TEADAL

D4.3 STRETCHED DATA LAKES

FINAL REPORT AND EVALUATION

Revision: 1.0

Work package	WP 4
Task	Task 4.1, 4.2, 4.3
Due date	31/05/2025
Submission date	31/05/2025
Deliverable lead	IBM
Version	V0.9
Authors	Katherine Barabash (IBM), Bruno Feitas (Ubiwhere), João Viegas (Ubiwhere), Temesgen Magule Olango (ALMAVIVA), Alessio Carelini (CEFRIEL), Gabriele Cerfoglio (Martel), Sergio Sestili (ALMAVIVA), Samantha Hine (ALMAVIVA)
Reviewers	Pierluigi Plebani (POLIMI) Boris Sedlak (TUW)
Abstract	Final technical summary of the control plane, data management, and trustworthy data flows addressing the defined KPIs
Keywords	Control plane, Data Lake, multi-cloud, multi-cluster, data pipelines

Document Revision History

Grant Agreement No.: 101070186
Call: HORIZON-CL4-2021-DATA-01
Topic: HORIZON-CL4-2021-DATA-01-01
Type of action: HORIZON-RIA

Version	Date	Description of change	List of contributor(s)
draft	01/03/2025	First draft	Katherine Barabash (IBM)
v0.1	18/03/2025	Initial ToC and contents	WP4 partners
V0.2	01/04/2025	First round of contribution to the ToC	Katherine Barabash (IBM), Bruno Feitas (Ubiwhere), João Viegas (Ubiwhere), Temesgen Magule Olango (ALMAVIVA)
V0.3	29/04/2025	Collected comments, added ASG section contents	Katherine Barabash (IBM)
V0.4	04/05/2025	Finalized the Executive Summary and Introduction	Katherine Barabash (IBM)
V0.5	06/05/2025	Finalized Sections 2 and 3	Katherine Barabash (IBM)
V0.6	12/05/2025	Catalogue Flows in Section 2 RBAC aspects in Section 2 Section 5	Alessio Carelini (CEFRIEL), Gabriele Cerfoglio (Martel), Katherine Barabash (IBM)
V0.7	14/05/2025	Section 3 Small updates to Section 2 Editing and formatting	Temesgen Magule Olango (ALMAVIVA), Sergio Sectile (ALMAVIVA), Gabriele Cerfoglio (Martel), Alessio Carelini (CEFRIEL), Katherine Barabash (IBM)
V0.8	15/05/2025	Internal review version	Katherine Barabash (IBM)
V0.9	25/05/2025	Internal review 1	Pierluigi Plebani (POLIMI), Katherine Barabash (IBM)
1.0	28/05/2025	Internal review 2	Boris Sedlak (TUW), Katherine Barabash (IBM)

DISCLAIMER

Funded by the European Union

Project funded by
Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra
Swiss Confederation
Federal Department of Economic Affairs, Education and Research EAER
State Secretariat for Education, Research and Innovation SERI

Funded by the European Union (TEADAL, 101070186). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them. This work has received funding from the Swiss State Secretariat for Education, Research and Innovation (SERI).

COPYRIGHT NOTICE

Nature of the deliverable:	R	
Dissemination Level		
PU	*Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page)*	✓
SEN	*Sensitive, limited under the conditions of the Grant Agreement*	
Classified R-UE/ EU-R	*EU RESTRICTED under the Commission Decision No2015/ 444*	
Classified C-UE/ EU-C	*EU CONFIDENTIAL under the Commission Decision No2015/ 444*	
Classified S-UE/ EU-S	*EU SECRET under the Commission Decision No2015/ 444*	

Funded by the European Union

EXECUTIVE SUMMARY

The TEADAL project empowers organizations to securely collaborate on data-driven tasks across distributed infrastructures. It introduces a novel architecture for federated data sharing, allowing participating organizations to contribute, transform, and consume datasets, efficiently and under well-defined policies. Key to this architecture is the concept of a Federated Data Product (FDP), which is a shareable REST endpoint for exposing organizational data, and of a Shared Federated Data Product (sFDP), created to expose the FDPs across organizations in a controlled way. TEADAL implements this architecture using Kubernetes-based deployments and GitOps practices to maintain consistency and security across the federation.

This document reports on the outcomes of Work Package 4 (WP4) within the TEADAL project. WP4 focuses specifically on realizing and managing cross-organizational data flows, originally conceptualized as "stretched data lakes" and evolved over time into a more modular and runtime-oriented architecture. The contributions described here include design, prototyping, and tooling that together allow shared federated data pipelines to be declaratively specified, consistently deployed, and programmatically observed. These results directly support the broader architectural goals of TEADAL and provide practical mechanisms to implement and govern data flows as part of shared infrastructures.

These are the main functional modules provided by WP4:

1. **Monitoring subsystem**, realized as AI-driven Performance Monitoring (AI-DPM) – a set of components responsible for collecting runtime operational data from nodes in TEADAL federation, including data on server performance and energy usage, analysing this data to create actionable insights such as predictions and alerts. Insights made available by these components are used by the control plane for selecting the most suitable deployment targets for the TEADAL data products.
2. **Automation subsystem**, realized as Automatic SFDP generation (ASG) – a set of components that assist developers in creating SFDPs to share data exposed by the existing FDPs according to agreements achieved between the data provider and the data consumer. ASG relies on generative AI capabilities for selecting data transformations that need to be applied to the source datasets. ASG is working as part of the SFDP creation flow facilitated through the TEADAL Catalogue, as briefly presented here for completeness. ASG also includes a runtime library for unified runtime execution and control of the generated SFDPs.
3. **Optimization subsystem**, responsible for selecting the deployment targets for SFDPs at runtime among the TEADAL Nodes in the federation, based on the operational data and insights provided by monitoring subsystem as well as on the labels attached to the infrastructure to signify the capabilities of individual infrastructure components.
4. **Deployment subsystem**, or the multi-node control plane responsible for deploying the data products and initiating their runtime monitoring.

In addition to presenting these major components, the report describes their relationship with the components developed in other technical work packages (WP3 and WP5), their roles for use cases presented by the TEADAL pilots, and their planned contribution to the KPI validation to be performed in the final stage of the project, M34-M36, as part of WP6.

Funded by the European Union

TABLE OF CONTENTS

Funded by the European Union

LIST OF FIGURES

Funded by the European Union

LIST OF TABLES

Funded by the European Union

ABBREVIATIONS

AI	Artificial Intelligence
AI-DPM	AI-driven Performance Monitoring
AIOps	Artificial Intelligence for Information Technology (IT) Operations
ASG	Automatic sFDP Generation
API	Application Programming Interface
BPMN	Business Process Model and Notation
CD	Continuous Delivery
CI	Continuous Integration
CRD	Custom Resource Definition
DAG	Direct Acyclic Graphs
FDP	Federated Data Product
GRU	Gated Recurrent Units
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IdM	Identity Management
IT	Information Technology
I/O	Input / Output
JWT	JSON Web Token
K8s	Kubernetes
KPI	Key Performance Indicators
LLM	Large Language Model
LSTM	Long Short-Term Memory Neural Network
MCC-C	Multi-cloud Computer Compiler
MCP	Model Catalog Platforms / Model Context Protocol
ML	Machine Learning
OIDC	OpenID Connect
OPA	Open Policy Agent
PoC	Proof-of-concept
PromQL	Prometheus Query Language
RBAC	Role-Based Access Control
REPL	Read-Eval-Print Loop
REST	REpresentational State Transfer
SDLC	Stretched Data Lake Compiler
SFDP	Shared Federated Data Product
TEE	Trusted Execution Environment

Funded by the European Union

TTMs	Tiny Time Mixers
YAML	Yet Another Markup Language

Funded by the European Union

1. INTRODUCTION

In an increasingly data-driven world, the ability of organizations to securely and efficiently collaborate across infrastructure and institutional boundaries is becoming a strategic imperative. From scientific research to smart manufacturing and public services, data is both the fuel and the product of digital operations. However, the realities of distributed infrastructures, heterogeneous technologies, and complex governance models often make cross-organizational data integration costly and brittle. The TEADAL project addresses this challenge by introducing a principled architecture in WP2 (see deliverables D2.1 [1], D2.2 [1][2], D2.3 [3], and the upcoming D2.4 [3][4]), and platform design in WP6 (see deliverables D6.1 [5] and D6.2 [6]) for the concept of **federated data sharing** that prioritizes trust, policy compliance, and operational efficiency.

At the heart of this vision is the concept of the Federated Data Product (FDP): a shareable, API-exposed data interface encapsulating a dataset maintained by an individual organization. FDPs are designed to respect local autonomy, security, and domain-specific control. Yet, to fully enable cross-organizational data reuse, TEADAL introduces a complementary concept, the Shared Federated Data Product (SFDP). SFDPs act as intermediaries: policy-enforced, transformed, and traceable views of underlying FDPs, accessible across organizational boundaries.

The broader TEADAL platform operationalizes this architecture using Kubernetes-based deployments, GitOps-style control, and declarative specifications. In this way, data flows become programmable, inspectable, and composable. Moreover, AI-powered monitoring and LLM-assisted development tools further reduce the barriers to adoption and help manage the complexity inherent in federated data landscapes.

This document reports on the final design and implementation of the technical components delivered as part of Work Package 4 (WP4). WP4 focuses specifically on the realization and governance of cross-organizational data flows, originally described as "stretched data lakes," and now more concretely implemented as runtime-deployable SFDPs. These contributions are designed to integrate seamlessly with the other TEADAL services (notably AI-DPM and Control Plane) and to serve the needs of pilot users across various domains. The following subsections provide a more detailed orientation to the objectives and structure of this report.

1.1 PURPOSE AND SCOPE

This report consolidates the software, design decisions, and implementation outcomes of WP4 within the TEADAL project. It covers three major innovations:

- The design and delivery of an AI-driven Performance Monitoring subsystem (AI-DPM), which collects and analyses operational data (e.g., performance, energy consumption) to guide infrastructure-aware deployment decisions.
- The development of an Automatic SFDP Generation (ASG) subsystem, which assists developers in specifying, generating, and executing SFDPs with the help of LLM-based tools and a shared runtime library.
- The updates and finalization of the TEADAL control plane's optimization and deployment subsystems, that use declarative specifications and monitoring feedback to select appropriate deployment targets and manage the lifecycle of SFDPs across the federation.

In addition to reporting on each component, the document explores their integration into the TEADAL architecture, their application in real-world pilot deployments, and their contribution to achieving the TEADAL project's requirements.

Funded by the European Union

1.2 CONTEXT: TOWARDS DECLARATIVE CONTROL OF THE AI-POWERED DATA PLATFORM

Our approach to Stretched Data Lakes and Control Plane reflects on and supports the important architectural decisions made as part of TEADAL:

1. **Declarative control**: Right from the beginning, TEADAL has selected Kubernetes as its underlying infrastructure controller. In WP4, we support this choice by leveraging the declarative Kubernetes control plane and its ability to define Custom Resource Definitions (CRDs) for implementing domain-specific capabilities. In addition, we reinforce this declarative approach through the ASG subsystem, which enables data users to specify "what" data products should do rather than "how" they should be implemented. As a result, SFDP creation becomes a largely hands-off, declarative process. The resulting SFDPs are standardized microservices that fit seamlessly into the Kubernetes-based control plane, ensuring modularity, automation, and reuse as part of the TEADAL Platform.

2. **Federated trust and control**: According to the TEADAL Architecture, each organization maintains sovereignty over its data and its FDPs, while the TEADAL platform ensures cross-cutting enforcement of data-sharing agreements. In WP4, we support this principle by creating SFDPs as standardized microservices that can leverage the same policy management and enforcement mechanisms developed for FDPs. Furthermore, our implementation of data transformation pipelines within SFDPs enables runtime policy controls, for example, by injecting concrete implementations of generic transform functions based on deployment context.

In addition to the above, our approach has evolved to adopt the emerging trend of applying AI for managing distributed infrastructures and supporting data engineering workflows.

AI and LLM integration: TEADAL embraces the growing role of AI in managing complexity across distributed data infrastructures. On the operational side, AI-driven observability and optimization, commonly known as AIOps, are used to monitor and control deployments based on real-time infrastructure conditions. In TEADAL, this is embodied in the AI-DPM subsystem, which applies AI models to performance and energy data to guide intelligent deployment decisions. On the development side, the rise of Large Language Models (LLMs) is transforming the way users interact with complex technical stacks. TEADAL's ASG subsystem explores how generative AI can be leveraged to automate key aspects of Shared Federated Data Product creation, including endpoint specification, transformation chaining, and runtime deployment configuration. This allows developers to work at a higher level of abstraction, using natural language prompts or assisted templates rather than manually writing code and YAML specifications, reducing the skill threshold for developers and accelerating the creation of SFDPs that conform to both technical standards and policy constraints. Together, these AI-enhanced capabilities position TEADAL at the forefront of intelligent, federated data platform design.

Emerging alignment with Model Context Protocol and Model Catalog Platforms (MCPs): An emerging direction, particularly relevant for the ASG subsystem, is the potential alignment with Model Context Protocol[1] and the Model Catalog Platforms powered by this protocol. Originally designed for sharing and managing AI/ML models, Model Catalog Platforms are now evolving to support broader ecosystems of reusable, composable assets, including data transformation methods, pipelines, and APIs. By adopting the open Model Context Protocol, created to enable seamless integration between LLM applications and external data sources

[1] Model Context Protocol · GitHub

Funded by the European Union

and tools, TEADAL could expose its transformation library as a searchable, machine-readable catalog. This would not only improve human usability and model-assisted discovery but also position TEADAL to integrate with other tools that rely on MCP conventions. As generative AI systems increasingly rely on structured interfaces to invoke external capabilities (e.g., tools, functions, plugins), MCP-style publication of TEADAL transformations could help unlock AI-powered orchestration and semi-automated data engineering. While still exploratory, this direction promises to reinforce TEADAL's commitment to openness, modularity, and future-ready design.

1.3 DOCUMENT STRUCTURE

The remainder of this document is organized as follows:

- Section 2, Stretched Data Lakes, provides an overview of how the initial "stretched data lakes" concept has evolved into the modular, modern, declarative, and runtime-oriented architecture presented in this report. In addition, this section presents the platform integration aspects of the WP4, including the Catalogue flows involved in SFDP creation and the RBAC integration.
- Section 3, The Monitoring Subsystem (AI-DPM), describes the monitoring subsystem of TEADAL Control Plane, an AI-driven performance monitoring (AI-DPM). The section presents the AI-DPM components, the data they collect, the algorithms they use, and the insights they provide for other Control Plane subsystems, such as runtime observability and optimization.
- Section 4, The Automation Subsystem (ASG), presents the automation aspect of the TEADAL Control Plane, an ASG subsystem. The section discusses the benefits of applying generative AI for automating data access and processing and describes the ASG subsystem including its design rationale, tooling, software architecture, and deployment model.
- Section 5, The Optimization and the Deployment Subsystem, iterates on the control plane aspects that were already presented in prior deliverables of this work package while refining them to be in line with the updated WP4 architecture and technology stack of the TEADAL Node in its finalized form.
- Section 6, Summary, concludes the document with a summary of the work performed, its integration with other work packages, and contributions to TEADAL's goals and validation KPIs. In addition, we outline the roadmap to impact, and present possible directions for follow up research.

Funded by the European Union

2. STRETCHED DATA LAKES

This section presents the architecture developed in WP4, explaining both how it supports key TEADAL project requirements ([2][3][4]) and how it integrates with the TEADAL platform ([5][6]). WP4's contributions are centered on automated creation of SFDPs according to data sharing contracts negotiated between the FDP Consumers and the FDP Providers, as well as on the deployment and the runtime management of the generated SFDPs.

2.1 REVISITING THE "STRETCHED DATA LAKES" CONCEPT

The term "stretched data lakes" was originally introduced during the early stages of project planning, with the intention to encapsulate challenges related to creating, deploying, and managing data flows and data transformation and processing pipelines across distributed and heterogeneous infrastructure environments. It was planned that "stretched data lakes" will establish the data lakes functionality for organizations participating in TEADAL Federations, while the challenges related to data sharing across organizations will be addressed by other work streams concerned with the federation aspects, such as trust plane, policy plane, etc. With time, this approach morphed into the need to view the "stretched data lakes" as something that can transparently control data flows and data pipelines both for cases of a single organization managing data across its own multiple distributed environments (the original, "stretched" data lakes) and also for cases where multiple organization deploy and share data across federation of such distributed environments.

While the concept of Shared Federated Data Products (SFDPs) was introduced to enable trusted cross-organizational data sharing, it also offers an opportunity to address the infrastructure-related challenges that would arise from exposing Federated Data Products (FDPs) directly. As part of our work on the TEADAL Stretched Data Lakes and its Control Plane, we have leveraged this opportunity to develop a complete toolchain for the creation and lifecycle management of SFDPs.

As a result, the "stretched data lakes" concept has evolved within WP4 into a more precise technical vision: the dynamic deployment, management, and execution of shared federated data pipelines across multiple organizational boundaries and across different TEADAL Nodes. This vision materializes primarily through separating the required functionality into four key subsystems of the TEADAL control plane: the **monitoring subsystem** for collecting, analysing, and externalizing the runtime performance data, the **automation subsystem** for automated creation and controllable execution of SFDPs, the **optimization subsystem** for selecting the deployment targets for SFDPs, and the **deployment subsystem** for ensuring the SFDP is deployed on a selected target and managing its runtime execution and lifecycle.

WP4's control-plane-oriented contributions have significantly shaped TEADAL's capacity to operationalize core concepts such as FDPs (Federated Data Products) and sFDPs (Shared Federated Data Products), making the automated process of FDP-to-SFDP creation a cornerstone of the federation's practical data-sharing capabilities. To set the context, we first briefly summarize TEADAL architectural concepts most relevant to WP4:

- **Federated Dataset Exposure** whereby organizations expose internal datasets through RESTful APIs known as Federated Data Products (FDPs). These endpoints encapsulate access control, metadata, and discovery logic and are designed to be published in a shared TEADAL Catalogue.
- **Data Sharing via sFDPs**, enabling one organization to re-share or transform data obtained from another organization's FDP. This capability supports the "negotiated access" model across federations and decouples data provisioning from local organizational control.

Funded by the European Union

- **Runtime Automation and Discoverability** with GitOps, where declarative configurations are stored in Git repositories and used by argoCD daemons to manage the lifecycle of FDPs and sFDPs on TEADAL Nodes.
- **Decentralized Policy Management** allowing identity and access control to remain under local organizational policies while FDPs and sFDPs are deployed across all the Nodes in TEADAL Federation. This way data sovereignty is ensured with no centralized data broker required at runtime, avoiding centralization wherever possible.

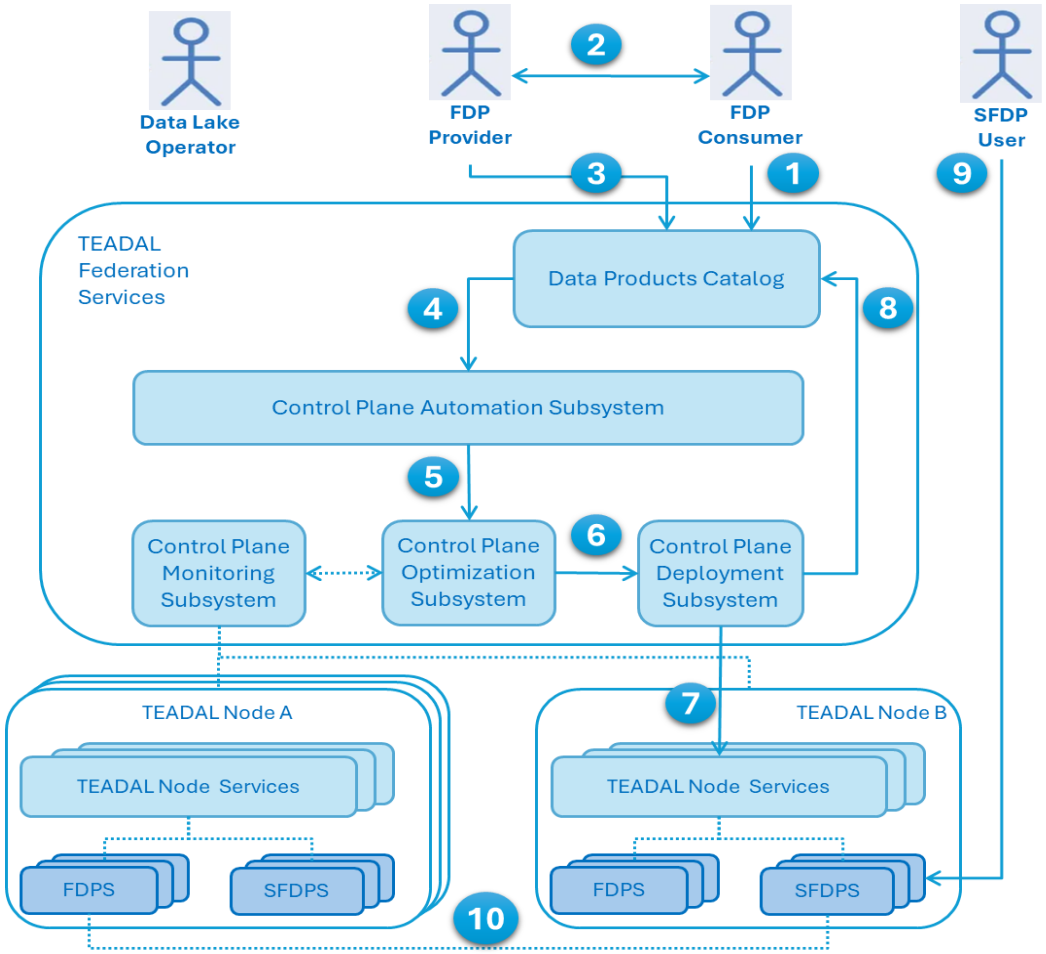

FIGURE 1 CONTROL PLANE COMPONENTS IN SFDP PRODUCTION AND USAGE

2.2 STRETCHED DATA LAKES ARCHITECTURE

WP4 architecture, with its four subsystems, is aligned with the principles listed above and, in addition, supports reproducibility, auditing, and automation of TEADAL Infrastructure and data products deployed on it. The monitoring subsystem acts as runtime planning assistant to the optimization subsystem that selects the deployment targets, the automation subsystem helps users in generating compliant transformation plans, automatically synthesizing sFDPs, and ensuring their compliance with their specifications at runtime. This architecture allows the deployment subsystem to be very simple with the only requirement to support the deployment of the ready to go SFDPs with their deployment manifests to the selected TEADAL Nodes.

Figure 1 presents the conceptual overview of the main control plane subsystems as part of simplified outlook on TEADAL Federation and shows, step-by-step, how these

Funded by the European Union

subsystems contribute to the end-to-end FDP-to-SFDP production and usage processes, helping to serve the needs of TEADAL actors:

1. FDP Consumer (a TEADAL actor defined to represent an organisation of the federation that searches for an FDP and negotiates the agreement to access it as SFDP), looks up the Catalogue and selects the FDP its organization needs access to.
2. The Catalogue notifies the FDP Provider (a TEADAL actor defined to represent an organisation of the federation with the right to access and share data; inside its organization, FDP Provider communicates to additional actors, the FDP Developer, and the FDP Designer) and initiates a process between the FDP Consumer and the FDP Provider to negotiate the terms and to specify the SFDP details.
3. As soon as the contract and the specification are ready, FDP Provider notifies Catalogue that the FDP-to-SFDP creation process can proceed to its next stages.
4. Next stage of the FDP-to-SFDP creation process invokes the automation subsystem of the TEADAL control plane to create the SFDP as a k8s-deployable server application.
5. When k8s-deployable server application is created, the control plane optimization subsystem is invoked to select the preferable deployment target for this SFDP, in this example, Node B. This is done ‘in consultation’ with the monitoring subsystem that collects runtime information about the federation’s Nodes and their workloads.
6. After the target TEADAL Node is selected, the control plane deployment subsystem is requested to take care of deployment on Node B.
7. Control plane deployment subsystem ensures Node B’s k8s control plane realizes the deployment of SFDP as a new service.
8. Catalogue is notified of the new SFDP readiness and finalizes the FDP-to-SFDP creation process, e.g. by notifying the involved agents.
9. As a result, users in FDP Consumer’s organizations can access the newly created SFDP and request the data.
10. From now on, SFDP serves data requests to its users while making data requests to its source FDP and transforming the fetched data according to the SFDP specification. At runtime, this is enabled by the automation subsystem of the control plane as explained in detail in Section **Error! Reference source not found.**.

Note that Figure 1 simplifies the presentation by not showing internal details of the above steps and by referring to only one TEADAL component other than control plane proper, the Catalogue. Additional TEADAL Platform subsystems and services play crucial roles in this process, e.g. for policy definition, k8s manifests creation, runtime policy enforcement, GitOps, etc.

2.3 SUPPORTING TEADAL REQUIREMENTS

From project wide business-level perspective, the motivations behind the SFDP concept are:

- facilitate internal data-sharing agreements without altering source systems
- provide governance, and traceability for reused data flows
- act as policy-compliant "middleware" between FDP Consumers and FDP Providers
- reduce time-to-data and development effort for data for FDP Consumers

At the work package level, we translate these motivations into a set of system-level objectives:

- enforce data-sharing contracts at the API level

Funded by the European Union

- apply required data transformations and policies before exposure
- preserve auditable boundaries between data ownership and consumption
- decouple FDP Consumers from the technical and semantic complexity of data sources
- represent transformed/contractual APIs as reusable, discoverable data products

Below we demonstrate how architectural contributions presented in the previous subsection align with and help addressing the architectural and the operational requirements specified across the TEADAL project in D2.2 [2]:

1. Enabling Dynamically Prepared Shareable Data (Req. 1.a in D2.2 [2])

WP4 work provides the concrete mechanisms to instantiate sFDPs as dynamic, deployable services. With TEADAL Platform's approach to declarative definitions, GitOps-based automation, and a Kubernetes-native control plane, TEADAL pipelines can be automatically materialized in accordance with organization-to-organization agreements. The automation subsystem facilitates this dynamic instantiation of sFDPs by combining transformation logic with access policies and endpoint specifications.

By treating sFDPs as deployable units tailored to specific sharing contexts, WP4 ensures that data sharing is both compliant with federated governance rules and adaptable to evolving requirements.

2. Simplifying Data Lake Management (Req. 1.b in D2.2 [2])

Rather than adopting a narrow "serverless" framework, WP4 advances the spirit of serverless computing by embracing declarative orchestration, GitOps workflows, and automation of routine operations. This approach relieves users from the burden of manually provisioning, scaling, or debugging services, instead enabling them to express intent at a higher level (e.g., "this dataset should be made available via this transformation, with these access rules"), allowing the TEADAL control plane to effectively abstract the operational complexity of multi-cluster infrastructure and aligning well with the goal of simplifying data lake management.

3. Supporting Data Discovery (Req. 1.d in D2.2 [2])

While responsibility for implementing the Data Catalogue mostly belongs to other work packages, WP4 ensures compatibility and interoperability with it. The automation and the deployment subsystems anticipate integration with the Catalogue for dataset and pipeline publication and discovery, as well as for triggering the SFDP generation and deployment processes. While architecturally, the monitoring and the optimization subsystem are internal to the control plane, their integration to 'actor-facing' components such as Catalogue can be beneficial in the future, e.g. for providing federation-wide observability and allowing actors to affect optimization goals and parameters. In addition, ongoing exploration into aligning transformation components with Model Catalog Platforms (MCPs) and Model Context Protocol, mentioned in Introduction and further explained in next sections, has a potential to further reinforce WP4's alignment with TEADAL's broader discoverability objectives.

4. Reducing Data-Sharing Friction (Req. 2.a in D2.2 [2])

WP4 directly addresses friction in federated data sharing by helping to automate SFDP negotiation and generation, and by taking care of runtime compliance of SFDP to its negotiated specification. The automation subsystem is created as a framework that captures sharing agreements and data transformations as structured, versioned, and reproducible templates, dramatically lowering the coordination and integration overhead typically required when bridging organizational boundaries. Monitoring and optimization subsystems contribute by

Funded by the European Union

streamlining the planning phase of data sharing, helping to generate appropriate configurations, exposing relevant constraints, and aligning local and federated policies. This design aligns well with work done in WP3 for transforming the abstract concept of “data-sharing friction” into a solvable engineering challenge.

5. Managing Data and Computation Placement (Req. 2.b in D2.2 [2])

Through the control plane, WP4 introduces mechanisms to flexibly schedule and deploy data pipelines close to data sources or downstream consumers, depending on cost, policy, or performance considerations. This elasticity directly supports the architectural goal of stretching the data lake across the computing continuum, optimizing placement of data and computation. With declarative pipeline definitions and cluster-aware deployment assisted by the telemetry (e.g., cost, latency, energy) implemented by the monitoring subsystem, TEADAL control plane design lays the groundwork for intelligent workload scheduling across the federation.

6. Contributing to Energy-Aware Orchestration (Req. 2.c in D2.2 [2])

Although energy modelling is addressed in another work package, WP4 provides the practical foundation needed to implement these models. The deployment subsystem, through declarative labels and telemetry provided by the monitoring subsystem that already collects some energy-related metrics, can implement the energy goals as part of selecting execution targets. The automation subsystem contributes by enabling the runtime selection of transformations the data goes through as part of the FDP-SFDP pipelines, based on how much energy each specific implementation of the given transformation requires. Additional well-known energy saving strategies, such as scaling deployments to zero when idle, and orchestrating transformations in energy-efficient locations, can be integrated in production systems. Future full integration with energy-aware metrics will empower TEADAL to act on optimization strategies in real-time, completing the feedback loop between analysis and execution.

Across all the above requirements, WP4 helps “operationalizing” TEADAL’s architectural vision and SFDP concept, while integrating with other work packages` outcomes, such as Catalogue, operational models (data friction, data gravity, performance and policy modelling), policy definition and enforcement tooling, trust subsystems, etc. WP4’s realization of the "stretched data lake" concept is created to ensure those ideas can be instantiated, managed, and evolved in real-world deployments and help positioning TEADAL as a pioneering effort in distributed, automated, and trustworthy data collaboration.

2.4 REALIZATION AND PLATFORM INTEGRATION

The TEADAL Platform, conceived at the architectural level in WP2 ([2][3]) and realized in WP6, has been evolving throughout the project. At the time of writing this deliverable, the TEADAL Node exists as capable and production-ready platform ready to be installed and supporting a range of enabling technologies and services:

- GitLab CI/CD and Argo CD for GitOps-driven automation,
- OPA (Open Policy Agent), Rego, and Keycloak for secure identity management,
- Istio for fine-grained policy enforcement,
- Prometheus, Thanos, Kepler, and Grafana for federated observability and resource monitoring.

The Stretched Data Lakes components described here have been designed to be compatible with this curated TEADAL technology stack and to interface with the broader TEADAL platform architecture, leveraging and respecting project-wide decisions and technology choices, e.g.:

Funded by the European Union

- Reliance on k8s control plane for in-cluster orchestration and placement: WP4 assumes that, with respect to control plane, TEADAL Nodes are k8s environments. Thus, there is no dedicated in-cluster runtime “control plane”; instead, this is handled by standard k8s operators within each organizational cluster extended by additional CRDs.
- Reliance on GitOps for realizing the deployments: WP4 assumes that, with respect to managing deployments, TEADAL Nodes are enabled with cluster-scoped GitOps tooling via Argo CD that can be leveraged for executing the deployments after the deployment target (as TEADAL Node) is selected by the optimization subsystem).
- Reliance on TEADAL Data Catalogue to coordinate the FDP-to-SFDP creation process starting from the FDP Consumer requesting access to a certain FDP and complete with the data users from the FDP Consumer’s organizations successfully access the data.
- Reliance on TEADAL tooling for policy definition, specification and enforcement.

Whenever feasible, WP4 components are integrated with the foundational services of the TEADAL Node, as explained in the following sections. Some components were created and demonstrated well before the TEADAL stack existed in its final state and need to be further adapted before their full integration. For example, the Multi-Cloud Computer Compiler (MCC-C) is not yet fully integrated at the time of writing this deliverable. We plan to complete the integration at the extent possible till the end of the project and in scope required to support major use cases related to multi-cluster (multi- TEADAL Node) federations. Integration status of WP4 components is as follows.

- The monitoring subsystem is integrated into TEADAL Nodes (see Section 3).
- The automation subsystem consists of several components and services, some fully integrated to TEADAL Nodes while some are expected to be run in development environments of FDP Developers/Designers (see Section 4).
- The optimization subsystem, as explained above, is not fully integrated yet but is integration ready in principle as it was developed to consume labelled descriptors of both workloads (e.g., sFDPs) and the infrastructure nodes (e.g., cluster capabilities) and to output placement suggestions (see Section 5).
- The deployment subsystem was initially planned to rely on Kubestellar to orchestrate deployments and state synchronization across TEADAL Nodes based on optimization subsystem’s output. However, this direction was abandoned mid-project as attention shifted toward AI-driven planning and automation and as the multi-cluster orchestration landscape has matured significantly, with numerous additional open-source and commercial solutions now available to manage cross-cluster workloads, policies, and data movement (see Section **Error! Reference source not found.**).

Catalogue-centered Views for SFDP Creation

To complete our explanation of the data-sharing process, this section presents the Catalogue-centered perspective on the creation and governance of SFDPs in TEADAL. Full description of Catalogue architecture and its interfaces can be found in D3.3 [10].

The TEADAL Catalogue is a web application that employs Business Process Model and Notation (BPMN), a standard for defining workflows that involve human and system activities, to coordinate multi-step actions. BPMN workflows are visually defined and executed by a workflow engine, but users only interact with them indirectly, via buttons, forms, or automatic notifications in the Catalogue UI (see Figure 2). This ensures that even complex coordination across federated organizations remains user-friendly and traceable. The Catalogue defines and manages the following types of data-sharing assets:

Funded by the European Union

- **Dataset**: A local asset that describes a data source (e.g., file, URL, database table, S3 bucket) and its associated metadata. A Dataset is visible only within the organization that owns it.
- **Federated Data Product (FDP)**: A public-facing service description built on one or more Datasets. It includes technical access information and data-sharing policies. FDPs are visible across the federation.
- **Shared Federated Data Product (sFDP)**: A customized data product instance created to fulfil a specific access request from another organization.
- **Agreement**: A recorded outcome of a negotiation process that documents the terms and results in the creation of an sFDP.

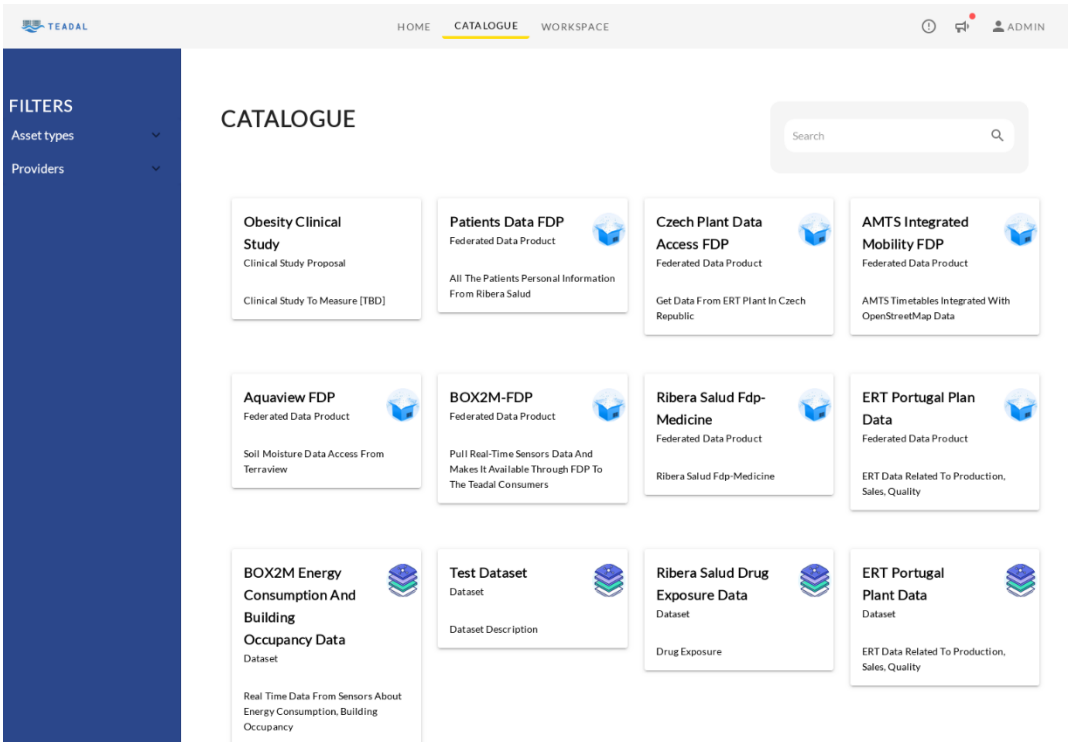

FIGURE 2 CATALOGUE – UI VIEW

When a user belonging to a federated organisation (FDP Consumer) discovers an interesting FDP via the Catalogue, they can initiate a data access negotiation process, also via the Catalogue. This triggers a BPMN workflow that takes care of informing the owner of the FDP (FDP Provider) about the access request and, optionally, supports the structured information exchange between the actors while negotiating, presented in Figure 3.

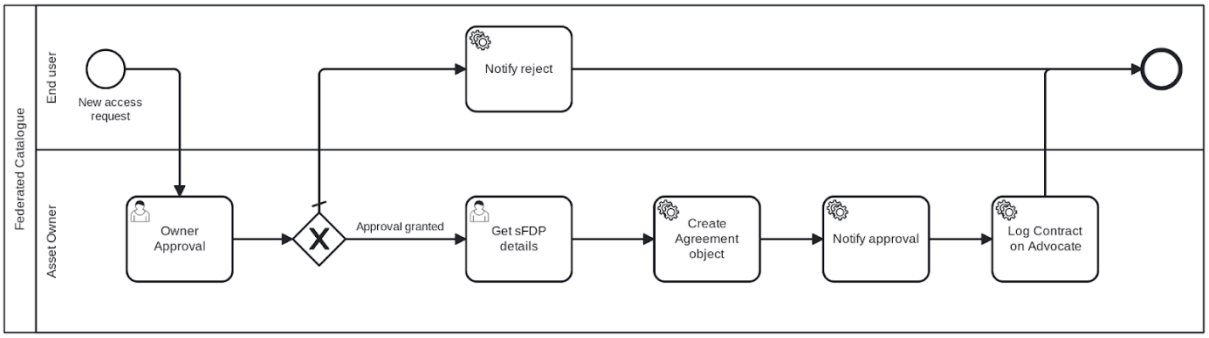

FIGURE 3 CATALOGUE – CONTRACT REQUEST PROCESS

Funded by the European Union

When the negotiation process is successfully finished, two things happen:

- The owner of the FDP creates a Shared Federated Data Product (sFDP) to allow the requester access to the FDP data according to the policy that has been negotiated between the parties. The sFDP is then described in the Catalogue and linked to its FDP
- The BPMN process that regulates the information exchange between FDP owner and the federated user asks the FDP owner to state the identifier of the newly created sFDP, sends such details to the requester, and then creates an Agreement object in the Catalogue. The Agreement object just records that an sFDP has been created as a reaction to a request by a specific user related to an FDP.

As a result, the FDP owner maintains visibility over all derived sFDPs and the reasons they were created. Moreover, as the Catalogue allows binding BPMN processes to asset types, we can handle contract termination. When inspecting an Agreement object, both parties can decide to terminate the data sharing agreement via a button that triggers a custom BPMN, presented in Figure 4.

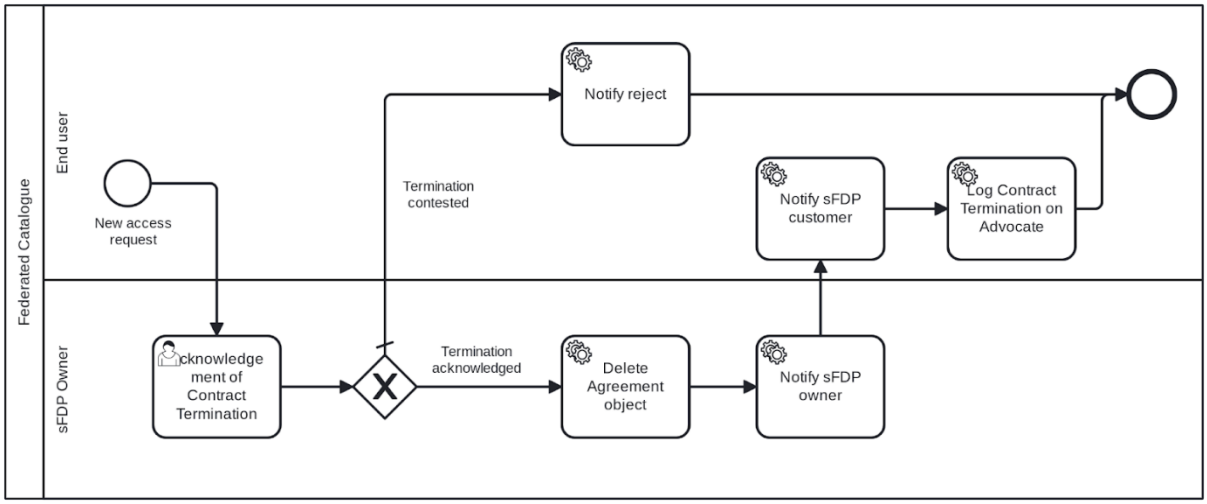

FIGURE 4 CATALOGUE – AGREEMENT TERMINATION PROCESS

All the events in the interactions described above are also tracked in the Advocate tool. Such tracking is implemented via service tasks in the BPMN processes that have been created to support the use cases. As a result, there's always evidence about any actions regarding Datasets, Federated Data Products and Shared Federated Data Products, related both to their lifecycle management and their usage by other users in the federation.

In addition to user-driven interactions through the Catalogue web UI, the TEADAL Catalogue also exposes a set of APIs that enable automated workflows and programmatic access to catalog functions. This allows internal services, CLI tools, or external platforms to interact with catalog assets (e.g., initiate access requests, create FDPs/sFDPs, or register agreements) without requiring manual UI steps. By supporting both UI and API-based interactions, the Catalogue accommodates a wide range of needs, from exploratory workflows to fully automated data sharing pipelines. The Catalogue exposes API methods supporting the following operations:

- CRUD operations on assets
- Sending notifications to users or groups of users
- Asset status management

Funded by the European Union

- Listing of assets types and their JSON Schema for validation purposes
- Performing controlled SPARQL queries as explained in D3.3
- Creating and managing User Requests, a system heavily inspired by ticketing that allows capturing user requests and linking them to the execution of BPMN processes

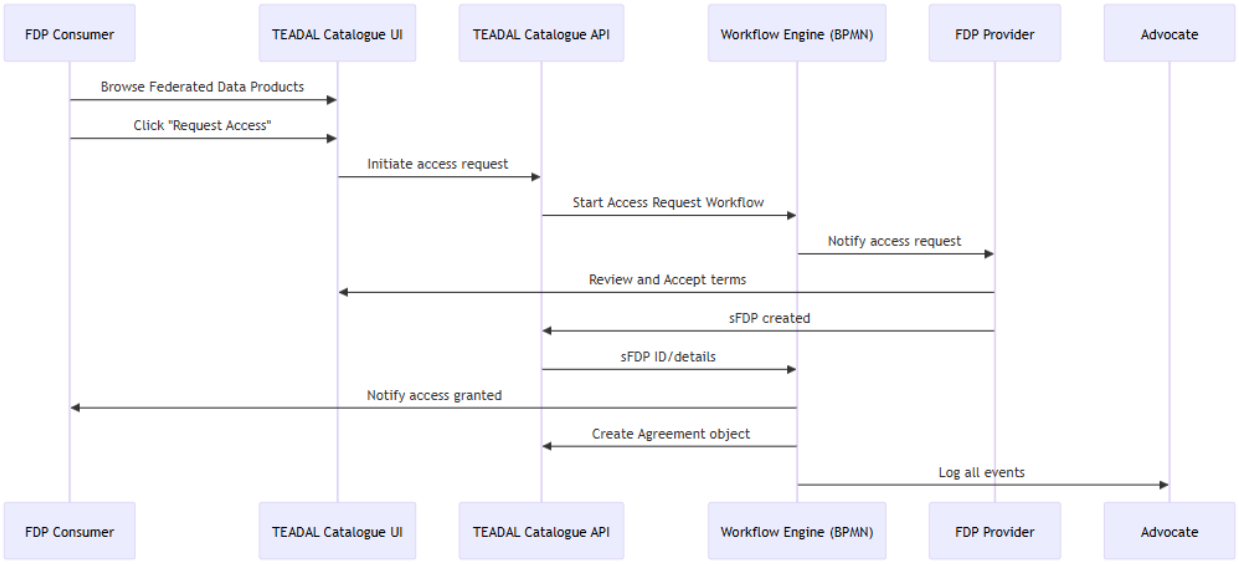

FIGURE 5 PROCESS COORDINATION THOUGH CATALOGUE UI AND API

Figure 5 presents a possible way to leverage Catalogue's ability to coordinate complex user flows both through the UI and through the API. In Figure 5, the SFDP creation flow, typically initiated by human actors using the UI, can proceed through automation steps involving other TEADAL components invoked by the or on behalf of the FDP provider. For example, ASG-tool is invoked to create the SFDP app and initial artefacts, followed by the policy generation, the deployment target selection, and the deployment. To maintain end-to-end integrity of this operation, components and tools can use the Catalogue API programmatically, to inform about the stages the request is going through and to notify the Catalogue about SFDP readiness.

From auto-generated SFDPs to running services

Control plane automation subsystem has a tool for auto-generating the SFDP server apps and a library for supporting the autogenerated SFDPs at runtime (see Section 4). In between, after the SFDP server app is generated and before it is accessible on its preferred TEADL Node, several further steps are taken by a broader TEADAL platform:

- Create YAML manifests for describing the required k8s resources
- Create kustomize files for firing the argoCD pipelines
- Create Rego files for encompassing the policies related to the SFDP access
- Optionally, annotate these artifacts with the runtime annotations, when for example the SFDP can benefit from running on GPU nodes or requires to be run in the trusted environments
- Select the deployment target for the new SFDP, among the TEADAL Nodes existing in the Federation (this is handled by the optimization subsystem of WP4 and is included here for completeness)

Funded by the European Union

- Deploy the new SFDP on the selected deployment target and initiate its runtime monitoring (this is also by handled by WP4 subsystems and is included here for completeness)

Following these steps, the Catalogue can be notified to complete the SFDP creation flow by publishing the new data product and alerting the FDP Consumer that have initiated the request for this new SFDP.

Role Based Access Control (RBAC) Framework

While the framework outlined in detail in D4.2[8] can be used to enforce any kind of access control, a setup was developed for TEADAL that also provides a built-in Role-Based Access Control (RBAC) framework. This framework dramatically reduces the effort needed to implement access control for RESTful services, while still leaving policy writers the freedom to extend the base framework with service-specific functionality.

Data lake users are managed through a federated, OpenID Connect or OIDC-compliant Identity Management (IdM) service. Consumer services act on behalf of users who have proved their identity through IdM-configured procedures such as credential challenges, multi-factor authentication, etc. Upon successful authentication, the IdM issues an identity token, more specifically a JSON Web Token (JWT), which certifies the user's identity. Consumer services attach the token to each data product service request by means of the Bearer HTTP Authorization header. Presently, TEADAL deploys Keycloak as an IdM service, although any other OIDC-compliant software could be used too as the RBAC framework only requires OIDC-compliance, making no assumption about the actual IdM implementation.

RBAC roles, users and policy rules are written in plain Rego. Thus, policy writers are empowered with a fully-fledged programming language which they can exploit to customize, abstract and reuse their roles and policies to an extent that is simply not possible with traditional, configuration-based, cloud Identity and Access Management solutions. Moreover, policy writers can implement automated Rego tests to verify their policies have the desired effect when evaluated or even do that interactively, for rapid prototyping, as the OPA runtime has both test and read-eval-print loop (REPL) facilities. Extensive, automated tests also prevent regression issues where modifying a rule may have an unforeseen, unwanted side-effect, possibly leading to a security incident. Again, this level of sophistication is extremely expensive, in terms of the required effort, to attain with traditional Identity and Access Management solutions.

The TEADAL "*authnz*" Rego library is a good case in point. Policy writers import this library in their code to automatically handle the evaluation of RBAC rules, user authentication, JWT validation, OIDC discovery as well as cryptographic keys download, verification and caching. The library allows policy writers to concentrate on defining their own, service-specific access control rules using an intuitive format.

For example, consider securing a simple FDP. The REST service exposes patient records as Web resources. There are three *paths:/patients* to list and add patients*, /patients/id/* to retrieve and delete a particular patient, and */patients/age* to retrieve a list with the ID and age of each patient but nothing else. Also, there is a */status* path which returns the current service status. We would like to define two roles. A product owner, which should be able to perform a GET, POST and DELETE on any URL path starting with */patients*, and a product consumer, which should only be allowed to GET patient ages and service status. Moreover, we would like to assign both the product owner and consumer roles to the user identified by the email of *jeejee@teadal.eu* whereas just the product consumer role to the user identified by the email of *sebs@teadal.eu*. In the TEADAL RBAC framework, all the above can be accomplished with the Rego code presented in Figure 6.

Funded by the European Union

```
# Role defs.
product_owner := "product_owner"
product_consumer := "product_consumer"

# Map each role to a list of permission objects.
# Each permission object specifies a set of allowed HTTP methods for
# the Web resources identified by the URLs matching the given regex.
role_to_perms := {
    product_owner: [
        {
            "methods": [ "GET", "POST", "DELETE" ],
            "url_regex": "^/patients/.*"
        }
    ],
    product_consumer: [
        {
            "methods": [ "GET" ],
            "url_regex": "^/patients/age$"
        },
        {
            "methods": [ "GET" ],
            "url_regex": "^/status$"
        }
    ]
}

# Map each user to their roles.
user_to_roles := {
    "jeejee@teadal.eu": [ product_owner, product_consumer ],
    "sebs@teadal.eu": [ product_consumer ]
}
```

FIGURE 6 EXAMPLE REGO CODE FOR SECURING AN FDP

To evaluate our RBAC rules against the request received from Envoy, we would simply import the TEADAL "*authnz*" library and call its *allow* function as exemplified by the Rego code snippet in Figure 7, where we tacitly assume the RBAC rules defined earlier are in a package imported as *rbac_db*.

```
default allow := false

allow = true {
    # Use the `allow` function from `authnz.envopa` to check our RBAC
    # rules against the HTTP request received from Envoy. The function
    # returns the user extracted from the JWT if the check is successful.
    user := envopa.allow(rbac_db)

    # Put below this line any service-specific checks on e.g. the
    # HTTP request received from Envoy.
}
```

FIGURE 7 METHOD FOR EVALUATING RBAC RULES AGAINST THE REQUEST

Funded by the European Union

As already mentioned, "*authnz"* automatically handles the evaluation of RBAC rules, user authentication, JWT validation, OIDC discovery as well as cryptographic keys download, verification and caching. Also of note, "*authnz"* provides built-in functions to evaluate user-defined RBAC rules interactively in the Rego REPL. This is useful for dry-run scenarios where a policy writer may want to see what the effect of their RBAC rules is before deploying them to the data lake.

In the previous example, roles are defined in Rego along with the mapping of users to roles. It is also possible to define roles in the IdM where users are kept and use the IdM's tools to associate users with roles. In this case, the Rego policy can be simplified to contain the *role_to_perms* map associating each role defined in the IdM to a list of permission objects as shown in Figure 8.

```
role_to_perms := {
    "product_owner": [
        {
            "methods": [ "GET", "POST", "DELETE" ],
            "url_regex": "^/patients/.*"
        }
    ],
    "product_consumer": [
        {
            "methods": [ "GET" ],
            "url_regex": "^/patients/age$"
        },
        {
            "methods": [ "GET" ],
            "url_regex": "^/status$"
        }
    ]
}
```

FIGURE 8 REGO POLICY EXAMPLE FOR ASSOCIATING PERMISSIONS WITH ROLES

This Rego code defines a policy that has the same effect as that presented earlier where users were explicitly associated with roles through the *user_to_roles* map.
A mixed scenario is also possible, where some roles are defined in the IdM and others in Rego policies but regardless of the approach, if some (or all) roles are managed in the IdM, then:

- the IdM must generate access tokens that include not only the authenticated user's ID, but also a list of roles the user belongs to; and
- *"authnz"* must be configured to read both the user ID and the roles from the access token.

In this setup, *"authnz"* merges any roles extracted from the token with the roles defined for that user in Rego. For added convenience, *"authnz"* treats each user as a singleton role. More precisely, *"authnz"* identifies every user *"u"* with a role named *"u"*, which contains only *"u"* as its member. For example, the user sebs@teadal.eu from the previous example implicitly has a corresponding role also named *sebs@teadal.eu*, with the user as its sole member. These implicit singleton roles allow policy writers to assign permissions directly to a user in the *role_to_perms* map, without needing to explicitly list the user as an additional role in the *user_to_roles* entry for that user.

Funded by the European Union

For example, suppose the policy writer wants to extend the previous policy with a rule specific to the user *sebs@teadal.eu*. As a product consumer, sebs@teadal.eu does not have access to service metrics. Without implicit singleton roles, the policy writer would need to manually add an entry to *user_to_roles* to associate *sebs@teadal.eu* with a role of the same name, in order to then add a corresponding entry for role *sebs@teadal.eu* to the *role_to_perms* map, as shown in Figure 9.

```
role_to_perms := {
    # ... same entries as earlier, plus
    "sebs@teadal.eu": [
        { "methods": ["GET"], "url_regex": "^/metrics/.*" }
    ]
}
user_to_roles := {
    # make `sebs@teadal.eu` a member of `sebs@teadal.eu` so the
    # above `role_to_perms` works.
    "sebs@teadal.eu": [ "sebs@teadal.eu" ]
}
```

FIGURE 9 EXAMPLE OF NOT USING IMPLICIT SINGLETON ROLES

While this works, it is cumbersome and places an additional burden on the policy writer—especially when roles are managed externally in an IdM system. Ideally, in such cases, the policy writer should only need to specify the *role_to_perms* map, without also maintaining the *user_to_roles* map. With implicit singleton roles, there is no need to explicitly map users to roles of the same name. The policy writer can simply rewrite the code as shown in Figure 10:

```
role_to_perms := {
    # ... same entries as earlier, plus
    "sebs@teadal.eu": [
        { "methods": ["GET"], "url_regex": "^/metrics/.*" }
    ]
}
```

FIGURE 10 EXAMPLE OF USING IMPLICIT SINGLETON ROLES

Policy Writing and LLM assisted Policy Generation and Bundling

While the RBAC framework provides a way to set up policies in relation to an OIDC compliant IdM, OPA (and consequently the Rego language) provides a lot more flexibility in defining policies. Multiple Rego files, defining complex policies that make very in-depth checks against a variety of data, can be packaged together into a bundle. This bundle can be deployed directly alongside OPA, or deployed elsewhere, with OPA configured to pull the bundle dynamically. This would enable policies to be updated and enforced continuously.

For this reason, we have experimented with a “Policy Editor” web app that enables authorized users to write and edit arbitrary policies and bundle them at once.

Funded by the European Union

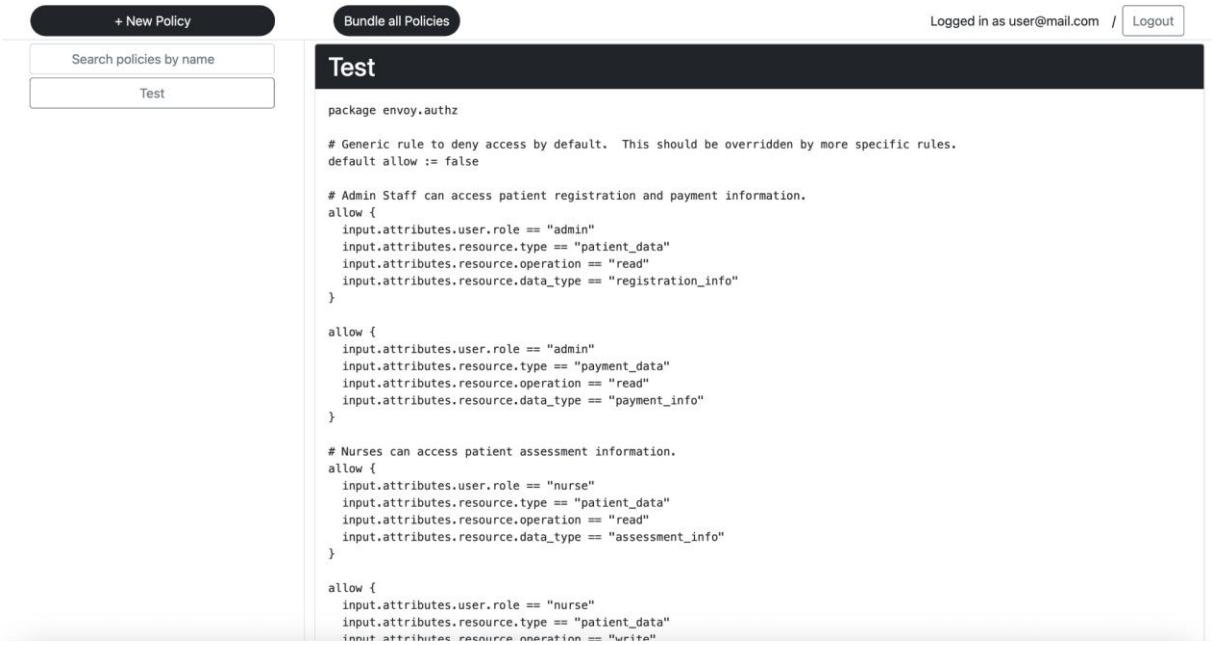

FIGURE 11 AUTHORING POLICIES IN THE "POLICY EDITOR" WEB APP

The policies authored in the app, e.g. as shown in Figure 11, once bundled, will automatically be picked up by OPA and thus enforced by it. Being a web app, it can easily be accessed by any browser. Only authorized users (with an account created for them) can access the editor and make changes to the policies. Because it allows for plain (valid) Rego code, it provides full flexibility on how the policies are written, be it simple checks on web token contents, or more complex API calls that can pull additional data to be used in formulating the policies.

Moreover, albeit in a very experimental way at this stage, the Editor also provides a way to generate Rego policies from natural language descriptions by leveraging AI. Figure 12 presents the view where the user inserts the name and the description of the policy to be generated and Figure 13 presents the view where the app displays the generated policy that user can inspect, manually edit, and, eventually, use.

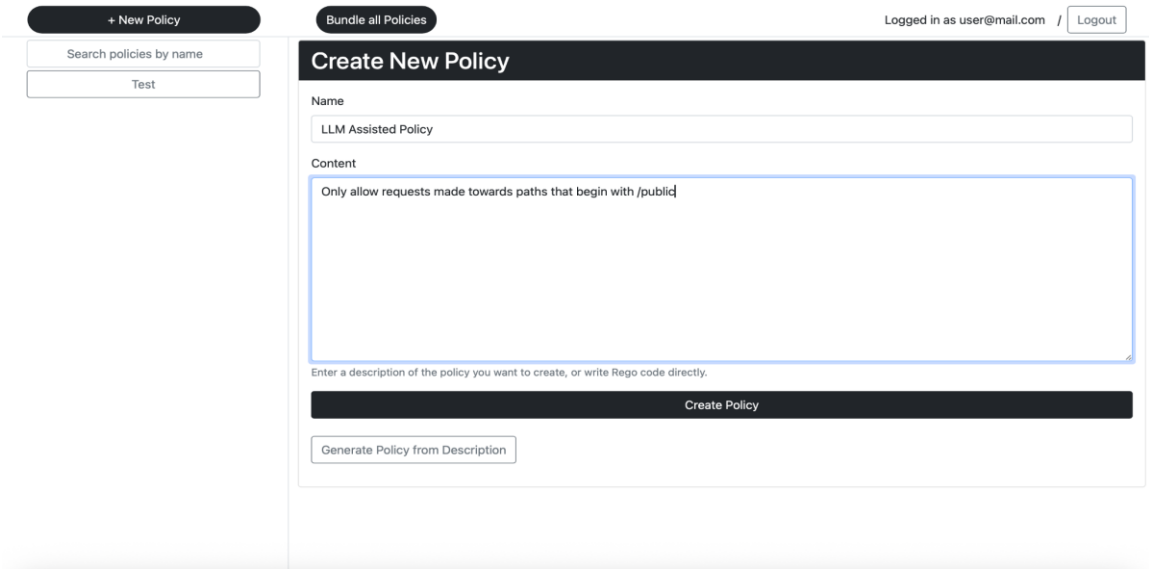

FIGURE 12 POLICY EDITOR – POLICY GENERATION REQUEST VIEW

Funded by the European Union

This is done by forwarding the description of the Policy to a middleware API, which in turn will interact with a configured LLM (either locally with Ollama available as part of TEADAL Nodes, or remotely towards one of the large models such as Gemini, Claude, GPT4.1, etc.). The API is configured to provide context for Rego code generation for OPA and, optionally, can be configured to interact with Model Context Protocol (MCP) servers. These can provide further specific context to the interactions with the LLMs, as well as tools that could enable the models to take some actions directly. While still a work in progress, there is the possibility to leverage this protocol to provide real time information on the state of a system that requires policies, allowing the AI to dynamically refer to it and generate meaningful relevant policies, streamlining the process.

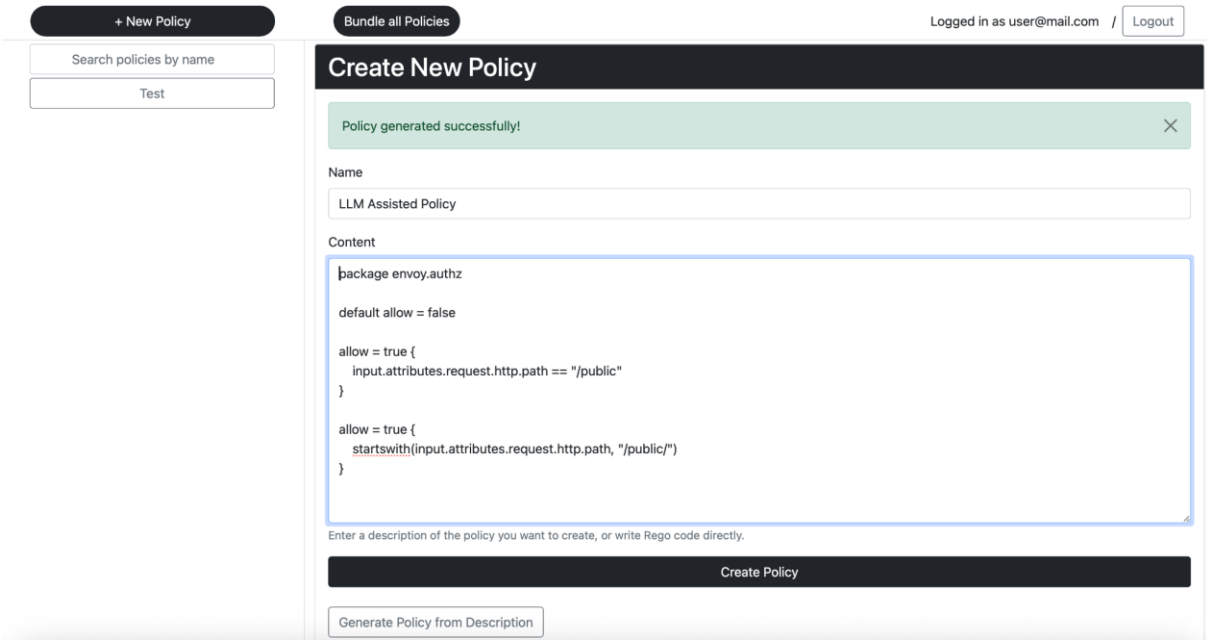

FIGURE 13 POLICY EDITOR – POLICY GENERATION RESULT VIEW

Note that a more structured and mature model driven approach to policy generation is fully presented in D3.3[10]. The LLM-based approach presented here is ultimately an exploration of alternative ways to assist the users in generating meaningful policies based on existing contextual data. The Policy Editor Web Application is available in TEADAL GitLab.

Funded by the European Union

3. THE MONITORING SUBSYSTEM (AI-DPM)

3.1 AI-BASED PERFORMANCE MONITORING PROCESS

In distributed Information Technology (IT) systems managed by platforms like Kubernetes, gaining visibility into resource utilization, such as CPU, memory, disk usage, energy consumption, and network behaviour, is essential for the proactive management of system functionality and performance. To contribute to addressing these needs in the TEADAL project, the component AI-DPM (Artificial Intelligence-Driven Performance Monitoring) has been produced. By applying Artificial Intelligence for IT Operations (AIOps), AI-DPM enables intelligent monitoring across federated environments. This component operates as part of the Control Plane, to generate insights on the performance of the TEADAL infrastructure and application environment resource utilization.

The AI-DPM relies on historical time-stamped metrics metadata from the TEADAL resource utilization system behaviour to **detect anomalies** and **generate predictive insights**. It was initially designed to provide insights on resource utilization to the Control Plane optimizer, to let it define and establish effective strategies for optimizing data flows in TEADAL. In the latter phases of the project activities, it has evolved as a standalone support tool developed as REST API services, and available for other TEADAL components, providing a wider set of different monitoring insights.

The progression of the AI-DPM experimental process consists of interrelated incremental phases. In the first iteration, the initial feasibility study and the foundational AI-DPM design, as described in Deliverable D4.1 [7], were presented. In the second iteration, the Proof of Concept (PoC) experimental results, based on classical AI models trained and tested using a publicly available metadata set, were shared in Deliverable D4.2 [8]. In this final iteration, AI-DPM tool has reached its full maturity, featuring incremental advancements:

- **Expanded metadata capture:** The scope of collected metadata has grown beyond traditional system resource performance metrics to include energy consumption data and service-mesh observability metrics within the TEADAL framework.
- **Enhanced Experimentation strategy:** To evaluate the performance of the models *µBench*-based experimentation environment was used to stress the system and collect the metadata. µBench is an open-source software that emulates real-world Kubernetes cluster scenarios. It is a tool designed to assess the performance of microservices within a Kubernetes environment. It allows users to simulate the behaviour of an application composed of multiple microservices, performing load tests and collecting metrics like CPU, memory, network, and disk space usage.
- **Enhanced AI model integration:** The original suite of statistical and classical AI models has been extended with both local and cloud-based large language models (LLMs), enabling improved analytical capabilities and comparative benchmarking against conventional classical methods.
- **Interactive experimentation dashboard:** In this phase, a dedicated custom dashboard has been added to support user experimentation with metrics selection, model training, testing, and evaluation, allowing users to compare performance, visualize the predictions, and anomaly detection results for actionable insights.

The approach

The AI-DPM approach relies on AIOps to collect multimodal metadata and apply different AI models for anomaly detection and prediction. It has been developed through incremental steps to refine machine learning (ML) algorithms, with recent iterations including the use of large

Funded by the European Union

language models (LLMs). Metadata sources included public data, VM nodes, µBench-based benchmarks, and anticipated TEADAL pilot data during validation. The **AI-DPM process** consists of five steps: (i) metadata collection, (ii) aggregation, (iii) use in the ML cycle, (iv) insight generation, and (v) insight serving. The schematic representation of the end-to-end AI-DPM approach is presented in Figure 14 below.

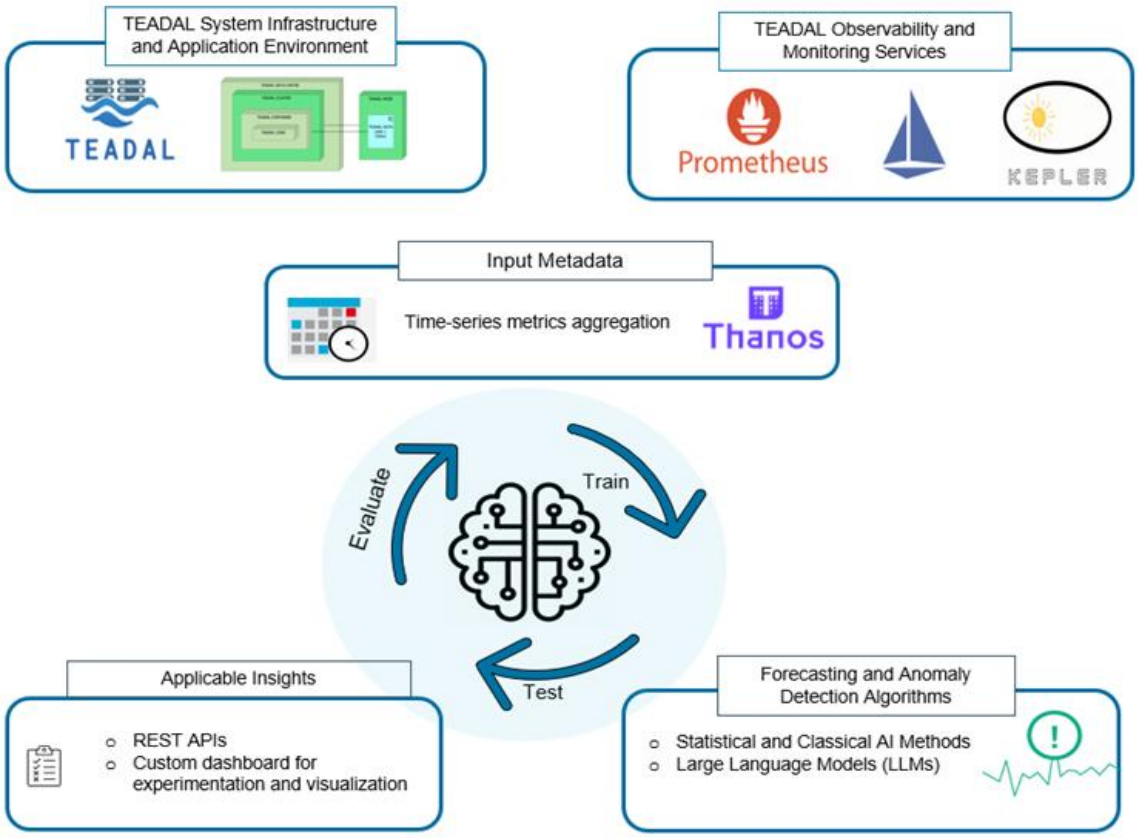

FIGURE 14 THE FIVE-STEP AI-DPM PROCESS

Types of Metadata Collected

The core of the AI-DPM process is managing time-stamped metrics metadata. This runtime metadata is collected across multiple layers of infrastructure and application environments using a stack of observability tools, such as Prometheus, Kepler, and Istio. These triads of monitoring and observability tools specialize in different sets of metadata:

Prometheus: Prometheus[2] is an open-source monitoring and alerting toolkit designed for reliability and scalability in dynamic cloud environments. It collects metrics primarily of resource utilisation from configured targets at specified intervals and supports powerful queries for analysis

Kepler (Kubernetes-based Efficient Power Level Exporter): Kepler[3] is an open-source tool that estimates and exports power consumption metrics in Kubernetes environments. It helps monitor and optimize energy usage across containers and nodes to support sustainable computing.

[2] https://prometheus.io/

[3] https://github.com/sustainable-computing-io/kepler

Funded by the European Union

Istio: Istio[4] is an open-source service mesh that provides a uniform way to secure, connect, and monitor microservices. It manages traffic flow, enforces policies, and offers observability into service communications.

The main types of metadata collected for anomaly detection and predictive insights in AI-DPM are system resource usage metrics, energy and sustainability metrics, and service-mesh observability metrics. The energy and service-mesh metrics were added to AI-DPM in a later phase of the incremental experimentation process to extend the monitored metric dimensions.

Resource usage metrics

Collected via Prometheus, the resource usage metrics capture the performance and the status of infrastructure components, such as nodes and containers. Key macro categories include **compute usage**, **memory consumption,** and **disk Input / Output (I/O)**. These metrics enable workload profiling, anomaly detection, and effective scheduling based on actual resource consumption patterns. Specific examples of resource usage metrics, along with their description and the insights they offer, are provided in Table 1. The complete list of over a thousand Prometheus-scraped metrics is provided as a supplementary file in the GitLab repository.

TABLE 1: PROMETHEUS SYSTEM RESOURCE MONITORING METRICS

Metric	Description
node_cpu_seconds_total	*Total time CPU cores spend executing processes. Used to track CPU utilization.*
node_memory_Active_bytes	*Amount of actively used memory. It is useful for detecting memory leaks.*
node_memory_MemAvailable_bytes	*Amount of memory available for starting new applications. Useful for memory pressure analysis.*
node_disk_read_bytes_total	*Total number of bytes read from disk. Helps monitor disk read I/O usage.*
node_disk_write_bytes_total	*Total number of bytes written to disk. Helps monitor disk write I/O usage.*

Energy and sustainability metrics

Energy and sustainability metrics are collected to provide insights into power usage and energy efficiency at various levels of the system. These metrics are collected with Kepler and include ***instantaneous power consumption***, **accumulated energy usage**, **component-level breakdowns, and platform-level energy metrics**. Collected at runtime, these metrics help assessing the energy consumption associated with different workloads and support sustainability reporting and planning. This way, AI-DPM contributes to energy efficiency goals of the project. Other energy-focused components of TEADAL Platform can use energy and sustainability metrics and insights provided by AI-DPM, for example, to implement to energy-aware scheduling or to select data transformation implementations most suitable to current conditions of the infrastructure and the workloads, as described in D3.3 [10] . A few examples of such Kepler metrics are described along with their insights in Table 2, with additional sets provided in the GitLab repository.

[4] https://istio.io/

Funded by the European Union

TABLE 2: KEPLER METRICS FOR ENERGY AND SUSTAINABILITY MONITORING

Metric	Description
kepler_node_package_joules_total	Total CPU energy consumed by the node (measured in joules). Helps to analyse energy efficiency.
kepler_container_gpu_joules_total	Measures GPU energy usage per container. Useful for AI/ML workloads.
kepler_node_dram_joules_total	Total RAM (DRAM) energy consumption. Helps track memory-intensive applications.
kepler_container_power_watts	Real-time power consumption (watts) of a running container. Useful for power-aware scheduling.
kepler_cpu_usage_ratio	Ratio of CPU usage to power consumed. Helps determine inefficient CPU workloads.

Service-mesh observability metrics

These metrics, shown in Table 3 are gathered from the service mesh layer and offer visibility into inter-service communication, latency, traffic volume, and security events. The main macro categories include ***traffic flow***, ***latency and performance***, ***error and reliability,*** and ***security telemetry.*** The metrics support the analysis of distributed workloads, help identify bottlenecks and validate the behaviour of service-level policies. The full list of these Istio metrics is provided along with their description in the GitLab repository.

TABLE 3: EXAMPLES OF ISTIO SERVICE MESH OBSERVABILITY METRICS

Metric	Description
istio_requests_total	Total number of HTTP requests received by a service. Used to track request load.
istio_request_duration_milliseconds	Measures the time taken to process requests (latency) in milliseconds. Helps detect slow services.
istio_tcp_sent_bytes_total	Total bytes sent over TCP connections by a service. Used for network performance analysis.
istio_requests_duration_seconds_bucket	A histogram of request durations, useful for analysing performance trends.
istio_policy_request_count	Tracks the number of requests that pass or fail Istio security policies. Useful for enforcing security rules.

3.2 AI MODELS

AI-DPM integrates an AIOps approach with both classical AI models and cutting-edge time series Large Language Models (LLMs). Time series forecasting and anomaly detection are common ML tasks that have recently seen significant advancements with the integration of LLMs. While classical statistical and AI approaches remain essential, combining them with LLMs unlocks enhanced predictive capabilities, providing a benchmarking framework for evaluating model performance. This hybrid implementation extends the capabilities of AI-DPM, enabling the utilization of the most robust and effective tools for monitoring TEADAL's data lake infrastructure and application environment, resulting in more accurate and resilient anomaly detection and forecasting solutions.

Funded by the European Union

Initially, as detailed in the Deliverable D4.2 [8] during the PoC phase, AI-DPM relied mainly on standard statistical methods and classical neural networks for predictive analysis and anomaly detection. In subsequent phases, we expanded the model suite by incorporating LLMs to strengthen results and benchmark traditional techniques. The following sections outline the specific models integrated into the AI-DPM tool.

Statistical and Classical AI Approach

Under this category, statistical models such as AutoRegressive Integrated Moving Average (ARIMA) and Prophet, along with Recurrent Neural Network (RNN) AI models, including Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM), are considered. Each of these has its advantages in specific contexts of forecasting and anomaly detection.

Statistical Models (ARIMA/Prophet)

ARIMA is a classical statistical model used for time series forecasting, particularly when data exhibits patterns such as trends or autocorrelation. It integrates three key components: AutoRegressive (AR), which models the relationship between current and past values; Integrated (I), which applies differencing to render the time series stationary; and Moving Average (MA), which considers past forecast errors. ARIMA does not require normalization of data and is effective for univariate forecasting tasks. Its strength lies in its simplicity, interpretability, and robust performance on well-behaved time series with linear patterns. More importantly, in the context of TEADAL, they generally demand fewer computational resources.

Prophet is an open-source time series forecasting tool developed by Facebook, designed for handling data with strong seasonal effects and historical trends. It is based on an additive model where components like trend, seasonality, and holidays are modelled separately and combined. Prophet is user-friendly, robust to missing data, and handles outliers well. It doesn't require extensive data preprocessing or normalization, making it ideal for business forecasting tasks. Its ability to incorporate domain knowledge through custom seasonality and event effects makes it highly flexible and interpretable.

RNN Models (GRU/LSTM). GRU and **LSTM** are advanced types of Recurrent Neural Networks (RNNs) widely used in time series forecasting and anomaly detection. Their architecture is designed to capture long-term dependencies in sequential data, making them ideal for modelling complex temporal patterns like system performance trends or usage fluctuations. In forecasting, they predict future values based on historical sequences, while in anomaly detection, they help identify deviations from learned patterns. Both models use gating mechanisms to control information flow—LSTM has separate input, output, and forget gates, while GRU simplifies this with update and reset gates—resulting in efficient learning. Their ability to handle noise, missing data, and non-linear dynamics makes them powerful tools for AI-driven monitoring systems like those in the TEADAL project.

Large Language Models (LLMs)

Large Language Models (LLMs), though originally built for text tasks, are increasingly being explored for time-series data analysis. By framing time-series problems as language modelling tasks, such as treating sequences of numerical values as tokens or generating textual descriptions of trends, LLMs can be adapted for forecasting, anomaly detection, and data summarization. We assessed various time-series-focused LLMs, e.g., TinyTimeMixer (TTM), LSTM-GPT, TimesFM. Our assessment focused on several parameters, including whether the LLMs are open or commercial. Eventually, we have integrated both the cloud-based and the on-premises LLMs, specifically, TimeGPT and Lag-Llama, alongside the classical AI models, for prediction and for anomaly detection.

Funded by the European Union

Lag-Llama/TimeGPT

TimeGPT, a ***cloud-based LLM*** tailored for time series tasks, introduces a new level of abstraction in predictive modelling for TEADAL. Unlike traditional models, it requires no local training or data preprocessing. Forecasts are generated through direct API calls after API key setup, enabling rapid deployment and scalability. TimeGPT exemplifies the integration of LLMs into classical AI pipelines, offering powerful predictive capabilities with minimal overhead.

Lag-Llama is a ***local LLM*** designed for time series forecasting. It uses a transformer-based architecture to model sequences of lagged inputs and generate future values. Forecasting is treated as a sequence completion task. Lag-Llama supports multivariate time series and can be used for both forecasting and anomaly detection. It runs locally, without requiring API calls or cloud services, unlike models such as TimeGPT.

3.3 ARCHITECTURE AND INTEGRATION OVERVIEW

The AI-DPM system has been designed upon the monitoring ecosystem of TEADAL nodes deployed on Kubernetes using open-source toolsets to ensure wide-ranging observability, alerting, and visualisation capabilities. This ecosystem primarily consists of Prometheus, Istio, and Kepler. Building upon this, the AI-DPM component consists of Thanos, AI algorithms, APIs, and GUI for experimentation and visualization. These elements of AI-DPM are part of the general AI-DPM architectural layers as shown in Figure 15 viz. Data Aggregation and Processing Layer (Thanos), AI Analytics Layer (AI algorithms), and the Serving Layers (APIs and UI). A complete description of the AI-DPM architecture can be referred from deliverable D2.4 [4].

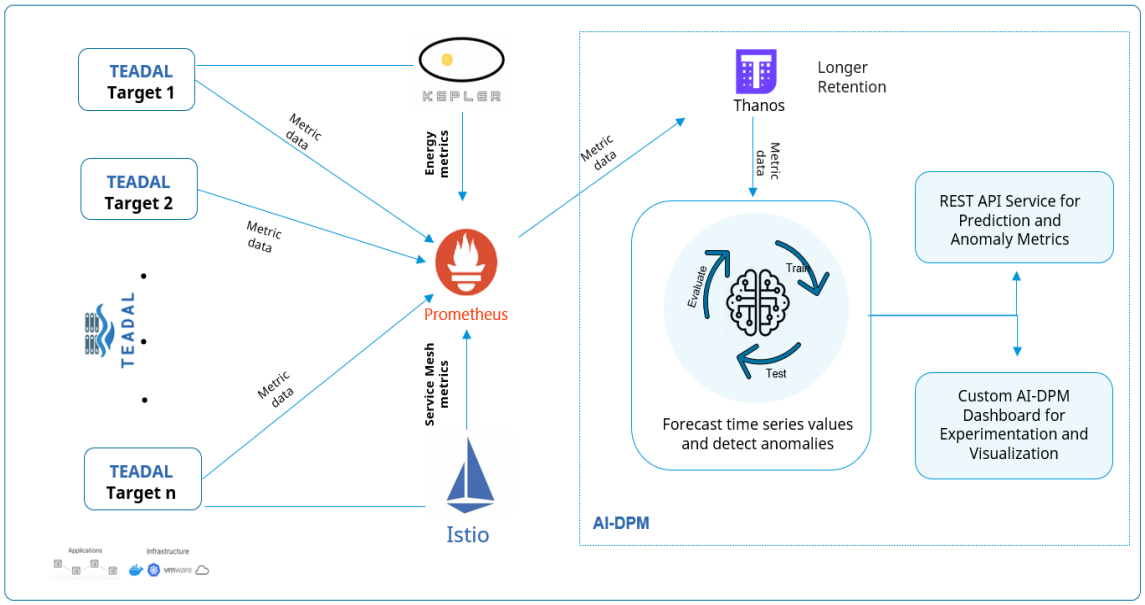

FIGURE 15 AI-DPM APPLICATION ARCHITECTURE (COMPONENTS) DIAGRAM

Starting from the observability and monitoring services of the TEADAL infrastructure and application environment, the metrics for AI-DPM flow through multi-layered architectural components:

- **Monitoring/Data Collection Layer**
  The AI-DPM data collection layer uses the TEADAL monitoring tools stack, such as Prometheus (for infrastructure metrics), Istio (for service mesh observability), and

Funded by the European Union

Kepler (for energy monitoring). This layer continuously provides historical time series metrics metadata regarding system resource usage, network traffic, and energy consumption. This layer is an integral element of TEADAL's system observability and monitoring services.

- **Data Aggregation and Processing Layer**

  This layer functions as a time series database. After data is collected through the data collection layer, it is adequately aggregated and persistently stored for future analytical processes. Here, the data is processed with Thanos, which provides long-term storage and scalability, ensuring that historical data is easily accessible for anomaly detection and predictive analysis.

- **AI Analytics Layer**

  This layer serves as the AI/ML engine of AI-DPM, where processed data is analysed using machine learning models and algorithms following the standard ML lifecycle (train-test-evaluate) to detect anomalies and predict trends. It incorporates classical ML models, statistical methods, and large language models (LLMs).

- **AI-DPM Serving Layer**

  This final layer allows sharing AI-DPM outputs to be served as REST APIs to be consumed. The API service features four main operations: (i) fetching historical data, (ii) model training on historical data, (iii) inference to generate predictions using trained models, and (iv) detecting anomalies. The */fetch* endpoint is used to retrieve data necessary for processing. The API endpoint for training the selected model on historical data is */train*, while generating predictions using the trained model is done by calling the */infer* endpoint. The */anomaly* endpoint serves for anomaly detection.

- **AI-DPM Dashboard**

  The AI-DPM Dashboard is a dedicated experimentation interface designed to clearly show the functionalities of the AI-DPM tool that support the basic AI workflow, including training, testing, and evaluation of machine learning models for time-series forecasting and anomaly detection. It integrates the multiple model implementations of the AI-DPM tool with configurable parameters, allowing users to execute training workflows, assess model performance, and compare results across different configurations. It enables quick comparison of AI-DPM multi-model performances and helps users choose the most suitable AI model for their specific context

3.4 EXPERIMENTS AND RESULTS

The experimental results from the PoC, as reported in Deliverable D4.2 [8], provided a foundational basis for the development of AI-DPM. However, the PoC analysis relied on public datasets sourced from systems that slightly differ from the TEADAL environment. While the dataset was relatively large, it was only a five-day data of monitoring, and the microservices used in the Kubernetes cluster were less comparable to those in TEADAL. To address these gaps of PoC, we implemented an incremental strategy of gathering metadata from systems that closely resemble the ***TEADAL pilots*** and utilizing tools like ***µ-bench*** for consistent microservice simulation. We set up two VM in our lab and implemented a 2 TEADAL node pilot-like scenario. Additionally, the integration of more advanced techniques, such as ***time-series LLMs***, was used to enhance forecasting and anomaly detection.

We present here the results of the AI-DPM tool. Monitoring metadata was collected from ***VM-nodes that closely mirror the TEADAL pilot setup***, and ***µ-bench*** was employed to simulate

Funded by the European Union

realistic microservice behaviour. Furthermore, both local and cloud-based time-series LLMs were incorporated into the AI-DPM algorithm stack to strengthen its analytical capabilities. At the time of writing this deliverable, AI-DPM has been deployed in two TEADAL Pilots, the Mobility and the Regional Planning. Additional deployments are planned during the integration and validation work.

Resource Usage Metadata Results

Resource usage metadata from Prometheus provides critical system-level metrics, including CPU, memory, and disk usage. For example, the results of memory resource use (*node_memory_MemAvailable_bytes*) metrics prediction using all six models available in AI-DPM, along with the anomalies in the pattern of available memory, are shown in the two figures below, Figure 16 and Figure 17. This metric indicates the amount of memory available for new applications without resorting to swapping. In the plot, the historical data (blue line) reveals a baseline of available memory hovering between 28.6 and 29.0 GB, with several significant spikes reaching up to 29.7 GB. The forecasting models demonstrate various prediction approaches for future memory consumption:

- ARIMA (green dashed line) predicts a gradual increase, reaching the highest forecast around 29.4 GB.
- Prophet (red dashed line) anticipates a moderate increase with slight fluctuations.
- GRU and LSTM (blue and orange dashed lines) forecast more conservative growth with tighter oscillations.
- LagLlama and TimeGPT (brown and purple dashed lines) predict declining memory usage, with TimeGPT showing the most pessimistic forecast, dropping below 28.8 GB.

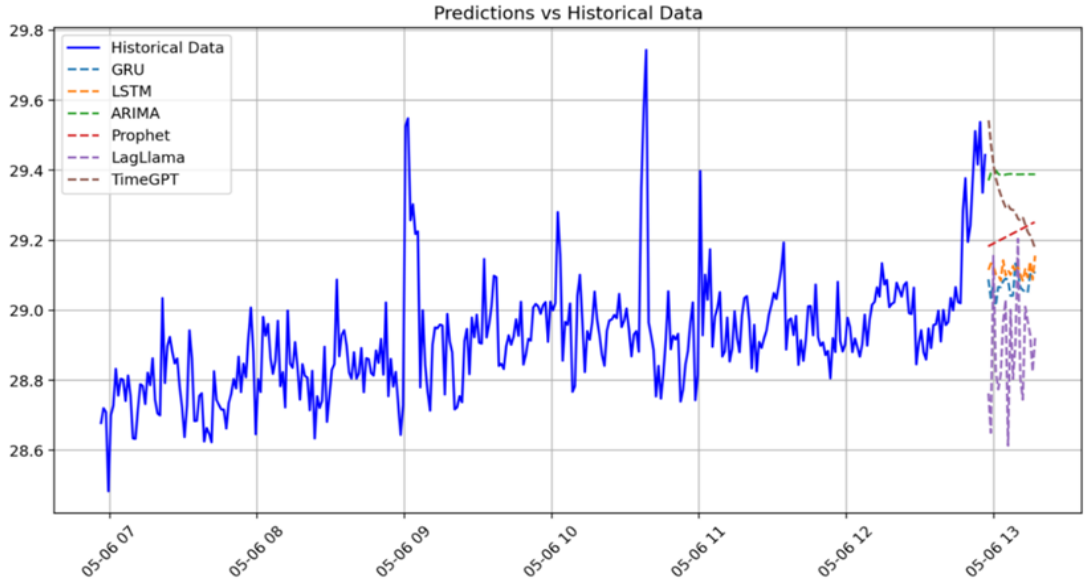

FIGURE 16 PREDICTIVE INSIGHT FOR MEMORY AVAILABILITY ON THE NODE

In terms of actionable insights, ***node_memory_MemAvailable_bytes*** offers valuable insight into the actual memory available for applications without resorting to swap. When this value remains consistently low, it indicates memory pressure and may signal an impending out-of-memory event, necessitating actions to offload tasks or auto scale to prevent performance degradation. Conversely, if memory availability is consistently high, it suggests over-provisioning, requiring a safe reduction in allocated memory resources to save energy. Additionally, by analysing trends over longer periods during low-demand phases, memory can

Funded by the European Union

be transitioned into lower power states, optimizing energy usage without impacting system responsiveness.

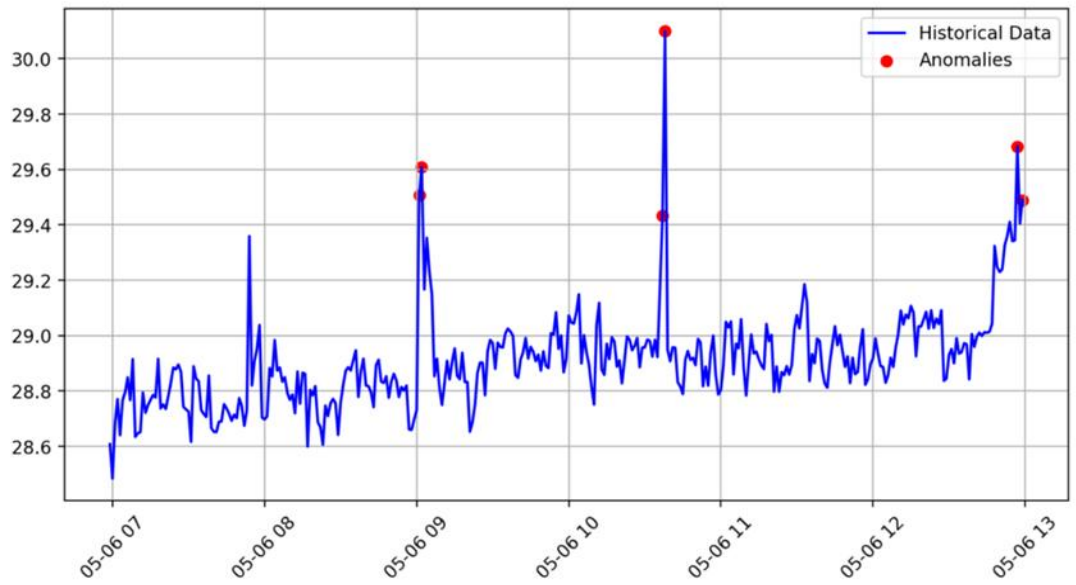

FIGURE 17 ANOMALY DETECTION FOR MEMORY AVAILABILITY ON THE NODE

The anomaly detection model implemented in AI-DPM has identified critical memory availability fluctuations that require attention to prevent potential system degradation. These anomalies represent statistically significant deviations from normal memory consumption patterns, likely indicating memory leaks, unexpected workloads, or system issues requiring investigation. The identified anomalies exclusively show memory availability increases rather than decreases, which counterintuitively may indicate problematic behaviour - applications terminating unexpectedly, service restarts, or major processes releasing memory abnormally. This analysis transforms what might appear as positive anomalies (more available memory) into actionable insights about potential application instability, allowing for proactive system reliability improvements.

Energy Sustainability Metadata

The energy metadata metrics are exposed and exported by Kepler. Kepler uses eBPF (extended Berkeley Packet Filter) to collect energy-related system stats and export them as Prometheus metrics. Several Kepler metrics provide granular power consumption data for Kubernetes pods, nodes, and containers. As an example, the Kepler Total power consumption per node (Watt) measures the total electrical power used by a compute node, recorded in watts. This includes power drawn by the CPU, memory, storage, and other components.

The predictive models demonstrate divergent forecasts for future power consumption. Traditional statistical approaches (ARIMA and Prophet, shown in green and red dashed lines) predict sustained elevated power usage at approximately 0.65-0.7 watts, suggesting the recent high-consumption pattern may continue. Meanwhile, neural network models (GRU and LSTM, in blue and orange) forecast a return to more moderate levels around 0.4-0.45 watts, indicating the spike may be transitory as shown in Figure 18. This predictive analysis enables proactive power management and can inform energy optimization strategies as part of sustainability initiatives, allowing operations teams to schedule workloads during periods of anticipated lower consumption or implement dynamic throttling during projected high-usage windows.

Funded by the European Union

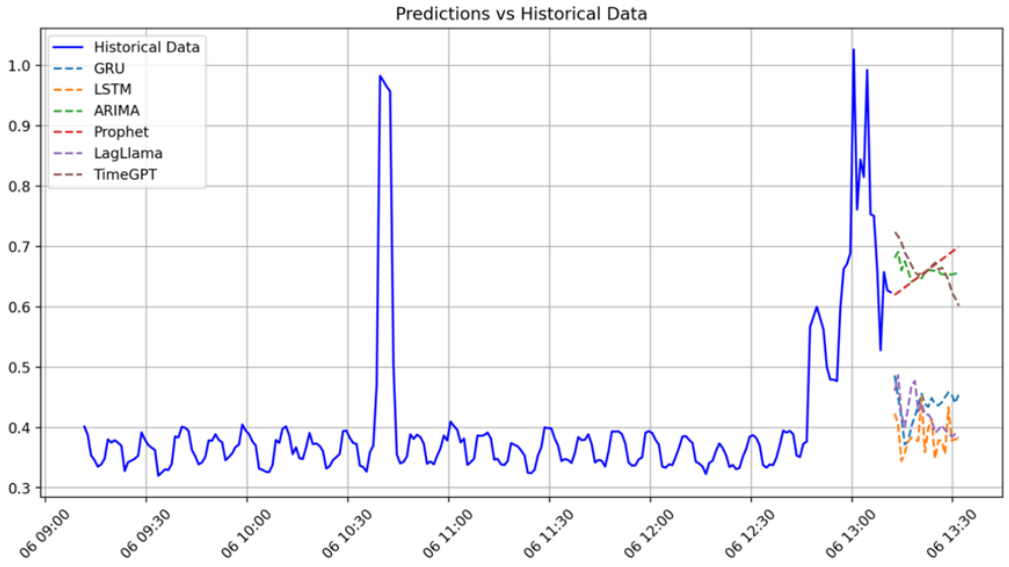

FIGURE 18 PREDICTIVE INSIGHT FOR POWER CONSUMPTION PER NODE

As shown in the time-series plot, node power consumption follows distinct daily patterns with baseline usage fluctuating between 0.35-0.4 watts during normal operations. The data reveals two significant power consumption spikes as shown in Figure 19. The two spikes were identified in the anomaly detection analysis as anomalous power consumption patterns that need further attention and monitoring.

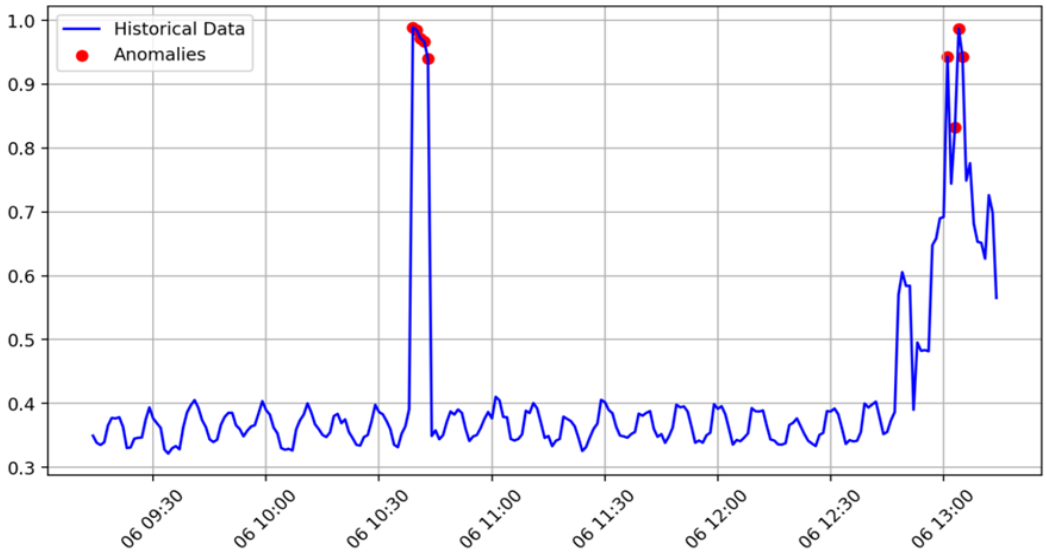

FIGURE 19 ANOMALY DETECTION FOR POWER CONSUMPTION PER NODE

Infrastructure and Network Metadata

The infrastructure and network metadata are metrics that help monitor service behaviour and are generated by Istio for all service traffic in, out, and within an Istio service mesh. These metrics provide useful information such as volume of traffic, error rates within traffic and the response times for requests. As an example, the results of different prediction models for the node network bandwidth metrics presented in Figure 20 and the anomaly detection results for the same metrics presented in Figure 21, reveal distinct usage patterns with predictable baseline traffic and recurring spike events. Our predictive modelling indicates that the network

Funded by the European Union

consistently maintains a baseline bandwidth of approximately 0.185 MB/s per node with regular spikes reaching 0.21-0.23 MB/s. More significant anomalous spikes occasionally reach 0.26 MB/s. This pattern suggests that scheduled processes or periodic system activities are driving network utilization cycles.

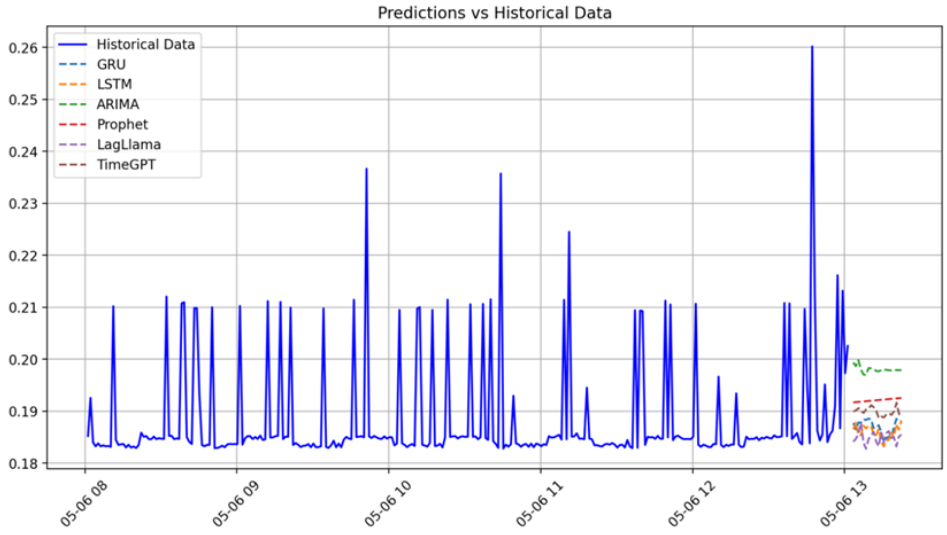

FIGURE 20 PREDICTIVE INSIGHT FOR ISTIO TRAFFIC PER NODE

The analysis identified four major bandwidth anomalies and one exceptional spike that significantly deviated from normal patterns. These represent opportunities for system optimization or potential issues requiring remediation. Among possible actions, to review application architecture to reduce inter-node dependencies and/or evaluate microservice deployment patterns to optimize network traffic flow. By implementing these recommendations, IT operations can expect more efficient resource utilization, reduced anomaly response times, and improved infrastructure stability, ultimately supporting better application performance and user experience.

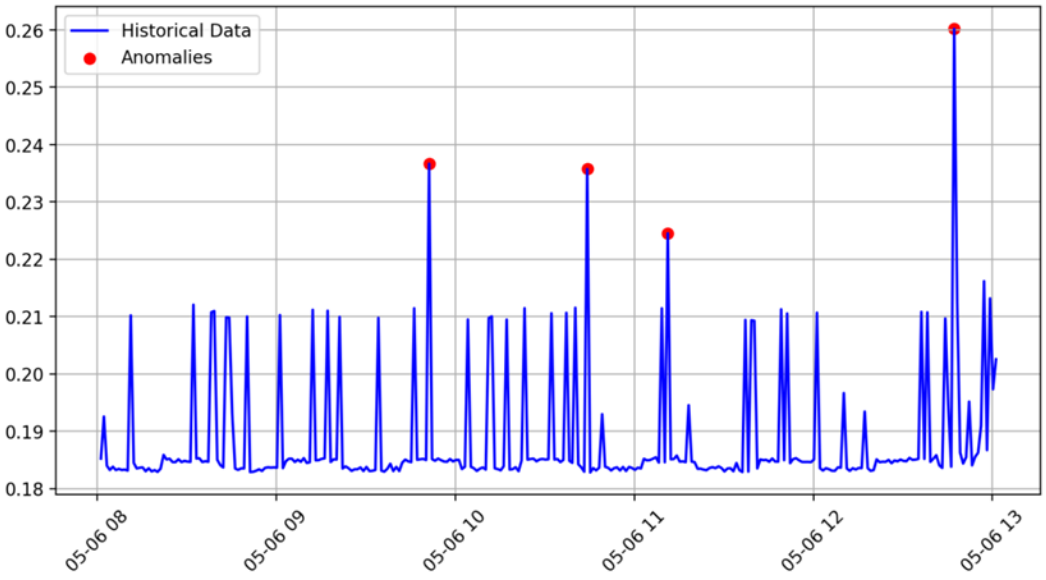

FIGURE 21 ANOMALY DETECTION FOR ISTIO NETWORK TRAFFIC PER NODE

Funded by the European Union

3.5 AI-DPM OUTPUTS

The AI-DPM outputs are served as REST APIs, and through an interactive experimentation and visualization GUI.

Rest APIs

AI-DPM is developed as a REST API service that provides time series forecasting capabilities using multiple models and integrates with persistently stored metrics in Thanos. The service supports multiple models, including a cloud API for TimeGPT LLM (requires tokens) and other local models, ranging from RNNs to classical statistical models and LLMs. In addition, the service offers a configurable training window and flexible data sourcing from Thanos via Prometheus Query Language (PromQL), a functional query language for selecting and aggregating time series data in real time.

The API service features four main operations (service endpoints):

1. **Fetching Historical Data (/fetch)**: This endpoint retrieves time series data based on a specified Prometheus query and time range (in hours). The request requires the query and duration and, optionally, accepts a Thanos URL. The response includes the historical data, which is essential for model training and evaluation.

2. **Model Training (/train)**: This endpoint trains forecasting models—either local (GRU, LSTM, ARIMA, Prophet) or LLMs (Lag-Llama)—using specified parameters like query, training duration, input/output steps, and model type. Local models are saved under the models/ directory for reuse. The API responds with a message indicating successful training.

3. **Inference (/infer)**: This endpoint generates predictions using a previously trained model. It requires the same parameters as training—query, time range, input/output steps, and model name. All the models, including classical as well as local and cloud-based LLMs are supported. The output is a list of future time-stamped predictions.

4. **Anomaly Detection (/anomaly)**: This endpoint identifies anomalies in time series data using the provided query and detection method. You can specify the confidence interval and detection duration. It returns a list of time-stamped values flagged as anomalies, helping detect unusual behaviour in monitored metrics.

5. **Model Evaluation (/compute_rmse):** This endpoint calculates the Root Mean Square Error (RMSE) across multiple predictive models. It returns a number for each model, with lower values indicating better performance. This metric helps to identify top-performing models, supports model comparison, enables ensemble decisions, and assists in model selection.

Collectively, these endpoints enable a complete AI-DPM workflow: from data retrieval and model training to predictive insights and anomaly monitoring. The detailed parameter examples and schema of endpoints are provided in the Swagger file.

AI-DPM Dashboard

The AI-DPM Dashboard is a dedicated experimentation interface designed to clearly show the functionalities of the AI-DPM tool that support the basic AI workflow, including training, testing, and evaluation of machine learning models for time-series forecasting and anomaly detection. It integrates the multiple model implementations of the AI-DPM tool with configurable parameters, allowing users to execute training workflows, assess model performance, and

Funded by the European Union

compare results across different configurations. It enables quick comparison of AI-DPM multi-model performances and helps users choose the most suitable AI model for their specific context.

Dashboard Overview

The AI-DPM service dashboard, presented in Figure 22, is organized into two primary sections: the *Global Configuration* panel on the left for setting global parameters, and the *Monitoring & ML Dashboard Panel* on the right, for executing specific workflows such as data retrieval, model training, and performance evaluation.

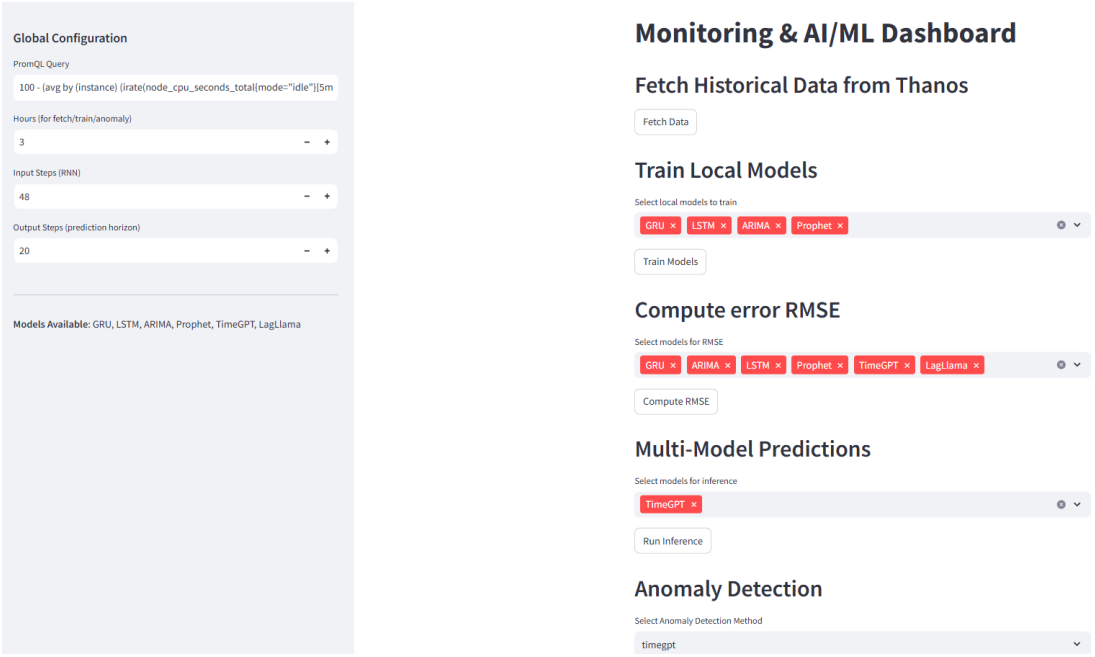

FIGURE 22 AI-DPM SERVICE DASHBOARD

Global Configuration Panel

The *PromQL Query* field allows to define the data source, defaulting to ***CPU idle metrics*** but customizable for all the timeseries metadata relevant for forecasting needs and gathered from *Prometheus*, *Kepler*, and *Istio*. Specifying the metrics to modify this query requires properly formatted and aggregated timeseries data suitable for the models available in the dashboard.

- The ***Hours for fetch/train/anomaly*** setting controls the historical time window used for analysis. The default 3-hour setting is appropriate for short-term patterns, but it can be adjusted using the +/- buttons. Consider shorter windows (1-6 hours) for immediate patterns, medium windows (12-48 hours) for daily patterns, and longer windows (72+ hours) for weekly patterns, keeping in mind that longer windows require more computational resources.

- ***Input Steps (RNN)*** determines how many previous points (in time) the model considers when making predictions. The default 48 steps can be modified as needed. This parameter significantly impacts model behaviour—larger values help capture long-term dependencies but increase computational demands, while smaller values process more efficiently but might miss extended patterns.

- ***Output Steps*** sets the prediction horizon—how far into the future the models will forecast. The default 20 steps can be adjusted according to requirements. Generally,

Funded by the European Union

prediction accuracy decreases as the horizon extends, so balancing is needed for longer-term forecasting against accuracy expectations.

The models available section simply displays all the available models that can be selected for experimenting: GRU, LSTM, ARIMA, Prophet, TimeGPT, and LagLlama.

Operations Panel

Located on the right side of the dashboard under the “Monitoring & AI/ML Dashboard” header, the Operational Panel consists of four sections:

Fetch Historical Data from Thanos: **this** section allows the retrieval of time-series data based on your configured query and time window. It includes a "*Fetch Data*" button that retrieves time series data from the Thanos backend. Before initiating any training, the first step is to ensure that the query is correctly set up.

Train Local Models: this section provides options to select the Model To use from various forecasting models: Statistical Models (ARIMA/Prophet), RNN Models (GRU/LSTM), and local LLM (Lag-Llama), along with a "*Train Models*" button to initiate model training.

Compute error RMSE: in this section, users can evaluate all the local models trained earlier in addition to the on-cloud LLM TimeGPT model performance of predictive models by selecting models such as GRU, LSTM, ARIMA, Prophet, TimeGPT, and LagLlama, and then clicking the "Compute RMSE" button to compute the Root Mean Square Error. This integrated setup enables users to systematically experiment with different models and configurations, making it a powerful tool for time series predictive monitoring analysis and model performance comparison

Anomaly Detection: this section enables users to identify unusual patterns in the selected metrics using two statistical methods: rolling z-score and prediction interval-based detection. Users can select a model and a metric, then visualize anomalies as highlighted regions overlaid on time series plots. This helps in quickly spotting deviations from expected behaviour based on model forecasts and statistical thresholds.

In the operational panel, in addition to fetching historical data for training, training local models, and evaluating their performance, prediction results, prediction, and anomaly plots are visualized. A tabular view of the predicted values with corresponding timestamps for each model is also made available through this panel.

Overall, the AI-DMP Dashboard is part of the distribution of the AI-DPM tool. It comes together with the AI-DPM API. Its usage is intuitive and can enable the leverage of all the functionalities provided by the API. We believe that it can be useful mainly for users interested in experimenting with the array of AI models available in AI-DPM. Configure training parameters, practice all the ML cycle to choose the Models that are more suitable for their needs and contexts. Eventually, evaluate the performance of the models and make an informed decision based on the appropriate monitoring metrics.

Funded by the European Union

4. THE AUTOMATION SUBSYSTEM (ASG)

The automation subsystem, referred to as Automated SFDP Generation (ASG), is a key component of the TEADAL Stretched Data Lakes architecture and its control plane. It operationalizes the SFDP-based data sharing model proposed by TEADAL, transforming high-level sharing intents into uniform, policy-compliant, and deployable data services. As part of the TEADAL platform, ASG realizes:

- A developer-facing service for producing standardized SFDPs.
- A controller for SFDP execution integrated into TEADAL GitOps, Catalogue-driven data product and agreements management, and policy injection and enforcement.

Innovation highlights:

- Declarative code generation using advanced generative AI technologies
- Unified runtime environment abstracting away low-level data access, processing and caching
- Transforms library supporting custom data transformations, including dynamic injection of their specific implementations
- Support for including deployment annotations (e.g., resource needs, TEE/GPU targeting)
- Support for runtime monitoring and observability via AI-backed analytics

Comparing with the alternatives:

- Manual SFDP creation would require lots of developer skill and effort, would be more time consuming and would result in inconsistent results and lack of code sharing between the SFDPs.
- While in principle OpenAPI tools5 could be used to generate and to parse SDFP specs as well as to generate the SFDP app code, this could result in fragile and opaque code, possibly lacking semantic awareness of TEADAL specifics.

This section provides the high-level design of the ASG subsystem, starting with the overview of its technology choices and dependencies in (ref), the componentization and the high-level design in (ref), and the implementation details in (ref).

4.1 THE ASG DEPENDENCIES AND TECHNOLOGY CHOICES

Manually authoring specifications and implementations for SFDPs would require users to understand REST semantics, data transformation logic, pagination schemes, input/output mappings, and more, making the creation process both time-consuming and error-prone, especially for non-expert users.

We notice that generative AI, particularly large language models (LLMs), can substantially lower this barrier by translating concise user intents expressed in natural language or guided prompts into valid, executable specs. LLMs trained on software patterns and API schemas can infer structure, resolve ambiguous terms, suggest transformation pipelines, and pre-fill

[5] OpenAPI.Tools - an Open Source list of great tools for OpenAPI.

Funded by the European Union

boilerplate based on partial context, acting as intelligent intermediaries across the data sharing lifecycle and helping to:

- Discover relevant APIs and datasets by interpreting user goals,
- Draft proxy or transform specifications from minimal input,
- Summarize or verify compliance with data access policies,
- Auto-generate test scenarios and documentation for shared data services.

In federated or multi-organizational settings, such as in TEADAL, this capability can promote interoperability and speed, reducing manual coordination and accelerating the creation of reproducible, governed data pipelines.

The GIN Library

As was already presented in D4.2 [8], SFDP generation relies on the LLM-driven GIN library, developed in IBM Research and becoming widely used internally both for research and for contributing to the upcoming data management products. Without repeating the already reported GIN library details, we present the main exported constructs of GIN library relevant to this document, for completeness:

- **GIN Connector Specification** – a set of pydantic models that cover internalization of every aspect of OpenAPI specification[6] and a module that parses the standard OpenAPI specification files to fill in parts of GIN Connector Spec.
- **GIN Spec Generator** – a module that generates the GIN Connector Specification for a given request.
- **GIN Spec Parser** – a module that parses the GIN Connector Specification embedded into the generated SFDPs, producing the list of endpoints to fetch from the origin FDP and the list of transforms to apply to the fetched dataset to produce the result expected by the SFDP spec.
- **GIN Spec Executor** – a module that takes in the parsed Connector Specification and executes it by 1. fetching the origin data and 2. loading and applying the required data transformations.

GIN Connector Specification module is central to the GIN library and is included in all the other modules. For completeness, we provide a full description of the ConnectorSpec model below. GIN Spec Generator module is closed source and is provided to the TEADAL project team as a dependency, in the form of a python package. GIN Spec Parser and GIN Executor are open and used as a basis for the TEADAL ASG-runtime library, also described below. Since its first presentation in D4.2 [8], the library has been adapted, specifically for the TEADAL use case, as will be described in what follows.

Gin ConnectorSpec

The ConnectorSpec model defines how to construct a selective proxy REST API based on an existing OpenAPI-defined data-serving backend. It allows specific endpoints to be re-exposed with argument mappings, runtime data transformations, and output reshaping. The ConnectorSpec model is used to configure:

- Which backend endpoints are exposed

[6] OpenAPI Specification v3.1.1

Funded by the European Union

- How parameters are passed or generated
- How to transform responses into structured datasets
- What transformations are applied before data is returned

Figure 23 presents a simplified and annotated ConnectorSpec schema summary. The schema is very much aligned with the OpenAPI specification and affords the following key capabilities:

- Selective Proxying: only whitelisted endpoints and only whitelisted data elements are exposed
- Flexible Input Mapping: inputs can come from constants, runtime inputs, or references to other calls/envs
- Data Transformation: output fields can be constructed from one or more input fields applying transformation functions
- Pagination Support: common pagination strategies are supported, including cursor and page-based
- Structured Output: outputs are mapped to named datasets, enabling consistent consumption downstream

```
apiVersion: "connector/v1"        # Fixed API version string
kind: "connector/v1"              # Model kind/type
metadata:                         # Basic descriptive info
  name: string                    # Human-readable name
  description: string             # Description of the connector
  inputPrompt: string?            # Optional natural language prompt

spec:                             # Main specification block
  apiCalls:                       # Dict of API call configurations
    <call_name>:
      type: "url"                 # Call type (currently only URL)
      endpoint: string            # Path of the backend endpoint
      method: "get"|"post"|...    # HTTP method
      arguments:                  # Optional list of input arguments
        - name: string
          argLocation:            # "parameter", "header", or "data"
          type: ArgTypeEnum       # Data type (string, int, object, etc.)
          source:                 # "constant", "runtime", "reference"
          value: any              # Argument value or reference
      pagination:                 # Optional pagination config
        type:                     # Pagination type (page, cursor, etc.)
        nextPath: string?         # Path to next page URL
        paginationParams:         # Dict of param names to value paths
        paramTranslation:         # Optional mapping of paging param keys
          pageRef: string
          pageSizePath: string
          totalSizePath: string

  output:                         # Describes how to construct outputs
    data:                         # Maps named outputs to API call responses
      <name>:
```

FIGURE 23 GIN CONNECTOR SCHEMA

Funded by the European Union

The structure of ConnectorSpec somewhat follows that of the OpenAPI specification and, while powerful and flexible, is also inherently verbose and complex. GIN Spec Generator module employs LLM calls to dynamically generate these specs provided with guided prompts. For example, given a prompt like *"Expose the /employees endpoint with a filter by role and only return name and email"*, an LLM can scaffold the relevant ApiCall object, define the necessary Arguments, and propose output field transformations, all aligned with the ConnectorSpec schema shown in Figure 23.

Ollama Service

To avoid reliance on expensive or intermittently available cloud-based LLM services, we focused on enabling offline-compatible generation of SFDPs. For this, we first evaluated a number of technology candidates available for the local model and inference serving and have selected Ollama among all the considered options. Ollama stood out due to its ease of use, cross-platform support, and built-in model management. However, our decision was informed by a broader review of local inference options, summarized in Table 4.

TABLE 4: COMPARISON OF THE LOCAL INFERENCE TOOLS AND FRAMEWORKS

Framework / Tool (with link)	Community & Ecosystem	License	Notes
Ollama https://ollama.com	Growing, GitHub activity	MIT	+ Easy to use; Docker support; fast to prototype; many models available
LM Studio https://lmstudio.ai	Small, mainly desktop users	Unknown	- GUI-focused, not suitable for automation
vLLM link	Active research/dev community	Apache 2.0	+ Excellent performance with batching; scalable; smart GPU management - learning curve, complexity
Text Generation Inference (TGI) link	Strong support from Hugging Face	Apache 2.0	Designed for production inference, full REST API
llama.cpp link	Very active, many wrappers	MIT	+ Lightweight and efficient - CLI or custom server required; needs REST/gRPC wrapper
GPT4All https://gpt4all.io	Moderate, good docs	Apache/MIT	- Better for desktop/offline GUI use
DeepSpeed-MII link	Research-focused	MIT	+ Great for high-performance inference - Non-trivial setup
AutoGPTQ / ExLlama link	Niche but growing	Apache/MIT	+ Optimized for quantized model inference - Still maturing

Each option was evaluated for its suitability in constrained, potentially air-gapped environments, as well as its compatibility with Kubernetes-based deployments. While more performant frameworks like vLLM and TGI offer production-grade scalability, they tend to have a steeper learning curve and more infrastructure requirements, e.g. advanced GPUs, etc. Ollama offers a pragmatic middle ground, with a strong local-first philosophy, simple deployment with Docker, and an evolving ecosystem of compatible models. In addition, Ollama

Funded by the European Union

community has proven to be one of the first to adopt the codebase to the latest feature additions of the OpenAI such as functions calling[7] and structured outputs[8].

Then, we have adapted the GIN library to be able to work with Ollama and have selected several models available in the community and capable of delivering the inference results required by the GIN library. Namely, GIN library, which in-house uses the advanced *granite-code-instruct* models family, relies on model's ability to support tool calling and structured output. In TEADAL, we have adapted the compatible granite-code models family available in the Ollama Models Library[9].

In addition, we have integrated the ollama service as part of the TEADAL Node, as an app-level-service that can be enabled on any TEADAL Node. If enabled, the service can be also used for other inference jobs, both by additional TEADAL services and, in the future, by the data analytics workflow put together by federation partners and users.

Data Transformations Library

The data transformations library is, architecturally, one of the major building blocks of the FDP-to-SFDP pipelines.

As part of the ASG subsystem, the transforms library is one of the links between the SFDP generator and the runtime execution of the generated SFDP. First, the generator 'understands' the user-provided specification of how SFDP should be derived from the FDP, 'reads' the descriptions of all the available transformations, and creates a plan of what transformations need to be applied and in what order. The generator then produces the spec (referred to as Gin Connector Specification) that is used at SFDP execution time to invoke the methods from the same library of transformations. So, for the generation step we basically need to only have the list of transforms with their descriptions, understandable by both the humans and by the LLMs, while at runtime we need to have the methods themselves ready to be loaded and executed. This link is facilitated by making the same transforms library available, both at the generation time and at the runtime.

In a production system, we envision the transforms library to be a standalone component responsible for the full lifecycle of the transforms and their implementations. Such component, deployed separately, would be accessed by the ASG subsystem, both at SFDP generation and at the SFDP execution. In addition to storing the transforms and providing access to them to the ASG actors, the library component will take care of ingesting the transforms upon their creation and introduction to the system, validating them from a perspective of correctness, security, etc., collecting their runtime performance data, and decommissioning them when they are no longer required. Such a system could be also integrated with providers and consumers of transformation implementations through protocols such as MCP. Creating a full-featured system like this is certainly beyond project resources and schedule but its very conception is one of the valuable project outcomes.

For our prototyping and experimentation, we have created the simplest possible transforms library as a collection of python functions that can be applied to the FDP data at runtime to produce the results expected by the SFDP. Some of these functions are generic data manipulation functions like filter, rename, slice, etc., and are built-in to the system. Some

[7] https://platform.openai.com/docs/guides/function-calling

[8] https://platform.openai.com/docs/guides/structured-outputs

[9] https://ollama.com/library

Funded by the European Union

functions can be added for specific FDP-SFDP pairs to implement the required domain specific functionality. Some other functions can be imposed by system operators to implement infrastructure-level functionalities such as, for example, compression and caching. Yet some others can even be dynamically adjusted to select the best possible implementation based on current system state, resources utilization etc. For example, there can be several implementations of the anonymization function, each requiring different types and amounts of runtime resources. In this and other similar cases, the SFDP generator will list the generic anonymization step as part of transformations pipeline and the runtime system, based on where the service is deployed and what resources are available, will select the suitable implementation and inject it to the library instead of the generic anonymization placeholder. In general, this will be done based not only on resource availability but also on additional parameters the system is constrained by, e.g. the considerations of data friction, data gravity, and energy consumption (as modelled and implemented in WP3). In our prototype, the library is delivered as a folder with python files containing all the transform functions in the library. To differentiate general-purpose Python functions (e.g., helpers) from transformation-specific functions, GIN imposes the requirement to decorate the transforms with a special *make_tool* decorator that will help picking only the transform functions, both at generation and at execution. In our implementation, the functions are loaded dynamically while serving the SFDP data endpoints which supports injecting specific realizations of certain functions at runtime as explained above.

4.2 THE ASG HIGH LEVEL DESIGN

ASG is designed to realise the uniform approach to the SFDP generation and execution outlined in the Introduction. Overall, the system pursues and realises the following design goals:

- Minimize developer effort required to stand up the SFDPs
- Ensure consistency across SFDPs via templated scaffolding
- Increase system maintainability by reducing the amount of custom code in the system
- Allow improving resource usage through sharing runtime components such as data caches, transforms library, etc.

ASG achieves its goals through its three major components:

- The ASG-tool – a service for generating the SFDPs using LLM-backed GIN library
- The ASG-SFDP – a thin templated FastAPI server to be deployed on TEADAL infrastructure
- The ASG-runtime – a library that backs-up the execution of the ASG- compliant SFDP servers, implementing all the heavy lifting: parsing the GIN connector Specification included in the SFDP; performing the http access to source FDPs; caching the data (with possibility of sharing among SFDPs deployed in the same environment); applying the transformation pipeline; error handling, etc., up to providing data to the SFDP endpoints at runtime.

Funded by the European Union

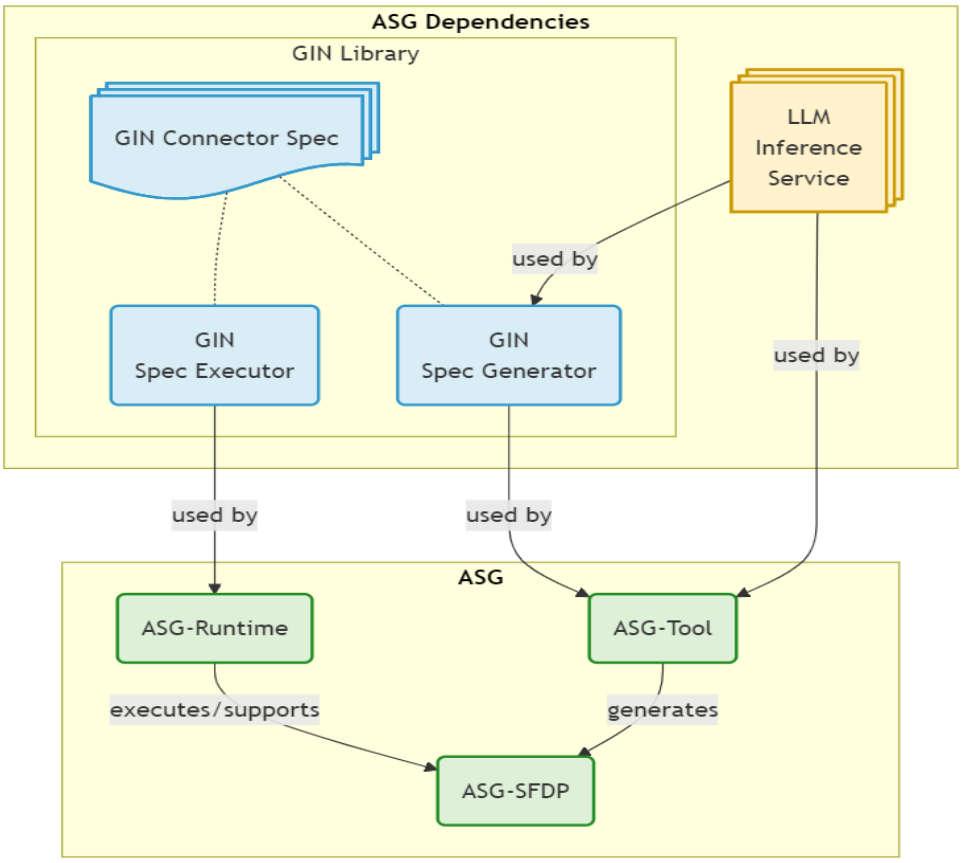

FIGURE 24 ASG COMPONENTS AND DEPENDENCIES

Figure 24 provides a visual representation of ASG system components, their external dependencies, and the relationships between them. In what follows we describe each of these components in more detail.

ASG-tool

ASG-tool is a developer-facing command-line tool (that can be exposed as web interface) capable of generating SFDPs, using the following inputs:

- Information about the origin FDP, including its OpenApi Specification document, its active URLs deployed in the TEADAL Federation, as well as additional items required to access the FDP data, e.g. auth keys.
- A minimal input spec defining the origin data source FDP and a set of endpoint configs that prescribe how endpoint's data is derived from the origin FDP data.
- A transformations library containing reusable data manipulation functions to choose from for deriving the SDFP data from the origin FDP data, e.g. built-in functions for reshaping, filtering, or aggregating the datasets.

ASG-tool outputs a working project (or repo) ready for validation and further processing, including:

- A FastAPI implementation of the SFDP server, as app.py file that includes all the data endpoints described in the input specification. For each endpoint, the app includes a generated GIN Connector Specification to be parsed and worked through at runtime.
- All the required boilerplate for standing up and testing the project locally, e.g. the requirements.txt, the README.md, etc., so the developer responsible for the SFDP

Funded by the European Union

can validate the generated SFDP is functioning up to the contract requirements and is ready for onboarding and publishing. This step is necessary to ensure validity of the generated services while reducing the overhead that would be required for creating SFDPs from scratch and ensuring uniformity of the resulting data products. In the future, when the technology matures, it'll be possible to include an automated validation step, to further reduce the need for human intervention.

- All the required boilerplate for creating a deployable image of the generated SFDP and pushing it to the TEADAL image registry, e.g. the Dockerfile and the CI workflow script to be enacted by the GitLab services as soon as the developer pushes the generated SFDP repo to the TEADAL's GitLab. In future production settings, the tool can be easily extended to support git integration, such as creating the remote repository and pushing the generated project artifacts thus triggering the CI workflow that will create and push new service's image.

Once an SFDP is generated, additional TEADAL Platform tools and services are required to prepare it for deployment, e.g., to add k8s resource manifests, policy files, kustomize scripts, etc. Next, the Control Plane of the TEADAL Federation is notified about the ready to be deployed SFDP and acts to select the deployment target and to realize the deployment. Next, the Catalogue is notified to finalize the SFDP creation process by informing the data user about new SFDP availability.

To summarize, the ASG-tool helps creating SFDPs as fully independent, policy-compliant, and version-controlled data services that are deployed like any other data products in the system, while being uniform and predictable by sharing the common template and the common, possibly shared, runtime services (e.g., caches, transform library instances, etc.).

ASG-SFDP

Each generated SFDP app:

- Lives in its own Git repo
- Embeds the generated connector spec and imported transformation logic
- Turned to a deployable image using the GitLab CI
- Deployed as a self-contained FastAPI service with:
    - Clearly defined endpoints
    - Stable response schemas
    - Business logic applied transparently via the ASG-runtime support

Ultimately, as part of the TEADAL platform, these apps are:

- Discoverable through the Catalogue services like other data products (FDPs)
- Deployable to TEADAL Nodes using existing practices the TEADAL Platform supports (k8s, argoCD, GitOps, etc.)
- Auditable thanks to versioned specs and centralized runtime behaviour
- Observable thanks to supporting service endpoints with stats and possibly telemetry postings (not yet implemented at the time of writing)

Figure 25 below presents the in-app documentation page for an example ASG-SFDP, showing the service endpoints, common to all the SFDPs, and the data endpoints, specific for each individual SFDP, whereby the data can be obtained as specified by the contract and realized by the combination of GIN Connector specification, the transform library, the ASG-runtime

Funded by the European Union

library and its configuration prescribing the caching policy, the http client parameters, and more.

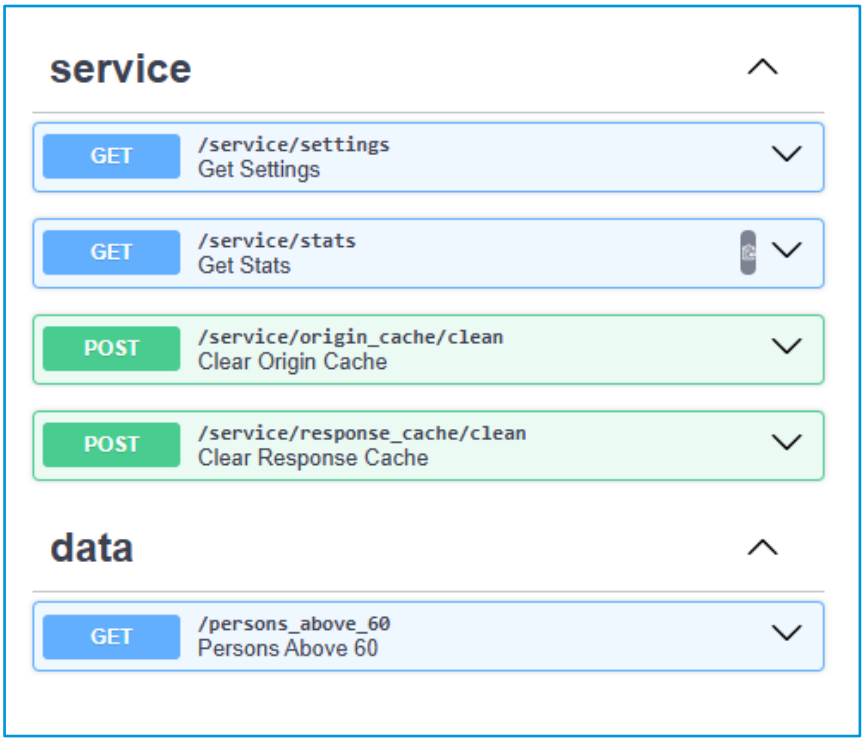

FIGURE 25 EXAMPLE OF SFDP /DOCS VIEW

ASG-runtime

ASG-runtime is a library created to serve as a runtime dependency imported by every generated SFDP, with the following key responsibilities:

- Declarative Configuration, based on Pydantic Settings, including:
  - HTTP client behaviour.
  - Caching strategies for origin and transformed data.
  - Serialization formats (e.g. orjson, pickle, or noop).
  - Logging and observability options.
- Two Level Caching, one for the origin FDP datasets and one for the transformed datasets. Both layers are backend-flexible (support LRU, disk, Redis), serialization pluggable (support pickle and orjson), optional, and allow for triggered purging. Caching is implemented for optimizing the network usage and saving energy related to data transfer and transformations (and addressing project KPIs 3.2 and 3.3).
- Transformation engine, dynamically loading the transforms library functions and executing them as specified in the GIN Connector specification of each endpoint.
- Observability support with built-in logging and statistics collection
  - HTTP statistics
  - Cache and serialization statistics
- Pluggability and Extensibility with modular design allowing introducing:
  - Custom serializers

Funded by the European Union

- Custom cache backends
- Alternate rest client implementations if needed

The library is designed to be modular, versioned, easily testable, and geared toward configurability, observability and fault isolation. The library is packaged as versioned releases to ensure the generated SFDP and the runtime are created using the same version of their common dependencies (the GIN library components such as GIN Connector Spec models and the code for generating and parsing the spec). The packaging allows two ways of usage:

- Pulling the sources or the build release (archive) from git and importing it into the environment where the SFDP will run. This can be useful for local testing by developers.
- Using the base image that includes the ASG-runtime for creating the SFDP image. This way is preferable for the automated image creation as it simplifies and speeds up image creation by eliminating the need to pull and build library sources every time SFDP image is created.

4.3 THE ASG SOFTWARE ARCHITECTURE

In this section we zoom into the lower level of abstraction and present how the ASG components are realised in software. All the code is written in python 3.12 and is available in TEADAL's GitLab.

ASG-tool

ASG-tool is implemented as a simple front end to the GIN connector generating module (GIN Spec Generator), adapted to the TEADAL use case. The additions are:

- a simple, OpenAPI-spec-like specification that describes the SFDP to be generated
- a module that parses this SFDP spec to retrieve the list of data endpoints to be exposed by the generated SFDPs. For each endpoint, the code invokes the GIN generator to create the suitable GIN Connector Specification
- a FastAPI app template (jinja2) that is used as a basis for the resulting SFDP server app by creating the boilerplate (e.g., for the app initialization, hooking into the ASG-runtime, service endpoints, etc.) of the app plus a placeholder for the required data endpoints, each to be filled in with the generated GIN Connector Specification
- generic artifacts to become part of the resulting SFDP project repo, e.g. the README.md, the requirements.txt, the Dockerfile, etc.

For completeness, we briefly present the SFDP Specification Format used by the ASG-tool that contains only one top-level entry, *sfdp_endpoints*, that lists all the data endpoints that will be available in the generated SFDP. Each item in this list represents a specific data endpoint along with info about how it is derived from the data obtained from the source FDP. The general structure of the specification file is presented in Figure 26 and explained below.

Funded by the European Union

```
sfdp_endpoints:
    # the fist endpoint to be generated
    - <endpoint1-name>:
        fdp_path: < path to the endpoint in the source FDP >
        sfpd_path: < path to the endpoint in the generated SFDP >
        sfdp_endpoint_description: < String describing the generated SFDP endpoint >

        schema:
            # what will be returned by the generated endpoint
            <schema-name>:
                type: < object | ? >
                properties:
                    # the first 'column' in the return data
                    <property1-name>:
                        type: < string | integer | number | ? >
                        example: < example value >
                        description: < string describing the property >

                    # more properties can follow

    # more enpoints can follow
```

FIGURE 26 SFDP SPECIFICATION SCHEMA

Each entry under *sfdp_endpoints* represents one data endpoint to be included in the generated SFDP, named as the entry itself. Each such entry is a dictionary with the following elements:

- fdp_path: This key is required and must contain a string representing the path to the corresponding endpoint existing in the source FDP. The string can contain placeholders for dynamic segments (e.g., /stops/stop_id/{stop_id}).
- sfdp_path: This key is required and must contain a string representing the path to the generated SFDP endpoint. It can mirror the placeholders in the source FDP path with similar placeholders (e.g., /stop_id/{stop_id}).
- sfdp_endpoint_description: This element is optional and can contain a string with the description of the generated SFDP endpoint. This is not required by the ASG tool and is used only as part of the OpenAPI specification of the generated SFDP, to help SFDP users by explaining the endpoints available from the generated SFDP.
- schema: This element is required and must contain a dictionary defining the schema for the data to be returned by the data endpoint. This dictionary describes the data properties along with their types and structures, as well as with the well-formed descriptions that will help GIN to derive the required data items from data exposed by the corresponding endpoint of the source FDP. The schema structure is a dictionary named as the data structure it describes, with the following keys:
    - *type*: (String) The type of the data described by the schema. The value is usually an *object* to indicate a structured data object.
    - *properties*: A dictionary that defines the properties of the data described by the schema with type: object. Each property must contain name, type, and description, formatted as specified next. Each data element returned by the generated SFDP endpoint is specified as an entry in the properties dictionary. The entries are named as the data elements they specify and contain the following keys:

Funded by the European Union

- *type*: This key is required and must contain a string defining the data type for the property (e.g., integer, string).
- *example*: This key is optional and can contain an example value for the property. This is not required by the ASG tool and is used only as part of the OpenAPI specification of the generated SFDP, to help SFDP users by illustrating a sample data value that conforms to the property's type.
- *description*: This key is required and must contain a string describing this property. This is used by the GIN generator as a guided prompt for LLM to derive the way this property can be constructed from the data exposed by the source FDP endpoint, namely, to select the right set of transforms for the transforms library to be applied to the origin dataset.

ASG-tool code and documentation complete with installation and usage instructions can be found in the project repository on GitLab[10].

ASG-runtime

ASG-runtime library is architected to ensure:

- Separation of concerns between the minimal templated FastAPI app that handles only the API layer and the bulk of SFDP functionality realised by the ASG-runtime
- Reusability: By putting the main class into your library, you make it easy to reuse across services that differ only in source URLs or transformation logic.
- Maintainability: Keeping cache backends and HTTP boilerplate in separate modules makes the system modular and easy to extend or test independently.
- Flexibility: You can easily swap in new cache backends, plug in new data sources, or change transformation logic with minimal impact to the FastAPI app code.

ASG-runtime code and documentation complete with installation and usage instructions can be found in the project repository on GitLab[11]. For ease of reference, Figure 27 shows module-level repository structure for the ASG-runtime project.

```
├── asg_runtime/
│   ├── executor.py         # Main library logic (Executor class)
│   ├── request_driver.py   # Helper for data request processing (depends on GIN)
│
│   ├── caches/         # Cache modules
│   ├── gin/            # Gin dependencies - GIN Connector spec, Gin Parser, and GIN executor
│   ├── http/           # HTTP client functionality
│   ├── models/         # Shared data models - defines structures for settings, cached, stats, etc.
│   ├── serializers/    # Serialization modules
│   ├── telemetry/      # Telemetry modules
│   ├── utils/          # Shared utilities (e.g., for logging)
```

FIGURE 27 ASG-RUNTIME MODULES

[10] https://gitlab.teadal.ubiwhere.com/teadal-tech/asg_generation_code

[11] https://gitlab.teadal.ubiwhere.com/teadal-tech/asg-runtime

Funded by the European Union

In what follows we briefly describe the most important individual modules that comprise the ASG-runtime library to realize its functionality.

The Settings Module

Settings mechanism is based on Pydantic v2 settings, allowing:

- Per-app customization to disable or enable the caches, to select the caching backend with its related parameters, to control logging, http behaviour, etc.
- Loading from environment variables or .env files
- Validation and the earliest possible error reporting
- Derived properties compute the effective configuration, e.g., when to bypass response cache or use specific serializers
- Easy integration into Kubernetes, via ConfigMaps or Secrets

```
service_name=test sfdp              # user friendly name for the service
transforms_path=test//transforms    # a path to where to load transform functions from

log_level=INFO                      # choice: INFO, DEBUG, etc
logging_flavor=rich                 # chose: rich, plain, jason

# caching
# note: disk cache requires 'pip install -e .[cache-disk]'
# note: redis cache requires 'pip install -e .[cache-redis]' and redis server
# note: only lru cache can work with noop encoder

origin_cache_enabled=true           # choice: disk, lru, redis
origin_cache_backend=disk           # choice: orjson, pickle, noop
origin_encoding=orjson
# backend specific settings , uncomment the relevant
origin_cache_lru_max_items=50
origin_cache_disk_path=./asg_cache_org
origin_cache_redis_url=redis://localhost:6379

response_cache_enabled=true
response_cache_backend=lru          # choice: disk, lru, redis
# backend specific settings , uncomment the relevant
response_cache_lru_max_items=20
response_cache_disk_path=./asg_cache_rsp
response_cache_redis_url=redis://localhost:6379

# update http client settings (timeout can be overriden by the spec)
http_timeout=33
http_retry_backoff=1.0
http_max_retries=3
```

FIGURE 28.ENV FILE EXAMPLE FOR CONFIGURING SFDPS AT RUNTIME

The settings are loaded only once at SFDP app startup, from inside the library, and used for the system initialization. In production environments where this can be a limitation, dynamic reloading and reconfiguration can be implemented. In TEADAL Platform, the declarative k8s control plane can be relied upon to reload the SFDP (basically, restart the old pods and start

Funded by the European Union

up new ones) when configuration change is required. Figure 28 presents the *.env* file example for configuring the ASG-runtime, and thus the SFDP, at runtime.

The Caches Module

The caching module is architected to support flexible selection of backend cache implementations and injection of the serialization to be used at the boundary of the cache and its user. There two main concepts developed to support this flexibility, the Cache Backends and the Cache Roles, are described here, the serialization support will be described next as it is a separate module and is used not only by the caches.

Cache Backends

The Cache Backends concept in the ASG-runtime enables flexible, maintainable, and future-proof caching support for Shared Federated Data Products (SFDPs). This abstraction allows:

- Flexibility in swapping out caching implementations without changing application logic
- Maintainability through a unified interface, decoupling core logic from backend specifics
- Extensibility for future cache mechanisms (e.g., cloud-native caches or hybrid local-remote setups)

The implementation is centered around a BaseCache class, which defines a common cache interface and a set of abstract methods to be implemented by specific backends. As of this writing, ASG-runtime provides three backends: **LRU**, **DiskCache**, and **Redis**, selected to cover a range of TEADAL use cases (stateless vs. stateful, single-node vs. multi-node, etc.). The architecture is deliberately open to supporting new backends in future iterations. Table 5 presents a brief comparison of several backends considered, with their pros and cons:

TABLE 5: CACHE BACKENDS CONSIDERED FOR INCLUSION

Option	k8s-ready	Persistent	Multi-pod Safe	Notes
Lru (in-memory)	Yes (per pod)	No	No	Fast, simple; per-process only; high memory use; no external setup; does not require encoding
diskcache	Yes (per pod)	Yes	No	Local disk-based; good for large objects; uses SQLite under the hood
Redis (external service)	Yes	Yes	Yes	Fast and scalable; shared cache; requires external Redis deployment and namespacing.
Redis (via fastapi-cache)	Yes	Yes	Yes	Uses Redis client within app; additional exposure and config needed.
Redis (sidecar pattern)	Yes	No	No	Easier setup than external; isolated per pod; limited utility.
Memcached (external)	Yes	No	Yes	Fast, multi-node; limited persistence by being memory-based and losing data on restart; external config required
Sqlite (file-based)	Yes (per pod)	Yes	No	Lightweight; easy local persistence; not shareable
Picle/joblib (local files)	Yes	Yes	No	Simple for small objects; no concurrency handling; dev-only use
Clous-native, e.g., AWS ElastiCache, GCP Memorystore	Yes	Yes	Yes	Fully managed; scalable; vendor lock-in; access and cost considerations; integration cab be difficult

Funded by the European Union

Cache Roles

The dual-level approach to caching, supports two roles, the origin cache and the response cache. Independent of the role the cache plays, all the caches reuse the same implementation (backend, configuration, serialization) but differ in how they compute the keys, how they decide on data revocation, and what driver is used to 'own' the cache.

Response Cache

- Stores transformed results for rapid repeated access and is especially useful when original data is stable, but transformations are costly
- Keys are computed by hashing the endpoints' GIN Connector Specification strings
- Cached objects are transformed and encoded response datasets

Origin Cache

- Stores raw data fetched from origin APIs as well as additional information optionally provided by the origin server, in our case, the caching http headers (other options considered described next)
- Keys are computed by hashing all the available REST call parameters used to retrieve the data, namely, the URL with its path elements, parameters, etc.
- Cached objects are of two types:
    - data: the dataset received from the FDP, cached under the origin cache key for this endpoint
    - headers: ETag/LastModified http headers, if they were provided by the origin FDP, cached under a special key, created by appending a header prefix to the origin cache key for this endpoint
        - ETag (Entity Tag) → A unique hash that changes if the data changes.
        - Last-Modified → A timestamp indicating when the resource was last updated.

- The flow:
    - when first fetching the data, we store, along with the data, the values returned in the ETag or the Last-Modified headers or both
    - when this dataset is requested again, we add data freshness validation headers to the request we send to the origin FDP
    - If-None-Match header when the ETag response header is available in the cache
    - If-Modified-Since header when the Last-Modified response header is available in the cache
    - if the server responds with 304 Not Modified, we use the cached data
    - if the server responds with 200 OK, get the new data and update the cache.

Alternatively, we could implement additional methods for validating the FDP data freshness, e.g., sending a lightweight "Version Check" Request to the origin FDP. This could be more predictable if supported by all the FDPs, e.g. by implementing data versioning and service endpoints to retrieve the version. In production systems we recommend implementing this functionality across all the FDPs in the TEADAL Federation and rely on it for cleaner origin caching.

Funded by the European Union

There are additional alternatives that we have considered but did not implement. For example, we could rely on TTL for refreshing the data, but this would be more complex and even less predictable than the ETag/LastModified option, and we recommend against relying on *TTLs/SETEX* for data freshness in production environments. Table 6 summarizes the options with their pros and cons.

TABLE 6: OPTIONS FOR VALIDATING FDP DATA FRESHNESS

Option	Requires FDP support	Overhead	Pull vs Push Mechanism	Notes
ETag/LastModified Headers	Yes (standard)	Very Low, just more headers	Pull	Widely supported by REST APIs; enables conditional GET requests
Version Check Endpoint	Yes (custom)	Low, need to issue small request	Pull	Requires FDPs to expose a version or checksum endpoint
Fetch on Expiry (TTL-based)	No	High	Pull	Works with any source; requires cache to handle TTL and auto-refresh
Polling with Diffing	No	Medium–High	Pull	App periodically re-fetches and compares content; can be inefficient
Pub/Sub or Webhook Notifications	Yes (custom)	Very Low	Push	FDP pushes invalidation/freshness notifications; requires infrastructure level trust
Signed Timestamp or Expiry Field	Yes (custom metadata)	Low	Pull	FDP embeds expiry metadata in responses; simple to parse and check

Both for the origin and for the response cache invalidation for removal of stale and unneeded data can rely on TTL and other eviction methods. We did not evaluate this aspect in the context of the current prototype, leaving it to be addressed in the production systems.

The Serializers Module

The serializers module is very simple and is described here for completeness only. Built similarly to the caching module, as a base calls with its different specific implementations, it allows injecting the most suitable encoding of the cached objects on a boundary between the cache and between its callers. For example, if the cache can store non-serialized objects, e.g. LRU cache, the system can be configured to use *noop* serializer for this cache to avoid possibly costly encoding/decoding on cache boundary. For cases, where the cache requires to receive *bytes* objects for caching, we inject an encoding serializer, depending on what type is preferred in the runtime settings. At the time of writing, the system supports three serializer implementations: a *noop* which avoids encoding/decoding, an *orjson* for fast encoding of the result datasets, and *pickle* encoding, as middle ground. As with caches, the library allows extensibility by adding additional custom serializers.

The Executor Module

The Executor is a singleton-like orchestrator, initialized once per SFDP app instance execution, during FastAPI lifespan startup, and acts as a lifespan-injected app context used by the app for serving all its endpoints. This is like having a service object in classic design patterns. In a web context, this gives separation of concerns and clarity, especially as the system grows. Note that Executor is the only ASG-runtime library object that SFDP apps must know about.

Funded by the European Union

- loading, validating, and interpreting the settings
- instantiating logging and runtime objects such as caches, serializers, origin fetcher, etc., according to the settings, ensuring initialization is centralized and robust
- communications with the FastAPI app executing the SFDP - returning results and errors according to a well-formed minimalistic interface (see below)
- managing the response cache and encoding of the result datasets into json format with *orjson* serializer
- coordinating fetch-transform-respond logic per data endpoint request, delegating data retrieval & transformation to a GIN-dependent per-request driver objects explained next
- manages runtime lifecycle and access to it state such as:
    - Shared caches
    - Global settings
    - Runtime stats / observability
    - Logging / tracing

Executor-app interface is created to be as minimal as possible, with only few hooks exposed by the Executor to the FastAPI app (SFDP):

1. async create method - load and initiate once; on failure, errors are logged and reported, SFDP initialization is prevented

2. async *get_endpoint_data* method - receives GIN Connector Spec for the endpoint and returns the result as a dictionary of "status" and details. The "status" can be "ok" or "error". For "ok" status, the details contain the already encoded and ready to be sent result dataset; in this case the app returns http 200 with the data with no additional json encoding. For "error" status, the details contain the problem description; in this case the app returns http 500 with the problem description. In production, return codes can be further refined.

3. synchronized methods to support service endpoints; currently implemented methods are:

- get_settings method - returns the currently applied settings as a dictionary
- get_stats method - returns the current values stored in the system wide counters
- clear_origin_cache and clear_response_cache - auxiliary convenience methods that can be used to purge cases without restarting the app if needed.

The Request Driver Module (GinHelper)

While the Executor object contains all the global state of the executing SFDP, each request is handled by a separate instance of the per-request driver object. As it is dependent on the GIN library, the object is called GinHelper.

GinHelper abstraction encapsulates gin-specific logic needed for a single request:

- use GIN Parser to parse the GIN Connector specification into, roughly, the following:
    - a list of objects that need to be returned in the response dataset
    - a list of origin endpoints that must provide data for computing the result datasets, including target FDP URL, endpoint path and any query parameters

Funded by the European Union

    - a list transforms that must be applied to origin datasets for computing the response datasets

- use global OriginFetcher to collect all the required origin data (the fetcher will cache data in a way transparent to the GinHelper)
- use GIN Executor to load and apply transformations

In addition, as part of GinHelper initialization for a specific request, after the GIN Connector spec is successfully parsed, GinHelper is also responsible to create a cache key to be used by the Executor for this request. This way, we decouple all the aspects of spec handling into the GinHelper object and relieve the Executor about knowing its details. On the other hand, all the caching done by the system is fully decoupled from the GinHelper and is transparent to it. Same is true for the http access done by the OriginFetcher on behalf of GinHelper.

To apply the transformation of origin datasets into result datasets, GinHelper uses transforms submodule from GIN Executor, almost as is. This code loads all the methods at runtime from files in the transforms path provided as settings, applies the required transformations, and returns the result. In the future, we envision refactoring this code to become a separate module that will encapsulate the transforms functionality even further. For example, this module could interface with an external Transforms Library service for obtaining the methods as well as for providing feedback on runtime performance of individual transforms back to the library, e.g. using protocol such as MCP. In addition, this module could cache the loaded transformations for cases when dynamic reloading is not required, saving time, energy, and compute resources. These advanced capabilities are out of the limited scope of the TEADAL prototyping; thus, the system is currently limited to only the simple pandas-based transformations that GIN Executor supports.

The http Module (OriginFetcher)

The http module implements all the boilerplate related to fetching data from origin FDP endpoints. The main object exposed by the module is OriginFetcher, responsible for interacting with the rest of the system, fetching the origin data with the help of its http client helpers, caching the fetched results and their caching headers, and returning them to the caller as transformation ready json datasets, so the caller does not have to deal with http objects such as Response, Headers, etc. In the original GIN Library that does not support caching, all http access is handled with synchronous requests library. For TEADAL, this code was refactored to extract the domain specific behaviour into OriginFetcher object and to replace the http client implementation with the asynchronous, httpx-based one. As a result, ASG-library features reliable, asynchronous http communications with the FDP servers, supporting retries with exponential backoff, paging, error handling, etc. In addition, refactoring allows for changing the http client code separately and replacing the implementation as needed, although we did not pursue exposing a clear interface for this in the scope of the project.

Serving SFDP Data Endpoints

We summarize the Software Architecture section by providing an overview of the steps involved in serving data endpoints of SFDPs as components-flow diagram presented in Figure 29 and described below. To complete the picture, Figure 30 presents a sequence diagram for the case the SFDP data request is served from the response cache and Figure 31 continues to show what happens on Response Cache Miss.

Funded by the European Union

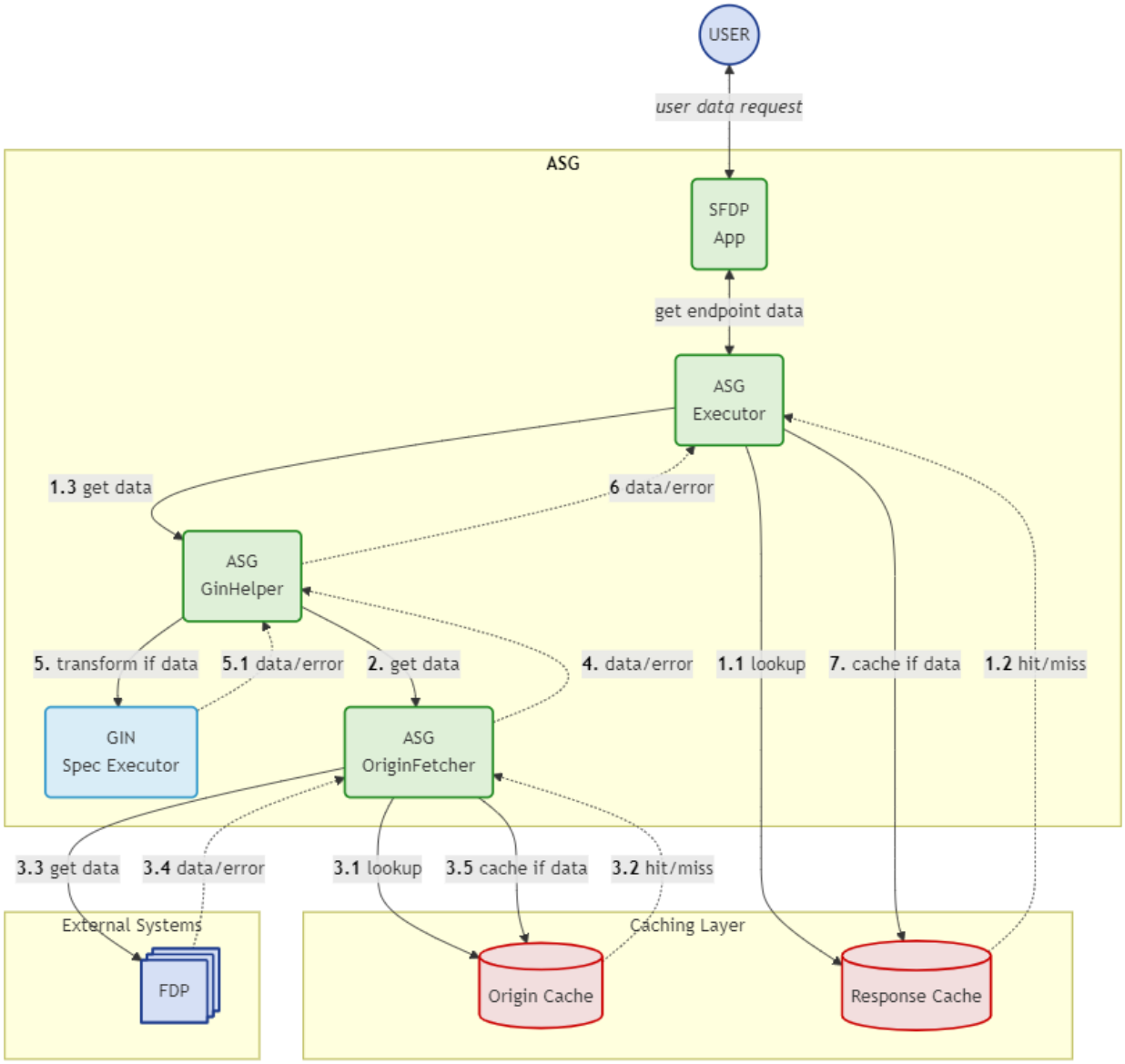

FIGURE 29 THE PROCESS OF SERVING THE SFDP DATA ENDPOINTS

- User issues a request to a running SFDP's data endpoint.
- SFDP app receives the request, obtains an Executor object from its runtime context, and invokes the *get_endpoint_data* method. This method receives a GIN Connector Specification string, embedded in each data endpoint (injected by the ASG-tool during SFDP generation).
- Executor processes the spec:
    - Creates a new GinHelper object to manage the request.
    - GinHelper parses the spec:
        - On error, informs the Executor → Executor informs the app → user receives an HTTP 500 response.
        - On success, informs the Executor and awaits further commands.
- Executor obtains caching key for the request from the GinHelper
- Executor checks the response cache (if enabled):

Funded by the European Union

- On response cache hit, it returns cached result to the app

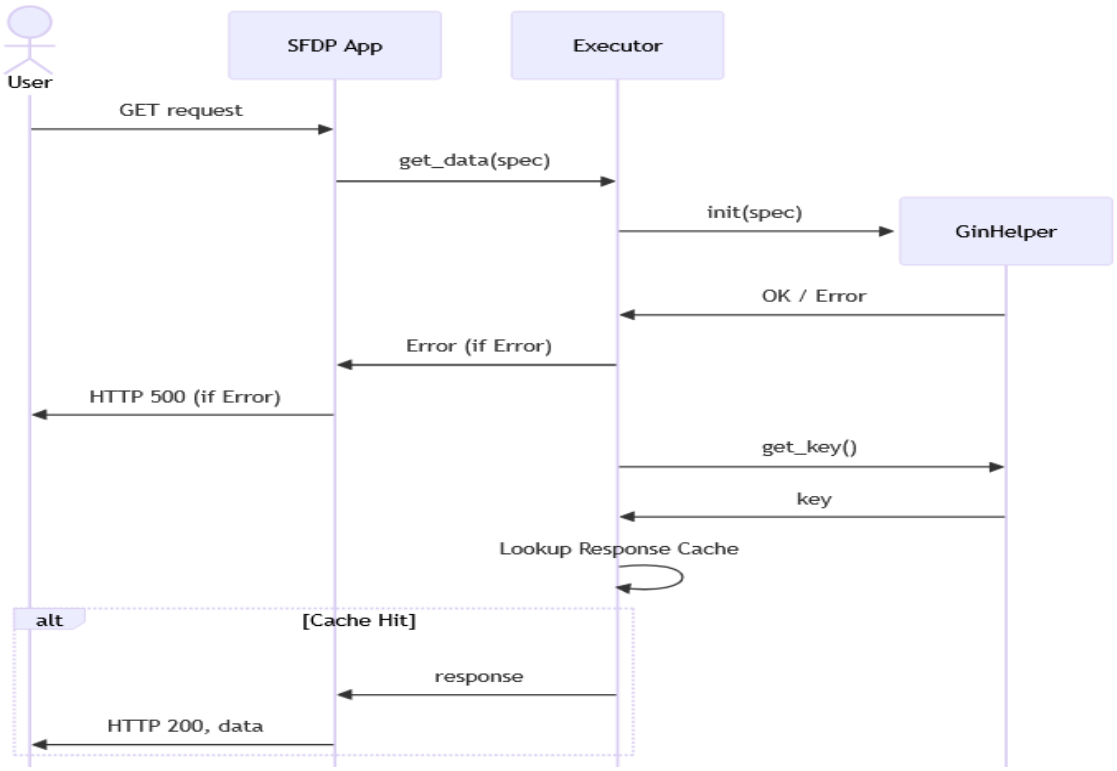

FIGURE 30 SERVING SFDP ENDPOINT DATA FROM THE RESPONSE CACHE

- On response cache miss, proceeds to request the result from GinHelper.
- GinHelper triggers data fetching:
  - Requests OriginFetcher to fetch origin data for each origin endpoint referenced in the spec.
- OriginFetcher fetches origin data for each endpoint:
  - Checks the origin cache (if enabled):
    - On cache hit, returns cached origin data.
  - On cache miss:
    - Uses its HTTP client to fetch JSON data from origin.
      - On error: reports failure up the chain.
      - On success, returns the data and caches it (if origin cache is enabled).
- GinHelper invokes GIN Executor:
  - Loads and applies all specified transforms
  - Returns the result to the Executor

Funded by the European Union

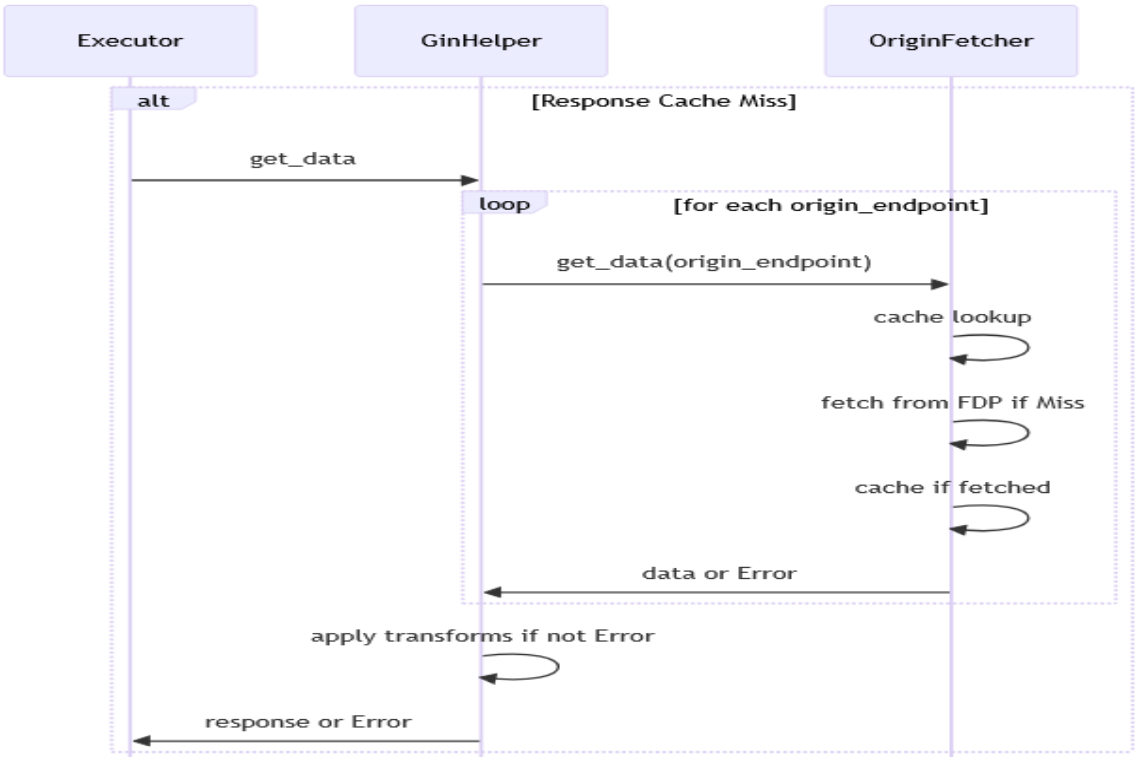

FIGURE 31 SERVING SFDP ENDPOINT DATA FROM THE ORIGIN

- Executor encodes the result using *orjson* encoder
- Executor stores the result in the response cache (if enabled)
- Executor returns the result or an error to the SFDP app
- SFDP app responds to the user with either the result or an error message

4.4 OPERATIONAL ASPECTS

After presenting the high-level design and the software architecture of the ASG system, we briefly describe its operational aspects as part of TEADAL Platform.

Packaging

ASG components are packages as follows:

- ASG-tool is provided as a command line utility that imports GIN modules as dependency and relies on Ollama service deployed on TEADAL as a driver for the generative AI capabilities of GIN. We plan to enhance the tool itself to be packaged as image deployable as TEADAL platform service as part of TEADAL Node and to expose a web interface in addition to CLI. Additional possible enhancement, as already mentioned above, is to integrate git repo creation and pushing.
- ASG-SFDPs are ultimately packaged as images deployable as TEADAL pilot services.
- ASG-runtime is packaged as a pip installable python package and as a base image the SFDP images can be built FROM.

TEADAL Node integration

ASG components are packages as follows:

Funded by the European Union

- ASG-tool is currently provided as a stand-alone tool not yet integrated to the TEADAL Node. ASG-tool main runtime dependency, Ollama service with the preloaded model, is integrated so developers that have access to the TEADAL Federation Nodes, can point their ASG-tool to this Ollama service.
- ASG-SFDPs are ready to be integrated into TEADAL Nodes like any other FDP
- ASG-runtime does not need to be integrated; instead, it is used as a library by all the deployed SFDPs.

SDFP Deployment and Configuration

After SFDP code is generated by the ASG-tool and its image is created and pushed to the registry, its deployment requires the creation of k8s YAML files, kustomize files, rego files, etc., according to the contract under which the SFDP was created. Parts of this process require human validation and approval and are thus manual. In addition, SFDP's deployment artefacts can be further annotated with infrastructure- or resource-specific labels that can influence the selection of deployment targets for the ready-to-go SFDPs.

To configure the SFDP, one should define values for the configuration parameters as ConfigMaps. Attention is required when deciding on how to configure SFDP caching and this deserves a separate discussion.

Enabling Caching

Selecting whether to cache the original data or the transformed data, or both, depends on the nature of the origin and the transformed datasets transformation, the network availability, the storage availability, the policy and additional system constraints. If the transformation is lightweight, e.g., simple filtering, renaming fields, the FDP data is stable (static) and network bandwidth is scarce, then caching the origin data and recomputing the transformation for each SFDP request can be the best solution. Same applies if the origin data is stable but the transformations are expected to be dynamic (injected for runtime loading and execution). On the other hand, when the transformations are stable but computationally expensive, e.g. e.g., encryption, heavy filtering, aggregation, or when the computed dataset is significantly smaller than the origin dataset and storage is scarce, it might be better to cache the response data. In cases where the origin FDP data is very dynamic, caching should be disabled, while in some other cases enabling both caches can benefit the system. In current implementation, caching is enabled per SFDP instance and is applied across all the endpoints. For production systems, the implementation can be easily adapted to make separate decisions per endpoint.

Selecting Cache Backends

Selecting the cache backend depends on the environment. After the backend is selected, additional configuration might be required:

1. *lru* backend - no need for additional setup; you can only adjust the *max_items* setting to regular memory size used by the cache

2. *diskcache* backend needs to write to a file so in k8s setting, there is a need to define persistent volumes for it and coordinate its configuration with a *disk_path* setting:

3. *redis* backend might be the most useful but also requires the most attention and configuration.

    - If redis service already exists in the TEADAL Federation, it can be made available to the running SFDP. In this case, we might need to define a namespace to isolate ASG-SFDP objects from other objects managed by other components.

Funded by the European Union

- If TEADAL Federation does not have redis service available, it can easily be created for ASG use and integrated in project GitOps like it's done for other platform or infrastructure services. In this case, we also might want to isolate spaces used by different SFDPs, depending on whether they belong to the same organizations and/or whether there are policies requiring this isolation.

- Alternatively, redis can be deployed as a side car together with the SFDP. In this case, no further namespace isolation is required.

- If redis service is going to be shared among SFDPs (the first two cases above), a multi-tenant redis caching strategy is required. We have already discussed the need for namespace isolation. In addition, a multi-tenant approach for cache expiration and cleanup is required, as well as maybe some additional synchronization (like, for example, disallowing any specific SFPD to purge caches if it is running in this kind of setup). These complexities will have to be handled in production TEADAL environments by infrastructure operators; covering all the possible options is out of the scope of this report.

Runtime requirements of different caching options are summarised in the following Table 7:

TABLE 7: CACHE BACKEND CONFIGURATION

Option	Configuration Parameter	Runtime Requirements
lru	'max_items' – maximum number of numbers to be stored in the cache	None; entirely in-memory and process-local
diskcache	'disk' – directory path for storing cache files	Requires a Persistent Volume (PV) to be mounted and configured for data durability
Redis as a Separate Service	'redis_url' – URL of the external Redis service	Requires an externally available Redis service; typically configured via ConfigMaps. Optional: namespace isolation for multi-tenant use

Runtime stats collection

ASG-runtime accumulates runtime stats for SFDP it backs up. For each cache, it accumulates hits and misses along with the serializer's stats such as the total amount of bytes before and after the encoding and total time taken by encoding/decoding. For the http client, it accumulates the number of requests issued, the number of bytes retrieved from the origin, and the total time spent retrieving the FDP data. At the app level, it also accumulates the amount of data requests received, served, and failed, the total bytes served, and the time taken by serving the requests.

Some of the counters are exposed to SFDP users through /service/stats endpoint, as shown in Figure 32. We do not spill out all the available counters, such as low level serializers stats, in order not to confuse and to overwhelm the users.

Funded by the European Union

```
{
  "app": {
    "requests_received": int,
    "requests_failed": int,
    "requests_served": int,
    "bytes_served": int,
    "processing_time": float
  },
  "rest": {
    "requests_issued": int,
    "bytes_received": int,
    "fetching_time": float
  },
  "response_cache": {
    "hits": int,
    "misses": int
  },
  "origin_cache": {
    "hits": int,
    "misses": int
  }
}
```

FIGURE 32 RUNTIME STATS EXPOSED BY ASG-RUNTIME ON BEHALF OF SFDPS

Telemetry

To enable observability at scale and support operational intelligence, the current runtime statistics infrastructure in ASG-runtime can be extended into full-fledged telemetry by integrating with Prometheus-based monitoring subsystem, enabling it to scrape, store, and visualize SFDP-related metrics. To expose the already implemented stats as metrics in a Prometheus-compatible format, we plan to use standard Prometheus Python client library and to register two types of metric values:

- Prometheus Counters for showing accumulated stats as they are, as all our runtime counters (hits, misses, byte counts, durations, etc.) monotonically increase during SFDP execution
- Prometheus Histograms/Summaries for tracking per-request averages for latencies and byte sizes over time (e.g., ‘request_duration_seconds’, ‘response_size_bytes’)

Once Prometheus metrics are available, dashboards can be created to offer insights useful for Data Lakes Operators, for example:

- Cache Effectiveness: hit/miss ratios over time per cache layer, helping to identify redundant fetches or poor reuse
- Network/Storage Efficiency Gains: aggregate bytes saved due to caching, distinguishing between response cache and origin cache contributions
- FDP Dependency Health: latency and error rates for each origin FDP; slowdowns or outages can be immediately visualized
- Load Characterization: requests per second, data served per SFDP, and peak usage windows

Funded by the European Union

- Encoding Overhead: time and size impact of different serializers across datasets

Metrics can be labelled as belonging either to the cache layer, to the http client, to the origin FDPs and to the SFDP app itself. One example of such mapping is shown in Table 8:

TABLE 8 : EXAMPLE STATS-TO- METRICS MAPPING

Runtime Stat	Prometheus Metric Name	Type	Labels
Requests received	`sfdp_app_requests_received_total`	Counter	sfdp
Response cache hits/misses	`cache_resp_hits_total` `cache_resp_misses_total`	Counter	cache
Origin cache hits/misses	`cache_orig_hits_total` `cache_orig_misses_total`	Counter	cache
Bytes Served	`sfdp_app_bytes_served_total`	Counter	sfdp, endpoint
Bytes Fetched	`fdp_app_bytes_fetched_total`	Counter	fdp, endpoint
Time fetching from FDP	`sfdp_rest_fetching_duration_seconds`	Histogram	sfdp, endpoint
Time encoding/decoding	`sfdp_serializer_duration_seconds`	Histogram	format, stage

To further assist Data Lakes Operators, the following can be implemented by the monitoring subsystem:

- Alerting rules: define thresholds for unusual patterns (e.g., 100% miss ratio, slow fetch times) to trigger alerts.
- Auto-tagging metrics: attach Git commit/version, SFDP name, or dataset type to metrics for deeper traceability.
- Push-based metrics: support for Prometheus Push gateway for edge deployments where scraping is not feasible.

Table 9 presents the two options we have identified for integrating the telemetry pipeline as part of ASG, with their pros and cons:

TABLE 9 : OPTIONS FOR INTEGRATING THE TELEMETRY PIPELINE

Option	Implemented In	Metrics Exposed Via	Advantages	Considerations
FastAPI app	In SFDP FastAPI (included in the ASG-tool template)	Served via /metrics endpoint (injected in static part of the template)	- Reuses existing HTTP stack - Easier integration with FastAPI routers	- Couples telemetry with app logic - Requires FastAPI metrics middleware or manual exposition
ASG-runtime library	Internal background HTTP server	Served via separate port (e.g., localhost:8001/metrics)	- Keeps observability self-contained - No changes to FastAPI - Metrics available even outside FastAPI	- Adds a second HTTP listener - Needs coordination with Prometheus scraping configuration
Sidecar Exporter	Separate container or thread	Metrics extracted from logs or API	- Externalizes all telemetry logic - Reuses existing exporters (e.g., log-based, HTTP proxy)	- Adds operational complexity - May lag behind real-time metrics or need custom integration

Funded by the European Union

With such telemetry pipeline, ASG-runtime can not only improve observability and debugging but also provide evidence of performance and efficiency improvements gained through its caching strategies, to help validating the project KPIs 3.2 and 3.2.

Given the TEADAL architecture's emphasis on modularity and clear separation of concerns, integrating Prometheus metrics collection directly in the ASG-runtime library offers a clean and reusable solution. It enables observability without modifying the lightweight SFDP FastAPI apps, keeping them focused solely on serving data. This approach aligns well with the goal of encapsulating SFDP runtime behaviour within the ASG layer. However, it introduces minor operational complexity (e.g., an additional internal port), so collaboration with the TEADAL monitoring subsystem team is recommended to ensure consistent Prometheus scraping setup across the Nodes. Alternatively, teams requiring tighter integration with the app logic or leveraging FastAPI-native monitoring may prefer the FastAPI-level implementation.

Deployment View

To summarize the operational aspects section, Figure 33 shows an example deployment diagram for the ASG components.

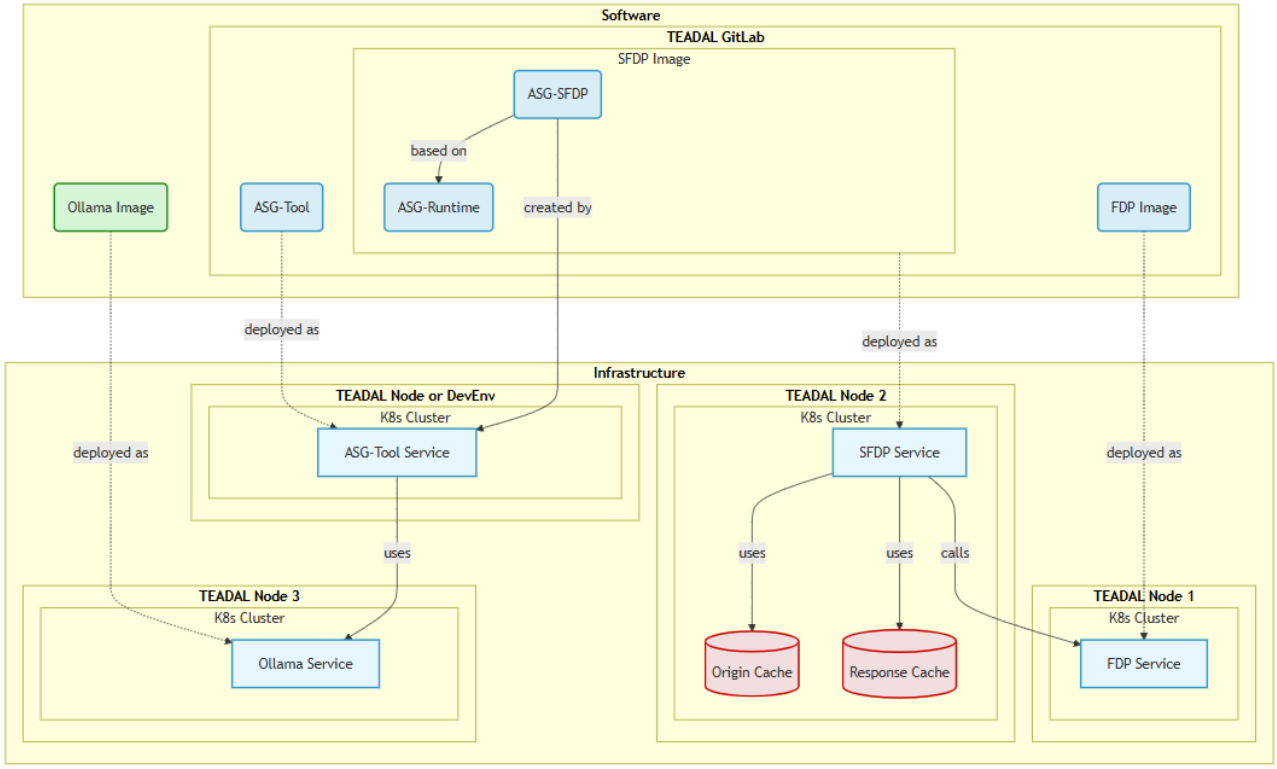

FIGURE 33 AUTOMATION SUBSYSTEM DEPLOYMENT VIEW

Funded by the European Union

5. THE OPTIMIZATION AND THE DEPLOYMENT SUBSYSTEMS

During the initial stages of the project (see D4.1 [7]), we envisioned data pipelines as explicit sequences of transformation operations, applied to data in transit from a source Federated Data Product (FDP) to a destination Shared Federated Data Product (sFDP). These pipelines were expected to fulfil the contractual transformation requirements of data sharing agreements while optimizing operational goals such as performance, energy efficiency, and locality, addressing challenges such as data gravity and friction. Advanced concepts such as *pipeline composition* (building complex pipelines from modular sub-pipelines), and *pipeline partitioning* (for enabling distributed deployment of sub-pipelines) were also introduced. However, most design attention was centered on optimizing, with Stretched Data Lake Compiler (SDLC) based on MCC-C, and orchestrating their deployment, via the Stretched Data Lake Executor (SDLE) based on Kubestellar, rather than on supporting their efficient and scalable construction.

As the project progressed, several critical limitations of this approach became apparent. First, manually constructing pipelines proved to be both time-consuming and skill-intensive, creating delays incompatible with TEADAL's vision of dynamic, user-driven federation workflows. Second, for non-trivial pipelines, the envisioned optimization, solving for placement of all pipeline components under multiple constraints, was not only computationally expensive but also increasingly energy-inefficient, undermining the optimization's own goals.

Understanding these limitations led to a fundamental shift in TEADAL's approach, resulting in the adoption of the ASG-based model described in this report. In this new approach, each FDP-to-SFDP data transformation, documented as an agreement between the FDP Consumer and the FDP Provider, is automatically created as a separately deployable data server component. At runtime, each such component retrieves the data from source and applies the transformations required by the agreement before serving the resulting data to its requestor, without requiring global pipeline planning or optimization.

As detailed in Section 4, these transformation components can be lightweight, dynamically loaded Python functions executed in-process (e.g., via REPL), or can invoke external transformation services via API calls (e.g., REST). This flexibility allows transformation logic to be substituted or adapted at runtime, by policy managers or infrastructure controllers, based on performance or locality needs. Moreover, the transformation functions can originate from pre-approved TEADAL libraries, user-provided scripts, or third-party components (e.g., discoverable via MCP interfaces).

This ASG-based approach affords three critical benefits. First, it drastically simplifies and automates pipeline construction by shifting the focus from manual specification of all the transformation steps towards the LLM-assisted generation of SFDPs based on negotiated agreements, including creation of transformation chains for each of its data endpoints. This makes SFDP creation feasible even for non-expert users and, in addition, simplifies their operation as a single deployable unit instead of a fragile sequence of processes/jobs. Second, it enables cross-pipeline optimization by recognizing shared or co-located transformation components and reusing or scaling them intelligently. Third, and most importantly, it defers certain optimization decisions to runtime. For instance, an sFDP can be deployed near the data source and the data requesters, optimizing latency and system load at that moment. If system conditions later change, instead of recomputing a full pipeline deployment, the runtime can adjust only the necessary transformation implementations or their placements, without breaking the data contract or disrupting service continuity. This makes the system significantly more adaptable and robust than the original, monolithic pipeline optimization model.

Funded by the European Union

As an additional bonus, transitioning to the ASG-based approach to data pipelines allowed us to re-scope both the optimization and the deployment subsystems of the TEADAL Control Plane. For example, for the newly created pipelines, the previous approach required global pipeline optimization followed by the distributed orchestration of deploying potentially many components on potentially several nodes. The new approach only requires a rather simple selection of the SFDP deployment target among the Federation's TEADAL Nodes, followed by dispatching the SFDP for deployment on a selected Node. Like previously, the placement decision should still be informed by system wide metadata such as policies, resource allocations, and monitored runtime data. In addition, like previously, runtime controller-watchers need to be installed to make the required runtime adjustments when needed. Still, the overall complexity of both the optimization and the deployment subsystems is greatly reduced as presented in the next two subsections that follow.

5.1 THE OPTIMIZATION SUBSYSTEM

In the ASG-based architecture, the role of the optimizer is substantially simplified compared to the original design, but it is no less essential. Its core responsibility is twofold: (1) initial placement decision-making for newly generated sFDP servers, and (2) continuous runtime observation and adjustment of deployed components to maintain operational efficiency and contract compliance under changing system conditions.

Initial Placement Decision

While the shift to ASG-based sFDP generation eliminates the need for a full-blown, centralized, multi-objective optimization of multi-component pipelines, it does not eliminate the need for placement optimization. Instead, it reframes the requirements towards a more localized, lightweight form of optimization focused on selecting the deployment target for the automatically generated sFDP server app.

With the ASG-based approach, when a new sFDP is automatically generated, typically in response to a data access agreement, it has to be deployed on one of the available TEADAL Nodes (i.e., Kubernetes clusters across the federation). So, the optimizer is basically responsible for selecting the most appropriate TEADAL Node (i.e., Kubernetes cluster) for hosting this new sFDP. This placement must account for multiple factors:

- **Proximity to data sources and expected consumers** (to minimize latency and cross-site data transfer costs),
- **Resource availability** on candidate Nodes (e.g., CPU, memory, GPU if needed),
- **Transform service locality**, i.e., whether relevant transformation services or reusable components are already running nearby,
- **Policy constraints**, such as jurisdictional data handling rules or energy-efficiency goals.

The decision is made once per sFDP generation and should be lightweight, fast, and explainable. It can leverage monitored metrics (e.g., current load, network Round Trip Times or RTTs, recent query patterns), as well as declarative resource requests encoded in the SFDP's deployment spec (e.g., via custom Kubernetes labels or annotations). After the TEADAL Node is selected, in-cluster placement is delegated to the local Kubernetes control plane, which handles finer-grained scheduling (e.g., pod-to-node assignment) using standard Kubernetes mechanisms, possibly enhanced with additional labels for declaring resource requirements.

Funded by the European Union

This lightweight placement optimization can leverage parts of the previously developed Stretched Data Lake Compiler (SDLC) particularly those components designed to ingest runtime metrics collected by the monitoring subsystem and declarative resource requirements (e.g., labels indicating memory needs, latency sensitivity, or preferred data regions). Runtime monitoring data can also inform decisions by helping estimate expected load, transformation reuse opportunities, or proximity to frequently queried data sources.

In summary, the optimization subsystem remains a key component of TEADAL's control architecture, but its scope is now streamlined. Rather than attempting to optimize entire transformation graphs globally, its role is to make fast, context-aware deployment decisions for individual sFDP servers, aligning them with dynamic system state and operational goals.

Runtime Adjustment via Optimization Controllers

After deployment, sFDPs may continue to run under evolving conditions: changes in load, new transformation capabilities becoming available, or shifts in data source access patterns. To handle this, dedicated optimization controllers monitor runtime state and can trigger adjustments when necessary. These adjustments may include:

- **Injecting or substituting transformation implementations** dynamically, e.g., switching from an in-process Python function to an external high-throughput service when load increases,
- **Rebinding transformation endpoints** to closer or more efficient service instances (e.g., when a cached transformation becomes hot and is replicated),
- **Triggering re-deployment** of an sFDP to a more suitable Node, in extreme cases where relocation offers significant benefit and is feasible under contract and SLA terms.

These controllers operate as Kubernetes operators: watching resource state and responding to declarative goals or policy conditions. They are aware of the ASG semantics and can reason about the relationship between data contracts, transformation chains, and deployment structure.

In effect, this runtime optimization capability allows TEADAL to defer certain placement or adaptation decisions until better runtime knowledge is available, trading pre-deployment complexity for runtime agility. The system becomes more resilient, elastic, and self-optimizing without requiring full pipeline redeployment, a key limitation of the original approach.

Architecture Updates

This lightweight placement optimization can leverage parts of the previously developed Stretched Data Lake Compiler (SDLC) particularly those components designed to ingest runtime metrics collected by the monitoring subsystem and declarative resource requirements (e.g., labels indicating memory needs, latency sensitivity, or preferred data regions). Runtime monitoring data can also inform decisions by helping estimate expected load, transformation reuse opportunities, or proximity to frequently queried data sources.

Eliminate Kubeflow Dependency

In the initial approach to data pipelines, they were conceived as Direct Acyclic Graphs (DAGs) of transformation steps, requiring end-to-end pipeline composition, partitioning across clusters, complex optimization across many stages, and often also model-based tuning (e.g., ML for placement, performance prediction). In that context, Kubeflow made sense with, for example Kubeflow Pipelines selected to model and orchestrate complex pipelines and Katib to

Funded by the European Union

automate hyperparameter-style search for optimal pipeline configurations or placement heuristics. Although Kubeflow is a functionally beneficial multi-step pipeline engine, its reliance on ML-heavy tooling turns it to a heavyweight overhead, adding complexity and degrading efficiency.

In the ASG-based model, we generate one deployable sFDP per agreement, not long convoluted pipelines. The transformation chains are still present but now as declarative specifications contained as part of every data endpoint of the new SFDP (GIN Connector Spec). As a result, placement decisions are made per individual self-contained data servers, not for complex DAGs. This allows the optimization to be far more lightweight than a full combinatorial problem before. Runtime optimization can be reactive and local, handled by custom controllers and not by the retrained ML models.

In short, adoption of the ASG-based approach to data pipelines, allows us to eliminate the dependency on Kubeflow and to overall achieve simple architecture, faster optimization integrations, potentially better explainability of the optimization decisions (this can be crucial for compliance in some industries), and lower resource consumption achieving the energy-efficiency goals better than before (see full description of the approach in D3.3 [10]).

Adapting the Stretched Data Lakes Compiler (SDLC)

TEADAL's already demonstrated SDLC[12] component, designed to analyse, optimize, and enrich data pipelines was designed to ingest Kubeflow pipelines and apply optimization to produce placement recipes, while addressing key optimization objectives, with a primary focus on resource optimization, and other crucial constraints.

To adapt it to the new ASG-based approach, it must be refactored to receive simpler inputs while still leveraging the various data and resource inventory services, including the Catalogue, as well as the Federation's resource inventory. Here is the list of metadata SDLC uses in order to compute the result:

- Metadata about source datasets. From the Catalogue, SDLC receives information and metadata about the source FDPs that the SFDP needs to get data from, e.g. FDP service location, dataset size, cardinality, and more.
- Metadata about the Federation. To perform the optimization, SDLC needs access to a detailed inventory of available locations (i.e., TEADAL Nodes or clusters) for pipeline execution. This inventory is planned as mostly static YAML manifests fully describing TEADAL Federation, its members, resources, users, policies, etc.
- Metadata about transformations, e.g., the input-output ratio of tasks, what columns being read and/or written, the nature of updates to the columns being written (e.g., append-only, overwrite). At first, the pipeline-based design was expecting users to annotate the transformations with this type of metadata. It was quickly understood, however, that having to specify some of the task characteristics could be too much of a burden to FDP developers/designers and also can result in inaccurate estimates/measurements. As a result of this understanding, and exploratory activity towards trying to predict these characteristics, possibly using LLMs, inspired by work done by the database optimization community. With an ASG-based approach, this data can be included as part of the transformations library, including by dynamically adding the collected performance characteristics.
- Metadata about the operational environment, e.g. the bandwidth between locations, the availability of hardware resources, and the geographic location of compute clusters.

[12] TEADAL Tech / stretched-data-lake-compiler · GitLab

Funded by the European Union

In addition to receiving the target data pipeline as an annotated Kubeflow manifest and the required metadata, the optimizer needs to know its optimization objectives (e.g., minimizing the data transfer, the energy consumption, the cost, the execution time, etc.)

Internally, SDLC computes internal representation of its inputs and executes the solver to resolve the constraints. The result of the SDLC execution is a Kubeflow pipeline enriched with optimization decisions indicating the most suitable clusters for their execution. This also needs to be adapted to just produce the prioritized list of deployment targets for the input SFDP.

To summarize, Table 10 presents the adaptations required to make the SDLC working as part of the ASG-based approach. Alternative options could be and will be considered if time/resources will permit.

TABLE 10 : ADAPTING SDLC TO WORK WITH ASG

Aspect	Current SDLC Design	Plan towards ASG-SDLC integration	Notes
Input: data pipeline for placement	Kubeflow Pipeline	Manifests generated as part of FDP-SFDP agreement	Easy to adapt. ASG-SFDP can be presented as a single step pipeline. Metadata annotations can be missing.
Input: Location constraints for tasks	Manual addition	Might be not needed.	Can be designed into the Transforms Library
Input: Metadata about datasets	Manual at first, the obtained through Catalogue APIs	Catalogue APIs	Can be obtained through the monitoring subsystem from the already deployed data products that support telemetry
Input: Federation's Resource Inventory	Kubestellar Resource Inventory	Inventory defined and maintained by Data Lake Operators as YAML manifests	Even in Kubestellar Resource Inventory would need to be based on source of truth provided by the operators

5.2 THE DEPLOYMENT SUBSYSTEM

In the ASG-based model, the Deployment Subsystem is responsible for realizing the optimizer's placement decisions and ensuring that sFDP components are correctly and consistently deployed across the Federation's TEADAL Nodes. While the first project iteration focused on sophisticated orchestration logic for deploying graph-shaped pipelines with complex interdependencies, the shift to single-component sFDP servers has significantly reduced the deployment complexity. However, the challenge of multi-cluster, policy-aware, and GitOps-compatible deployment orchestration remains central to TEADAL's operational model.

Evolving Landscape of Multi-Cluster Orchestration

Since TEADAL's inception, the Kubernetes ecosystem has seen significant advancements in multi-cluster management. Organizations now routinely operate numerous clusters across diverse environments, including cloud, on-premises, and edge locations. This proliferation has led to the emergence of several tools and managed platforms designed to simplify multi-cluster operations, some by well-established and trusted vendors and cloud providers:

Funded by the European Union

- Rancher[13]: Provides centralized management for multiple Kubernetes clusters, offering features like unified authentication, access control, and monitoring.
- Google Anthos[14]: Enables consistent application deployment and operations across hybrid and multi-cloud environments, integrating with existing Kubernetes clusters.
- Azure Arc[15]: Extends Azure management capabilities to Kubernetes clusters running outside of Azure, facilitating unified governance and policy enforcement.
- Spectro Cloud Palette[16]: Offers a platform for managing Kubernetes clusters across various infrastructures, emphasizing flexibility and customization.

These and other solutions underscore the industry's shift towards more sophisticated and scalable multi-cluster management approaches. Most solutions available as a service are feature rich and affordable making investment in creating a do-it-yourself solution less attractive than few years ago when the solution space was fresh and open to innovations and there many competing open-source projects exploring the space (see the survey included in the previous deliverable of this work package [8]).

While initially it was planned to use Kubestellar, due to change of direction taken by the project. Looking for a replacement project to integrate, we created the survey reported in D4.2 [8] and discovered that most projects are fading away, either turning into vendor-backed managed solutions or just disappearing. As our WP4 focus has pivoted towards AI-driven automation and declarative methodologies, the initial plan to utilize open-source tools like Kubestellar for cross-cluster orchestration was set aside, in alignment with project priorities and with the broader industry trend of leveraging existing, mature tools rather than developing bespoke solutions from scratch.

As a result, we envision that in production TEADAL Platform will be realized using production solutions of choice selected by the Data Lake Operators. For prototyping and demonstration purposes, we have decided to scaffold simple yet functionally sufficient do-it-yourself multi-cluster deployment subsystem relying on a lightweight control loop built around two custom TEADAL CRDs:

- *TeadalFederation*, which represents federation-wide configuration, including the list of participating TEADAL Nodes (clusters), their access configurations (e.g., kubeconfig secrets), and shared policy metadata.

- *TeadalDataProduct*, which represents a deployment intent for a single sFDP, including its container image, resource requirements, labels, and transformation metadata.

A TEADAL Controller component, written in Python and deployed either centrally on a dedicated Node or replicated across the Federation's Nodes, is initiated with the (mostly static) *TeadalFederation* resource that contains information about the Federation buildup. In addition, the controller watches for new *TeadalDataProduct* resources. Upon detecting one, it performs the following sequence:

1. **Node Selection**: Using the metadata in the TeadalFederation and the placement decision produced by the Optimization Subsystem, it selects the target Node (cluster)

[13] Enterprise Kubernetes Management Platform & Software | Rancher

[14] Anthos Powers Enterprise Container Platforms | Google Cloud

[15] Azure Arc – Hybrid and Multi-Cloud Management and Solution

[16] What is Palette? | Palette

Funded by the European Union

for deployment. Note that in the pipeline-based design, final selection of the target Node for the pipeline deployment out of the prioritized list of nodes computed by the SDLC, is also under the responsibility of the Stretched Pipeline Executor.

2. **GitOps Dispatching**: It pushes the corresponding sFDP deployment YAML (or Kustomize patch) to the Git repository associated with the selected Node. This repository is continuously watched by a GitOps agent (e.g., Argo CD) running on that Node.

3. **Deployment Execution**: The GitOps agent applies the new resource manifests to the local cluster, ensuring that the sFDP server is started, monitored, and automatically re-synced if drift occurs.

4. **Status Tracking and Reconciliation**: The TEADAL Controller periodically queries the Kubernetes API of each Node to monitor deployment status and health. If failures or delays are detected, it can trigger re-dispatch, fallback logic, or even re-optimization if allowed by the sFDP's contract.

This architecture ensures a clean separation of concerns:

- Optimization logic is kept lightweight and dynamic.
- Deployment logic is delegated to GitOps mechanisms per cluster, ensuring local consistency and minimal central coordination.
- Monitoring and adaptation happen through federated observation of runtime metrics, fed back into both optimization and deployment decisions.

Importantly, this structure also supports TEADAL's vision of zero trust. Since each TEADAL Node has autonomy over its local GitOps, deployment compliance with organizational constraints is preserved, avoiding the need for a centralized orchestrator with direct write access to remote clusters. Finally, this TEADAL deployment flow integrates tightly with the ASG toolchain. As soon as a new sFDP is generated, its deployment spec is automatically registered as an SFDP CRD, closing the loop from agreement creation to operational deployment. This supports dynamic, on-demand federation scenarios where data sharing workflows can be rapidly created, deployed, monitored, and adapted, entirely through declarative resources and lightweight control logic.

Note also that in the pipeline-based approach, the deployment component (the Stretched Pipeline Executor) was also assumed responsible for orchestrating the communications between the different sub-pipelines, especially when they are set to run in separate clusters, sending output of one sub-pipeline to another pipeline as input. With an ASG-based approach, we do not envision the need for dedicated connectivity management beyond what is already provided by the TEADAL Platform that ensures policy-driven connectivity in a way implicit to the data users.

Funded by the European Union

6. SUMMARY

Final architecture and realization of the TEADAL control plane described in this report puts forward several major innovations.

First, the automation subsystem enables SFDPs to be implemented as lightweight, self-contained API-driven data services that act as controlled intermediaries between FDPs and their consumers. These SFDPs are generated using a templated application framework and at runtime are backed by a shared library. This approach facilitates rapid development, consistent deployment, and unified runtime management of SFDPs across the TEADAL Federation, in alignment with the rest of the TEADAL Platform, including TEADAL Nodes integration, policy management, Catalogue flows, etc.

Next, to further enhance the operational capabilities of SFDPs and the TEADAL infrastructure at large, the platform integrates intelligent monitoring and optimization strategies. These are driven by rich runtime metadata and AI-based analysis pipelines. The AI-Driven Performance Monitoring (AI-DPM) framework presented in (section ref) complements the Stretched Data Lakes by enabling proactive resource management, anomaly detection, and performance prediction across the TEADAL Nodes, all critical to achieving the platform's trust, sustainability, and efficiency goals.

Another important innovation is related to exploring applicability of LLMs across the board of the WP4 tasks. As a result of this exploration, we present the ASG-tool, the RBAC policy generation app, and the LLM based data analytics.

Additional innovation is related to the Transformations Library Concept that will certainly be further explored towards better automation, e.g. annotating the tools with their functional and performance characteristics, and towards potential alignment and integration with emerging industry trends such as MCP. Future work includes exploring MCP-compatible exposure of TEADAL's transformation assets, enabling LLM-based orchestration and discoverability of reusable components across the federation, potentially turning TEADAL into a reference model or early prototype of MCP-aligned architectures.

Assets created by the WP4 team are designed in alignment with the overall TEADAL Architecture and Platform design. Some components are already integrated into the platform and available for pilots while some are in the process of integration and validation.

Funded by the European Union

REFERENCES

[1] TEADAL Consortium, D2.1, Requirements of the Pilot Cases, Mar 2023

[2] TEADAL Consortium, D2.2, Pilot Cases Intermediate Description and Initial Architecture of the Platform, Nov 2023

[3] TEADAL Consortium, D2.3, Pilot Cases Final Description and Intermediate Architecture of the Platform, Aug 2024

[4] TEADAL Consortium, D2.4, Final General Architecture, due May 2025

[5] TEADAL Consortium, D6.1 Testbed Design, November 2023

[6] TEADAL Consortium, D6.2 Integration Report, August 2024

[7] TEADAL Consortium, D4.1 Stretched Date Lakes, First Release Report, Jan 2024

[8] TEADAL Consortium, D4.2 Stretched Date Lakes, Second Release Report, Oct 2024

[9] TEADAL Consortium, D3.2 Reducing energy footprint in federated stretched data lakes, 2024

[10] TEADAL Consortium, D3.3 Privacy/confidentiality definition approach, due May 2025

Funded by the European Union