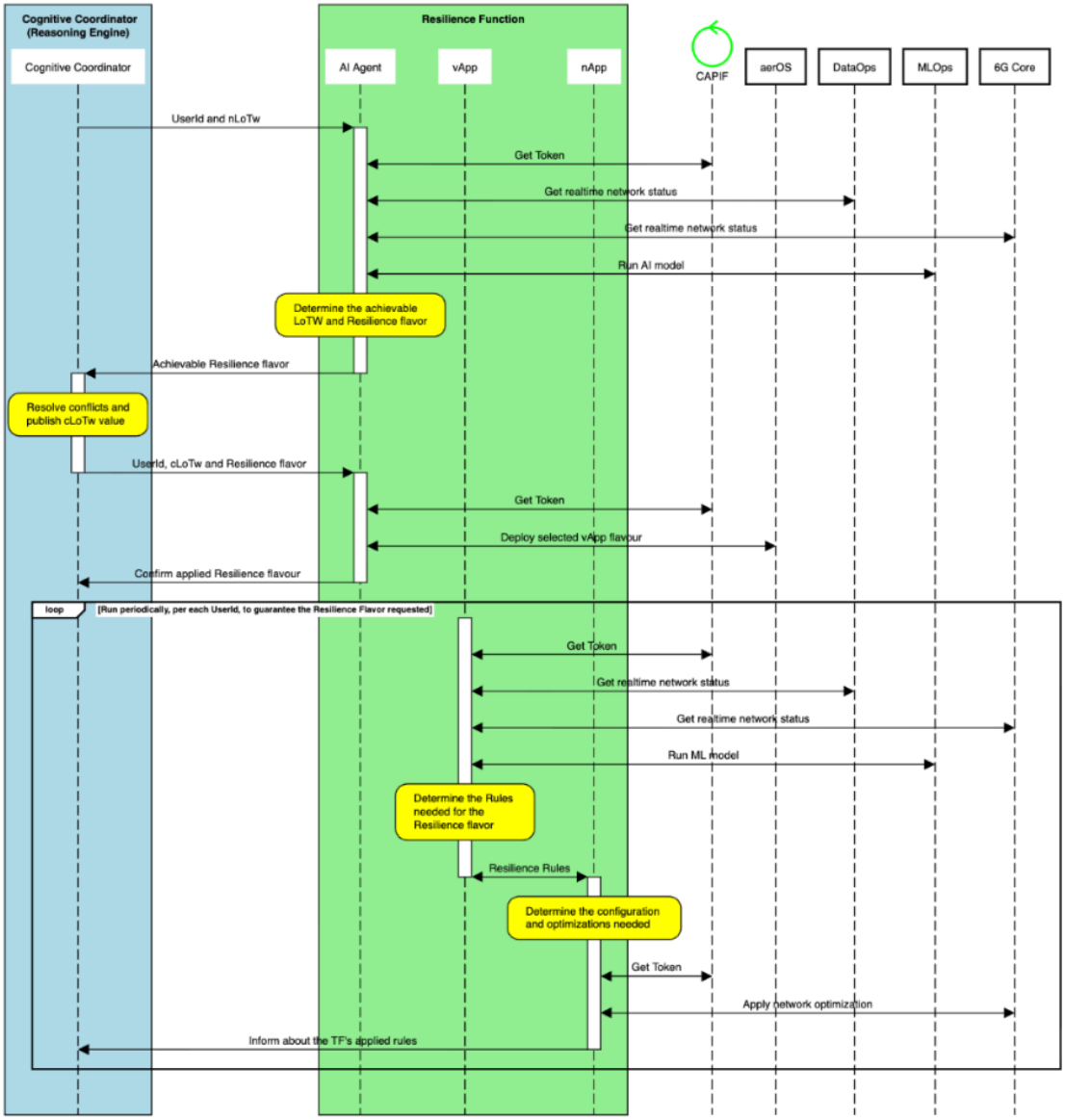


Figure 86 Resilience Trust Function Sequence Diagram

4.9.3 Integration with other Components

The Resilience TF interacts with the components of the SAFE-6G platform through the following interfaces:

Component	Protocol	Action	Details
CoCo	**Kafka**	Subscribe	Local AI Agent gets nLoTw from CoCo
		Publish	Local AI Agent publishes the achievable flavor of Resilience for the CoCo
		Subscribe	Local AI Agent gets cLoTw from CoCo
		Publish	Local AI Agent publishes the activated Resilience Flavour for CoCo
		Publish	nApp publishes executed rules and actions for CoCo

CAPIF	RESTful API	Read	Gets the auth tokens required to consume the APIs of other modules
MLOps	RESTful API	Read	Local AI Agent and vApp retrieves the more recently trained AI models
DataOps	RESTful API	Read	Local AI Agent and vApp get the required data to feed the models
aerOS	RESTful API	Write	Local AI Agent deploys and instantiates the required vApps
6G Core	CMC API	Read	Local AI Agent and vApp get the required data to feed the models
	CMC API	Write	nApp applies the network configuration required by the Rules
	NEF API	Write	nApp applies the network configuration required by the Rules

Table 15 Resilience TF communication with the SAFE-6G components

4.9.4 Unit Tests

This section outlines the unit tests developed to verify the algorithms and logic within the components of the Resilience Function. These tests are designed to ensure that the models yield consistent outcomes when executed under controlled conditions. Unit tests isolate the targeted software module for evaluation, employing mocks for any dependent modules required during testing.

4.9.4.1 AI Models

The Resilience TF AI models consume the status of the network, cloud continuum and use case applications relevant to the UE with the goal to predict what is the level of Resilience that can be provided for the UE. The following Unit Test simulates known context situations where the achievable level of Resilience is known and check whether the AI model prediction is correct.

Prerequisites
A mock-up is required to simulate the DataOps component, so that the AI Models can be fed with the data representative for the Unit Test.

Resilience-UT1 – Resilience Level and Flavour Predictor

Objective:	**To validate the Resilience Flavour Predictor AI Model**
Components	a. AI Model b. DataOps mock-up service
Requirements	N/A
Features to be tested	Output of the AI model is as expected
Test Steps	1. Activate AI Model with input context information 2. Check that the AI Model reads the relevant information from the DataOps mock-up 3. Get AI Model output

Test case 1: XR glasses, aLoTw low value and flavor low
Verification Checklist: the AI Model must indicate the following results

Step	**Description**	**Yes**	**No**	**Comments**
1	Achievable LoTw	☒	☐	

2	Resilience flavor: low	☒	☐	

Test case 2: XR glasses, aLoTw medium value and flavor medium
Verification Checklist: the AI Model must indicate the following results

Step	Description	Yes	No	Comments
1	Achievable LoTw	☒	☐	
2	Resilience flavor: medium	☒	☐	

Test case 3: XR glasses, aLoTw high value and flavor high
Verification Checklist: the AI Model must indicate the following results

Step	Description	Yes	No	Comments
1	Achievable LoTw	☒	☐	
2	Resilience flavour: high	☒	☐	

Test case 4: Smartphone, aLoTw low value and flavor low
Verification Checklist: the AI Model must indicate the following results

Step	Description	Yes	No	Comments
1	Achievable LoTw	☒	☐	
2	Resilience flavour: low	☒	☐	

Test case 5: Smartphone, aLoTw medium value and flavor medium
Verification Checklist: the AI Model must indicate the following results

Step	Description	Yes	No	Comments
1	Achievable LoTw	☒	☐	
2	Resilience flavor: medium	☒	☐	

Test case 6: Smartphone, aLoTw high value and flavor high
Verification Checklist: the AI Model must indicate the following results

Step	Description	Yes	No	Comments
1	Achievable LoTw	☒	☐	
2	Resilience flavor: high	☒	☐	

Execution Output:
The 6 Test Cases of the Unit Test 1 of the Resilience Function were executed as defined in the sections above. They verified that the achievable level of trustworthiness and the achievable flavor of Resilience were correctly predicted when the deterministic input was read from the mock of the DataOps. This demonstrated that the AI Model was properly trained, at least for the specific conditions of the Test Cases. Figure 87 shows the summary of the tests results.

```
============================================================ test session starts =============================================================
platform darwin -- Python 3.13.3, pytest-9.0.2, pluggy-1.6.0
cachedir: .pytest_cache
configfile: pyproject.toml
plugins: anyio-4.10.0
collected 6 items

tests/ai_models_unit_test.py::TestAIModels::test_phone_high_alotw PASSED                                                             [ 16%]
tests/ai_models_unit_test.py::TestAIModels::test_phone_low_alotw PASSED                                                              [ 33%]
tests/ai_models_unit_test.py::TestAIModels::test_phone_medium_alotw PASSED                                                           [ 50%]
tests/ai_models_unit_test.py::TestAIModels::test_xr_high_alotw PASSED                                                                [ 66%]
tests/ai_models_unit_test.py::TestAIModels::test_xr_low_alotw PASSED                                                                 [ 83%]
tests/ai_models_unit_test.py::TestAIModels::test_xr_medium_alotw PASSED                                                              [100%]
============================================================= 6 passed in 0.07s ==============================================================
```

Figure 87 Resilience Unit Tests Outputs

4.9.5 Integration Tests

This section describes the integration tests created to ensure the correct interaction between internal Resilience TF modules and the SAFE-6G components. The tests use synthetic, predefined inputs and rely on mocks for dependent modules. The expected output is then compared with the actual output to confirm whether the test passes or not.

4.9.5.1 Local AI Agent

The Resilience TF Local AI Agent orchestrates all the actions required to determine the achievable flavor of Resilience based on the UE and level of trustworthiness demanded by the CoCo and the actual network conditions. Once the CoCo confirms the Resilience flavor to be enforced, it boots up the appropriate vApp. The following Unit Tests simulate the interactions from the CoCo and validate if the Local AI Agent processes them correctly and generates the expected outputs.

Prerequisite
Mock-ups are required to simulate the components that interact with the Local AI Agent:

1. MLOps component: to run the required AI models to calculate the achievable LoTw and the vApp flavour.
2. aerOS component: to discover and deploy the vApp and nApp.
3. Message Broker: to simulate the inputs and read the outputs of the Local AI Agent.

Resilience-UT2 – AI Agent Initial Feedback Flow

Objective:	**To validate the use case when the Local AI Agent receives the result from the AI model and returns the achievable LoTw to the CoCo via Message Broker.**
Components	a. Local AI Agent b. MLOps mock-up service c. Message Broker mock-up service
Requirements	N/A
Features to be tested	Activation of the Flow between Message broker, AI Agent and AI Model
Test Steps	1. Publish a JSON message in the Message Broker mock-up that indicates the userID and required LoTw 2. Local AI Agent activates the MLOps mock-up 3. MLOps mock-up provides an achievable LoTw that will be sent in JSON format.

Test case 1: Message processed by AI Agent with feedback flow.

Verification Checklist: test_message_processing_flow

Step	Description	Yes	No	Comments
1	AI Agent receives information from the Kafka topic mock	☒	☐	
2	AI Agent sends the information to MLOPs mock and receives the result of it.	☒	☐	
3	The AI Agent publishes a JSON message in the Message Broker through Kafka with the flavor and LoTw	☒	☐	

Test case 2: Validates batch processing of multiple users with different resilience scores, same process as **Test case 1**.

Verification Checklist: test_multiple_messages_processing

Step	Description	Yes	No	Comments
1	AI Agent receives information from the Kafka topic mock	☒	☐	
2	AI Agent sends the information to MLOPs mock and receives the result of it.	☒	☐	
3	The AI Agent publishes a JSON message in the Message Broker through Kafka with the flavor and LoTw	☒	☐	

Resilience UT3 – AI Agent Final Feedback Flow

Objective:	**To validate the use case when the Local AI Agent can provide the LoTw and activate the vApp with correct parametrization executing the respective nApp.**
Components	a. Local AI Agent b. MLOps mock-up service c. aerOS mock-up service d. Message Broker mock-up service
Requirements	N/A
Features to be tested	Activation of the vApp
Test Steps	1. Publish a JSON message in the Message Broker mock-up that indicates the required LoTw 2. Local AI Agent activates the MLOps mock-up 3. MLOps mock-up provides an achievable LoTw and flavor that matches with the aLoTw 4. Local AI Agent calls the aerOS mock-up to discover the vAPP a. aerOS activates the vApp if it was not running 5. Local AI Agent provides the LoTw and flavor to the vAPP 6. Local AI Agent publishes a JSON message in the Message Broker mock-up that informs about the activated flavour and LoTw.

Test case 3: Validates deployment strategy for users with low resilience scores (< 40).

Verification Checklist: test_final_score_low_deployment

Step	Description	Yes	No	Comments
1	AI Agent receives the final score from the Kafka topic mock	☒	☐	
2	AI Agent deploys the flavour that has received from CoCo, deploying the corresponding vApp and nApp	☒	☐	Low vApp and nApp configuration

3	The AI Agent receives the status of the vApp and nApp deployment and publishes a JSON message back to the CoCo through Kafka	☒	☐	

Test case 4: Validates deployment strategy for users with medium resilience scores (40-80).
Verification Checklist: test_final_score_medium_deployment

Step	Description	Yes	No	Comments
1	AI Agent receives the final score from the Kafka topic mock	☒	☐	
2	AI Agent deploys the flavour that has received from CoCo, deploying the corresponding vApp and nApp	☒	☐	Medium vApp and nApp configuration
3	The AI Agent receives the status of the vApp and nApp deployment and publishes a JSON message back to the CoCo through Kafka	☒	☐	

Test case 5: Validates deployment strategy for users with high resilience scores (≥ 80).
Verification Checklist: test_final_score_high_deployment

Step	Description	Yes	No	Comments
1	AI Agent receives the final score from the Kafka topic mock	☒	☐	
2	AI Agent deploys the flavour that has received from CoCo, deploying the corresponding vApp and nApp	☒	☐	High vApp and nApp configuration
3	The AI Agent receives the status of the vApp and nApp deployment and publishes a JSON message back to the CoCo through Kafka	☒	☐	

Execution Output:
All the Test Cases of the Unit Tests 2 and 3 of the Resilience Function were successfully executed, and they verified that the Local AI Agent can interact with its required component interfaces, including MLOps, aerOS, vApp, and the Message Broker. Figure 88 shows the summary of the test results.

```
============================================= test session starts ==============================================
platform darwin -- Python 3.10.17, pytest-9.0.2, pluggy-1.6.0
cachedir: .pytest_cache
collected 5 items

test/aiagent_unit_test.py::UnitTestAIAgent::test_final_score_high_deployment PASSED                    [ 20%]
test/aiagent_unit_test.py::UnitTestAIAgent::test_final_score_low_deployment PASSED                     [ 40%]
test/aiagent_unit_test.py::UnitTestAIAgent::test_final_score_medium_deployment PASSED                  [ 60%]
test/aiagent_unit_test.py::UnitTestAIAgent::test_message_processing_flow PASSED                        [ 80%]
test/aiagent_unit_test.py::UnitTestAIAgent::test_multiple_messages_processing PASSED                   [100%]
============================================== 5 passed in 0.06s ===============================================
```

Figure 88 Resilience Integration Tests Outputs

4.9.6 Next Steps

The team is engaged in an ongoing, iterative process to develop the modules of the Resilience function. The primary focus is on refining both the architecture and functionality of these modules to ensure they are closely aligned with the overarching objectives of the project. The planned next steps are:

1. Evolve the Resilience TF. Evolve the AI models, which were initially constructed using synthetic datasets, to consume and handle real data from the Cellular (RAN and Core) networks.

Collaboration is ongoing between Telefónica, Cumucore, and the Polytechnic University of Valencia to finalize the lists of KPIs used to assess the network resilience, as well as to agree on the full catalogue of corrective and preventive actions triggered, including network subscription optimization, quality of service management and NF (Network Function) selection optimization. Insights from each development cycle will be fed back into the process to steadily improve resilience prediction accuracy.

2. Verification and validation are prioritized by developing comprehensive tests in parallel with model iterations, ensuring correctness, stability, and intended behavior under varied conditions.

    a. Unit test coverage is being expanded across all critical modules, including the AI models, the Local AI Agent, and vApp/nApp. Special attention will be given to edge cases, such as nulls, out-of-range values, malformed inputs, and fault-injection scenarios

3. Integration and E2E validation. Integrate the modules with the rest of the SAFE-6G components: CoCo, CAPIF, MLOps, DataOps, Core API, … Once integrated, test cross-domain interactions under both normal and degraded operating conditions.

4.10 Reliability Trust Function

4.10.1 Overview

The Reliability TF exploits AI/ML methods to ensure the reliability of applications deployed in 6G networks, contributing in this way to the overall trustworthiness of 6G. Within SAFE-6G, it operates based on the user's intent by first performing multi-layer monitoring across the different planes of the system (edge-cloud continuum plane, system openness plane and service/application plane). Then, it leverages AI/ML mechanisms, such as deep neural networks, to continuously predict application reliability and, when needed, takes actions (e.g. scale up/out of system resources) to ensure that the reliability level requested by the user (via the LoTw) is maintained throughout the entire service lifecycle.

4.10.2 Architecture

The internal operation of the Reliability TF and its interaction with other SAFE-6G components is pictorially described in the sequence diagram of Figure 89.

The workflow begins when the CoCo publishes the nLoTw score to the AI Agent of the Reliability TF via the CoCo's message broker (Kafka/RedPanda). This score encapsulates the user's desired trustworthiness level for the deployed service. Upon receiving the score, the AI Agent semantically maps the numerical LoTw value to a trust tier, low, medium, or high, establishing the operational baseline for subsequent decision-making.

Next, the AI Agent initiates the TF by acquiring a secure token from CAPIF to authenticate its actions. It then issues deployment requests to the aerOS orchestrator, which provisions the selected vApp and

nApp instances across the SAFE-6G infrastructure. The vApp retrieves the appropriate AI/ML model from the Model Store (MinIO), while both vApp and nApp independently acquire CAPIF tokens to validate their configuration and API access rights.

Once deployed, the nApp begins collecting telemetry and contextual data from multiple distributed sources:

- The Edge-Cloud Continuum for infrastructure-level metrics.
- The 5G Core for network-level performance indicators and session data.
- The Metaverse Manager for application-level metrics and immersive service context.

The nApp processes the raw input and forwards structured data to the vApp. The vApp applies the selected AI/ML model to analyze the data and generate reliability predictions, such as forecasting anomalies, performance degradation, service instability relative to the user's LoTw requirement, etc. These predictions are then transmitted back to the AI Agent.

Based on the predictions, the AI Agent determines whether proactive reliability actions are required (e.g., scaling resources). If so, it publishes the proposed actions to the Message Broker. The Conflict Resolution module consumes these actions, and through its internal operations it confirms or rejects them. Approved actions are executed by the AI Agent of the TF across multiple domains:

- Through aerOS for orchestration and deployment.
- Via the Metaverse Manager for application-level enforcement.
- Directly on the 5G Core for network-level optimization.

Finally, the AI Agent transmits the executed reliability actions and outputs back to the CoCo, completing the trust orchestration loop.

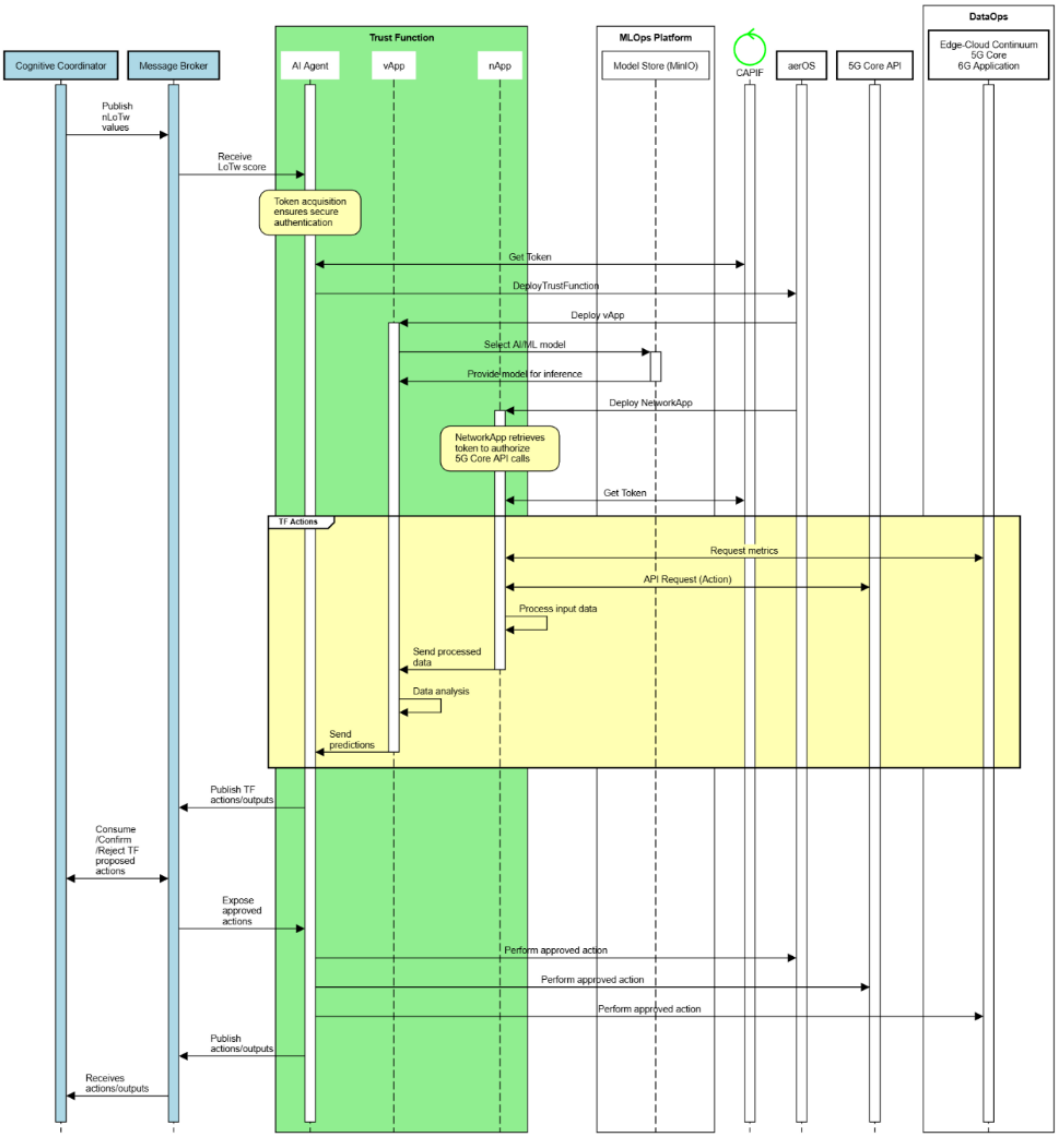


Figure 89 Reliability Function Sequence diagram

4.10.3 Integration with other Components

The Reliability TF interacts with the components of the SAFE-6G platform through the following interfaces:

Component	Direction	Protocol/Topic	Details
CoCo	AI Agent	Kafka/RedPanda	Receives nLoTw from CoCo
AI Agent	aerOS	aerOS API	Deploy TF
aerOS	vApp	aerOS API	Deploy vApp
vApp	MLOps	REST	Select AI/ML model
MLOps	vApp	REST	Provide model for inference
aerOS	nApp	aerOS API	Deploy Network App
nApp	DataOps	REST	Request Metrics
nApp	5G Core API	REST	API request

nApp	vApp	Internal Message Broker	Send processed data
vApp	AI Agent	Internal Message Broker	Send predictions
AI Agent	CoCo	Kafka/RedPanda	Publish TF actions/outputs
CoCo	AI Agent	Kafka/RedPanda	Sends back executed actions/outputs

Table 16 Reliability TF communication with the SAFE-6G components

In Figure 90, the Gitlab pipeline of the Reliability TF executes automatically both the unit and integration tests. The following picture depicts the Gitlab pipeline dashboard of the nApp jobs that have been executed successfully. The lint tests check the structure of the code for any formatting errors. The unit tests check the functionality of isolated modules/components. The integration tests check that the communication between modules/components works correctly.

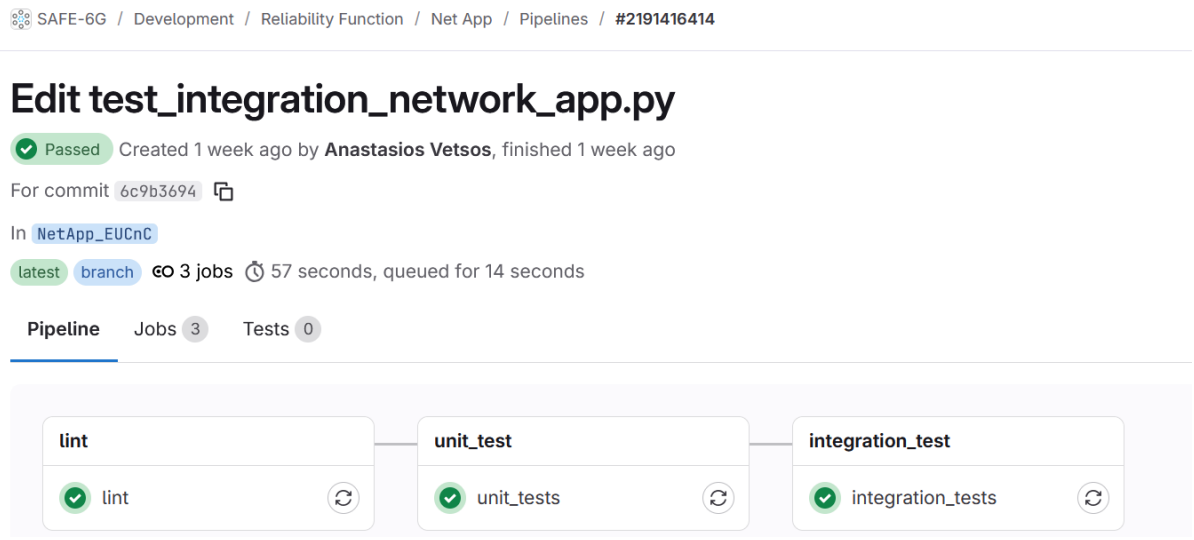


Figure 90 Integration Test

4.10.4 Unit Tests

Reliability-UT1: Split time window

Objective	To split the time windows for optimal data fetching
Components	a. Reliability nApp
Requirements	1. nApp deployed 2. Start and end timestamps defined
Features to be tested	Time window splitting logic
Test Steps	1. Define the start and end timestamps 2. Call “split_time_window” function for a 1 minute chunk 3. Verify the count and boundaries of the chunks

Verification Checklist for Reliability-UT1:

Step	Description	Yes	No	Comments
1	Returns the correct number of chunks for the split time window	☒	☐	
2	Returns each chunk of equal length	☒	☐	

Execution Output:

The Reliability-UT2 ensures that the data is fetched optimally and efficiently by splitting the time window for the data requests; any change in the time window could result in data loss so a unit test for a required time window was created. In Figure 91 two messages are shown, the first notifies about the time window and the number of chunks that the message will be split. The second about the successful outcome of the test.

```
$ python -m unittest tests.test_unit_network_app_UT${TEST_NUMBER}
==================================== test session starts =====================================
Ran 1 test in 0.000s
OK
Reliability-UT1: Split time window
   ✓ Time window splitting test passed
Cleaning up project directory and file based variables
Job succeeded
====================================== test session ends ======================================
```

Figure 91 Reliability-UT1 output

Reliability-UT2: Successfully fetch metric data

Objective	To test the data fetching process
Components	a. Reliability nApp
Requirements	1. nApp deployed 2. Required namespaces defined
Features to be tested	Fetching of the metrics successfully
Test Steps	1. Mock a successful response 2. Filter the required namespaces 3. Call “fetch_metric_data” function 4. Verify the results

Verification Checklist for Reliability-UT2:

Step	Description	Yes	No	Comments
1	Returns a successful status	☒	☐	
2	Filters only the required namespaces	☒	☐	
3	Verify data structure	☒	☐	

Execution Output:

The Reliability-UT2 validates that the nApp fetches the data from the DataOps and filters only the data from the required namespaces through the “fetch_metric_data” function. In Figure 92 four messages are shown, the first shows the time window and the selected metric(s) being fetched. The second and third messages follow the requirements mentioned in the verification steps and the fourth notifies about the outcome of the test.

```
$ python -m unittest tests.test_unit_network_app_UT${TEST_NUMBER}
==================================== test session starts =====================================
Ran 1 test in 0.006s
OK
Reliability-UT2: Successfully fetch metric data
   ✓ Returns a successful status
   ✓ Data structure and required namespaces are valid
   ✓ Successful metric data fetch test passed
Cleaning up project directory and file based variables
Job succeeded
====================================== test session ends ======================================
```

Figure 92 Reliability-UT2 output

Reliability-UT3: Fetch metric data errors

Objective	To testing error handling
Components	a. Reliability nApp
Requirements	1. nApp deployed
Features to be tested	Handling of various errors
Test Steps	1. Mock a HTTP 500/JSON decode error responses 2. Call “fetch_metric_data” function 3. Verify error handling

Verification Checklist for Reliability-UT3:

Step	Description	Yes	No	Comments
1	Returns error status	☒	☐	
2	Status code error capture	☒	☐	

Execution Output:

The Reliability-UT3 was executed to handle of the errors that may occur when the “fetch_metric_data” function is requesting data, this is a significant step that requires verification. In Figure 93 there are three messages show. The first one notifies about the handling of HTTP errors and the other two about the any JSON error.

```
$ python -m unittest tests.test_unit_network_app_UT${TEST_NUMBER}
===================================== test session starts ======================================
Ran 2 tests in 1.004s
OK
Reliability-UT3: Fetch metric data errors
   ✓ HTTP error handling test passed
   JSON decode error on attempt 1 for metric 'container_memory_usage_bytes' (2025-11-06T14:00:00Z -
2025-11-06T14:01:00Z), retrying in 1s...
   ✓ JSON error test passed
Cleaning up project directory and file based variables
Job succeeded
====================================== test session ends =======================================
```

Figure 93 Reliability-UT3 output

Reliability-UT4: Fetch metric data for window

Objective	To test the data fetching function with the split time windows
Components	a. Reliability nApp
Requirements	1. nApp deployed
Features to be tested	Metric fetch with split time windows
Test Steps	1. Mock “fetch_metric_data” for two time windows 2. Call “fetch_metric_data_for_window” function 3. Verify results

Verification Checklist for Reliability-UT4:

Step	Description	Yes	No	Comments
1	Returns a successful status	☒	☐	
2	Return results for both windows	☒	☐	
3	Verify the grouping of the collected metric data	☒	☐	

Execution Output:
The Reliability-UT4 validates the combination for the data fetching functionality with the splitting of the time window into equal chunks using the "fetch_metric_data_for_window" function. By calling this function, the test verifies that the metric data received by all the chunks of the time window are correct. There are five messages shown in Figure 94. The first shows the time window and the selected metric(s) being fetched. The rest of the messages follow the requirements mentioned in the verification steps and the fifth notifies about the outcome of the test.

```
$ python -m unittest tests.test_unit_network_app_UT${TEST_NUMBER}
==================================== test session starts =====================================
Ran 1 test in 0.001s
OK
Reliability-UT4: Fetch metric data for window
   ✓ Returns a successful status
   ✓ Returns results for both windows
   ✓ Verify the grouping of the collected metric data
   ✓ Metric fetching with split time windows test passed
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends ======================================
```

Figure 94 Reliability-UT4 output

Reliability-UT5: Grouping of metrics

Objective	To test the grouping of the metrics
Components	a. Reliability nApp
Requirements	1. nApp deployed 2. Required namespaces defined
Features to be tested	Grouping of the metrics
Test Steps	1. Create mock test input data with different pod label names 2. Call "combine_metrics_by_metric_and_pod" function 3. Verify grouping of the metrics

Verification Checklist for Reliability-UT5:

Step	Description	Yes	No	Comments
1	Successful grouping of the metrics by metric name and pod name	☒	☐	
2	Handles both "pod" and "pod_name" labels	☒	☐	

Execution Output:
The Reliability-UT5 validates that the data received by the ML model in the vApp need to be correctly formatted and with proper grouping of the metrics. This test ensures that any formatting or grouping issues that may occur in this step don't propagate to the vApp component. The test is performed by combining the identified metric(s) and verifying correct label handling, in this case by splitting the metrics into two groups 'streamer-1' and 'streamer-2', these values represent the names of the 'pod' label and the 'pod_name' label. There are three messages shown in Figure 95. The first message shows the name(s) of the metrics to be combined. The second message shows if the requirement mentioned in the verification step 2 is valid and the third about the outcome of the overall test.

```
$ python -m unittest tests.test_unit_network_app_UT${TEST_NUMBER}
==================================== test session starts =====================================
Ran 1 test in 0.000s
OK
```

```
Reliability-UT5: Grouping of metrics
   ✓ Handles both 'pod' and 'pod_name' labels correctly
   ✓ Metric combination test passed
Cleaning up project directory and file based variables
Job succeeded
====================================== test session ends ======================================
```

Figure 95 Reliability-UT5 output

Reliability-UT6: Calculation of metric ratios

Objective	To calculate the metric rations
Components	a. Reliability nApp
Requirements	1. nApp deployed 2. Required namespaces defined 3. Memory usage metrics defined 4. "MAX_MEMORY_BYTES" value defined
Features to be tested	Memory metrics ration calculation
Test Steps	1. Create mock test memory usage metric data 2. Call "add_memory_metrics_ratio" function 3. Verify the calculation of the ratios

Verification Checklist for Reliability-UT6:

Step	Description	Yes	No	Comments
1	Required memory usage metrics names added to the keys	☒	☐	
2	Rations calculated correctly	☒	☐	

Execution Output:

The Reliability-UT6 ensures that any operation made using the "MAX_MEMORY_BYTES" variable are correctly applied and do not effect the data send to the vApp. In Figure 96 two messages are shown. The first one shows the number of metrics the memory rations are being calculated and the values of the pod labels. The second shows the outcome of the test.

```
$ python -m unittest tests.test_unit_network_app_UT${TEST_NUMBER}
==================================== test session starts ====================================
Ran 1 test in 0.000s
OK
Reliability-UT6: Calculation of metric ratios
   ✓ Memory ratio calculation test passed
Cleaning up project directory and file based variables
Job succeeded
====================================== test session ends ======================================
```

Figure 96 Reliability-UT6 output

Reliability-UT7: Get pod suffix

Objective	To Test the suffix extraction function
Components	a. Reliability nApp
Requirements	1. nApp deployed
Features to be tested	Pod suffix generation
Test Steps	1. Test different pod names 2. Call "get_pod_suffix" function for each given pod name 3. Verify correct suffix extraction from pod name

Verification Checklist for Reliability-UT7:

Step	Description	Yes	No	Comments
1	Verification of all the pod name cases	☒	☐	

Execution Output:
The Reliability-UT7 confirms that the suffix on the pod names is correctly extracted, as it is required in the formatting of the data send to the vApp. In Figure 97 two messages are shown. The first message indicates the name of the pod label name being extrated and the second shows the outcome of the test.

```
$ python -m unittest tests.test_unit_network_app_UT${TEST_NUMBER}
==================================== test session starts =====================================
Ran 1 test in 0.000s
OK
Reliability-UT7: Get pod suffix
   ✓ Pod suffix generation test passed
Cleaning up project directory and file based variables
Job succeeded
===================================== test session ends ======================================
```

Figure 97 Reliability-UT7 output

Reliability-UT8: Feature extraction

Objective	To test the feature extraction function
Components	a. Reliability vApp
Requirements	1. vApp deployed
Features to be tested	Feature extraction output validation
Test Steps	1. Create test input data 2. Call “feature_extraction” function 3. Verify output shape and values

Verification Checklist for Reliability-UT8:

Step	Description	Yes	No	Comments
1	Verify the function handles the input	☒	☐	
2	Check for dimension errors	☒	☐	
3	Verify correct values	☒	☐	

Execution Output:
The Reliability-UT8 validates that the feature extraction. This processes critical to the training of the ML model. The test verifies that each feature is extracted correctly with the expected values in regard to the test input data. The message shown in Figure 98 notifies if the test was successful.

```
$ python -m unittest tests.test_unit_vapp_UT${TEST_NUMBER}
==================================== test session starts =====================================
Ran 2 tests in 0.006s
OK
Reliability-UT8: Feature extraction
    ✓ Feature extraction values test
Cleaning up project directory and file based variables
Job succeeded
===================================== test session ends ======================================
```

Figure 98 Reliability-UT8 output

Reliability-UT9: Predict scale up

Objective	To test the predictions of the ML model
Components	a. Reliability vApp
Requirements	1. vApp deployed 2. ML model required
Features to be tested	ML model predictions
Test Steps	1. Call “predict_scale_up” function 2. Test with low memory usage input data 3. Test with high memory usage input data 4. Verify predictions

Verification Checklist for Reliability-UT9:

Step	Description	Yes	No	Comments
1	Verify if high memory usage triggers scale up decision	☒	☐	
2	Verify if low memory usage doesn't triggers scale up decision	☒	☐	

Execution Output:

A wrong prediction from the ML model could have major problems in the operation of the Reliability Function, for that reason the Reliability-UT9 tests the decision making of the ML model by feeding it with specific input data to verify each scale up decision. In Figure 99, there are four messages depicted, the first message shows the average memory ratio of the test input data in this case ‘0.9' indicated for a high memory usage scenario which later triggers a scale up decision as shown in the second message. In the last two messages the scenario with ‘0.1' average memory usage is tested and the model correctly doesn’t trigger a scale up.

```
$ python -m unittest tests.test_unit_vapp_UT${TEST_NUMBER}
==================================== test session starts =====================================
Ran 1 test in 0.005s
OK
Reliability-UT9: Predict scale up
    ✓ High memory usage triggers scale up decision
    ✓ Low memory usage doesn't triggers scale up decision
Cleaning up project directory and file based variables
Job succeeded
===================================== test session ends ======================================
```

Figure 99 Reliability-UT9 output

Reliability-UT10: Cognitive Consumer low reliability

Objective	To test the AI Agent “lotw” sorting for low reliability
Components	a. Reliability AI Agent
Requirements	1. AI Agent deployed
Features to be tested	”lotw” threshold processing
Test Steps	1. Call the cognitive_consumer “callback” function with a low “lotw” 2. Verify that the function was called with low “lotw”

Verification Checklist for Reliability-UT10:

Step	Description	Yes	No	Comments
1	The cognitive_consumer “callback” function was called with a low “lotw”	☒	☐	

Execution Output:
The Reliability-UT10 confirms that when the AI Agent receives a low "LoTw" it properly interprets the input and issues a request for low flavour deployment of the components. Two messages in Figure 100, depict the logs made by the cognitive_consumer "callback" function receiving the low LoTw value.

```
$ python -m unittest tests.test_unit_ai_agent_UT${TEST_NUMBER}
==================================== test session starts ====================================
2026-01-03 13:52:38,414 - INFO - Received Low lotw: 15
Ran 1 test in 0.515s
OK
Reliability-UT10: Cognitive Consumer low reliability
    ✓ reconciling_deployment function was called with "low" lotw
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends =====================================
```

Figure 100 Reliability-UT10 output

Reliability-UT11: Cognitive Consumer medium reliability

Objective	To test the AI Agent "lotw" sorting for medium reliability
Components	a. Reliability AI Agent
Requirements	1. AI Agent deployed
Features to be tested	"lotw" threshold processing
Test Steps	1. Call the cognitive_consumer "callback" function with a medium "lotw" 2. Verify that the function was called with medium "lotw"

Verification Checklist for Reliability-UT11:

Step	Description	Yes	No	Comments
1	The cognitive_consumer "callback" function was called with a medium "lotw"	☒	☐	

Execution Output:
The Reliability-UT11 confirms that when the AI Agent receives a medium "LoTw" it properly interprets the input and issues a request for medium flavour deployment of the components. Two messages in Figure 101, depict the logs made by the cognitive_consumer "callback" function receiving the medium LoTw value.

```
$ python -m unittest tests.test_unit_ai_agent_UT${TEST_NUMBER}
==================================== test session starts ====================================
2026-01-03 13:52:39,328 - INFO - Received Medium lotw: 50
Ran 1 test in 0.518s
OK
Reliability-UT11: Cognitive Consumer medium reliability
    ✓ reconciling_deployment function was called with "medium" lotw
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends =====================================
```

Figure 101 Reliability-UT11 output

Reliability-UT12: Cognitive Consumer high reliability

Objective	To test the AI Agent "lotw" sorting for high reliability
Components	a. Reliability AI Agent
Requirements	1. AI Agent deployed

Features to be tested	”lotw” threshold processing
Test Steps	1. Call the cognitive_consumer “callback” function with a high “lotw” 2. Verify that the function was called with high “lotw”

Verification Checklist for Reliability-UT12:

Step	Description	Yes	No	Comments
1	The cognitive_consumer “callback” function was called with a high “lotw”	☒	☐	

Execution Output:

The Reliability-UT12 confirms that when the AI Agent receives a high “LoTw” it properly interprets the input and issues a request for high flavour deployment of the components. Two messages in Figure 102, depict the logs made by the cognitive_consumer “callback” function receiving the high LoTw value.

```
$ python -m unittest tests.test_unit_ai_agent_UT${TEST_NUMBER}
==================================== test session starts ====================================
2026-01-03 13:52:37,253 - INFO - Received High lotw: 85
Ran 1 test in 0.485s
OK
Reliability-UT12: Cognitive Consumer high reliability
    ✓ reconciling_deployment function was called with “high” lotw
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends ======================================
```

Figure 102 Reliability-UT12 output

Reliability-UT13: Scale up prediction “1”

Objective	To test the scale up action when prediction is “1”
Components	a. Reliability AI Agent
Requirements	1. AI Agent deployed
Features to be tested	Scaling logic based on the ML models output
Test Steps	1. Set “scale_up” variable to “False” 2. Call the model_consumer “callback” function with prediction “1” 3. Verify scaling action

Verification Checklist for Reliability-UT13:

Step	Description	Yes	No	Comments
1	Verify “reconciling_deployment” was called with the correct “clotw”	☒	☐	
2	Verify that “scale_up” was set to “True”	☒	☐	

Execution Output:

The Reliability-UT13 validates the scale up logic of the AI Agent. An unwanted scale up action could desynchronize the components, so to validate that the scale up action correspond to the ML model output of a “1” prediction. There are three messages in Figure 103, that show the reliability of a test service and the actions taken for a scale up decision.

```
==================================== test session starts ====================================
$ python -m unittest tests.test_unit_ai_agent_UT${TEST_NUMBER}
2026-01-03 13:52:44,320 - INFO - Predicted value for service ID service-123 received.
```

```
2026-01-03 13:52:44,320 - INFO - Service service-123 is not reliable. Scaling up.
-------------------------------------------------------------------------------
Ran 1 test in 0.434s
OK
Reliability-UT13: Scale up prediction “1”
    ✓ reconciling_deployment function was called with “medium” clotw for service-123
Cleaning up project directory and file based variables
Job succeeded
======================================== test session ends ========================================
```

Figure 103 Reliability-UT13 output

Reliability-UT14: Scale up prediction “0”

Objective	To test the scale up action when prediction is “0”
Components	a. Reliability AI Agent
Requirements	1. AI Agent deployed
Features to be tested	Scaling logic based on the ML models output
Test Steps	1. Call the model_consumer “callback” function with prediction “0” 2. Verify that no scaling action occured

Verification Checklist for Reliability-UT14:

Step	Description	Yes	No	Comments
1	Verify “reconciling_deployment” was not called	☒	☐	
2	Verify the log message about the reliability of the service	☒	☐	

Execution Output:

The Reliability-UT14 validates the scale up logic of the AI Agent. An unwanted scale up action could desynchronize the components, so to validate that the scale up action correspond to the ML model output of a “0” prediction. We can see one message in Figure 104, that shows the reliability of a test service and that no actions were taken for a scale up decision.

```
$ python -m unittest tests.test_unit_ai_agent_UT${TEST_NUMBER}
======================================== test session starts ======================================
Ran 1 test in 0.422s
OK
Reliability-UT14: Scale up prediction “0”
    ✓ Service-123 is reliable. No action taken.
Cleaning up project directory and file based variables
Job succeeded
======================================== test session ends ========================================
```

Figure 104 Reliability-UT14 output

4.10.5 Integration Tests

Reliability-IT1: Network App RabbitMQ Integration

Objective	To test the input-output operations of the nApp in relation with the Internal RabbitMQ
Components	a. Reliability nApp b. Internal RabbitMQ
Requirements	1. RabbitMQ message broker deployed 2. nApp deployed
Features to be tested	RabbitMQ Integration with the nApp
Test Steps	1. Establish RabbitMQ connection 2. Create test queue

	3. Publish test message to the queue 4. Consume message 5. Send response to the response queue 6. Clean test queues

Verification Checklist for Reliability-IT1:

Step	Description	Yes	No	Comments
1	Successful connection to RabbitMQ	☒	☐	
2	Message published without errors	☒	☐	
3	Message received and response published	☒	☐	
4	Response successfully sent to the response queue	☒	☐	
5	Queues cleaned up after all the tests	☒	☐	

Execution Output:

The Reliability-IT1 confirms that the nApp and the internal message broker are successfully integrated. A failure in the connection to the internal RabbitMQ or any queue within the RabbitMQ would separate the communication of the nApp to the vApp. The four messages depicted in Figure 105 follow the requirements mentioned in the verification steps.

```
$ python -m unittest tests.test_integration_network_app_IT${TEST_NUMBER}
===================================== test session starts ======================================
Ran 1 test in 0.235s
OK
Reliability-IT1: Network App RabbitMQ Integration
   ✓ Successful connection to RabbitMQ
   ✓ Message published
   ✓ Message received and response published
   ✓ RabbitMQ test completed successfully!
Cleaning up project directory and file based variables
Job succeeded
====================================== test session ends =======================================
```

Figure 105 Reliability-IT1 output

Reliability-IT2: Fetch metric data with DataOps connection

Objective	To test the connection between the nApp and DataOps
Components	a. Reliability Network App b. DataOps
Requirements	1. DataOps API reachable 2. Network App deployed
Features to be tested	DataOps API integration with the Network App
Test Steps	1. Make DataOps API call 2. Query memory usage metrics for a 1 minute window 3. Call “fetch_metric_data” function 4. Verify results

Verification Checklist for Reliability-IT2:

Step	Description	Yes	No	Comments
1	API call returns successful status	☒	☐	
2	Verify the filtering of the required namespaces is successful	☒	☐	
3	Message received in the callback function	☒	☐	

4	Verify the handling of errors	☒	☐	

Execution Output:
The Reliability-IT2 validates the connection to the DataOps. Without a connection to the DataOps no data could be processed in the nApp and send to the vApp. This test verifies that the nApp successfully performs an API call to the DataOps and receives the requested data. The Figure 106 depicts four messages, the first two show that the API call was successful and the last two show that the results are properly filtered and the structure of the requested data is valid.

```
$ python -m unittest tests.test_integration_network_app_IT${TEST_NUMBER}
==================================== test session starts ====================================
Ran 1 test in 0.037s
OK
Reliability-IT2: Fetch metric data with DataOps connection
   ✓ Status 'success' is valid
   ✓ DataOps API call succeeded
   ✓ All results properly filtered to 'demo-app' namespace
   ✓ Message structure is valid with data and result fields
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends ====================================
```

Figure 106 Reliability-IT2 output

Reliability-IT3: vApp RabbitMQ Integration

Objective	To test the input-output operations of the vApp in relation with the Internal RabbitMQ
Components	a. Reliability vApp b. Internal RabbitMQ
Requirements	1. RabbitMQ message broker deployed 2. vApp deployed
Features to be tested	Queue operation with test queue and messages
Test Steps	1. Establish RabbitMQ connection 2. Create required queues 3. Send test message from producer to the input queue 4. Process message 5. Send prediction output queue 6. Verify the prediction

Verification Checklist for Reliability-IT3:

Step	Description	Yes	No	Comments
1	Successful connection to RabbitMQ	☒	☐	
2	Verify if producer can send messages to input queue	☒	☐	
3	Verify if consumer can retrieve messages to input queue	☒	☐	
4	Verify if the prediction was received and if the value is correct	☒	☐	

Execution Output:
The Reliability-IT3 confirms that the vApp and the internal message broker are successfully integrated. A failure in the connection to the internal RabbitMQ or any queue within the RabbitMQ would separate the communication of the vApp to the nApp and the AI Agent. The four messages depicted in Figure 107 follow the requirements mentioned in the verification steps.

```
$ python -m unittest tests.test_integration_vapp_IT3
==================================== test session starts ====================================
Ran 1 test in 4.890s
OK
Reliability-IT3: vApp RabbitMQ Integration
   ✓ Connected to RabbitMQ
   ✓ Producer sent message to reliability-vapp-test-service-001
   ✓ Found message in input queue
   ✓ Consumer processed message, prediction: 1
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends ====================================
```

Figure 107 Reliability-IT3 output

Reliability-IT4: aerOS API endpoint reachability

Objective	To ensure reachability with the aerOS API
Components	a. Reliability AI Agent b. aerOS
Requirements	1. aerOS deployed 2. AI Agent deployed
Features to be tested	aerOS reachability
Test Steps	1. Make GET request to aerOS API endpoint 2. Verify expected response

Verification Checklist for Reliability-IT4:

Step	Description	Yes	No	Comments
1	Verify that the API endpoint is reachable	☒	☐	
2	Status code is one of expected values	☒	☐	

Execution Output:

The Reliability-IT4 ensures that the aerOS API is reachable. The API availability is required to deploy any flavour of the vApp and nApp. A single message that shows if the status if the endpoint is depicted in Figure 108.

```
$ python -m unittest tests.test_integration_ai_agent_IT${TEST_NUMBER}
==================================== test session starts ====================================
Ran 1 test in 0.133s
OK
Reliability-IT4: aerOS API endpoint reachability
   ✓ aerOS endpoint is reachable
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends ====================================
```

Figure 108 Reliability-IT4 output

Reliability-IT5: Cognitive Consumer scaling action

Objective	To test scaling action to aerOS based on cognitive consumer input
Components	a. Reliability AI Agent b. aerOS
Requirements	1. aerOS deployed 2. AI Agent deployed 3. TOSCA template structure
Features to be tested	Scaling action with aerOS API

Test Steps	1. Create TOSCA template 2. Call "reconciling_deployment" function from the "cognitive_consumer" script 3. Handle exception types 4. Verify scaling action

Verification Checklist for Reliability-IT5:

Step	Description	Yes	No	Comments
1	TOSCA template created with proper structure	☒	☐	
2	Verify successful scaling action log output from "reconciling_deployment" function	☒	☐	

Execution Output:

The Reliability-IT5 verifies that aerOS deploys the correct flavors of the components based on the LoTw input that the AI Agent receives from CoCo. There are two messages shown in Figure 109, the first shows a log from the "reconciling_deployment" function triggered by a low LoTw mocked by a test TOSCA template and the second one if the API call was successful.

```
$ python -m unittest tests.test_integration_ai_agent_IT${TEST_NUMBER}
===================================== test session starts ======================================
2026-01-03 13:53:16,000 - INFO - Triggered aerOS for deployinglow level of trust.
Ran 1 test in 0.818s
OK
Reliability-IT5: Cognitive Consumer scaling action
   ✓ Triggered aerOS for deployinglow level of trust.
   ✓ API call completed
Cleaning up project directory and file based variables
Job succeeded
====================================== test session ends =======================================
```

Figure 109 Reliability-IT5 output

Reliability-IT6: Model Consumer Scaling Action

Objective	To test scaling action to aerOS based on vApp output
Components	a. Reliability AI Agent b. aerOS
Requirements	1. aerOS deployed 2. AI Agent deployed 3. TOSCA template structure
Features to be tested	Scaling action with aerOS API
Test Steps	1. Create TOSCA template 2. Call "reconciling_deployment" function from the "model_consumer" script 3. Handle exception types 4. Verify scaling action

Verification Checklist for Reliability-IT6:

Step	Description	Yes	No	Comments
1	TOSCA template created with proper structure	☒	☐	

2	Verify successful scaling action log output from "reconciling_deployment" function	☒	☐	

Execution Output:

The Reliability-IT6 validates that the scale up decision that the AI Agent received from the vApp is correctly translated into the scale up action of the resources that aerOS initiates. There are two messages depicted in Figure 110. The first message shows the log from the "reconciling_deployment" function of the triggered aerOS deployment for a test service and the second one if the API call was successful.

```
$ python -m unittest tests.test_integration_ai_agent_IT${TEST_NUMBER}
==================================== test session starts ====================================
2026-01-03 13:53:17,177 - INFO - Triggered aerOS for scaling service test-service-123.
Ran 1 test in 0.272s
OK
Reliability-IT6: Model Consumer scaling action
   ✓ Triggered aerOS for scaling service test-service-123.
   ✓ API call completed
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends ======================================
```

Figure 110 Reliability-IT6 output

Reliability-IT7: RabbitMQ Connection

Objective	Complete AI Agent RabbitMQ communication flow
Components	a. Reliability AI Agent b. Internal RabbitMQ
Requirements	1. RabbitMQ message broker deployed 2. AI Agent deployed
Features to be tested	Queue operation with test queue and messages
Test Steps	1. Establish RabbitMQ connection 2. Create test queue 3. Create test message 4. Publish test message in the queue 5. Acknowledge and consume the test message 6. Clean test queues

Verification Checklist for Reliability-IT7:

Step	Description	Yes	No	Comments
1	Successful connection to RabbitMQ	☒	☐	
2	Message properly acknowledged	☒	☐	
3	Test queues cleaned	☒	☐	

Execution Output:

The Reliability-IT7 confirms that the AI Agent and the internal message broker are successfully integrated. A failure in the connection to the internal RabbitMQ or any queue within the RabbitMQ would separate the communication of the AI Agent to the vApp. The four messages depicted in Figure 111 that follow the requirements mentioned in the verification steps.

```
$ python -m unittest tests.test_integration_ai_agent_IT${TEST_NUMBER}
```

```
==================================== test session starts ====================================
Ran 1 test in 0.020s
OK
Reliability-IT7: RabbitMQ Connection
   ✓ Connected to RabbitMQ
   ✓ Test message published
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends =====================================
```

Figure 111 Reliability-IT7 output

Reliability-IT8: Redpanda Connection

Objective	To complete AI Agent Redpanda communication flow
Components	a. Reliability AI Agent b. Redpanda/Kafka
Requirements	1. Redpanda message broker deployed 2. AI Agent deployed
Features to be tested	Kafka producer functionality and validation
Test Steps	1. Configure KafkaProducer 2. Fetch topic metadata 3. Create test message 4. Send message with callback

Verification Checklist for Reliability-IT8:

Step	Description	Yes	No	Comments
1	KafkaProducer configured successfully	☒	☐	
2	Metadata fetched	☒	☐	
3	Callbacks registered	☒	☐	

Execution Output:

The Reliability-IT8 validates the connection between the AI Agent and CoCo's message broker. A failure in the connection to the CoCo's message broker would separate the communication of the AI Agent with the CoCo and prevent the LoTw message from being exchanged. There are five messages depicted in Figure 112. The first confirms that the producer was created. The second one verifies that the metadata was fetched from the relevant topic. The third and fourth message verify that the test message was sent to the relevant topic and the fifth message about the success of the overall test.

```
$ python -m unittest tests.test_integration_ai_agent_IT${TEST_NUMBER}
==================================== test session starts ====================================
Ran 1 test in 0.114s
OK
Reliability-IT8: Redpanda Connection
   ✓ Producer created, testing connection...
   ✓ Topic 'trust-scores' metadata fetched
   ✓ Message sent to partition 0, offset 112
   ✓ Successfully sent test message to trust-scores
   ✓ Message Broker connection test completed
Cleaning up project directory and file based variables
Job succeeded
==================================== test session ends =====================================
```

Figure 112 Reliability-IT8 output

4.10.6 Next Steps

The next steps, based on the integration plan in Section 3.3 *Integration Plan*, are to further extend the unit and integration tests of the components, complete the E2E integration test using the standard initial template of the pipeline mentioned in Section 2.3 *SAFE-6G GitOps*. In Addition, the incorporation of the Open CAPIF into the pipeline and the production-ready DevOps and MLOps systems will be finalized. Furthermore, the development of the high LoTw flavour for the Reliability TF will continue in alignment with the integration plan.

4.11 Metaverse Applications

4.11.1 Overview

Three main components are directly related to the metaverse UC1 and UC2. Since each of them was further detailed in D2.3, we only give in the following section a summary. First, two applications (one per use-case) were developed with the Unity framework [47] during the project. When launching these apps, end users first arrive in a virtual XR lobby. They must then authenticate themselves before interacting with the SAFE-6G chatbot to define their priorities. Finally, they are allowed to access to the selected content (factory DT for UC1, machine procedure formation for UC2). Finally, the metaverse manager was developed as an intermediate component between the Unity UC apps and the rest of the SAFE-6G architecture. The goal of the metaverse manager is to provide endpoints and expose UC APIs while being easy to containerize and deploy during validation phases. At M24, the intermediate versions of both the Unity UC apps and the metaverse manager were released.

4.11.2 Architecture

The technical architectures of the Unity applications are detailed in D2.3. Since these applications cannot be containerized, they are a special case with respect to WP5 activities. They run directly on the user-worn XR devices (for instance, Quest 3 headsets [48]) or on a dedicated workstation running CloudXR to stream the resulting video to the XR device (UC1).

The metaverse manager is a Python application based on Flask. Two major Flask subcomponents are present:

- The main one coordinates web endpoints and the three categories of API handlers: *chatbot*, *system* and *UC-specific* handlers.
- The *Request manager* coordinates the communication with the *EndUserManager* module and with the Unity applications (through Transmission Control Protocol (TCP) sockets).

Beyond its web graphical user interface, the metaverse manager thus exposes APIs for the rest of the SAFE-6G components, requesting data from the Unity UC apps or triggering effects on them to affect end-users. Contrary to the Unity apps, the metaverse manager can be containerized, deployed at UNIWA and/or NCSRD premises and follow the standard CI/CD approach conducted in SAFE-6G.

The two figures below illustrate the behaviour of the metaverse components and the related metaverse APIs. The first one showcases the fact that the Unity apps send QoS metrics logs to the

metaverse manager on a regular basis. If a TF needs to have access to real-time metrics about the XR headset, it can thus request a subscription to the metaverse manager it order to receive those logs.

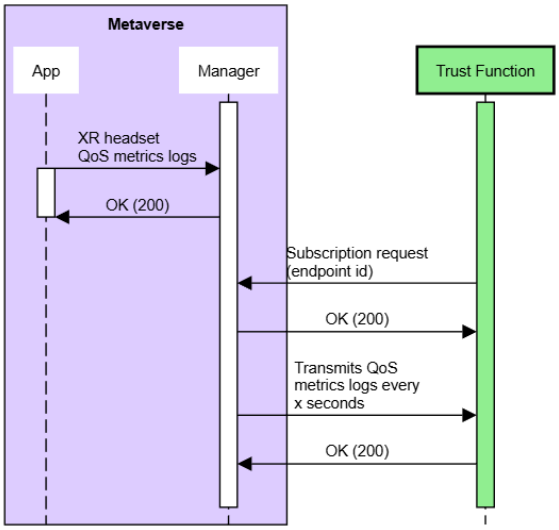


Figure 113 Example of UC API workflow with the QoS metrics API

The second figure illustrates how a TF can have an impact on the Unity apps and the activity of end users. By calling an API like the *BlacklistUser* from the metaverse manager, the TF can for instance trigger a request on it to isolate a connected user until further notice. When the user has been successfully disconnected from the metaverse lobby, the Unity app sends an acknowledgment to the metaverse manager, which will then do the same towards the TF.

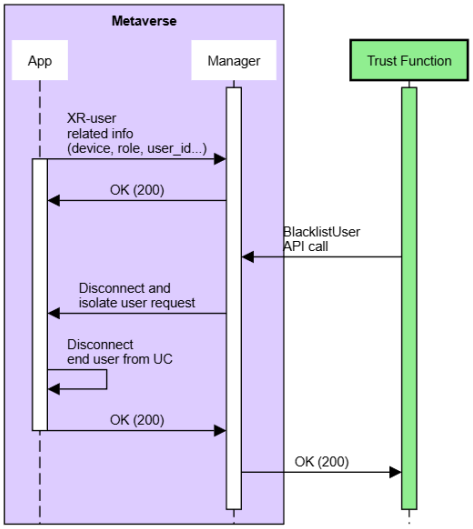


Figure 114 Trust Function interaction with Metaverse App

4.11.3 Integration with other Components

The metaverse manager communicates with the UC Unity apps through TCP sockets. To communicate with the rest of SAFE-6G components (chatbot and TFs), it either consumes or exposes REST APIs.

Component	Direction	Protocol/Topic	Details
Unity UC apps	Metaverse manager	Websockets	
Metaverse manager	Chatbot	REST	
Metaverse manager	TFs	REST	Metaverse APIs exposed by the metaverse manager

Table 17 Metaverse Application communication with the SAFE-6G components

4.11.4 Unit Tests

MetApp-UT1: Metaverse manager web GUI status

A first important test for the metaverse manager consists in checking if the main Flask app was correctly initialized, with basic routes in place and accessible on the expected url and port. A simple way to do so is to assert if the homepage of the metaverse manager GUI is available and complete.

Objective	To check if the metaverse manager web GUI is reachable as expected on the given url & port
Components	a. Metaverse manager
Requirements	None
Features to be tested	Web GUI initialization and reachability (GET request)
Test Steps	1. Start the Metaverse manager 2. Trigger a GET request to obtain the content of the web page at the expected url and port 3. Check if the request status code is 200 (OK) and that the content of the page looks correct (for instance, the welcome message is present).

Verification Checklist for MetApp-UT1:

Step	Description	Yes	No	Comments
1	Request status code is 200 (OK)	☒	☐	
2	Html page content is correct (welcome message present, UC selection too)	☒	☐	

Execution output:

Upon executing the test, the two steps are validated. First, a GET request is successfully executed to obtain the content of the homepage. Then, this Html content is checked to make sure the page is complete and allows to select the desired use-case. *The metaverse manager web GUI is accessible and complete.*

```
bailly@IMM-PC-CHB d/Charles/Code/SAFE-6G/MetaverseManager/metaverse-manager (dev) $ ./UnitTests.sh
========================================= test session starts =========================================
Python 3.11.8, pytest-6.2.4, py-1.11.0, pluggy-0.13.1 rootdir: D:\Charles\Code\SAFE-
6G\MetaverseManager\metaverse-manager plugins: parallel-0.1.1
collected 1 item
test\unit\webapp\test_root.py
    • Serving Flask app 'src.python.network.MainFlaskThread'
    • Debug mode: off Serving Flask app 'src.python.network. authenticationsAPIs.
      AuthenticationHandler' *Debug mode: off
    • Serving Flask app 'src.python.network.immAPIs.unityAPIs. UnityCommHandler' *Debug mode: off

Step success: managed to reach the metaverse manager landing web page
Step success: landing page content is complete

==== 1 passed, 2 warnings in 12.85s ====
================================== 1 passed, 2 warnings in 12.85s      ============================
```

Figure 115 MetApp-UT1 output

MetApp-UT2: Forbid factory DT updates

A second major feature that must be tested concerns the available APIs. The metaverse manager exposes APIs to let other SAFE-6G components interact with the use-case applications. It is thus important to assert that these APIs are reachable at the expected path after being declared in the corresponding Flask app.

To do so, one of the best candidate API for this is the dt-update-authorization API, as it makes sense in the context of UC1 to initially block modifications on the factory DT until users are authenticated.

Objective	To test one of the available APIs (for instance, factory DT update API) provided by the metaverse manager to check if the Flask routes were correctly initialized
Components	a. Metaverse manager
Requirements	The metaverse manager must allow to enable or disable factory DT updates for UC1, even without any users connected yet.
Features to be tested	dt-update-authorization API
Test Steps	1. Start the Metaverse manager 2. Call the dt-update-authorization API with "areDTUpdatesAllowed" parameter set to false 3. Check if the metaverse manager answers OK (200)

Verification Checklist for MetApp-UT2:

Step	Description	Yes	No	Comments
1	Metaverse manager answers OK (200) to the dt-update-authorization call	☒	☐	

Execution output:

When executing the test, a POST request with a JSON message is sent on the expected dt-update-authorization API url. The request is successful as the API is reachable and handheld correctly the JSON content. Since the boolean for authorizing DT updates was set to false in this JSON content, DT factory updates are now blocked for all users.

```
bailly@IMM-PC-CHB MINGW64 /d/Charles/Code/SAFE-6G/MetaverseManager/metaverse-manager (dev) $
./UnitTests.sh
========================================= test session starts ==========================================
Python 3.11.8, pytest-6.2.4, py-1.11.0, pluggy-0.13.1 rootdir: D:\Charles\Code\SAFE-
6G\MetaverseManager\metaverse-manager plugins: parallel-0.1.1
collected 1 item
test\unit\webapp\test_root.py
    • Serving Flask app 'src.python.network.MainFlaskThread'
    • Debug mode: off Serving Flask app 'src.python.network. authenticationsAPIs.
      AuthenticationHandler' *Debug mode: off
    • Serving Flask app 'src.python.network.immAPIs.unityAPIs. UnityCommHandler' *Debug mode: off

DT factory updates successfully turned off.
================================== 1 passed, 2 warnings in 22.71s ============ ==================
```

Figure 116 MetApp-UT2 output

4.11.5 Integration Tests

MetApp-IT1: Checking metaverse manager <-> SAFE-6G chatbot connectivity

The first main role of the metaverse manager is to make the link between the Unity UC applications and the SAFE-6G chatbot, transmitting user requests and giving back the system answers to inform end-users in XR. The first integration test to perform should thus focus on checking that communications between the metaverse manager and the chatbot were correctly established.

Objective	Once deployed, the metaverse manager should be able to communicate with the SAFE-6G chatbot to transmit user requests and their results
Components	a. Metaverse manager b. SAFE-6G chatbot
Requirements	None
Features to be tested	Metaverse manager <-> SAFE-6G chatbot connectivity
Test Steps	1. Deploy and run the SAFE-6G chatbot 2. Deploy and run the Metaverse manager 3. Using the web GUI, trigger the “dummy user request” function through the corresponding button (UC1 or UC2) 4. Check that the Metaverse manager calls the chatbot API 5. Check that the SAFE-6G chatbot sent a reply to the metaverse manager

Verification Checklist for Integration MetApp-IT1:

Step	Description	Yes	No	Comments
1	Action 1	☒	☐	The reply should contain the text to be displayed for the end user

MetApp-IT2: Checking metaverse manager <-> Unity UC app connectivity

Following the first integration test about the connectivity with the chatbot, the second integration test must assert the connectivity between the metaverse manager and the Unity UC app. This way, a complete communication pipeline between the Unity apps and the SAFE-6G chatbot can be verified.

Objective	To assess that the metaverse manager can properly communicate with any of the Unity UC app
Components	a. Metaverse manager b. Unity UC app (UC1 or UC2)
Requirements	None
Features to be tested	Metaverse manager <-> SAFE-6G chatbot connectivity
Test Steps	1. Deploy and run the Metaverse manager 2. Launch the selected Unity UC app

	3. Enter the initial identification step by entering end-user basic info through the dedicated XR GUI (user id, role, UC). 4. Check that the Metaverse manager receives the corresponding end-user data

Verification Checklist for MetApp-IT2:

Step	Description	Yes	No	Comments
1	Action 1	☒	☒	Check that the end-user data matches exactly with what we provided on the Unity side

Figure 117 Result of the test performed to check the connectivity between the Unity UC app and the SAFE-6G chatbot.

4.11.6 Next Steps

The intermediate version of the metaverse manager has been containerized through Docker. Current work involves finishing to deploy it at UNIWA and/or NCSRD premises. Once deployed, the next major activity will be to test the new authentication pipeline with EVIDEN components and the Security TF.

The deployment of the UC applications is more complex to handle since these Unity applications cannot be containerized and follow the same CI-CD approach than other SAFE-6G components. The 5G connectivity setup was already successfully tested at NCSRD premises for both the Unity UC apps, Current discussions involve the possibility to use lightweight versions of the Unity UC app as a first step to perform integration tests remotely at NCSRD premises, without sending XR headsets at this first stage. This way, remote integration tests will be performed to first validate the complete pipeline between the metaverse component and the SAFE-6G chatbot. The metaverse APIs will then be tested,

iterated if necessary and validated one by one until the final release of the metaverse components at M30.

5 DIGITAL TWINS OF THE COMPONENTS AND NETWORK

Digital Twin (DT) paradigm -understood as a virtual model of a physical object, system, or phenomenon that is represented in the digital world- has already been adopted by other industry sectors with the aim to model complex dynamic systems as they offer the advantage of accurate modelling of such cases. Although it is extensively instrumented by other sectors, it still remains relatively new for 5G/6G networks, despite the obvious support it could provide in taming the complex 5G/6G environment, regarding development, deployment and management aspects.

Especially, the Network DT (NDT), which is considered in SAFE-6G, extends the notion of DTs in the domain of networking, where data regarding network operation and traffic patterns are collected or mined from cloud to edge and are integrated into one system to facilitate the live replica of distinct processes or of the whole 5G/6G network. Data are populating models that represent accurate digital versions of physical entities constituting the real network peers and AI-powered functions are producing results that can achieve an autonomous feedback loop between the 6G physical network and its NDT.

Being the NDT a faithful copy of the real-world network, any input value set can be provided for testing even if these values is possible to cause service disruptions. NDT is executed in a safe environment isolated from the real network. It does not affect real world network operation and only when best models and scenarios are explored, they are fitted to 5G/6G network for adaptation. Thus, the unique capabilities of DTs make them a powerful technology for the design, analysis, diagnosis, simulation, and control of 6G wireless Networks. Several tools provide such emulation and simulation capabilities although in the SAFE-6G ecosystem it became apparent that the best candidates are the EXata tool and the ns3.

5.1 EXATA TOOL

EXata [42] is a real-time network emulation platform designed to accurately reproduce the behaviour of communication networks under controlled, repeatable conditions. It provides a hybrid environment in which simulated network components operate alongside real hardware, applications and protocols, enabling hardware-in-the-loop and software-in-the-loop experimentation. EXata achieves high fidelity by executing protocol stacks, wireless channel models, routing algorithms and mobility patterns in real time, allowing users to observe how a network reacts to operational events such as congestion, failures, interference, mobility or security attacks.

In a deeper dive, Figure 118 EXata architecture illustrates the EXata architecture, where the Simulation Kernel provides scalability for high-fidelity simulations across various platforms. Users interact via an API to develop protocol models. The Emulation Kernel, meanwhile, connects real applications and hardware, processing internal and external events in real-time, ensuring seamless integration with physical network components. Users engage with it through interfaces like Simple Network Management Protocol (SNMP) and Packet Sniffing.

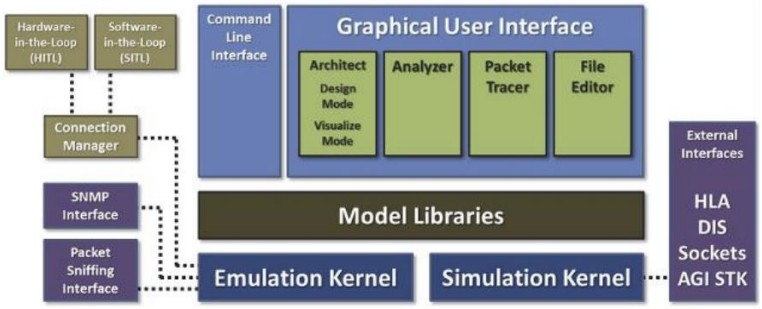


Figure 118 EXata architecture

The tool supports a wide range of communication technologies, including wired, wireless, tactical and mobile ad-hoc networks, and is commonly used in research, defence and industrial settings where deterministic and time-synchronised network experimentation is required. EXata also allows users to import traffic, integrate with external devices through API interfaces, and perform real-world testing without risking service interruption. Its key value lies in its ability to emulate complex network scenarios that behave and "feel" like real systems while still providing the controllability, repeatability and safety of a virtualised environment.

5.2 NS-3 TOOL

NS-3 [43] is an open-source, discrete-event network simulator widely used in academic and industrial research to model the behaviour of communication networks. It provides highly granular simulation of network protocols, physical-layer mechanisms, mobility models, queueing behaviour, scheduling algorithms and E2E application flows. As a simulator rather than an emulator, ns-3 does not operate in real time but instead advances internal events according to a controlled simulation clock, enabling detailed exploration of protocol behaviour, performance metrics and large-scale network configurations.

The ns-3 framework includes comprehensive models for Wi-Fi, Long-Term Evolution (LTE), 5G NR, wired networks, TCP variants, routing protocols and application-level traffic. Its modular architecture allows researchers to extend or replace protocol modules, conduct repeatable experiments, and perform statistically rigorous evaluation of new algorithms or architectural proposals. Because it runs entirely in software, ns-3 is particularly suitable for experimentation with system-level hypotheses, algorithmic comparisons, and large-scale network topologies that would be difficult or costly to reproduce physically.

Core components of ns-3 include Nodes, Packets, and Channels. A Node represents a network element hosting Applications, Protocol Stacks, and NetDevices, which connect to Channels to send and receive packets. Nodes can interface with multiple Channels via NetDevices, using protocols to process data efficiently. Sockets enable Applications to interact with protocol stacks, mirroring real world networking. The high-level architecture is presented in Figure 119.

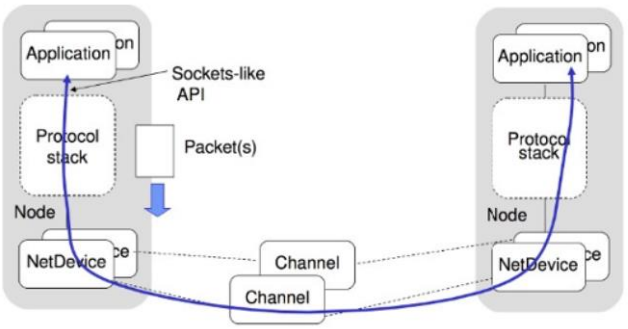


Figure 119 ns-3 architecture

Additionally, NS-3 topologies can be visualized by utilising the NetAnim tool [32] . It enhances NS3 simulations by graphically representing network behaviour, including packet transmissions, node mobility, and routing events. It offers an intuitive view of network dynamics, aiding analysis of communication events and packet routing.

5.3 Digital Twin Integration in SAFE-6G Scenarios

5.3.1 EXATA Scenario

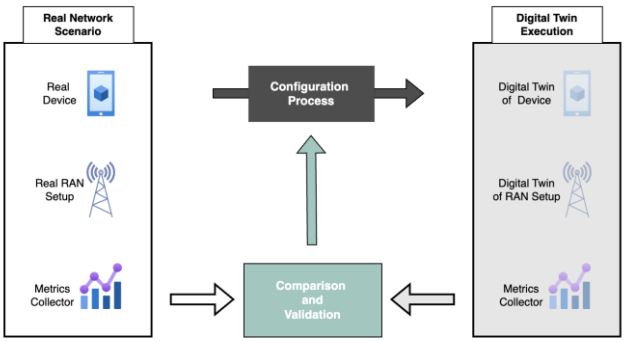


Figure 120 EXata scenario

In SAFE-6G, by utilising the EXata tool, DTs serve as high-fidelity virtual counterparts of real network elements, enabling controlled and measurable experimentation on realistic scenarios. A core use case involves creating a DT of a Radio Access Network (RAN) connection for a mobile device or XR/ Virtual Reality (VR) headset. The DT is configured using the same antenna parameters, radio frequency (RF) conditions, deployment attributes, device profiles, and mobility patterns as the real-world network. They are derived from it. This ensures that the virtual RAN behaves as closely as possible to the physical infrastructure. The real device executes a target scenario, such as estimating the quality of a user's connection based on location, while the same scenario is reproduced within the DT using an identical network topology. The Exata-based simulator models realistic wireless channel impairments, including path loss, shadowing, multipath fading, interference, and noise, and reflects their impact through time-synchronized Physical Layer (PHY), MAC, and Radio Resource Control (RRC) events, signal quality measurements (e.g., Reference Signal Received Power (RSRP), Signal-to-Interference-

plus-Noise Ratio (SINR)), scheduling decisions, and handover dynamics captured in the dataset. Such a topology could be extracted by EXata and an example of NCSRD premises is provided below:

Figure 121 Scenario Definition – User navigation to be tested as shown on Exata app

Figure 122 Topology extracted from EXata including RAN and active UEs for the predefined scenario

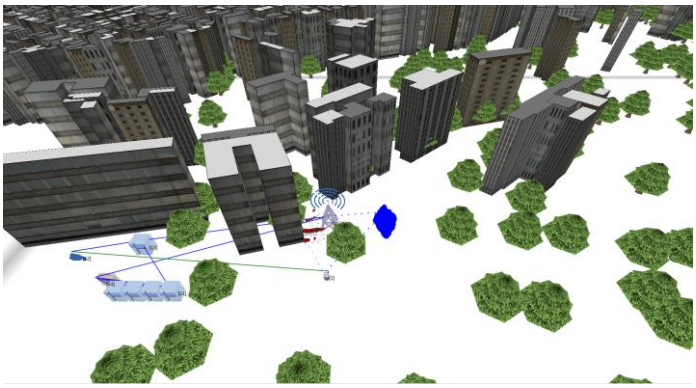


Figure 123 The same topology in a 3D view for better representation as extracted from EXata

Once both the physical environment (real network configuration) and its corresponding DT have been established, scenario executions are performed in parallel. During each execution, SAFE-6G systematically collects and aligns performance metrics from both twins, including throughput, latency, jitter, signal quality, and handover behaviour. The scenario concludes with a quantitative comparison between real and virtual measurements, enabling the assessment of the DT's accuracy, the identification of deviations, and the detection of modelling aspects that require refinement. Based on these observations, iterative adjustments are applied to the DT configuration, progressively reducing metric deviations across successive executions. An indicative configuration example is provided on Figure 124.

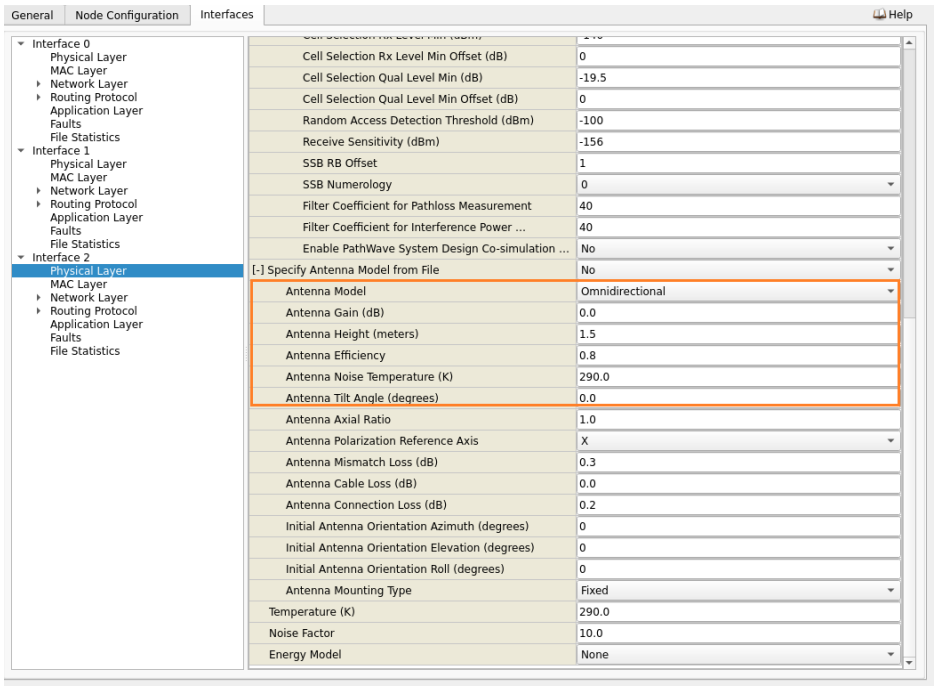


Figure 124 Antenna configuration for the Digital Twin on EXata

When this iterative refinement process converges, the DT reaches a finalized configuration that closely mirrors the behaviour of the physical network. At this stage, the DT becomes an operational decision-making component of SAFE-6G. Specifically, the finalized DT is used to evaluate the real-time

connectivity conditions of an end user, based on the user's location, mobility context, and observed radio and network performance metrics. By emulating the expected network behaviour under these conditions, the DT determines whether the user can sustain an undisturbed and uninterrupted connection that satisfies the trustworthiness requirements of the requested SAFE-6G service.

This DT-based evaluation is executed prior to CoCo's predictions and its interaction with the TFs. In effect, the DT acts as a pre-validation gate: if the simulated user connection metrics meet the required thresholds, CoCo is allowed to proceed with the remainder of the SAFE-6G workflow; otherwise, the flow is halted or adapted accordingly. In this way, the DT ensures that only users whose connectivity conditions are deemed adequate are permitted to advance through the SAFE-6G trustworthiness pipeline.

In addition, the DT can be exploited to proactively evaluate corner cases and rare operational conditions that are difficult, costly, or unsafe to reproduce in the physical network. These include extreme mobility patterns, transient radio impairments, network congestion scenarios, or adversarial conditions affecting trustworthiness KPIs. In this way, the DT enables SAFE-6G to assess robustness margins and anticipate trust degradation before such conditions manifest in real deployments.

A high-fidelity emulator such as EXata is typically used for such purposes because it supports hardware-in-the-loop and software-in-the-loop execution, real-time behaviour, and detailed protocol modelling consistent with real deployments. With EXata, the degree of correlation between real-world and emulated metrics becomes a key validation measure for the reliability of the SAFE-6G ecosystem.

Such methodology enables SAFE-6G to use DTs not only for validation but also for predictive analysis and safe experimentation. Once the DT is calibrated so that its output closely matches real behaviour, it can be used to test "what-if" configurations, evaluate the impact of different mobility patterns or antenna setups, and assess service behaviour under adverse conditions without risking disruptions in the real network.

5.3.2 NS3 Scenario

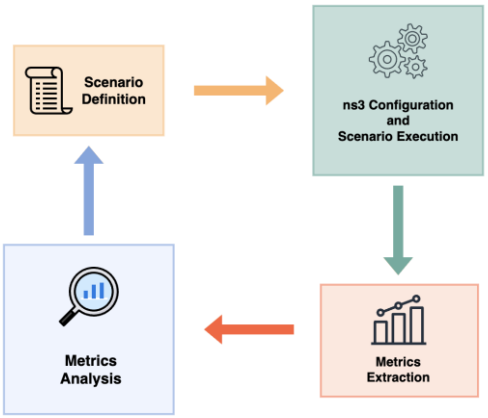


Figure 125 NS3 Scenario

In contrast to high-fidelity network emulators such as EXata, ns-3 does not provide the ability to directly mirror or “map” real antenna configurations, beam patterns, or RF propagation parameters with the same degree of physical realism. NS-3 operates as a discrete-event network simulator and, although it includes advanced 5G/NR modules, it is not designed to ingest full real-world RAN configurations or replicate exact antenna behaviour. Nevertheless, NS-3 remains extremely valuable within SAFE-6G because it supports the construction of complete E2E network scenarios, including UE, gNB, and a full 5G Core implementation using the 5GC modules.

The step-by-step ns3 scenario could be described as follows: First, the scenario is defined (traffic, mobility, UEs, etc.) and then NS3 DT is configured accordingly for the same topology as shown in Figure 126 and simulation is executed.

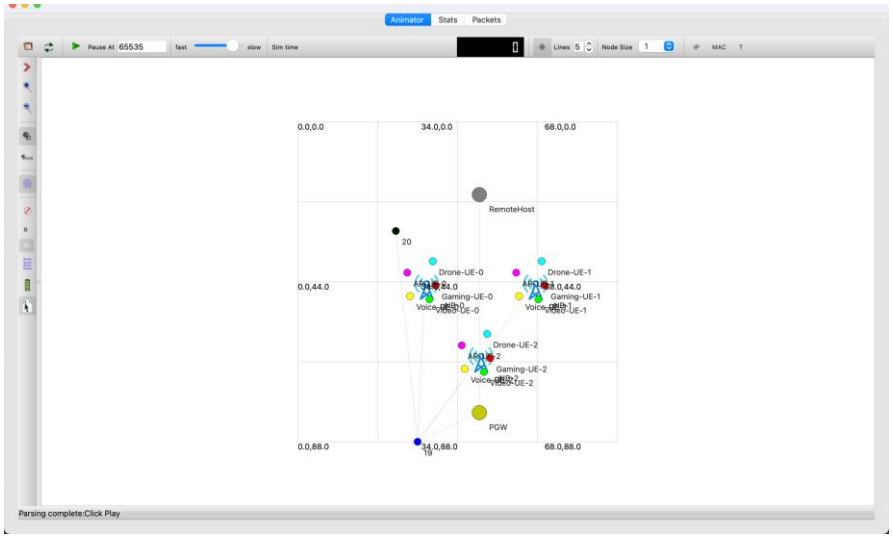


Figure 126 Network Topology Visualization in NetAnim

```
[0/2] Re-checking globbed directories...
ninja: no work to do.

Network-Level Metrics:
Packet Delivery Ratio (PDR): 81.3955%
Packet Loss Ratio (PLR): 18.6045%
Average Delay: 5.32394 ms
Throughput: 214.44 Mbps

Drones Metrics:
Packet Delivery Ratio (PDR): 74.2543%
Packet Loss Ratio (PLR): 25.7457%
Average Delay: 8.82553 ms
Throughput: 13.0239 Mbps

Smartphones Metrics:
Packet Delivery Ratio (PDR): 37.8341%
Packet Loss Ratio (PLR): 62.1659%
Average Delay: 9.07369 ms
Throughput: 15.484 Mbps

Gaming Metrics:
Packet Delivery Ratio (PDR): 98.3797%
Packet Loss Ratio (PLR): 1.62032%
Average Delay: 51.0025 ms
Throughput: 3.19491 Mbps

Video Metrics:
Packet Delivery Ratio (PDR): 79.6671%
Packet Loss Ratio (PLR): 20.3329%
Average Delay: 1.26832 ms
Throughput: 100.475 Mbps

AR Metrics:
Packet Delivery Ratio (PDR): 81.5637%
Packet Loss Ratio (PLR): 18.4363%
Average Delay: 6.4342 ms
Throughput: 82.2628 Mbps
```

Figure 127 Example of NS-3 metrics

After the completion of each execution, the collected metrics are systematically analysed and used for what-if studies, configuration tuning, and the generation of new scenarios, allowing the simulated

environment to progressively reflect real use cases more accurately and to be re-tested. Within such an NS-3–based scenario, SAFE-6G can proactively explore representative patterns, traffic profiles, and service flows, enabling the collection of a wide range of mainly network level performance metrics, including packet loss, jitter, latency, throughput, scheduling behaviour, and handover performance.

Although the fidelity of the wireless layer does not fully replicate a real antenna configuration, ns-3 provides a controlled, repeatable, and analytically precise environment for evaluating protocol behaviour, traffic dynamics, and E2E service delivery. Importantly, ns-3 does not perform reactive or runtime decision-making; instead, it is used to proactively identify critical operating regions, corner cases, and performance degradation patterns. The knowledge derived from these simulations is then encoded into SAFE-6G policies, thresholds, and decision rules, enabling the framework to know how to interpret and act upon real-time metrics should such conditions arise in operational deployments.

In this sense, ns-3 complements DT activities by serving as a simulation-driven, design-time decision support tool, providing validated behavioural insights that guide how SAFE-6G responds to real network conditions, even though exact RF replication and real-time reactions are outside the scope of the simulator, unlike EXata.

5.4 Digital Twin Generated Dataset

Within the scope of a DT framework, a dataset was created to capture and analyze system behavior under realistic simulated conditions. The detailed structure, content, and characteristics of the dataset are described below.

Dataset Description

This dataset captures cross-layer network behavior in a 5G RAN generated through a digital twin simulation. It records radio signal measurements, protocol-layer events, connectivity states, scheduling decisions, and PHY transmission activity across two gNB nodes and UE).

The dataset provides a time-synchronized, multi-layer trace of network operations, enabling detailed inspection of radio performance, protocol dynamics, handover behavior, and resource allocation processes. Events are logged across the PHY, MAC (Layer 2), and RRC/Layer 3 stacks, together with RF configuration metadata and antenna parameters, offering a comprehensive view of network behavior under dynamic radio conditions.

Scenario Description

This scenario models a dynamic 5G network environment with multiple gNB nodes and mobile UEs, capturing realistic radio propagation, connectivity evolution, and handover behavior. It enables the study of multi-PHY, multi-antenna configurations and UE-driven serving cell selection under time-varying signal conditions.

Network Topology

- Multiple gNB nodes identified by NodeId

- Support for multi-PHY and multi-antenna configurations
- Dynamic UE connectivity with evolving serving cell association

UE Behavior

- UE moves across coverage regions within the topology
- Radio signal quality varies over time due to propagation dynamics
- Measurement reports influence serving cell selection and handover decisions

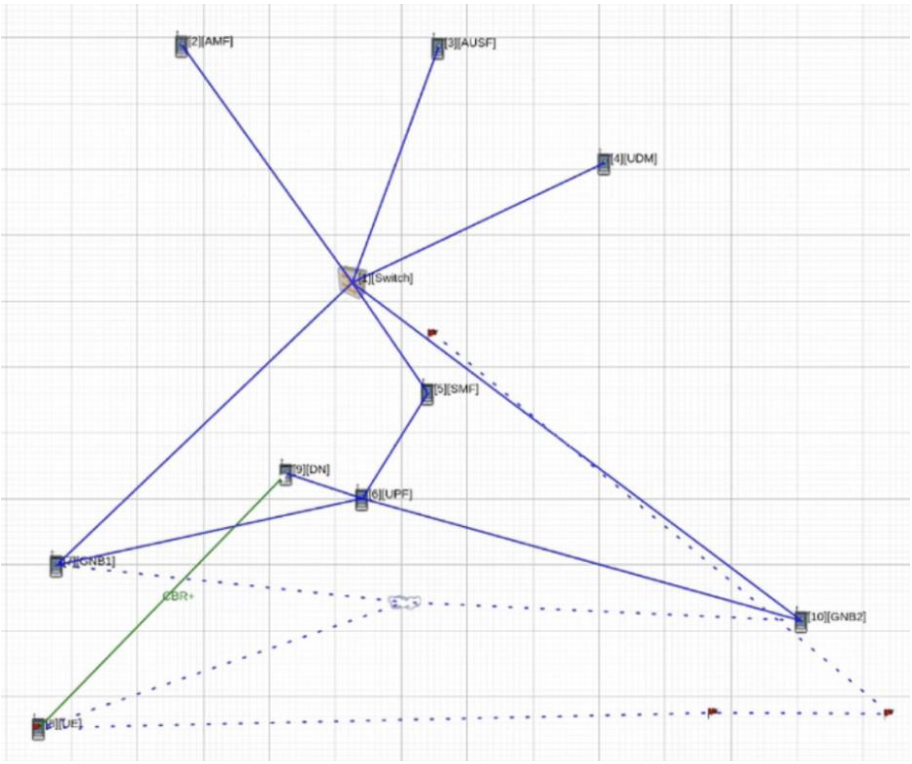


Figure 128 Network Simulation Topology

Dataset Structure (Core Tables and Logged Layers)

- **FiveG_Connectivity** (RRC / Layer 3): Records UE–gNB association and serving cell state
- **FiveG_Measurement_Events** (RRC: Stores RSRP, SINR, and radio signal quality measurements
- **FiveG_Layer3_Events** (Layer 3): Logs handover procedures and RRC signaling events
- **FiveG_Layer2_Events** (MAC / Layer 2): Contains Hybrid Automatic Repeat Request (HARQ) activity, retransmissions, and MAC protocol events
- **FiveG_Scheduler_Events** (MAC): Captures Resource Block allocation, Modulation and Coding Scheme (MCS) selection, and scheduling decisions
- **PHY_Events** (PHY): Records transmission, reception, and channel-related events
- **PHY_Connectivity** (PHY): Tracks physical-layer connectivity states
- **PHY_Summary** (PHY): Provides aggregated link-level performance metrics
- **PHY_Description** (PHY): Stores PHY model configuration parameters
- **ANTENNA_Description** (RF): Contains antenna gain, orientation, and RF characteristics

- **Message_Id_Mapping** (System): Maps event identifiers to protocol semantics

Key Feature Groups

- **UE and Node Identity Metrics**
  - NodeId: Network node identifier (UE or gNB)
  - PhyIndex: PHY interface identifier
  - InterfaceIndex: Network interface mapping
- **Time and Event Tracking Metrics**
  - Timestamp: Simulation time (seconds or milliseconds)
  - RowId: Unique event record index
  - MessageId: Event type and protocol mapping
- **Radio Signal and Channel Metrics** (from FiveG_Measurement_Events and PHY_Summary): These metrics describe signal strength, interference conditions, and propagation loss
  - RSRP: Reference Signal Received Power (dBm)
  - SINR: Signal-to-Interference-plus-Noise Ratio (dB)
  - NoisePerHz: Noise power spectral density (W/Hz)
  - SystemLoss: RF system attenuation (dB)
- **Connectivity and Serving Cell State** (from FiveG_Connectivity and PHY_Connectivity) These metrics enable identification of connectivity transitions and handover boundaries
  - Serving gNB ID: Currently connected base station
  - Connectivity state: Active, Idle, or Switching
  - PHY link status: Up, Down, or Reselection
- **PHY Transmission and Reliability Metrics** (from PHY_Events): These metrics reflect link reliability, retransmission behavior, and communication stability.
  - Transmission and reception event timestamps
  - HARQ retransmission indicators
  - Packet decoding success or failure flags
  - PHY error states, including CRC failures
  - Channel impairment and degradation events

This dataset was generated using the **Keysight EXata Network Modeling and Simulation Platform**, which provides high-fidelity digital twin capabilities for wireless and 5G network simulation. EXata was used to model the 5G RAN topology, radio propagation environment, protocol stack behavior, and event-level telemetry captured in this dataset. The dataset is currently available to consortium partners for internal use and validation, and will be released later as an openly accessible public dataset.

6 Next Steps

The SAFE-6G integration plan follows a phased approach through milestones M18 to M33, with current progress at M25 (intermediate V0.5). The next period toward M26–M33 prioritizes maturing integrations, E2E testing, and validation across WP3/WP4 components, emphasizing DevSecOps, interface standardization, and KPI/KVI validation of the system and delivery of the final version of the

SAFE-6G platform (v1.0). The integration activities will focus on advancing CI/CD pipelines and GitOps automations for containerized deployments across SAFE-6G continuum. Key actions include finalizing OpenCAPIF integration to enable API discovery, authorization, and invocation across all TFs and subsystems, while enhancing the NDT for handling real and synthetic data in AI/ML training. Chatbot will test alternative LLMs for performance and will updates APIs for E2E use-case support with refined formats, validation, and error handling. CoCo will be integrated with xAI to perform tests for fault tolerance against delays, congestion, and multitenancy. Safety TF will expand unit test coverage for Local AI Agent logic alongside E2E black-box validation of trust evaluation cycles from vApp to nApp, ensuring message propagation accuracy and failure handling. Privacy TF will provide more tests for QoS validation, high-load behaviors, and K8s recovery simulating user journeys to verify 5G Core slice parameters and pod lifecycles. Resiliency TF will continue the iterative process to develop its modules. Reliability TF will enhance its resilience, accuracy, and operational efficiency, by providing more enhanced testing and validation procedures and will extend unit/integration tests via GitOps pipelines, incorporate OpenCAPIF, finalize production DevOps/MLOps, and provide the high flavor of the vApp. Metaverse Applications will support the validation phase and will provide their final version on M30. Thus, for next period the primary focus is on refining both the architecture and functionality of these modules to ensure they are closely aligned with the overarching objectives of the project. In parallel, the DevOps practices will ensure zero-downtime scalability, prioritizing interface/code separation, standardized protocols, and use-case demonstrations of the capabilities of the SAFE-6G.

7 CONCLUSION

This deliverable documents the integration of SAFE-6G modules developed collaboratively across the project consortium. SAFE-6G adopts DevSecOps practices, which extend the DevOps methodology by embedding security considerations throughout the entire software development and operational lifecycle, to orchestrate a coherent end-to-end platform integration grounded in Agile software development principles.

The deliverable also establishes the integration foundation through three key contributions: first, a detailed exposition of integration guidelines, deployment architectures, and the SAFE-6G laboratory environments (development, testing, and production); second, a systematic assessment of integration and validation status across the platform's core components; and third, the design framework for the Digital Twin capability.

Further, the current document tracks the progressive combination of components developed in WP3 and WP4 into a unified system architecture within the WP5 integration framework. This analysis documents the maturity level, interoperability readiness, and deployment status of each module as instantiated in the SAFE-6G production environment. The integrated components comprise: (i) the communication and orchestration layer, encompassing the Chatbot, Cognitive Coordinator, and Message Broker; (ii) the trustworthiness foundation, consisting of the Safety, Security, Privacy, Resilience, and Reliability Trust Functions; and (iii) the end-user service interface, realized through the Metaverse application.

The deliverable introduces the Digital Twin architecture designed to extend beyond validation use cases. SAFE-6G leverages Digital Twins for predictive analysis of network performance, dynamic trustworthiness estimation, and controlled experimentation across the trustworthiness framework, enabling the platform to anticipate failure modes, optimize resource allocation, and validate edge-cloud continuum behavior before live deployment.

Overall, this incremental approach establishes the technical foundation for the subsequent WP5 tasks, enabling continuous validation against project objectives while maintaining the flexibility to incorporate emerging requirements and lessons learned from the metaverse-based pilots.

8 REFERENCES

[1] Docker, "Docker: Empowering App Development for Developers.," [Online]. Available: https://www.docker.com/.
[2] Docker Docs, "Docker image overview.," [Online]. Available: https://docs.docker.com/get-started/overview/.
[3] Kubernetes, "Production-Grade Container Orchestration.," [Online]. Available: https://kubernetes.io/.
[4] GitLab, "GitLab: The complete DevSecOps platform," [Online]. Available: https://gitlab.com/.
[5] PyCQA, "Bandit: Security linter for Python.," [Online]. Available: https://bandit.readthedocs.io/.
[6] Semgrep, "Semgrep: Lightweight static analysis for many languages.," [Online]. Available: https://semgrep.dev/.
[7] 6G SANDBOX Project, "6G SANDBOX: Athens Platform Overview.," [Online]. Available: https://6g-sandbox.eu/.
[8] aerOS Project, "aerOS: Meta Operating System for the IoT edge cloud continuum.," [Online]. Available: https://aeros-project.eu/.
[9] Proxmox Server Solutions, "Proxmox Virtual Environment: Open source server virtualization platform.," [Online]. Available: https://www.proxmox.com/en/proxmox-virtual-environment.
[10] WireGuard, "WireGuard: Fast, modern, secure VPN tunnel.," [Online]. Available: https://www.wireguard.com/.
[11] OPNsense, "OPNsense: Open-source firewall and routing platform.," [Online]. Available: https://opnsense.org/.
[12] Apache Software Foundation, "Apache HTTP Server Project.," [Online]. Available: https://httpd.apache.org/.
[13] OpenEBS, "OpenEBS: Container Attached Storage for Kubernetes.," [Online]. Available: https://openebs.io/.
[14] Linux Foundation, "NFS: Network File System.," [Online]. Available: https://wiki.linuxfoundation.org/nfs/.
[15] cert-manager, "cert-manager: Kubernetes native certificate management.," [Online]. Available: https://cert-manager.io/.
[16] Let's Encrypt, "Let's Encrypt: Free, automated, open certificate authority," [Online]. Available: https://letsencrypt.org/.
[17] Prometheus Authors, "Prometheus: Monitoring system & time series database," [Online]. Available: https://prometheus.io/.
[18] Grafana Labs, "Grafana: Open-source analytics & monitoring platform," [Online]. Available: https://grafana.com/.
[19] Grafana Labs, "Loki: Log aggregation system," [Online]. Available: https://grafana.com/oss/loki/.
[20] NGINX, "NGINX Ingress Controller for Kubernetes," [Online]. Available: https://www.nginx.com/products/nginx-ingress-controller/.
[21] LoxiLB, "LoxiLB: Cloud-native load balancer for 5G and edge," [Online]. Available: https://loxilb.io/.

[22] Cilium, "Cilium: eBPF-based networking, observability, and security," [Online]. Available: https://cilium.io/.
[23] FluxCD, "Flux: GitOps toolkit for Kubernetes," [Online]. Available: https://fluxcd.io/.
[24] University of West Attica, "UNIWA AI Innovation Hub: Development of UNIWA Infrastructure for Research and Innovation in Artificial Intelligence and its Applications," [Online]. Available: https://ai-innohub.uniwa.gr/.
[25] OpenStack Foundation, "OpenStack: Open source cloud computing platform," [Online]. Available: https://www.openstack.org/.
[26] Canonical, "Ubuntu: Linux operating system," [Online]. Available: https://ubuntu.com/.
[27] Rook Authors, "Rook: Storage Orchestration for Kubernetes," [Online]. Available: https://rook.io/.
[28] Ceph Foundation, "Ceph: Open-source distributed storage system," [Online]. Available: https://ceph.io/.
[29] Argo Project, "Argo CD: Declarative GitOps for Kubernetes," [Online]. Available: https://argo-cd.readthedocs.io/.
[30] Rocky Enterprise Software Foundation, "Rocky Linux," [Online]. Available: https://rockylinux.org/.
[31] NVIDIA, "NVIDIA DGX A100: Universal system for AI workloads.," [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-a100/.
[32] Bright Computing, "Bright Cluster Manager.," [Online]. Available: https://www.brightcomputing.com/.
[33] NVIDIA, "CUDA-X: GPU-accelerated libraries for AI and HPC.," [Online]. Available: https://developer.nvidia.com/cuda-x.
[34] Kubeflow Authors, "Kubeflow: The Machine Learning Toolkit for Kubernetes.," [Online]. Available: https://www.kubeflow.org/.
[35] MinIO, "MinIO: High performance object storage.," [Online]. Available: https://min.io/.
[36] Dex, "Dex: OpenID Connect Identity Provider.," [Online]. Available: https://dexidp.io/.
[37] Istio Authors, "Istio: Open platform for service mesh.," [Online]. Available: https://istio.io/.
[38] Netgate, "pfSense: Open-source firewall and router.," [Online]. Available: https://www.pfsense.org/.
[39] HAProxy Technologies, "HAProxy: High performance TCP/HTTP load balancer.," [Online]. Available: https://www.haproxy.org/.
[40] Software Radio Systems, "srsLTE / srsRAN: Open source 4G/5G software radio suite," [Online]. Available: https://www.srslte.com/.
[41] OAIBOX, "OAIBOX: 5G/6G Open RAN and Core Network Solutions," [Online]. Available: https://www.oaibox.com/.
[42] Open5GS, "Open5GS: Open Source 5G Core Project.," [Online]. Available: https://open5gs.org/.
[43] Amarisoft, "Amarisoft 5G Software Suite.," [Online]. Available: https://www.amarisoft.com/.
[44] Apache Software Foundation, "Apache Kafka: Introduction and Getting Started," [Online]. Available: https://kafka.apache.org/41/getting-started/introduction/.
[45] Redpanda Data, "Kafka Architecture and Kafka Broker Guide," [Online]. Available: https://www.redpanda.com/guides/kafka-architecture-kafka-broker.

[46] ETSI, "OpenCAPIF Framework (OCF) Portal," [Online]. Available: https://ocf.etsi.org/.
[47] Unity Technologies, "Unity Real Time 3D Development Platform," [Online]. Available: https://unity.com/.
[48] Meta Platforms, "Meta Quest 3: Mixed Reality Headset," [Online]. Available: https://www.meta.com/quest/quest-3/.
[49] Keysight Technologies, "EXata Network Modeling and Simulation Software," [Online]. Available: https://www.keysight.com/.
[50] ns-3 Consortium, "NetAnim: Network Animator for ns 3," [Online]. Available: https://www.nsnam.org/wiki/NetAnim.

ANNEX 1

Description of message broker topics and messages

Topic Name	Message Name	Publisher	Consumer	Message structure
reliability_trust_score	RL_LoW	COCO	Relaibility Func	{ "service _id": "2072e0f0-4e67-45ad-b1cf-dc6fa5ce28ee", "user_id": "ASD23456", "nlotw": { "Reliability": 25.283350918358117 }, "clotw": { "Reliability": 39.99999999999999 } }
resilience_trust_score	RS_LoW	COCO	Resilience Func	{ "timestamp": "2025-10-05T12:00:00Z", "nlotw": { "Resilience": 90.05, }, "userID": "340601234567890", "userType": "XR Glasses", "service_id": "xxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" }
privacy_trust_score	PR_LoW	COCO	Privacy Func	{ "timestamp": "2025-10-05T12:00:00Z", "clotw": { "Resilience": 90.05, }, "flavor": "high", "userID": "340601234567890", "userType": "XR Glasses", "service_id": "xxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx", }
security_trust_score	SE_LoW	COCO	Security Func	{ "service_id": "123456", "user_id": "ABCDEF", "idempotency_id": UUID, "nlotw": { "Privacy": 80 }, "clotw": { "Privacy": 60

				} }

safety_trust_score	SA_LoW	COCO	Safety Func	{
				"nlotw": {
				"Security": 55.55,
				},
				"clotw": {
				"Security": 89.0,
				},
				"service _id": "xxxx-xxxxxxxx-xxxx",
				"user_id": "xxxx-xxxxxxxx-xxxx",
				}

reliability_report	RL_EVT	Relaibility Func	COCO	{
				"SUPI": {
				"value": "supi",
				"description": "Subscription Permanent Identifier",
				"required": true
				},
				"MSISDN": {
				"value": "msisdn",
				"description": "Mobile Station International Subscriber Directory Number",
				"required": false
				},
				"IMEI": {
				"value": "imei",
				"description": "International Mobile Equipment Identity",
				"required": true
				},
				"USR": {
				"value": "usr",
				"description": "User Role",
				"required": true
				},
				"Target_App_ID": {
				"value": "*********",
				"description": "Can be used as DNN or Application IP address and port",
				"required": true
				},
				"clotw": {
				"value": "****",
				"description": "Level of trustworthiness",
				"required": true
				}
				}

resilience_report	RS_EVT	Resilience Func	COCO	{
				"service _id":"2072e0f0-4e67-45ad-b1cf-dc6fa5ce28ee",
				"user_id":"ASD23456",
				"timestamp":1760609692,
				"event":{
				"action":"Horizonal scale up",
				"desrcription":"Reliability predicted lack of memory resources and proceed with a scale up action"
				}
				}

privacy_report	PR_EVT	Privacy Func	COCO	{
				"timestamp": "2025-10-05T12:00:00Z",
				"service _id": "xxxx-xxxxxxxx-xxxx",
				"user_id": "xxxx-xxxxxxxx-xxxx",
				“resilience_flavour": “high",
				"alotw": {
				"Resilience": 90.05,
				}
				}

security_report	SE_EVT	Security Func	COCO	{
				"timestamp": "2025-10-05T12:00:00Z",
				“clotw": {
				"Resilience": 90.05,
				},
				"flavor": "high",
				"userID": "340601234567890”,
				"userType": “XR Glasses”,
				"service_id": "xxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
				}

safety_report	SA_EVT	Safety Func	COCO	{
				"timestamp": "2025-10-05T12:00:00Z",
				"service _id": "xxxx-xxxxxxxx-xxxx",
				"user_id": "xxxx-xxxxxxxx-xxxx",
				"resilience_flavour": “high”,
				"userType": “XR Glasses”,
				"action": {
				"RB1": xxx,
				"RB2": yyy
				}
				}