TALON

Autonomous and Self-organized Artificial Intelligent Orchestrator for a Greener Industry 4.0

Deliverable

D5.2 Initial TALON Platform Setup, Operation, Continuous Integration & Maintenance Report

Actual submission date: 01/08/2024

Project Number:	101070181		
Project Acronym:	TALON		
Project Title:	Autonomous and Self-organized Artificial Intelligent Orchestrator for a Greener Industry 4.0		
Start date:	October 1st, 2022	**Duration:**	36 months
D5.2 Initial TALON Platform Setup, Operation, Continuous Integration & Maintenance Report			
Work Package:	WP5		
Lead partner:	INTRA		
Author(s):	Magda Foti (INTRA)		
Reviewers:	Konstantinos Ntontin, Ons Aouedi (UL) Stylianos Trevlakis, Vasileios Kouvakis, Lamprini Mitsiou, Theodoros Tsiftsis (IC)		
Due date:	31/07/2024		
Deliverable Type:	R	**Dissemination Level:**	PU
Version number:	1.0		

Revision History

Version	Date	Author	Description
0.1	17/06/2024	INTRA	Table of Contents release
0.2	01/07/2024	UBI	Contributions in Sections 3 & 4
0.3	15/07/2024	INTRA	Contributions in Sections 1 & 2
0.4	17/07/2024	INTRA	Version ready for review
0.5	24/07/2024	UL, IC	Deliverable reviews with embedded comments
0.6	24/07/2024	INTRA, UBI	Review comments addressed; Sent for quality review
0.7	31/07/2024	INTRA, UBI	Final version; Quality check comments addressed
1.0	31/07/2024	ENG	Final coordinator review before submission

Table of Contents

List of figures

Definitions and acronyms

AI	*Artificial Intelligence*
APIs	*Application Programming Interfaces*
AR	*Augmented Reality*
CA	*Consortium Agreement*
CI	*Continuous Integration*
CD	*Continuous Deployment (or Continuous Delivery)*
DoA	*Description of Action*
EC	*European Commission*
EU	*European Union*
EC2	*Amazon Elastic Compute Cloud*
E2C	*Edge-to-cloud*
FL	*Federated Learning*
GA	*Grant Agreement*
JSON	*JavaScript Object Notation*
K8S	*Kubernetes*
KPI	*Key Performance Indicator*
MS	*Milestone*
NG-SDN	*Next Generation – Software Defined Network*
PaaS	*Platform as a Service*
SLOs	*Service Level Objectives*
TrL	*Trust Level*
UATV	*Automatic unnamed arial and terrestrial vehicle*
UC	*Use Case*
UI	*User Interface*
QoS	*Quality of Service*
XAI	*Explainable Artificial Intelligence*
VR	*Virtual Reality*
WP	*Work Package*

Disclaimer

Executive Summary

The vision of TALON is to design and develop next-generation industrial systems in terms of performance, adaptation, explainability, trustworthiness and transparency. TALON aims at sculpturing the road towards the next industrial revolution by developing a fully automated AI architecture capable of bringing intelligence near the edge in a flexible, adaptable, explainable, energy and data efficient manner. In this direction, TALON researches and develops a rich set of components that enable intelligent, energy efficient, trusted, human-centred, AI-based orchestration of resources across the cloud-edge continuum. Most importantly, the project integrates and demonstrates these components in several real-life use cases and demonstrators. The implementation of such demonstrators is a challenging task, as it requires the integration of diverse and technologically heterogeneous components, which asks for a structured planning of the integration work. Likewise, the evaluation of the demonstrators asks for a structured methodology that considers technological, integration and user-centred aspects.

In this context, the present **"Deliverable D5.2 – Initial TALON Platform Setup, Operation, Continuous Integration & Maintenance Report**" aims to describe the platform set-up and integration steps, along with an overview of deployment of the TALON infrastructure. Specifically, D5.2 provides a detailed description on how the TALON Platform setup and operation process was conducted *until month 22* of the project. The work carried out within the scope of this deliverable has been pivotal for the successful realisation of the integration process of key technologies and functionalities, which required the systematic deployment of the TALON infrastructure.

In addition, TALON infrastructure along with the integration of its components will be deployed in each of the following use cases: (i) Automatic UATVs Coordination, (ii) I5.0 Automation and Planning, (iii) AR/VR for Training and Maintenance and (iv) Human-Robot Collaboration, specifying the architectural modules that will be tested during the demonstration activities after each release.

Finally, a robust deployment and testing approach is employed to ensure an effective integration process, leading to the release of both an early prototype and a final version of the TALON platform.

I Introduction

1.1 Objective of the Deliverable

The main objective of this deliverable is to provide an overview of how the TALON Platform setup and operation process is conducted. The work carried out within the scope of this deliverable has been pivotal for the successful realisation of the integration process of key technologies and functionalities, which required the systematic deployment of the TALON infrastructure.

Firstly, interfaces are developed, followed by a step-by-step integration of modules to adhere to the specified development timeline. The ultimate goal is to create a fully functional platform that integrates all elements seamlessly, resulting in a prototype that is ready for comprehensive testing and validation.

To achieve this, a robust iterative approach is employed, ensuring that the platform meets the practical needs and expectations, enhancing its functionality and reliability. Throughout the evolution of the integration process, specific structures and templates for the developed components are adopted. This adoption is facilitated by means of specialised tools for version control, testing, bug tracking, and more, which are essential for maintaining the integrity and efficiency of the development process. Furthermore, a comprehensive Unit Testing and Early Platform Integration Testing plan was set up to support the technical verification of the TALON Platform, including unit testing and integration testing.

In summary, D5.2 is dedicated to ensuring that the TALON platform is not only successfully integrated but also operates efficiently and evolves continuously.

1.2 Relation to other Work Packages

This is the second deliverable of “WP5 - Integration, Validation and Demonstration” series of deliverables. The work reported in this document is part of **Task 5.2: TALON Platform Setup, Operation, Continuous Integration & Maintenance** (M19-M36). Task 5.2 and its output (i.e. D5.2) utilises the following input: D3.1, D3.2, D3.3 and D4.1. In addition, the outputs of this deliverable will be used as part of T3.1: Overall Architecture & Platform Design; T3.2: Enabling Common NG-SDN & Distributed Intelligence Functionalities; T3.3: Zero-Touch AI-Orchestrator; T3.4: AI-based Resource Coordination Through Computation Offloading & Social Aware Caching; and T4.4: Security & Privacy Blockchain Mechanisms.

Completing D5.2 marks the attainment of Milestone 5 (MS5) - the first, early release of the TALON Integrated platform.

The operations of Task 5.2 will continue after the submission of D5.2 until the end of the project (M36) in order to support the integration, operation and demonstration of the TALON platform. The results of the application of the formulation described in this task will be finally discussed on D5.4: Final TALON Platform Setup, Operation, Continuous Integration & Maintenance Report (M36).

1.3 Structure of the Document

This document is structured in 5 Chapters:

- **Chapter 1** provides an overall introduction to the deliverable objectives and positioning within the project.
- **Chapter 2,** titled "*Initial TALON Platform Setup*," provides an in-depth look at the TALON Platform setup and operation, emphasizing the deployment of the infrastructure essential for the successful realization of the integration process. This chapter describes key concepts related to Kubernetes and its components, including containers, pods, nodes, and clusters and provides the description of an example application deployed in TALON infrastructure.

- **Chapter 3,** titled "*Continuous Integration and Continuous Deployment*", explores the CI/CD process for the TALON Platform's components, elements, and modules It delves into how CI/CD methods are applied to build, test, deploy, and monitor iterative code changes in TALON. Additionally, it describes how GitLab's tools automate these processes, detect bugs early, and ensure that the code meets production standards, thereby enhancing development efficiency and quality.
- **Chapter 4**, titled "Unit Testing and Early Platform Integration Testing," addresses the testing processes for TALON's technological components. It covers unit testing for verifying individual code units and early platform integration testing for evaluating component interactions. The chapter explains how these methods are used to develop initial and final prototypes.
- Finally, **Chapter 5** concludes the deliverable along with the outlook.

2 Initial TALON Platform Setup

2.1 Kubernetes

To efficiently perform integration for the TALON platform, we have followed the following methodology, utilizing the Kubernetes framework. We have: (i) set up a Kubernetes (K8S) cluster; (ii) defined the application resources via YAML files for the deployment of the various services / pods; (iii) configured persistent storage and inter-pod communication via networking mechanisms; and (iv) implemented secure policies, access mechanisms and abundant monitoring of the infrastructure.

K8S is an open-source system that allows TALON developers to manage containerized workloads and services. We have used K8S in TALON to automate the deployment and serving of APIs and services, and therefore allow applications to adjust to dynamic changes automatically, with speed, efficiency and minimal downtime. Also, it allows in collaboration with the Continuous Integration (CI) and Continuous Deployment (CD) principles (detailed in Section 3) to manage and update applications or services in production incorporating improvements without disrupting the functionality

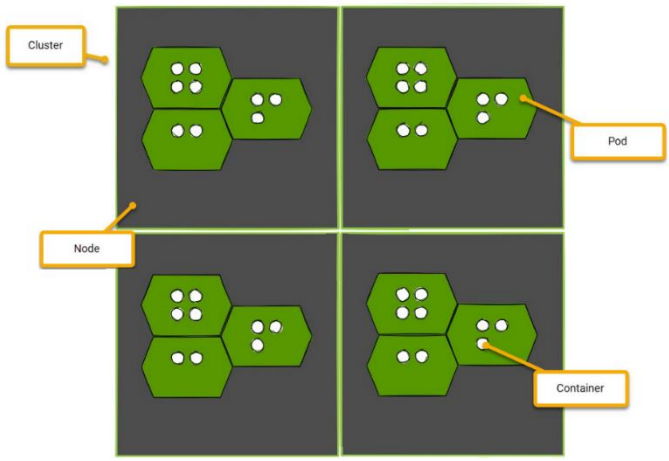


Figure 1: Kubernetes hierarchy. Adopted from here[1]

of other services. Kubernetes consists of the following key concepts working together (Figure 1):

(i) ***Containers*** are ready-to-run software packages containing code, runtime content, settings, and system libraries;

(ii) ***Pods*** are collections of these containers;

(iii) ***Nodes*** house the pods and execute workloads;

(iv) ***Clusters***, in turn, contain multiple nodes, orchestrating the entire system.

2.1.1 Containers

Containers are the lowest level of the Kubernetes hierarchy and an efficient way to bundle and run the applications. They are packages of applications or services that are bundled together with their execution environments. Containerized applications act the same whether they're on a laptop or distributed server.

The TALON components will run as containers that are deployable and maintainable through the advanced utilities of the Kubernetes framework.

However, it is crucial to manage these containers to prevent downtime in TALON's development and deployment environment. For instance, if a container fails, another replacement container should automatically start. Kubernetes simplifies this process by offering a framework for running the

distributed system resiliently. It carries out scaling, failover, deployment patterns such as canary deployment ensuring the applications run smoothly and efficiently.

2.1.2 Pods

Pods are the smallest deployable units of computing in Kubernetes. They consist of one or more application containers that share network and storage resources, making them relatively tightly coupled. A pod can also include *init containers* that run during the pod's startup.

A pod consists of several key components: (i) Application Containers, which are the primary containers running the TALON applications, (ii) Init Containers, which run before the main application containers during pod startup, and (iii) Shared Storage Volumes, which all containers within a pod can access. These shared volumes allow containers to share data and ensure persistent data survives if a container needs to be restarted.

2.1.3 Nodes

A node is a machine that runs docker containers in pods. In a hosted Kubernetes, a virtual machine is used to run workloads.

Typically, a Kubernetes cluster consists of several nodes. Each node is managed by the control plane and contains the essential services required to run pods. Specifically, each node runs the following:

(i) **Kubelets**: An agent that monitors the state of the node, ensuring that your containers are healthy.

(ii) **Workloads**: The containers and pods that hold your applications, as well as other types of deployments.

2.1.4 Clusters

Clusters use the Kubernetes container-orchestration system to deploy, maintain, and scale Docker containers. A cluster is a group of computers that work together as a single system. Therefore, a Kubernetes cluster consists of components that represent the control plane and a set of machines called nodes.

The core of Kubernetes' control plane is the application programming interface (API), which lets you query and manipulate the state of objects in Kubernetes. These objects are persistent entities that Kubernetes uses to represent the state of your cluster. In a hosted Kubernetes cluster, resources are isolated using a geographical region and a virtual private network.

2.1.5 E2C AI Orchestrator within Kubernetes

Integrating the E2C AI Orchestrator within Kubernetes offers numerous benefits, leveraging Kubernetes' robust container orchestration capabilities to enhance efficiency, scalability, and automation. By deploying the orchestrator on Kubernetes, it can seamlessly collect data from end devices, manage AI models, and datasets dynamically across nodes, and optimize resource allocation based on network conditions and application requirements. Kubernetes' ability to automate the deployment, scaling, and operation of application containers facilitates a zero-touch approach, providing an autonomous, adaptable, and transparent resource orchestration solution. Additionally, Kubernetes' ecosystem includes powerful monitoring tools like Prometheus, InfluxDB, and Kepler, which are essential for real-time pod stats monitoring. These tools enable the orchestrator to gather detailed metrics on resource utilization, detect patterns in computing resource availability, and ensure optimal energy efficiency. The orchestrator can make data-driven decisions to reduce energy consumption while maintaining high performance and reliability. By leveraging Kubernetes and its monitoring stack, the E2C AI Orchestrator can deliver a scalable and efficient solution that meets the evolving demands of modern network environments.

As a first demonstration of the capabilities of the E2C AI Orchestrator, we have developed a PoC that allows users to set optimisation parameters for an interpolation task. This initial version of the

orchestrator allows the user to choose between two different algorithms based on different aspects of the task. The orchestrator will then apply the algorithm that best meets the user's requirements and effectively solves the task.

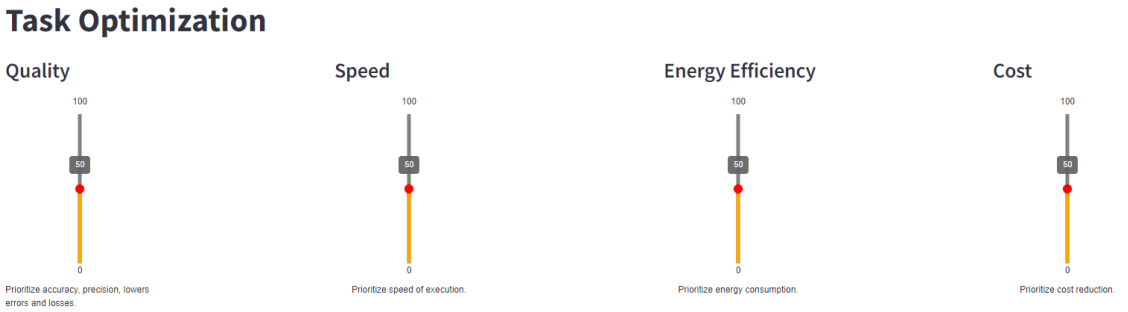


Figure 2: Task optimization parameters

Once deployed, Kubernetes enables monitoring of the algorithm's resource and energy consumption. This capability allows capturing a snapshot of the cluster at any given time, ensuring that future deployments can be optimized to best meet requirements and adapt to the cluster's current status.

Kubernetes' robust orchestration capabilities, coupled with advanced monitoring tools such as Prometheus, InfluxDB and Kepler, provide real-time insight into resource utilisation and energy consumption. This ensures that deployments are continuously optimised to meet evolving requirements and adapt to the state of the cluster. The orchestrator's ability to dynamically deploy the most appropriate algorithms based on user input further enhances its effectiveness and user-centric approach. Overall, the use of Kubernetes enables the E2C AI Orchestrator to deliver a high-performance, energy-efficient and autonomous resource management solution that meets the demands of modern network environments.

2.2 Current Deployments in TALON Infrastructure

In the TALON context, the deployed applications are represented or conceptualised as Directed Acyclic Graphs (DAGs). Each DAG is a pod deployed on a Kubernetes node following the concepts presented above. Figure 3 depicts a custom application implemented as an indicative E2C example to collect the initial data and verify the functionalities of the frameworks that will be extended within the TALON's lifecycle. This example application is similar to the E2C TALON Use Cases and facilitate to progress the technical development of the NG-SDN and Distributed Intelligence functionalities and other components within the project. The edge producers are applications which generate data. The message broker (e.g., RabbitMQ[1]) manages this data and directs them in the cloud, either in a relational database (e.g., PostgreSQL[2]) or in a reporting database (e.g., MongoDB[3]).

We have deployed two producer applications that run on the edge and stream data to the RabbitMQ server deployed on the cloud. Then a Spring boot[4] consumer retrieves the data from the message queue and stores them to a MongoDB server. Moreover, the Spring boot application periodically receives HTTP requests and stores the results on a PostgreSQL server. This can be considered as a typical application workflow that both handles online and offline data. In the rest of this document, we will focus on the online workflow of the application which is more challenging, and the AI algorithms can detect patterns in the generated time series.

[1] https://www.rabbitmq.com/documentation.html
[2] https://www.postgresql.org/docs/
[3] https://www.mongodb.com/docs/
[44] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

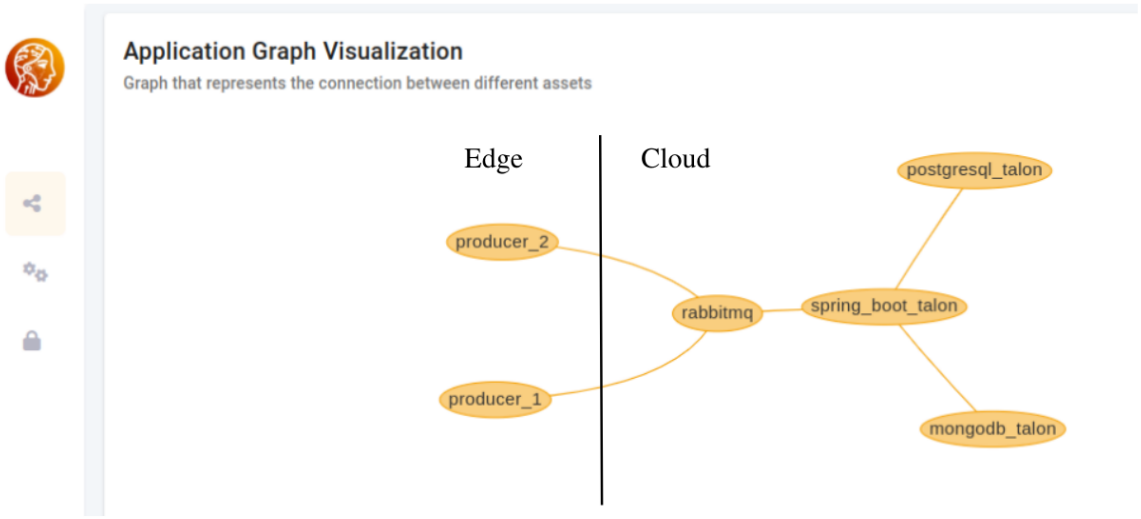


Figure 3: Example E2C Application deployed in TALON infrastructure.

3 Continuous Integration and Continuous Deployment

CI/CD, which stands for Continuous Integration and Continuous Deployment (or Continuous Delivery), is a continuous method of software development, where you continuously build, test, deploy, and monitor iterative code changes aiming to automate and improve the process of software development reducing the chance that you develop new code based on buggy or failed previous versions.

The TALON project implements CI/CD in the context of GitLab[5], using GitLab's integrated tools to streamline the workflow from code development to deployment. GitLab CI/CD help TALON developers catch bugs early in the development cycle and help ensure that the code deployed to production complies with the established code standards.

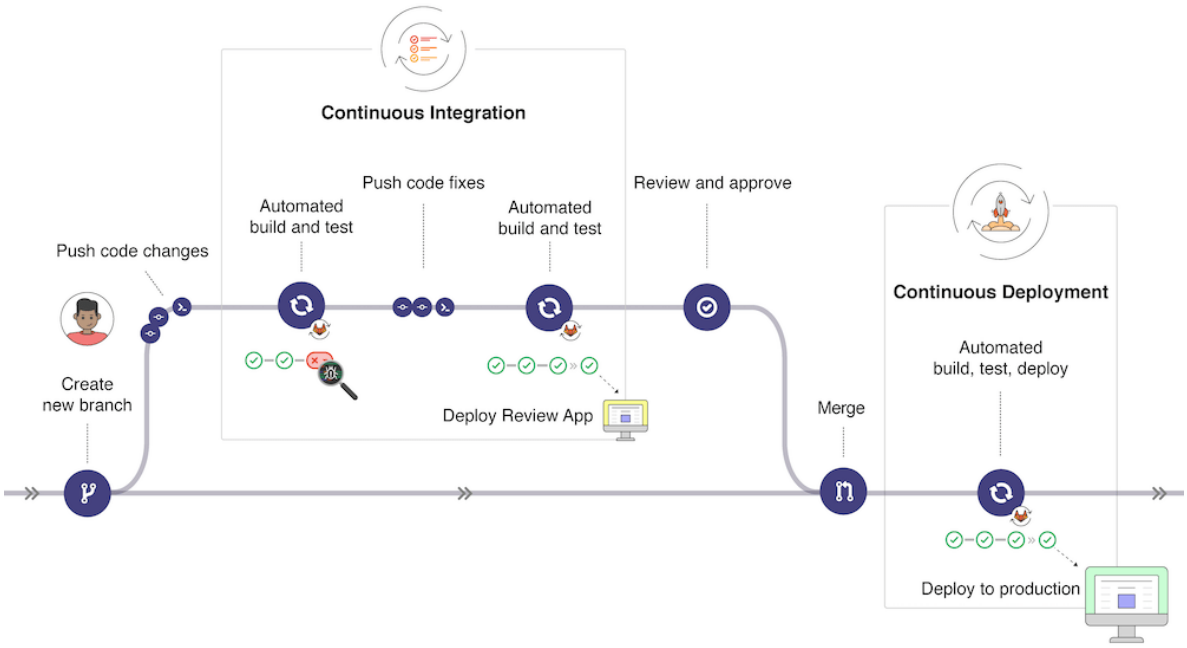


Figure 4: CI/CD Lifecycle in GitLab. Adopted from here[6].

3.1 Continuous Integration (CI)

Continuous Integration is the practice of automatically building and testing code changes as they are committed to the repository. In GitLab CI, this is achieved through the following:

GitLab CI/CD Pipelines: These are defined in a ".gitlab-ci.yml" file located in the root of the repository. This file specifies the steps (jobs) to be executed whenever a change is pushed to the repository.

Jobs and Stages: A pipeline consists of multiple stages (e.g., build, test, deploy), and each stage contains one or more jobs that run sequentially. Jobs in the same stage run in parallel.

Runners: GitLab Runners are agents that execute the jobs defined in the pipeline. Runners can be shared, specific to a project, or dedicated to a particular group or instance.

A simplified ".gitlab-ci.yml" file used in one of the TALON's modules is presented below, specifying the CI/CD pipelines, the relevant jobs and stages and the variables.

[5] https://about.gitlab.com/
[6] http://www.bioinfotiget.it/gitlab/help/ci/introduction/index.md

```yaml
stages:
  - dependencies
  - build
  - test
  - publish
  - analyze
install_dependencies_job:
  stage: dependencies
  image: node:$NODE_VERSION
  script:
    - npm install --legacy-peer-deps
test_job:
  stage: test
  image: node:$NODE_VERSION
  before_script:
    …
  script:
   ...
additional-test-job:
  stage: test
  image: trion/ng-cli-karma
  allow_failure: false
  before_script:
    …
  script:
    …
build-job:
  stage: build
  image: node:$NODE_VERSION
  script:
  only:
    - master
    - dev
    - tags
    policy: pull
publish_job:
  stage: publish
  image:
    name: ….
    entrypoint: [""]
  script:
    …
analyze_job:
  stage: analyze
  only:
    - master
  allow_failure: true
  image:
    …
  variables:
    HOST_URL: ${HOST_URL}
    LOGIN: ${TOKEN}
  script:
    …
    policy: pull
```

Figure 5: Simplified .gitlab-ci.yml file used in one of the TALON's modules.

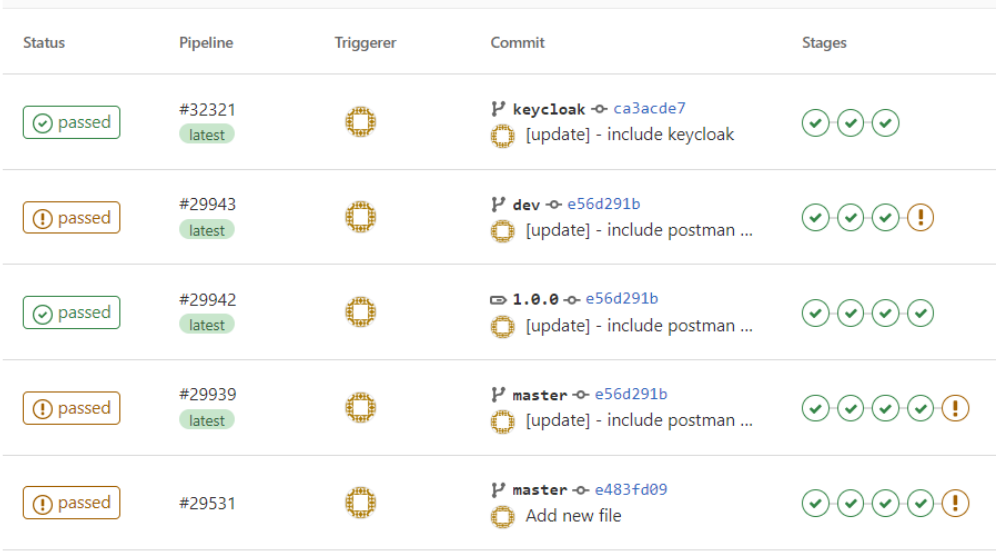


Figure 6: Pipeline Execution in GitLab.

3.2 Continuous Deployment/Delivery (CD) of TALON Platform

Continuous Deployment and Continuous Delivery are practices that automate the deployment of code to production (or staging) environments.

Continuous Delivery ensures that every change to the codebase is tested and ready for deployment to production, but the actual deployment may require a manual approval step.

Continuous Deployment goes a step further by automating the entire deployment process, including pushing changes to production automatically after passing the tests.

In TALON, we have adopted the CD practices to maximise team collaboration, prepare and deploy flexible workflows, and develop a solution which continuously incorporates improvements and updates. The technical solution of the (currently, i.e., as of M22) deployed components at the TALON cloud-edge infrastructure is depicted in Figure 7. As it is illustrated, users can access the TALON Platform through the TALON UI component (i.e., this is the Visualization Dashboard), which is protected by the Keycloak[7] framework. Every backend component or service has been configured to be authorized by Keycloak before exposing data or results onto the TALON UI. This inter-process communication takes place either directly (such as anonymization, or federated learning components), or through the Krakend[8] gateway (the rest of the components). Krakend acts as a gateway and provides an API that integrates all the other APIs being exposed to the TALON Platform. We highlight that Krakend has been configured on the side of Keycloak to provide a secure gateway API to all TALON components, such as the AI Orchestrator, Energy Intelligence, and more. We clarify that some components have been directly configured with Keycloak as they also deliver a user interface and tools easing the user interaction with them, while other components have been safeguarded behind Krakend because they only expose reporting APIs or results. For instance, the end users can view the energy metrics of the K8S cluster via the Energy Intelligence component, and the network metrics of the cluster via the NG-SDN and Distributed Intelligence component. These two components provide metrics in terms of Service Level Objectives (SLOs) to the AI Orchestrator component to optimize them, derive smart decisions and enforce policies either applied by itself (e.g., regarding energy, cost, resource allocation optimisations) or by the Self-Healing component (e.g., regarding network intelligence vertical scaling, QoS adaptation and AI-fuelled vertical scaling).

[7] https://www.keycloak.org/
[8] https://www.krakend.io/

Besides, the Anonymization component provides a tool to anonymize text, numerical data and dates, while the Federated Learning (FL) component presents the training results of the Federated Learning being performed at multiple edge nodes. Last but not least, the XAI component showcases different Trust Levels, i.e., from Trust Level 1 to Trust Level 4 (TrL1 – TrL4), as far as it concerns the quality, inconsistencies, veracity and balance of the datasets.

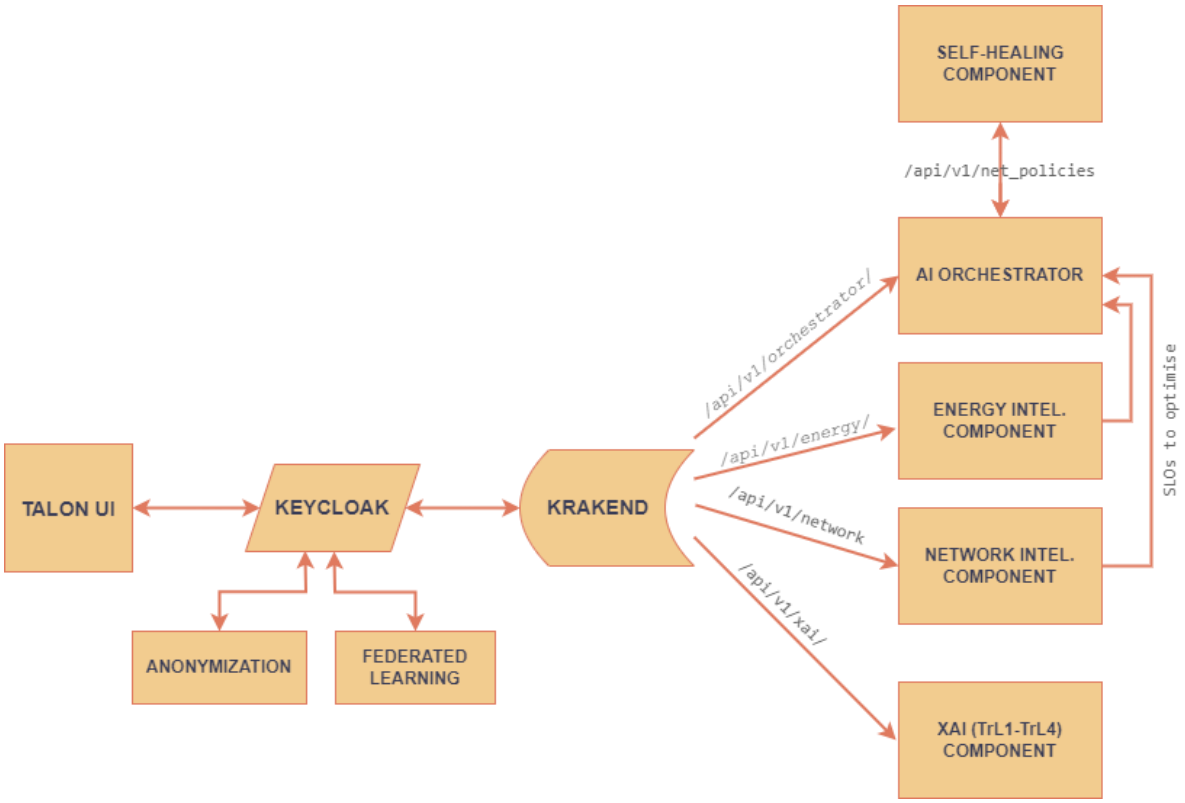


Figure 7. Integrated APIs and Components in TALON Platform.

In the meantime, we have set up and provided to all TALON technical partners access to the common code repository of GitLab[9]. The technical partners commit on a regular basis the source code of their components. More specifically, Figure 8 presents the common report and the projects of which it consists, while Figure 9 presents the activity logs for the various projects and their branches.

[9] https://about.gitlab.com/

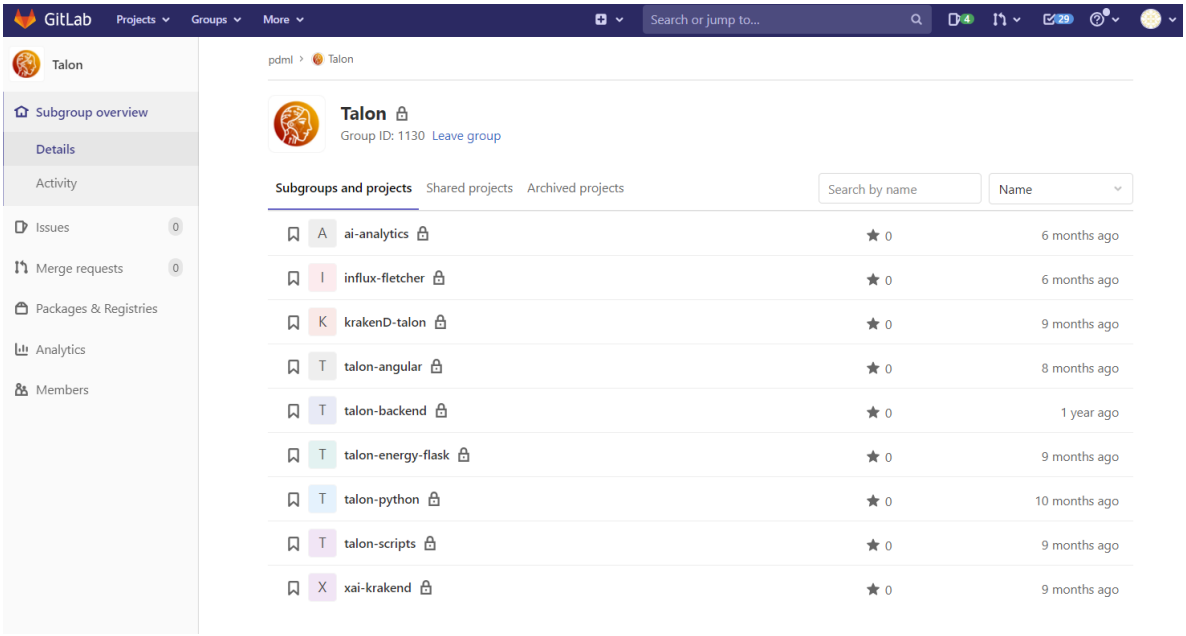


Figure 8: The common TALON GitLab repository.

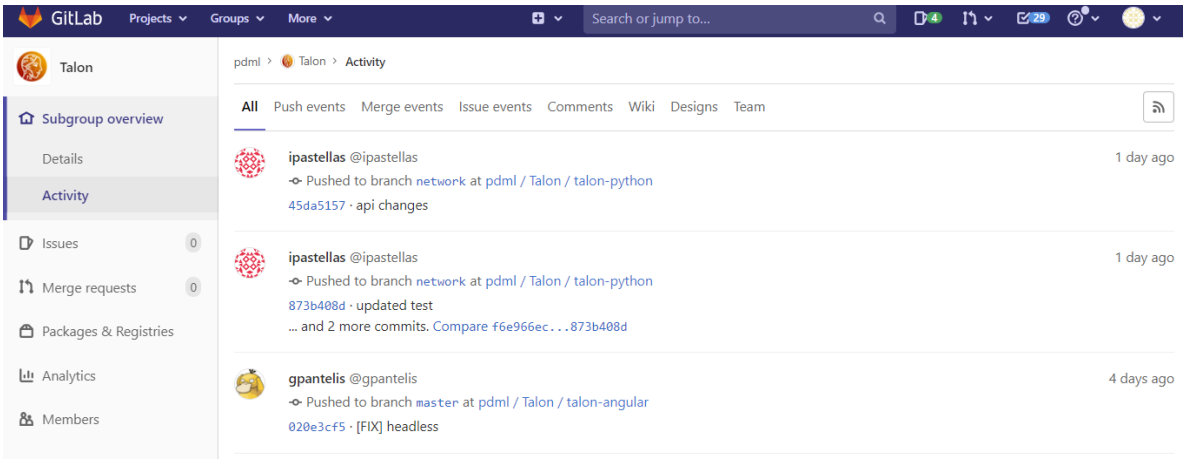


Figure 9: GitLab Activity Log.

4 Unit Testing and Early Platform Integration Testing

This section presents the first steps we performed for the technical verification of the TALON Platform, including unit testing and integration testing.

4.1 Unit Testing

The process of testing the smallest functional units of code is called unit testing. It ensures that each unit behaves as expected and it's a prerequisite for integration testing. For unit testing, several testing frameworks are used depending on the requirements of each component and the programming language or languages being used. The majority of the TALON's components are python projects, utilizing the standard unit testing framework in Python, unittest (unittest, n.d.). The main advantage of unittest is that it is part of the standard library, ensuring stability and compatibility with various Python versions. For the User Interface (UI) of TALON, we use the Angular web application framework, which has a built-in testing framework, Jasmine (Jasmine, n.d.), and a default test runner, Karma (Karma, n.d.), which executes tests written in Jasmine.

For each component that is built with python we have created a file named "test_unit.py" that includes all the necessary unit tests. An example of unit test is shown in Figure 10. This Python test function, test_network_corr_post, uses the patch decorator to mock the behaviour of an API client, ensuring the function can be tested in isolation without actual network interaction. The test setup involves a mock response simulating a successful API call, returning a JSON object indicating success with a 200-status code. The test payload includes specific dates and pod identifiers, relevant to the test case, which queries network and CPU usage measurements. Within the test, the application's post method is overridden to return the mock response, and the test function then invokes this mocked method with the payload, checking if the response is as expected. The test asserts the correct status code and validates that the response data includes the necessary keys and values, verifying the integrity and functionality of the network correlation feature in the application.

```python
@patch('api.app.test_client')  # Adjust the import path based on your app structure
def test_network_corr_post(self, mock_test_client):
    # Set up the mock response
    mock_response = Mock()
    mock_response.status_code = 200
    mock_response.data = json.dumps({
        'code': ResponseCode.SUCCESS.value,
        'returnobject': {}
    }).encode('utf-8')

    mock_client_instance = Mock()
    mock_client_instance.post.return_value = mock_response
    mock_test_client.return_value = mock_client_instance

    payload = {
        "date_start": "2024-07-09 00:00:00",
        "date_end": "2024-07-09 23:00:00",
        "pod_1": "pod1",
        "pod_2": "pod2",
        "measurements": ["container_network_receive_bytes_total", "container_cpu_usage_seconds_total"]
    }

    # Call the actual method to test
    with patch.object(self.app, 'post', return_value=mock_response):
        response = self.app.post(BASE_URL + '/corr/pod', data=json.dumps(payload), content_type='application/json')

        self.assertEqual(response.status_code, 200)
        data = json.loads(response.data)
        self.assertEqual(data['code'], ResponseCode.SUCCESS.value)
        self.assertIn('returnobject', data)
```

Figure 10: Example of unit testing.

All unit tests have successfully run in the CI/CD pipeline, as illustrated in Figure 11, showing that all tests passed without any issues.

```
$ flask unittests
 * Tip: There are .env or .flaskenv files present. Do "pip install python-dotenv" to use them.
.........
----------------------------------------------------------------------
Ran 10 tests in 0.038s
OK
Initialize client:  None
http://127.0.0.1:8080/
Saving cache for successful job
```

Figure 11: Unit tests execution in Flask.

The same approach is applied to other Python components. In addition to executing tests, we generate test coverage reports for each component, using the “coverage” library (coverage.py, n.d.). For example, we present test coverage reports for the energy and network intelligence components in Figure 12 and Figure 13, respectively.

Coverage report: 67%

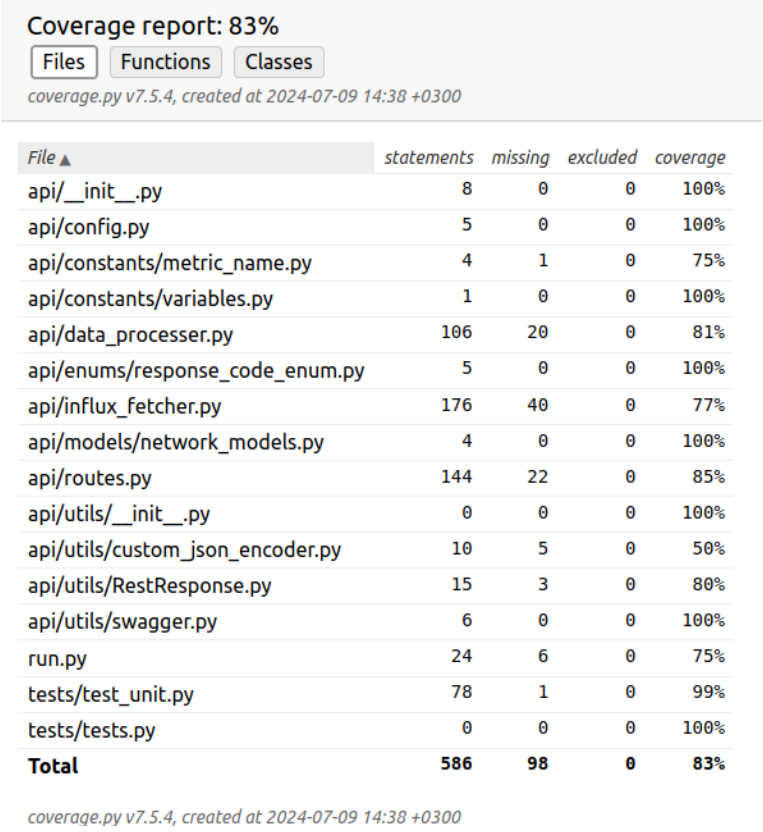
Files | Functions | Classes

coverage.py v7.5.4, created at 2024-07-10 15:48 +0300

File ▲	statements	missing	excluded	coverage
api/__init__.py	8	0	0	100%
api/config.py	5	0	0	100%
api/constants/variables.py	1	0	0	100%
api/data_processer.py	28	21	0	25%
api/enums/response_code_enum.py	7	0	0	100%
api/influx_fetcher.py	62	32	0	48%
api/routes.py	26	1	0	96%
api/utils/__init__.py	0	0	0	100%
api/utils/custom_json_encoder.py	10	5	0	50%
api/utils/RestResponse.py	15	3	0	80%
api/utils/swagger.py	6	0	0	100%
run.py	24	6	0	75%
tests/test_unit.py	26	5	0	81%
tests/tests.py	0	0	0	100%
Total	**218**	**73**	**0**	**67%**

coverage.py v7.5.4, created at 2024-07-10 15:48 +0300

Figure 12: AI Orchestrator (Energy Intelligence) Coverage Report.



Coverage report: 83%

Files | Functions | Classes

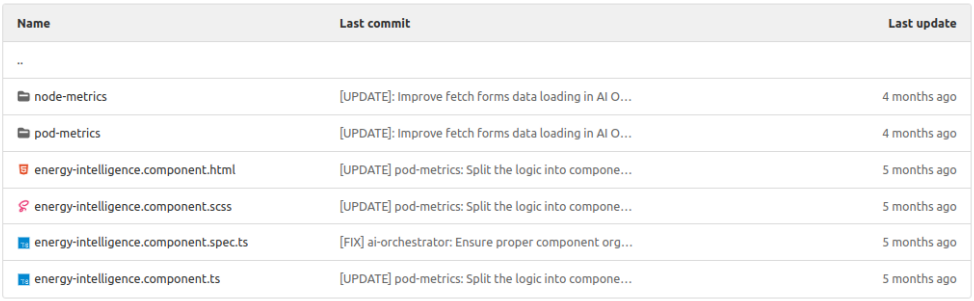
coverage.py v7.5.4, created at 2024-07-09 14:38 +0300

File ▲	statements	missing	excluded	coverage
api/__init__.py	8	0	0	100%
api/config.py	5	0	0	100%
api/constants/metric_name.py	4	1	0	75%
api/constants/variables.py	1	0	0	100%
api/data_processer.py	106	20	0	81%
api/enums/response_code_enum.py	5	0	0	100%
api/influx_fetcher.py	176	40	0	77%
api/models/network_models.py	4	0	0	100%
api/routes.py	144	22	0	85%
api/utils/__init__.py	0	0	0	100%
api/utils/custom_json_encoder.py	10	5	0	50%
api/utils/RestResponse.py	15	3	0	80%
api/utils/swagger.py	6	0	0	100%
run.py	24	6	0	75%
tests/test_unit.py	78	1	0	99%
tests/tests.py	0	0	0	100%
Total	**586**	**98**	**0**	**83%**

coverage.py v7.5.4, created at 2024-07-09 14:38 +0300

Figure 13: NG-SDN and Distributed Intelligence Component Coverage Report.

We use the Jasmine framework to write unit tests in Angular. The name of the test files must end with ‘spec.ts’ in order to be able to be collected and run from the Karma test runner. Because we use the Angular CLI to generate components and services, these test files are automatically generated as well. An example of an angular component is shown in Figure 14.



Name	Last commit	Last update
..		
node-metrics	[UPDATE]: Improve fetch forms data loading in AI O…	4 months ago
pod-metrics	[UPDATE]: Improve fetch forms data loading in AI O…	4 months ago
energy-intelligence.component.html	[UPDATE] pod-metrics: Split the logic into compone…	5 months ago
energy-intelligence.component.scss	[UPDATE] pod-metrics: Split the logic into compone…	5 months ago
energy-intelligence.component.spec.ts	[FIX] ai-orchestrator: Ensure proper component org…	5 months ago
energy-intelligence.component.ts	[UPDATE] pod-metrics: Split the logic into compone…	5 months ago

Figure 14: Example of angular component with test file (.spec.ts).

Another advantage of Angular is that it also creates a basic structure for test cases. This structure includes the describe function, which groups the test cases; the *it function*, used to write individual test cases; and the Expect function. The expect function takes a single argument - the value we want to test - and a matcher function that compares this value with the expected one. The Figure 15 shows the spec.ts file for the AI Orchestrator (and specifically the Energy Intelligence) component as it was created from the Angular CLI.

```typescript
import { ComponentFixture, TestBed } from '@angular/core/testing';

import { EnergyIntelligenceComponent } from './energy-intelligence.component';

describe('EnergyIntelligenceComponent', () => {
  let component: EnergyIntelligenceComponent;
  let fixture: ComponentFixture<EnergyIntelligenceComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [EnergyIntelligenceComponent]
    });
    fixture = TestBed.createComponent(EnergyIntelligenceComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Figure 15: Test file for AI Orchestrator (Energy Intelligence) component.

We can add test cases based on what we want to verify. For example, the test case shown Figure 16 verifies the functionality of the changeTimeZone method in the component. This test ensures that the method accurately converts and formats dates to GMT+0, maintaining the correct date and time values.

```typescript
it( expectation: 'should convert a date to GMT+0 timezone', assertion: () : void => {
  const date : Date = new Date( value: '2024-07-09T12:34:56Z'); // Example date
  const transformedDate : string | null = component.changeTimeZone(date);
  expect(transformedDate).toBe( expected: '2024-07-09 12:34:56');
});
```

Figure 16: Test case for changeTimeZone function.

As of now, we have 29-unit tests that are running automatically in the CI/CD pipeline, as depicted in Figure 17, alongside the coverage summary.

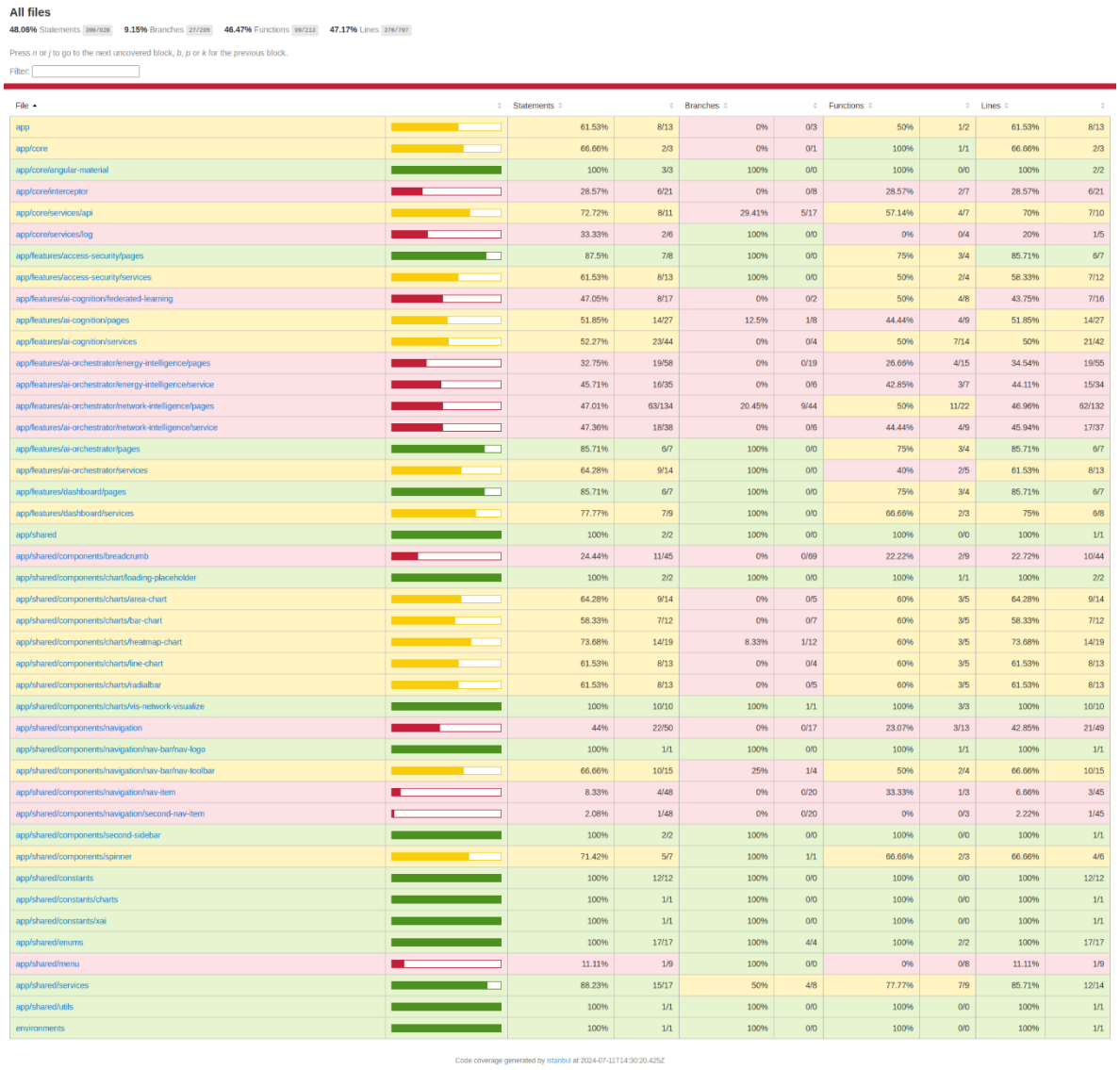
```
$ ng test --watch=false --code-coverage --browsers=ChromeHeadless
- Generating browser application bundles (phase: setup)...
✔ Browser application bundle generation complete.
11 07 2024 14:12:55.999:INFO [karma-server]: Karma v6.4.3 server started at http://localhost:9876/
11 07 2024 14:12:56.013:INFO [launcher]: Launching browsers ChromeHeadless with concurrency unlimited
11 07 2024 14:12:56.057:INFO [launcher]: Starting browser ChromeHeadless
11 07 2024 14:12:57.222:INFO [Chrome Headless 126.0.6478.126 (Linux x86_64)]: Connected on socket Wea2ViOPVdeOqYnqAAAB with id 20661520
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 0 of 29 SUCCESS (0 secs / 0 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 1 of 29 SUCCESS (0 secs / 0.346 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 2 of 29 SUCCESS (0 secs / 0.458 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 3 of 29 SUCCESS (0 secs / 0.585 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 4 of 29 SUCCESS (0 secs / 0.586 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 5 of 29 SUCCESS (0 secs / 0.648 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 6 of 29 SUCCESS (0 secs / 0.804 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 7 of 29 SUCCESS (0 secs / 0.832 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 8 of 29 SUCCESS (0 secs / 0.875 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 9 of 29 SUCCESS (0 secs / 0.917 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 10 of 29 SUCCESS (0 secs / 0.936 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 11 of 29 SUCCESS (0 secs / 0.978 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 12 of 29 SUCCESS (0 secs / 1.048 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 13 of 29 SUCCESS (0 secs / 1.377 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 14 of 29 SUCCESS (0 secs / 1.628 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 15 of 29 SUCCESS (0 secs / 1.711 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 16 of 29 SUCCESS (0 secs / 2.368 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 17 of 29 SUCCESS (0 secs / 2.379 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 18 of 29 SUCCESS (0 secs / 2.404 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 19 of 29 SUCCESS (0 secs / 2.441 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 20 of 29 SUCCESS (0 secs / 2.476 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 21 of 29 SUCCESS (0 secs / 2.487 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 22 of 29 SUCCESS (0 secs / 2.492 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 23 of 29 SUCCESS (0 secs / 2.498 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 24 of 29 SUCCESS (0 secs / 2.502 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 25 of 29 SUCCESS (0 secs / 2.685 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 26 of 29 SUCCESS (0 secs / 2.73 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 27 of 29 SUCCESS (0 secs / 2.74 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 28 of 29 SUCCESS (0 secs / 2.816 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 29 of 29 SUCCESS (0 secs / 2.822 secs)
Chrome Headless 126.0.6478.126 (Linux x86_64): Executed 29 of 29 SUCCESS (3.18 secs / 2.822 secs)
TOTAL: 29 SUCCESS
✔ Browser application bundle generation complete.
============================== Coverage summary ==============================
Statements   : 48.06% ( 398/828 )
Branches     : 9.15% ( 27/295 )
Functions    : 46.47% ( 99/213 )
Lines        : 47.17% ( 376/797 )
==============================================================================
```

Figure 17: Output of test execution in CI/CD pipeline.

Except for that output, we also generate an analytical Coverage Report, Figure 18, which includes the following coverages:

- Line: measures the percentage of lines of codes that have been executed by tests.
- Statement: measures the percentage of executable statements that have been executed by tests
- Branch: measures the percentage of the branches of the control structures (If-else) that have been executed by tests.
- Function: measures the percentage of the functions that have been called by the tests

As of M22 the test coverage is approximately around 75%), By adopting automatic CI/CD pipelines and incorporating enhancements into our codebase we plan to increase this threshold to a greater coverage.



All files

48.06% Statements 398/828 9.15% Branches 27/295 46.47% Functions 99/213 47.17% Lines 376/797

Press n or j to go to the next uncovered block, b, p or k for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
app	61.53%	8/13	0%	0/3	50%	1/2	61.53%	8/13
app/core	66.66%	2/3	0%	0/1	100%	1/1	66.66%	2/3
app/core/angular-material	100%	3/3	100%	0/0	100%	0/0	100%	2/2
app/core/interceptor	28.57%	6/21	0%	0/8	28.57%	2/7	28.57%	6/21
app/core/services/api	72.72%	8/11	29.41%	5/17	57.14%	4/7	70%	7/10
app/core/services/log	33.33%	2/6	100%	0/0	0%	0/4	20%	1/5
app/features/access-security/pages	87.5%	7/8	100%	0/0	75%	3/4	85.71%	6/7
app/features/access-security/services	61.53%	8/13	100%	0/0	50%	2/4	58.33%	7/12
app/features/ai-cognition/federated-learning	47.05%	8/17	0%	0/2	50%	4/8	43.75%	7/16
app/features/ai-cognition/pages	51.85%	14/27	12.5%	1/8	44.44%	4/9	51.85%	14/27
app/features/ai-cognition/services	52.27%	23/44	0%	0/4	50%	7/14	50%	21/42
app/features/ai-orchestrator/energy-intelligence/pages	32.75%	19/58	0%	0/19	26.66%	4/15	34.54%	19/55
app/features/ai-orchestrator/energy-intelligence/service	45.71%	16/35	0%	0/6	42.85%	3/7	44.11%	15/34
app/features/ai-orchestrator/network-intelligence/pages	47.01%	63/134	20.45%	9/44	50%	11/22	46.96%	62/132
app/features/ai-orchestrator/network-intelligence/service	47.36%	18/38	0%	0/6	44.44%	4/9	45.94%	17/37
app/features/ai-orchestrator/pages	85.71%	6/7	100%	0/0	75%	3/4	85.71%	6/7
app/features/ai-orchestrator/services	64.28%	9/14	100%	0/0	40%	2/5	61.53%	8/13
app/features/dashboard/pages	85.71%	6/7	100%	0/0	75%	3/4	85.71%	6/7
app/features/dashboard/services	77.77%	7/9	100%	0/0	66.66%	2/3	75%	6/8
app/shared	100%	2/2	100%	0/0	100%	0/0	100%	1/1
app/shared/components/breadcrumb	24.44%	11/45	0%	0/69	22.22%	2/9	22.72%	10/44
app/shared/components/chart/loading-placeholder	100%	2/2	100%	0/0	100%	1/1	100%	2/2
app/shared/components/charts/area-chart	64.28%	9/14	0%	0/5	60%	3/5	64.28%	9/14
app/shared/components/charts/bar-chart	58.33%	7/12	0%	0/7	60%	3/5	58.33%	7/12
app/shared/components/charts/heatmap-chart	73.68%	14/19	8.33%	1/12	60%	3/5	73.68%	14/19
app/shared/components/charts/line-chart	61.53%	8/13	0%	0/4	60%	3/5	61.53%	8/13
app/shared/components/charts/radialbar	61.53%	8/13	0%	0/5	60%	3/5	61.53%	8/13
app/shared/components/charts/vis-network-visualize	100%	10/10	100%	1/1	100%	3/3	100%	10/10
app/shared/components/navigation	44%	22/50	0%	0/17	23.07%	3/13	42.85%	21/49
app/shared/components/navigation/nav-bar/nav-logo	100%	1/1	100%	0/0	100%	1/1	100%	1/1
app/shared/components/navigation/nav-bar/nav-toolbar	66.66%	10/15	25%	1/4	50%	2/4	66.66%	10/15
app/shared/components/navigation/nav-item	8.33%	4/48	0%	0/20	33.33%	1/3	6.66%	3/45
app/shared/components/navigation/second-nav-item	2.08%	1/48	0%	0/20	0%	0/3	2.22%	1/45
app/shared/components/second-sidebar	100%	2/2	100%	0/0	100%	0/0	100%	1/1
app/shared/components/spinner	71.42%	5/7	100%	1/1	66.66%	2/3	66.66%	4/6
app/shared/constants	100%	12/12	100%	0/0	100%	0/0	100%	12/12
app/shared/constants/charts	100%	1/1	100%	0/0	100%	0/0	100%	1/1
app/shared/constants/xai	100%	1/1	100%	0/0	100%	0/0	100%	1/1
app/shared/enums	100%	17/17	100%	4/4	100%	2/2	100%	17/17
app/shared/menu	11.11%	1/9	100%	0/0	0%	0/8	11.11%	1/9
app/shared/services	88.23%	15/17	50%	4/8	77.77%	7/9	85.71%	12/14
app/shared/utils	100%	1/1	100%	0/0	100%	0/0	100%	1/1
environments	100%	1/1	100%	0/0	100%	0/0	100%	1/1

Code coverage generated by Istanbul at 2024-07-11T14:30:20.425Z

Figure 18: TALON UI Analytical Coverage Report.

4.2 Early Platform Integration Testing

The main objective of the early platform integration testing is to verify the interactions and interfaces between the TALON ‘s components as a group. Until now the following components have been integrated into the Visualisation Dashboard component:

- Authentication and Authorisation (Component ID: TALON_AS_AA)
- NG-SDN and Distributed Intelligence (Component ID: TALON_AS_SDNDI)
- Self-healing and Self-correcting (Component ID: TALON_AIC_SHSC)
- XAI, Monitoring and Reporting (TrL1-TrL4) (Component ID: TALON_AIC_XAIM)
- Data Anonymisation (Component ID: TALON_AS_DA)

The plan for early platform integration testing involves defining workflows to test the integration of each component with the Visualization Dashboard. Since each component is made up of different tools, these workflows will also test the integration of the various tools within each component, ensuring comprehensive integration testing.

The following three indicative workflows have been selected to present the platform integration points across the core technologies and frameworks, and programming APIs of TALON.

4.2.1 Workflow 1 – Validation of Authentication and Authorisation

The Authentication and Authorisation component is being implemented using the Keycloak framework. To test the integration with the Visualization Dashboard, we define the following workflow:

1. The user navigates to the public domain of the Visualization Dashboard (https://talon-ui.euprojects.net).
2. The system should automatically redirect the user to the Keycloak login page.
3. After a successful login, the user should be redirected back to the Visualization Dashboard.
4. The user can logout.

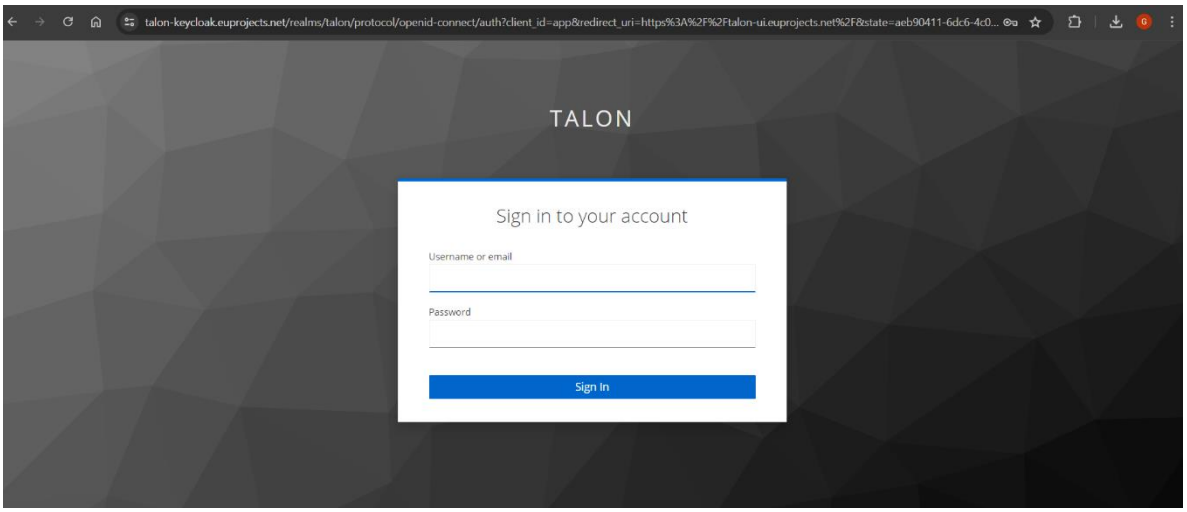


Figure 19: TALON Login Page.

As we can see in Figure 19, the system successfully redirects the user to the login page. After signing in with the correct credentials, it redirects the user to the TALON Platform (Figure 20).

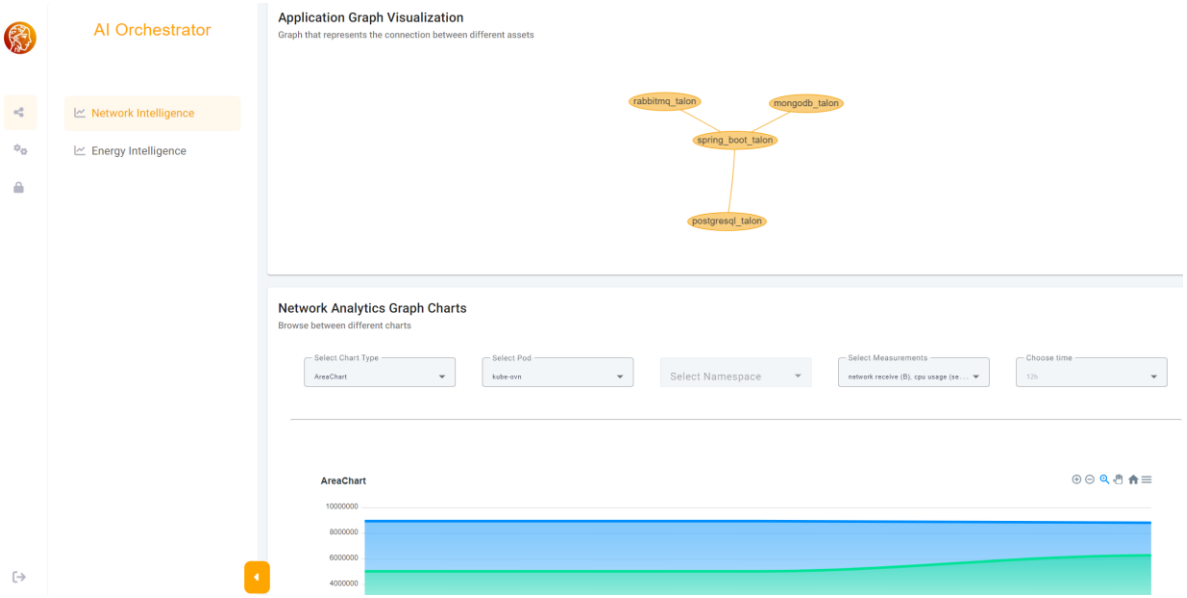


Figure 20: TALON Platform.

Also, pressing the logout button logs the user out and redirects them to the login page, verifying the successful integration of the Authentication and Authorization component.

4.2.2 Workflow 2 – Validation of NG-SDN and Distributed Intelligence component

The NG-SDN and Distributed Intelligence component provides different measurements from the Kubernetes cluster for specific pods in a visualized way. The workflow is as follows:

1. The authenticated user requests specific data from the backend gateway (Krakend).
2. Krakend passes the request to the NG-SDN and Distributed Intelligence component,
3. The NG-SDN and Distributed Intelligence component searches the InfluxDB for the requested data.
4. The reverse flow takes place to return the result.

To test the integration of these components, we will use the Postman *(Postman, n.d.)* tool to make a request to Krakend, as shown in Figure 21. We can see that the correct data has been returned, indicating successful integration of the involved components.

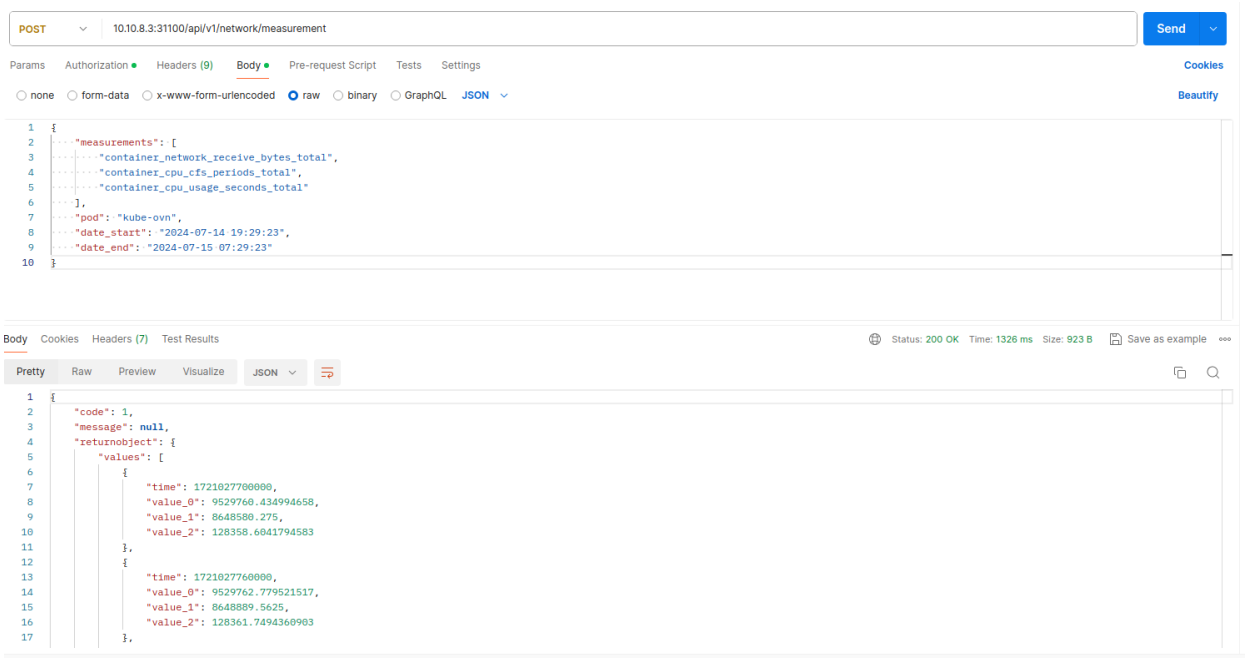


Figure 21: Request to NG-SDN and Distributed Intelligence component.

4.2.3 Workflow 3 – Validation of Data Anonymisation component

The Data Anonymisation component has being integrated into the Visualisation Dashboard, and the workflow is a follow:

1. The authenticated user navigates to the anonymisation tab.
2. Fills the form and sends to the backend gateway (Krakend) the request.
3. The Krakend passes the request to the Anonymisation component
4. The Anonymisation component performs the necessary actions to anonymise the text.
5. The reverse flow takes place to return the result.

To test the integration between the involved components we fill the form with data and perform the request in the Visualisation Dashboard (Figure 22):

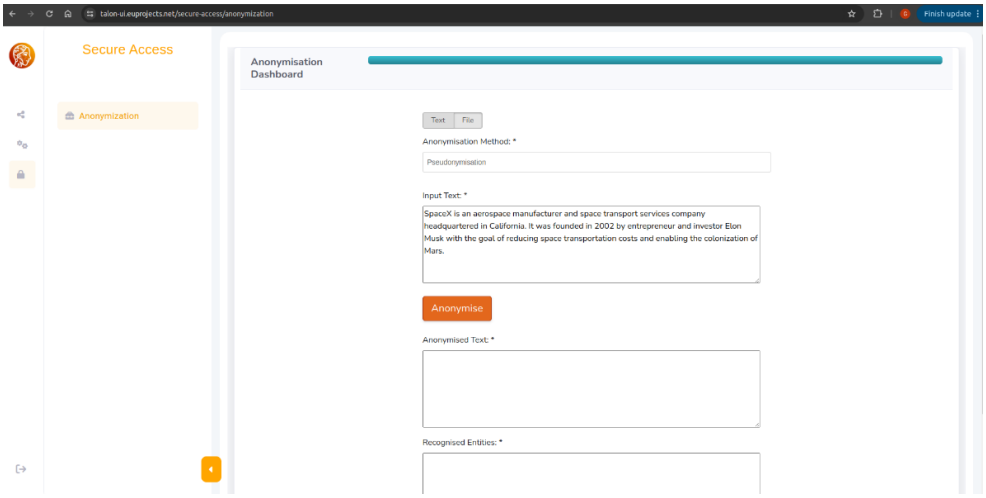


Figure 22: Anonymisation Component before the request.

We validate the returned result, shown in Figure 23, indicating the successful integration of these components.

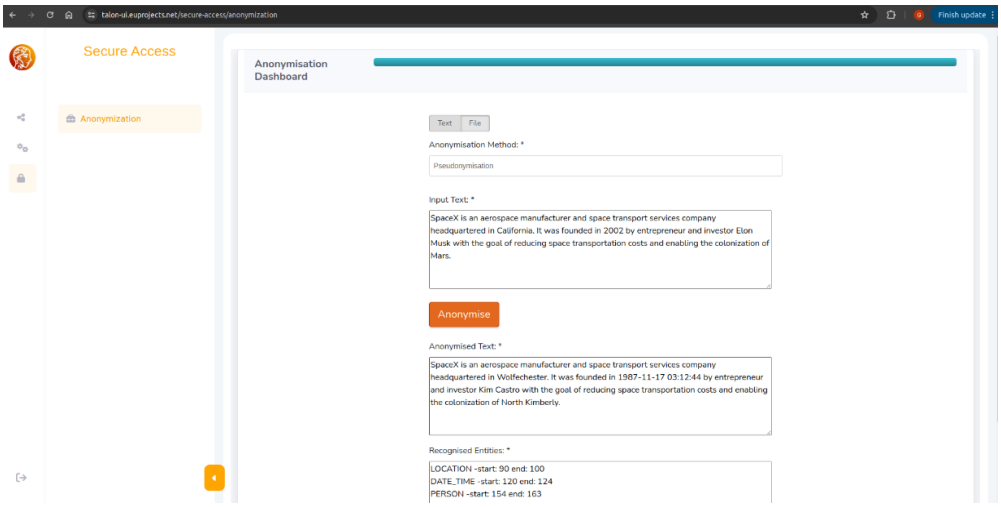


Figure 23: Anonymisation Component after the request.

Similar workflows have been defined and tested, to validate the integration of the currently deployed and integrated components (remark: as of M22).

4.3 Integration Traceability Matrix

As of M22, we have identified the all the TALON components and we have also designed an integration traceability matrix, as presented below, in order to monitor how all components interact among them. In the upcoming months the content of this matrix will be populated to reflect the different interactions among the different TALON components, and it will be continuously monitored until the end of the project.

Our plan until M24 is to populate the integration traceability matrix with all the interactions that are taking place directly or indirectly among the different TALON components. The latter will further help us to perform unit and integration testing and incorporate improvements in the TALON components, heading towards their validation by the Use Case / Pilot partners.

	Authentication and Authorisation	Data Anonymisation	DLTs for Securing AI/ML models weights	Anomaly Detection	Service Modelling	NG-SDN and Distributed Intelligence	Orchestration	AI Swarm Orchestration	Resource Allocation and Deployment	Definition, Customisation and Monitoring of Metrics	Smart Policy Manager	Data Monitoring, Collection and Aggregation	AI Model Training and SLOs Optimisation	Self-healing and Self-correcting	Hybrid and Optimised Learning	AI Capabilities and Transfer Learning	Data Operations	Digital Twins	XAI, Monitoring and Reporting	Data Lifecycle Management	Visualisation Dashboard
Authentication and Authorisation																					
Data Anonymisation																					
DLTs for Securing AI/ML models weights (DLTs for Security and Privacy)																					
Anomaly Detection																					
Service Modelling (aka “Configurations) and Enactment																					
NG-SDN and Distributed Intelligence																					
Orchestration																					
AI Swarm Orchestration																					
Resource Allocation and Deployment																					
Definition, Customisation and Monitoring of Metrics																					
Smart Policy Manager																					
Data Monitoring, Collection and Aggregation																					
AI Model Training and SLOs Optimisation																					
Self-healing and Self-correcting																					
Hybrid and Optimised Learning																					
AI Capabilities and Transfer Learning																					
Data Operations																					
Digital Twins																					
XAI, Monitoring and Reporting																					
Data Lifecycle Management																					
Visualisation Dashboard																					

Figure 24: Integration Traceability Matrix.

5 Conclusion and Future Outlook

Deliverable D5.2 – Initial TALON Platform Setup, Operation, Continuous Integration & Maintenance Report documents the work carried out in the context of **Task 5.2: TALON Platform Setup, Operation, Continuous Integration & Maintenance** of TALON Project. The main objective of this deliverable was to provide an overview of how the TALON Platform setup and operation process was conducted.

The document outlines the following activities:

- A comprehensive review on the TALON platform setup and operation, emphasizing the essential infrastructure deployment for the successful realization of the integration process, and describing key Kubernetes concepts;
- Information about the continuous integration and continuous deployment approach with the automating tools (i.e. GitLab) employed by the TALON Project to achieve an iterative process in alignment with the implementation of each use case;
- A detailed testing planning for TALON's technological components, including the unit testing for verifying individual code units and early platform integration testing for evaluating component interactions;

D5.2 is the second out of four deliverables related to the "Work Package 5: Integration, Validation & Demonstration" of TALON, marks the attainment of Milestone 5 (MS5), the first, early release of the TALON Integrated platform in M22. It will be followed by "D5.3 Pilot Specific TALON Platform Setup, Operation, Continuous Integration & Maintenance Report" which will be dedicated to TALON Use Cases.

Funded by
the European Union

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101070181