



ExtremeXP

**Experiment Driven and user Experience Oriented Analytics for
Extremely Precise Outcomes and Decisions**

**D5.2: Core framework services - Data and
knowledge management**

Project Acronym/Title	Experiment Driven and user Experience Oriented Analytics for Extremely Precise Outcomes and Decisions
Grant Agreement No.:	101093164
Call	HORIZON-CL4-2022-DATA-01
Project duration	36 months 01 January 2023 – 31 December 2025
Deliverable title	Adaptive experiment-driven complex analytics framework
Deliverable reference	D5.2
Version	1.0
WP	5
Delivery Date	30 April 2024
Dissemination level	PU - Public
Deliverable lead	ICCS
Authors	Petar Kochovski (UL), Ilias Gerostathopoulos (VUA), Keerthiga Rajenthiram (VUA), Eleni Zarogianni (ICOM), Tomas Bures (CUNI), Petr Hnetynka (CUNI), Andrews Cordolino Sobral (AE), Marcela Tuler de Olivera (TUDelft), Orestis Almpantoudis (ICCS), Yiannis Verginadis (ICCS), Dimitris Apostolou (ICCS)
Reviewers	Vasilios Theodorou (ICOM) Stavros Maroulis (ARC)
Abstract	This deliverable documents the intermediate status of the core ExtremeXP framework services responsible for designing, configuring, and deploying experimentation workflows. Moreover, it describes the current status of the secure and distributed management of datasets and knowledge pertaining the experimentation-based workflows, as well as the data authorisation mechanism supporting the ExtremeXP framework. The deliverable includes links to the software code and libraries that implement the aforementioned services. It also documents their implementation and provides usage Information.
Keywords	experiment planning, experiment execution, knowledge management, access control

Dissemination Level	
PU	Public, fully open
Type	
DEM	Demonstrator, pilot, prototype, plan designs

Version History

Version	Date	Owner	Editor(s)	Changes to previous version
0.1	2024-01-30	ICCS	Dimitris Apostolou	ToC and allocation of writing
0.2	2024-04-08	ICCS	Dimitris Apostolou	Draft for review
0.3	2024-04-28	ARC, ICOM, VUA	Stavros Maroulis (ARC), Ilias Gerostathopoulos (VUA), Vasilios Theodorou (ICOM)	Internal review
0.4	2024-04-29	ALL	All Contributors	Addressed reviewers' comments
1.0	2024-04-30	ARC	Giorgos Giannopoulos, George Papastefanatos, ARC	Final version for submission

Table of contents

Version History	3
Table of contents	4
Table of Figures	7
Table of Listings	8
List of Acronyms.....	10
Executive Summary.....	11
1 Introduction.....	12
1.1 Objectives.....	12
1.2 Structure.....	15
1.3 Map of Tools and Prototypes.....	15
2 Experimentation Framework and Domain Specific Language	17
2.1 Framework overview	17
2.2 DSLs.....	18
2.2.1 Workflow meta-model updates.....	18
2.2.2 DSL for modeling workflows.....	19
2.2.3 DSL for modeling experiments	24
2.3 Command-line tools	26
2.3.1 Workflow DSL visualizer	26
2.3.2 Validator of workflows	28
2.3.3 Generator of espace.....	28
2.4 Design Model Storage (DMS).....	29
2.5 Graphical editor for experiment specification	29
2.5.1 Login	31
2.5.2 Dashboard	31
2.5.3 Editor	33
3 Continuous Adaptive Experiment Planning	38
3.1 Parsing.....	38
3.2 Reading Workflows	38
3.3 Reading Assembled Workflows Data.....	40
3.4 Generating Assembled Workflows	40
3.5 Streamlining the assembled workflows.....	41
3.6 Reading Experiment Spaces.....	43
3.7 Creating Experiments	44
3.8 Executing Experiments	44
4 Data Abstraction Layer	46

4.1	Strategy and process for traceability and repeatability of experiments and their results ..	46
4.2	Data model used by the Data Abstraction Layer	46
4.3	Reference architecture of the Data Abstraction Layer	48
5	Runtime for Scheduling Complex Workflows	50
5.1	Introduction	50
5.1.1	Overall Architecture of the PWS	50
5.2	ProActive AI Orchestration	51
5.3	Scheduling Abstraction Layer and Proactive Python SDK	52
5.3.1	Applications and Examples:	53
5.3.2	Getting Started with the ProActive Python SDK	53
5.3.3	Supported programming languages	54
5.3.4	Task dependencies	55
5.3.5	Job and task variables	56
5.3.6	Workflow Data management	58
5.3.7	IDEKO Use Case (UC5)	61
6	Context-aware Access Control for Experiment-driven Analytics	64
6.1	Relevant technologies	64
6.1.1	Attribute-Based Access Control Model language	64
6.1.2	Hyperledger Besu and Smart contracts	65
6.1.3	Keycloak for authentication and JWTokens	65
6.2	Access Control System Architecture	66
6.2.1	Users roles	68
6.2.2	Application	68
6.2.3	Smart contracts	68
6.2.4	Context handling	69
6.2.5	Modelling the access policies	70
6.2.6	Access control enforcement flow	70
6.3	Demonstration of a Context-aware Access Control for Experiment-driven Analytics - UC5 IDEKO	72
6.3.1	User roles	72
6.3.2	Network and Nodes	72
6.3.3	Modelling access control policies	73
6.3.4	Example tokens issued for IDEKO	74
6.3.5	The Middleware (PEP)	76
6.3.6	The Smart Contracts	79
6.4	Next steps	82
6.4.1	Time complexity test and performance evaluation	82
6.4.2	Implementation of additional context handlers for UC5	82
6.4.3	Policy Modeling and Context Handling for Other Use Cases (UC1-UC4)	82
6.4.4	Designing the Translator Middleware for Smart Contract Policy Mapping	82
7	Management of Decentralized Data and Knowledge for and from Experiments	83
7.1	Approach	83
7.2	Conceptual architecture	83
7.3	Components Functionality	85
7.3.1	Zenoh Backends and Volumes	87

7.3.2	Zenoh taxonomy of components.....	87
7.4	Prototype Zenoh Configuration & Rest API.....	88
7.5	Relevant Technologies and Investigations.....	92
7.5.1	Interplanetary File System (IPFS).....	92
7.5.2	Decentralized DB.....	93
7.5.3	Dataset Catalog.....	93
7.5.4	Distributed Ledger Data	93
7.5.5	API	93
7.5.6	NFT Provenance and Traceability	93
8	Conclusion and Future works.....	95
9	References.....	96
10	Appendices.....	97
10.1	Appendix 1: API of the Data Abstraction Layer	97
10.1.1	Retrieval of executed workflows overview	97
10.1.2	Addition of an executed workflow to the overview	97
10.1.3	Modification of an executed workflow in the overview	97
10.1.4	Retrieval of executed tasks overview	98
10.1.5	Addition of an executed task to the overview	98
10.1.6	Modification of an executed task in the overview	98
10.1.7	Retrieval of input datasets overview	99
10.1.8	Addition of an input dataset to the overview	99
10.1.9	Modification of an input dataset in the overview	100
10.1.10	Retrieval of output dataset overview	100
10.1.11	Addition of an output dataset to the overview.....	100
10.1.12	Modification of an output dataset in the overview	101
10.1.13	Retrieval of metrics overview.....	101
10.1.14	Addition of a metric to the overview.....	101
10.1.15	Modification of a metric in the overview	102
10.1.16	Retrieval of parameters overview	102
10.1.17	Addition of a metric to the overview.....	102
10.1.18	Modification of a parameter in the overview	103
10.2	Appendix 2: TextX Grammar.....	104
10.3	Appendix 3: Zenoh API endpoints	106

Table of Figures

Figure 1. ExtremeXP Architecture and Reported Components	12
Figure 2. UC5 experiment workflow for the binary classification problem.....	15
Figure 3. Framework overview	17
Figure 4. Workflows meta-model	18
Figure 5. Visualization of Use Case 5.....	27
Figure 6. Visualization of the workflow (Use Case 4)	28
Figure 7. The structure of the graphical editor.	29
Figure 8. The user flow of the graphical editor.	30
Figure 9. The login page.	31
Figure 10. Dashboard experiments page.	32
Figure 11. Dashboard tasks page.	32
Figure 12. Editor overview.	33
Figure 13. "Save model" functionality.	34
Figure 14. "Save as" functionality.	34
Figure 15. Composite task editing.....	35
Figure 16. Label editing.....	36
Figure 17. Task configuration panel - name editing.	36
Figure 18. Add task variant.	37
Figure 19. Task variant selection.....	37
Figure 20. Overview of Continuous Adaptive Experiment Planning Module	38
Figure 21. Data layer meta-model	47
Figure 22. Meta-model of data used in the Data abstraction layer	48
Figure 23. IVIS architecture.....	48
Figure 24. Architecture of ProActive Workflows & Scheduling System	50
Figure 25. Designing Predictive Models with ProActive's Machine Learning Studio	52
Figure 26. IAM system landscape	67
Figure 27. IAM container view	67
Figure 28. Policy definition template	70
Figure 29. Flow of Authentication and Authorization (forUC5 IDEKO).....	71
Figure 30. Flow of processing the policies	71
Figure 31. First Node in the Hyperledger Besu Network.....	72
Figure 32. Last (Fourth) Node in the Hyperledger Besu Network	73
Figure 33. Policy definition for IDEKO	73

Figure 34. Access token generation for authentication	74
Figure 35. Resource Access token (RPT) generation for authorization (parameters).....	75
Figure 36. Resource Access token (RPT) generation for authorization (Header)	75
Figure 37. Decentralised data management conceptual architecture	84
Figure 38. Zenoh positioning on ISO/OSI Model	85
Figure 39. Zenoh peers and routes	86
Figure 40. Zenoh url filtering leveraging selectors (Key1: First variable name; Key2: Second variable name; Value1: First property value; Value2: Second property value;?: Query string begins; =: Value separator; &: Parameter separator)	88
Figure 41. Prototype Rest API for file management.....	89
Figure 42. Zenoh API Client.....	91
Figure 43. Zenoh API Client UI	92

Table of Listings

Listing 1. An example of a UC 5 (provided by IDEKO) workflow definition in our DSL.	19
Listing 2. Examples of UC5 assembled workflow definition in DSL.	21
Listing 3. Use case 4 modeled in DSL	21
Listing 4. An example of an UC4 assembled workflow definition in DSL.....	24
Listing 5. An example in Experimentation space DSL for UC5.	24
Listing 6. An example in Experimentation space DSL for UC4.	25
Listing 7. Code segment to read DSL file and create a meta model	38
Listing 8. Code segment to read workflows from the DSL specifications.	39
Listing 9. Code segment to read assembled workflows data from the DSL specifications	40
Listing 10. Code segment to generate final assembled workflows	41
Listing 11. Code segment to streamline the assembled workflows	41
Listing 12. Code segment to find the tasks within a subworkflow	42
Listing 13. Code segment to order the tasks in sequence	42
Listing 14. Code segment to read experiment spaces.....	43
Listing 15. Code segment to create experiments according to the experiment method	44
Listing 16. Code segment to submit experiment workflows on Proactive	44
Listing 17. Creating a virtual environment	53
Listing 18. Connecting to a ProActive server.....	54
Listing 19. Creating a task	55
Listing 20. Creating a task dependency	55

Listing 21. Creating job and task variables	56
Listing 22. Producing a global variable	57
Listing 23. Consuming a global variable	57
Listing 24. Producing a result variable	57
Listing 25. Consuming the results variable.....	58
Listing 26. Transferring data to the user space	59
Listing 27. Transferring data from the user space.....	59
Listing 28. Transferring to global space.....	60
Listing 29. Importing from global space	60
Listing 30. Uploading files	60
Listing 31. Downloading task results	61
Listing 32. UC5 example setup	61
Listing 33. Validating access permissions.....	76
Listing 34. Enforcing Smart contract-based Authorizations	78
Listing 35. Implementing Access Control and Event Logging in Smart Contracts	79

List of Acronyms

Abbreviation	Meaning
ABAC	Attribute-Based Access Control
AI	Artificial Intelligence
AutoML	Automated Machine Learning
CNC	Computer Numerical Control
CNN	Convolutional Neural Network
DAL	Data Abstraction Layer
DDM	Decentralized Data Management
DMS	Design Model Storage
DSL	Domain Specific Language
Espace	Experimentation Space
IBFT	Istanbul Byzantine Fault Tolerance
IDE	Integrated Development Environment
JWTs	JSON Web Tokens
LSTM	Long Shot Term Memory
ML	Machine Learning
NN	Neural Network
PAIO	ProActive AI Orchestration
PAPSC	Policy Administration Point Smart Contract
PDPSC	Policy Decision Point Smart Contract
PEP	Policy Enforcement Point
PIPSC	Policy Information Point Smart contract
PoA	Proof of Authority
QBFT	Quorum Byzantine Fault Tolerance
RNN	Recurrent Neural Network
SAL	Scheduling Abstraction Layer (SAL)
UC	Use Case
VP	Variability Points
XML	Extensible Markup Language

Executive Summary

This deliverable documents the intermediate status of the core ExtremeXP framework services responsible for designing, configuring, and deploying experimentation workflows. Moreover, it describes the current status of the secure and distributed management of datasets and knowledge pertaining the experimentation-based workflows, as well as the data authorisation mechanism supporting the ExtremeXP framework. The deliverable includes links to the software code and libraries that implement the aforementioned services. It also documents their implementation and provides usage Information.



1 Introduction

1.1 Objectives

The objective of WP5 is to develop and deploy the core components of the ExtremeXP experiment-driven complex analytics framework. Deliverable 5.2 reports on the intermediate status of the following core services of the framework, which will continue to improve and evolve for the next twenty months:

- (i) Adaptive planning of experiments, i.e., the framework component that enables the modelling of experiments using variants that determine which complex analytics variant should be executed, given a user intent, profile, and context as input (Experiment Modeling block of Figure 1).
- (ii) Selection of best complex analytics variant for each interaction. (Experiment Modeling block of Figure 1)

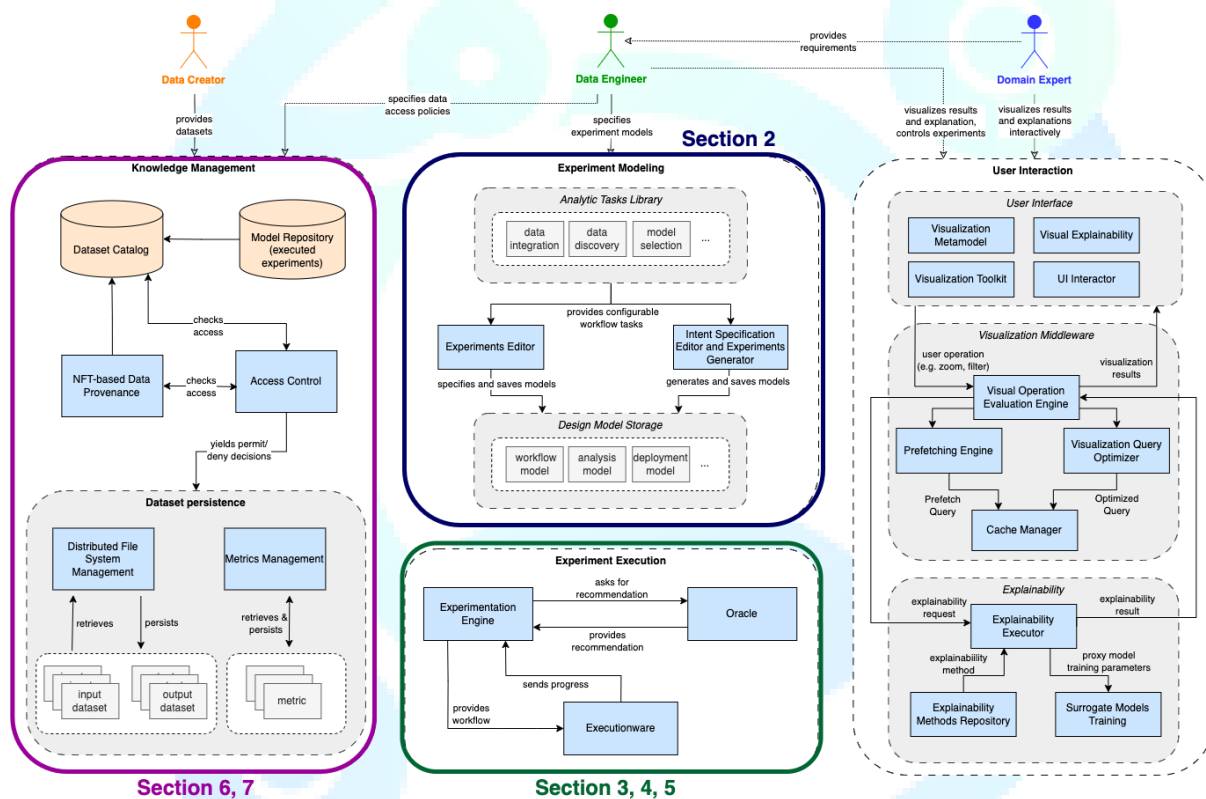


Figure 1. ExtremeXP Architecture and Reported Components

(iii) Monitoring of analytics workflows and users. (Experiment Execution block of Figure 1). Experimentation engine contains the core artifacts of the framework that are related to modelling and planning experiments, enhancing experiment descriptions with context information, scheduling the execution of complex analytics workflows in the available infrastructure, and monitoring their execution as well as their properties (using both system metrics and user metrics).

(iv) Scalable scheduling of analytics workflows, optimally orchestrating the application (workflow) deployment, considering different techniques of virtualisation and heterogenous execution environments. (Experiment Execution block of Figure 1)

(v) Secure management of data and knowledge assets, such as datasets, user profiles, results, learning outcomes, good practices, etc. that are the inputs and outputs of the complex analytics workflow. (Knowledge Management block of Figure 1). Methods, Tools, and Illustrative Example

Adaptive experiment planning is based on the Monitor-Analyse-Plan-Execute over Knowledge loop. An experiment takes the form of a loop in which variants of a complex analytics workflow are monitored and analysed. Both system and user-feedback metrics related to a variant are recorded and plans for the next variant to run are made. Adaptive experiment planning is facilitated by both a graphical editor as well as a Domain Specific Language.

Experiment execution lies at the core of WP5 because it enables mapping between the experiment design and the operational environment for complex analytics. This component is based on Activeeon's ProActive Workflows and Scheduling technology, on the Scheduling Abstraction Layer (SAL) and allows for the execution of workflows in heterogeneous and virtualized execution environments. In this deliverable we also report on the specific APIs developed in the project for the integration with monitoring framework and control access tools in addition to APIs for external integration with other frameworks and tools such as Big Data frameworks.

To enable experiment management and learning from experiment, WP5 develops a secure and distributed knowledge management tool that supports the complete life cycle of knowledge asset management, i.e., knowledge elicitation, representation, persistence, organisation and sharing, and will enable users to access securely relevant know-how to design, perform and evaluate experiments. To facilitate knowledge management, WP5 develops a Data Abstraction Layer, i.e., an interface which unifies communication between the ExtremeXP components and their data persistence technologies as well as the knowledge management tool.

To illustrate the intended purpose and capabilities of the WP5 components, we utilise simple examples from the project Use Case 5 "Failure Prediction in Manufacturing". As previously detailed in D6.1 "Use Case Requirements", this UC aims to investigate anomaly and failure scenarios in a manufacturing setting, by incorporating self-diagnostic procedures. In these examples, we will be showing how adaptive experiment planning can be used to facilitate the development of data-driven analytics experiment workflows, how the experiment execution can enact the various steps of a data-driven analysis such as training and hyper-parameter tuning, as well as how knowledge about the experiment datasets and configurations is managed.

Use Case 5 focuses on a fleet of machines and the running of diagnostic tests on the machines to identify faults by gathering data from a custom testbed. Three types of failures have been characterised by UC5, emerging from faults in different parts of the machine: (i) faults in machine's screw; (ii) faults in the machine's bearings; and (iii) faults in the machine motor. Faults in the machine's screw and bearing constitute a Mechanical anomaly, whereas fault in the machine's motor constitute an Electrical anomaly. To characterize anomalies in both cases, high frequency data are generated periodically by the machine's control and collected at a rapid pace when the machine performs a series of predefined, non-destructive movements.

UC5 has two main objectives:

1. Detect anomalies, given inputs of a mechanical anomaly scenario and specifically resulting from faults in the machine's screw (binary classification problem),

2. Identify among the different types of anomalies, that is mechanical, electrical anomaly or not anomaly at all (multi-class classification problem).

At the time of writing this deliverable only the first objective (binary classification problem) has been in-depth investigated and therefore is reported. The steps comprising this workflow are presented below and shown in Figure 2.

1. Import Data (anomalous and not anomalous data). Advanced machines transmit data using highly accurate embedded sensors. High frequency data are generated periodically by the machine's CNC (computer numerical control) when the machine performs a series of specific predefined movements. This data is often used to know the current "health" state of the machines and to identify issues or faults in the machine. Data used for model training is labelled into anomalous or not anomalous.
2. Add padding. Padding is added to equal the length of the timeseries or to complete incomplete series. This step involves the addition of 0s and NaN values to complete incomplete timeseries and end up with equal-length ones.
3. Split data. This is a generic step before the ML training, where data is divided into a training and testing sets. (Note that this step could introduce some variability in the choice of ratio for the split. Usually a 70% / 30% ratio is chosen for the split).
4. Train a ML model. Currently four ML methodologies have been implemented:
 - a. Neural Network (NN)
 - b. Convolutional Neural Network (CNN)
 - c. Recurrent Neural Network (RNN)
 - d. Long Shot Term Memory (LSTM)
5. ML method testing and evaluation.
6. Visualization of the results.

As described in D2.1 "Initial architecture, languages and models for complex experiment-driven analytics", variability points constitute an aspect of a workflow that can be changed, i.e. different task implementations (such as different ML algorithms), different hyperparameters, or different task inputs (e.g., different datasets). In this specific UC workflow, the variability space includes (i) the different ML algorithms and (ii) the hyperparameters and their optimal value identification (i.e. parameters such epochs and batch size for all four pre-defined ML methodologies that can be tuned).

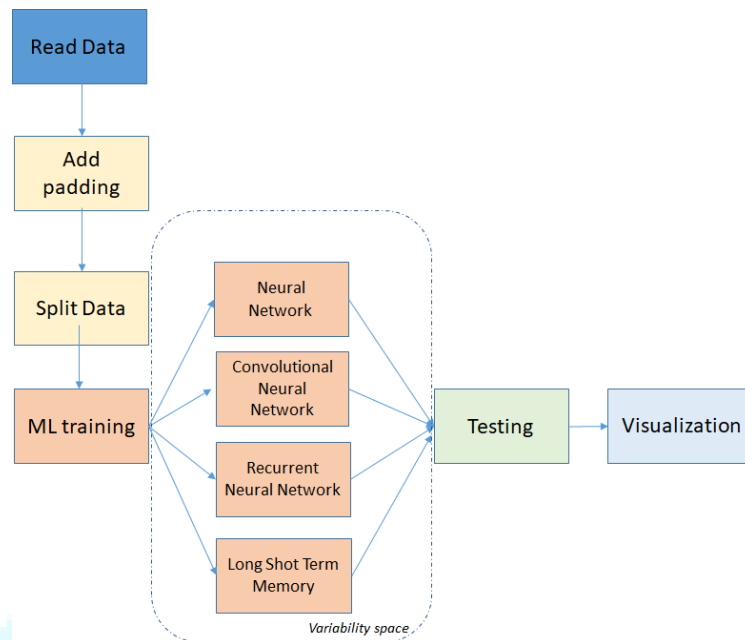


Figure 2. UC5 experiment workflow for the binary classification problem.

1.2 Structure

The rest of the deliverable is structured as follows: **Section 2** presents the current state of the developed experimentation framework overall, together with the created description languages (DSLs) and tools for the preparation and execution of experimentation workflows. It also presents a user-driven view of the tools and the workflows as well as the Graphical User Interface used for modeling experiments in the project. **Section 3** focuses on the experiment planning component of the framework. **Section 4** presents the Data Abstraction Layer. **Section 5** describes the workflow runtime component. **Section 6** presents the adopted Access Control mechanism while **Section 7** describes the Data and Knowledge management component. Finally, **Section 8** draws conclusions and presents our next implementation steps.

1.3 Map of Tools and Prototypes

If not stated otherwise, all tools and prototypes developed or extended within WP5 are available in the ExtremeXP Gitlab organization at <https://colab-repo.intracom-telecom.com/colab-projects/extremexp>. Table 1 provides an overview of them with links to the corresponding sections in this deliverable.

Table 1. Overview of Tools and Prototypes of WP5.

#	Tools/SW Module	Description	Section
1	PWS/PAIO and ProActive Python SDK ¹	Workflow scheduler and Execution Engine - Executionware in Architecture.	5
2	Experimentation Engine	Orchestrates the execution of experiments as specified by their models.	3
3	NFT-based Data provenance	Tags with metadata any input datasets used for experiment-driven analytics	7
4	Access Control	Implementation of the Hyperledger Besu network to host the nodes of the private blockchain that runs the ABAC system's Smart Contracts (4 nodes)	6
5	Middleware to connect Authentication and Authorization modules	PEP middleware with the wrappers for Smart contracts generated with Web3j to connect the Keycloak authentication server to Smart contracts to yield access/deny decisions	6
6	Data Abstraction Layer / IVIS platform	This repository stores data about every executed experiment – in particular, it stores the serialized workflow, the pointers to input datasets (which we consider as versioned and thus immutable), parameters used to parameterize the workflow, and human-in-the-loop inputs during the experiment.	4
7	design-model-storage	Saves EMF models in ExtremeXP framework server	2
8	experiment-dsl, experiment-dsl-cmdline-tools, experiment-dsl-lang-server, extremexp-dsl-framework	These projects hold the dsl examples and related tools (language server, command-line tools for parsing the specification, etc.) A new group should be created on Gitlab to host them.	2
9	Prototype Zenoh Configuration & Rest API	A network of Zenoh nodes, operating in peer-to-peer mode, as well as a Flask-based API on top of the Zenoh network to provide a unified interface for external clients to interact with the components of the Zenoh Network.	7
10	ExtremeXP Graphical DSL editor ²	Holds the graphical editor that can be used for specifying ExtremeXP experiment using the DSL.	2

¹ Available at <https://github.com/ow2-proactive/proactive-python-client>² Available at <https://github.com/ExtremeXP-VU/ExtremeXP-graphical-editor>

2 Experimentation Framework and Domain Specific Language

This section provides an overview of the experimentation framework of ExtremeXP from the perspective of domain-specific languages that the users of the framework (data scientist, domain experts) need to use for specifying and executing their experiments. The methods and processes described in this section are supported by tools #7, #8, and #10 in Table 1.

2.1 Framework overview

Figure 3 shows a user perspective of the current state of the framework. The individual tools and languages are described in the rest of the section.

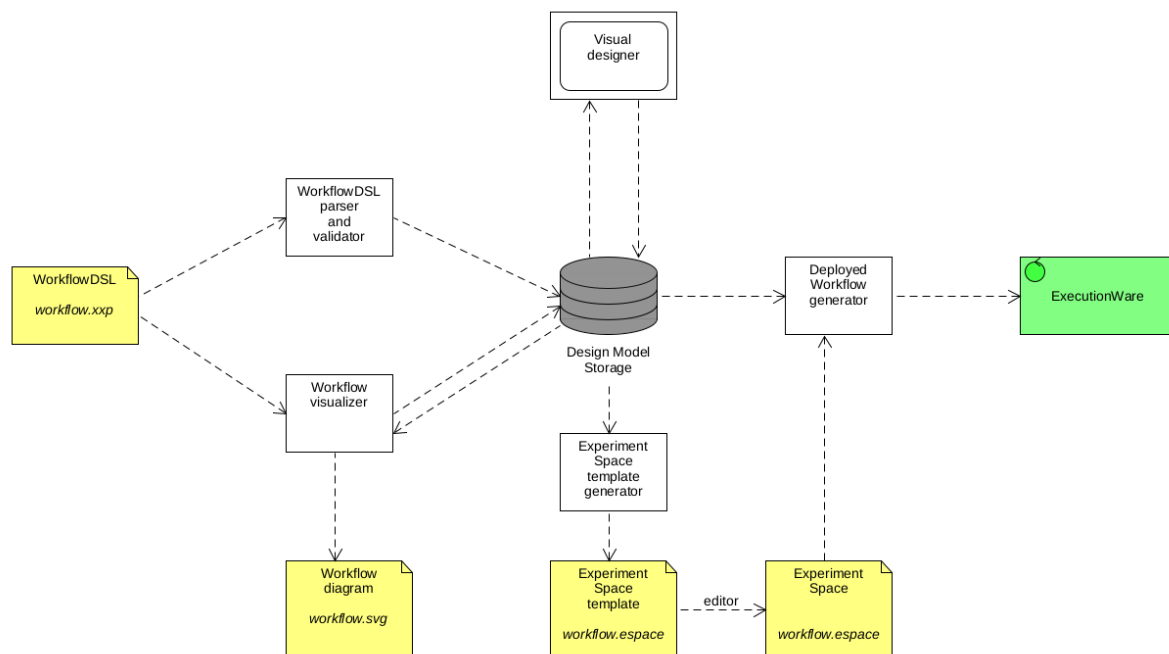


Figure 3. Framework overview

The framework is used as follows: A user creates a workflow. The workflow can be created either in a visual designer tool (Section 2.5) or via a set of command line tools (Section 2.3). Both the visual tool and command line tools will provide the same set of functionalities (the visual tool primarily used for development, command line tools primarily for running tests, repeated executions, etc). All the entities (modelled workflows, tasks, experiment spaces, etc.) are stored in the Design Model Storage (DMS – Section 2.4 and project architecture in Deliverable D2.1). The command line tools primarily work with files containing descriptions in various DSLs. The most prominent is DSL for modelling workflows (Section 2.2.2). Names of the files containing the workflow descriptions are expected to have the **.xxp** extension (an acronym from eXtremeXP).

After a workflow is created, it is verified and stored in DMS. Typically, a single experiment is composed of several (sub)workflows. When all necessary workflows are created, the experimentation space needs to be created – either in visual designer or textually in DSL for describing the experimentation space (Section 2.2.3); the expected file name extension is **.espace**). The experiment space can be seen as a set of all possible configuration values needed for the workflow execution.

A template for the experimentation space is generated automatically from DMS; the user fills in actual values for parameters and other configurations. Then, from the experimentation space, the deployed workflows are generated and submitted for execution.

2.2 DSLs

In this section, the designed DSLs (together with necessary updates of the project's meta-model) are described.

2.2.1 Workflow meta-model updates

The meta-model for describing workflows has been created in the scope of WP2 and thoroughly described in Deliverable D2.1. Nevertheless, during the work on tools, and especially on DSL for modeling workflows (see Section 2.2.2), the need to update the meta-model arose. Namely, these were the needs to allow for explicit development of workflows composed from other existing ones, already stored in DMS (see Section 2.4).

Figure 4 shows the updated Workflows meta-model. For easier orientation, the newly added elements are depicted in red. They are **AssembledWorkflow** and **SubstitutedTask**. Their meaning is as follows.

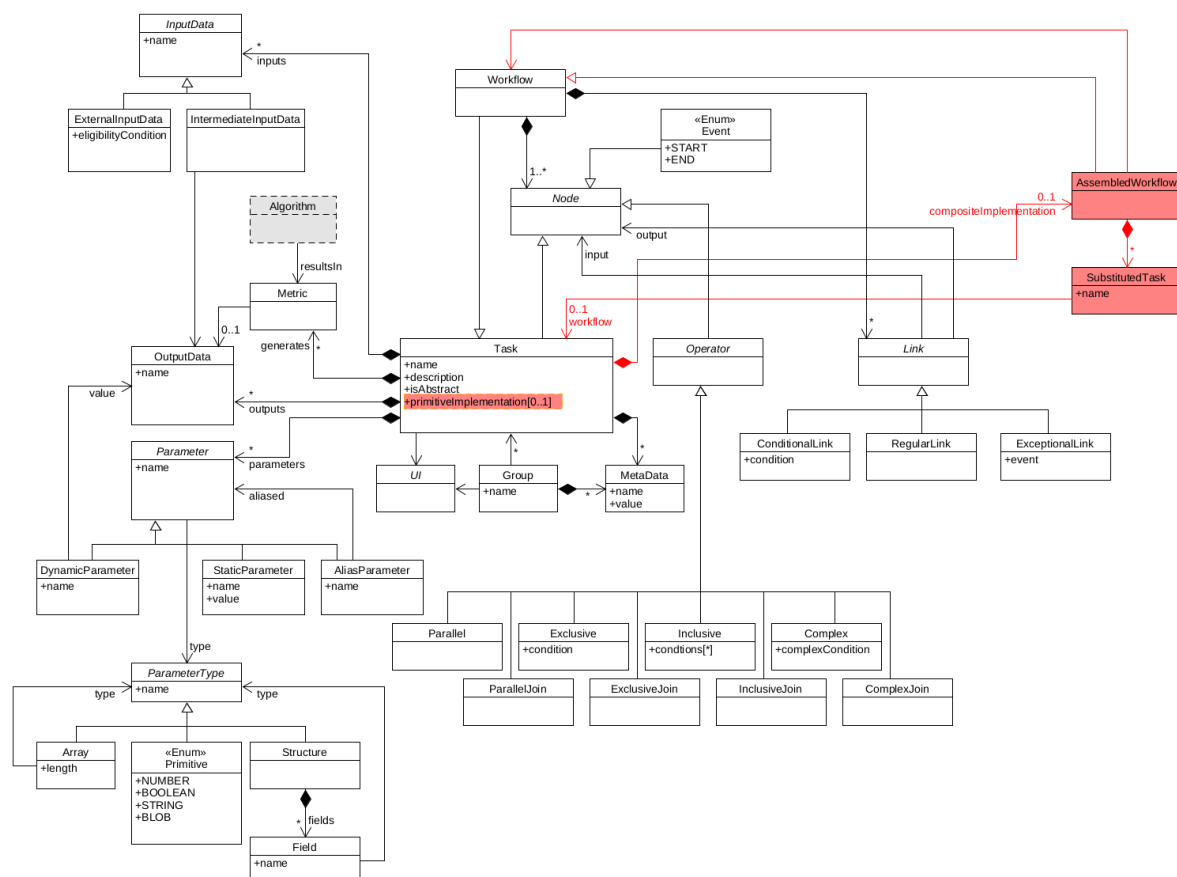


Figure 4. Workflows meta-model

- **AssembledWorkflow**, as the name suggests, represents a workflow assembled from other workflows. It is always based on another workflow, in which the AssembledWorkflow allows designers to configure the tasks – mainly to set the task's implementation (in a workflow, a task's implementation may not be set – it is then an abstract task) and also to set particular values of parameters. This is meant as an extension mechanism which allows providing concrete

parameters to abstract tasks/workflows and allows overriding parameters in non-abstract tasks/workflows.

- **SubstitutedTask** is then the configuration of an abstract task in the workflow.

The **Task** class has been modified around its possible implementation in the following way. The implementation can be either set via the attribute **primitivImplementation** (which would refer to an executable file, services, etc.) or it can be set via the association **compositeImplementation** (which would refer to another workflow). If neither is set, then the task is abstract (and the **isAbstract** attribute is set to true).

2.2.2 DSL for modeling workflows

Even though the modeling of workflows is primarily envisioned via the visual IDE-like tool overviewed in Section 2.5, we have also created a human-friendly textual DSL for modeling workflows in order to speed up the creation of the toolchain (creating tools with text-based input is faster than creating visual tools) and thus allows for earlier results.

We showcase the textual DSL for specifying workflows on the first version of IDEKO's use case (UC5) which deals with binary classification for failure prediction in manufacturing (Section 1). The case consists of five sequential tasks which together constitute the ML training pipeline of IDEKO. In our workflow specification (Listing 1), TrainModel is an abstract task which is only concretized when creating assembled workflows (Listing 2).

Listing 1. An example of a UC 5 (provided by IDEKO) workflow definition in our DSL.

```
package ideko;

workflow BinaryClassification {

    // TASKS
    define task ReadData;
    define task AddPadding;
    define task SplitData;
    define task TrainModel; // abstract task
    define task EvaluateModel;

    configure task ReadData {
        implementation "tasks/IDEKO/read_data.py";
    }

    configure task AddPadding {
        implementation "tasks/IDEKO/add_padding.py";
    }

    configure task SplitData {
        implementation "tasks/IDEKO/split_data.py";
    }

    configure task EvaluateModel {
        implementation "tasks/IDEKO/evaluate_binary_classification.py";
    }

    // DATA
    define data InputData;
    define data RawData;
    define data PaddedData;
    define data TrainingData;
    define data TestData;
```

```
// TASK CONNECTIONS
START -> ReadData -> AddPadding -> SplitData -> TrainModel -> EvaluateModel ->
END;

// DATA CONNECTIONS
InputData --> ReadData --> RawData --> AddPadding --> PaddedData --> SplitData;
SplitData --> TrainingData;
SplitData --> TestData;
TrainingData --> TrainModel --> TrainedModelData;
TestData --> EvaluateModel;
TrainedModelData --> EvaluateModel;
}
```

We believe that the syntax of DSL is relatively self-explanatory, and thus, we only provide an overall description here. The structure is similar to commonly used programming and definition languages in that: (i) blocks are enclosed in curly braces, (ii) statements are terminated by a semicolon, and (iii) comments are either from double-slash till end-of-line or enclosed in slash-star brackets. Below is a list of the keywords and their explanations.

- **package** – defines a namespace
We plan to use the same organization of files with definitions as it is used, for example, in Java. i.e., each workflow is defined in its own file with the same name as the name of the workflow (plus the extension **xxp** – a shortcut for ExtremeXP). The package is then a hierarchical name corresponding to directories in a path where the file is stored.
- **workflow** – defines a workflow
Workflow is a block containing the definition and configuration of nodes, data, links, operators, and parameters of the workflow.
- **define task** – defines a task
- **configure task** – defines the task's configuration
Within the configuration block, the task's parameters and implementation can be defined (the **param** keyword and the **implementation** keyword). The implementation can be (i) a string pointing to an executable file, service URL, etc. (a primitive task), or (ii) a name of another workflow (a composite task). Also, the implementation can be omitted and then the task is abstract (its implementation is set later in an assembled workflow).
- **define data** – defines datasets (inputs/outputs of tasks)
The structure of a dataset is either defined externally (by a schema) or can be defined directly.
- **->** (an arrow) – defines a link between tasks, i.e., control flow between tasks
It is possible to link a sequence of tasks in a single statement.
- **-->** (double arrow) – defines data flow between tasks (aka data links)
The data link is always between a data and task (origin/target depends on whether the data are input or output).
- **START** – a special “singleton” task, where the workflow execution starts.
No link can end in this task.
- **END** – a special “singleton” task, where the workflow execution terminates.
No link can start from this task.

The example in Listing 1 defines a workflow. Listing 2 shows an example of a definition of two assembled workflows. As defined in the meta-model, an assembled workflow is in fact a configuration of another workflow. This is expressed by the **from** keyword followed by the name of the workflow that is configured. The limitation of the assembled workflow is that it cannot contain any new definitions but only configurations (typically implementations of abstract tasks and parameters' values). Listing 2 depicts two possible assembled workflows of UC5 corresponding to two possible

models that can be trained: neural network or recurrent neural network. Technically, this is done by setting the implementation of the TrainModel task to a Python script.

Listing 2. Examples of UC5 assembled workflow definition in DSL.

```
package ideko;

// corresponds to training a neural network (NN) model
assembled workflow BinaryClassification_NN from BinaryClassification {
  configure task TrainModel {
    implementation "tasks/IDEKO/train_nn.py";
  }
}

// corresponds to training a recurrent neural network (RNN) model
assembled workflow BinaryClassification_RNN from BinaryClassification {
  configure task TrainModel {
    implementation "tasks/IDEKO/train_rnn.py";
  }
}
```

To show more examples, Listing 3 presents Use Case 4 (provided by MOBY), modelled using the DSL (the same workflow used as an example in Deliverable D2.1).

Listing 3. Use case 4 modeled in DSL

```
package moby;

workflow MobyWorkflow {

  define task SensorReading;
  define task TripDetection;
  define task TripDetectionUserValidation;
  define task ModeDetection;
  define task ModeDetectionUserValidation;
  define task ActivityDetection;
  define task ActivityDetectionUserValidation;
  define task ModeDetectionClassifierTraining;
  define task ActivityDetectionClassifierTraining;

  configure task SensorReading {
    param samplingFrequency;
    param timeToStopCollectingData;
    implementation "file://sensor-reading.exe";
  }

  configure task TripDetection {
    implementation "file://trip-detection.exe";
  }

  configure task TripDetectionUserValidation {
    implementation "file://trip-detection-user-validation.exe";
  }

  configure task ModeDetection {
    implementation "file://mode-detection.exe";
  }

  configure task ModeDetectionUserValidation {
    implementation "file://mode-detection-user-validation.exe";
  }
}
```

```

configure task ActivityDetection {
    implementation "file://activity-detection.exe";
}

configure task ActivityDetectionUserValidation {
    implementation "file://activity-detection-user-validation.exe";
}

configure task ModeDetectionClassifierTraining {
    param period;
    // no implementation, i.e., it is abstract
}

configure task ActivityDetectionClassifierTraining {
    param period;
    // no implementation, i.e., it is abstract
}

START -> SensorReading -> TripDetection;

define data SensorData {
    schema "file://sensor-data.schema";
    // or define explicitly ... longitude,latitude,altitude,timestamp,speed,...
}

SensorReading --> SensorData;
SensorData --> TripDetection;

TripDetection ?-> SensorReading {
    condition "not exists tripData";
}
TripDetection ?-> PARALLEL_1 {
    condition "exists tripData";
}

define data TripData {
    schema "file://trip-data.schema";
}

TripDetection --> TripData;
TripData --> TripDetectionUserValidation;
TripData --> ModeDetection;
TripData --> ModeDetectionClassifierTraining;

PARALLEL_1 -> EXCLUSIVE_1;
PARALLEL_1 -> ModeDetectionClassifierTraining;
PARALLEL_1 -> ActivityDetectionClassifierTraining;
EXCLUSIVE_1 -> TripDetectionUserValidation;
TripDetectionUserValidation -> EXCLUSIVE_END_1;
EXCLUSIVE_1 -> EXCLUSIVE_END_1 {
    condition "validation not needed";
}

define data ModeData {
    schema "file://mode-data.schema";
}

EXCLUSIVE_END_1 -> ModeDetection -> EXCLUSIVE_2;
ModeDetection --> ModeData;

ModeData --> ActivityDetection;
ModeData --> ModeDetectionUserValidation;
ModeData --> ActivityDetectionClassifierTraining;

```

```

EXCLUSIVE_2 -> ModeDetectionUserValidation -> EXCLUSIVE_END_2;
EXCLUSIVE_2 -> EXCLUSIVE_END_2 {
    condition "validation not needed";
}

define data ActivityData {
    schema "file://activity-data.schema";
}

EXCLUSIVE_END_2 -> ActivityDetection -> EXCLUSIVE_3;
ActivityDetection --> ActivityData;

ActivityData --> ActivityDetectionUserValidation;

EXCLUSIVE_3 -> ActivityDetectionUserValidation -> EXCLUSIVE_END_3;
EXCLUSIVE_3 -> EXCLUSIVE_END_3 {
    condition "validation not needed";
}

EXCLUSIVE_END_3 -> PARALLEL_END_1;

group UserPerformed {
    TripDetectionUserValidation;
    ModeDetectionUserValidation;
    ActivityDetectionUserValidation;
}

ModeDetectionClassifierTraining ?-> ModeDetectionClassifierTraining {
    condition "retrain needed";
}
ModeDetectionClassifierTraining -> PARALLEL_END_1;
define data MLModelMode {
    schema "file://ml-model-mode.schema";
}
ModeDetectionClassifierTraining --> MLModelMode;
MLModelMode --> ModeDetection;

ActivityDetectionClassifierTraining ?-> ActivityDetectionClassifierTraining {
    condition "retrain needed";
}
ActivityDetectionClassifierTraining -> PARALLEL_END_1;
define data MLModelActivity {
    schema "file://ml-model-activity.schema";
}
ActivityDetectionClassifierTraining --> MLModelActivity;
MLModelActivity --> ActivityDetection;

PARALLEL_END_1 -> END;
}

```

Listing 3 showcases some more keywords and constructs of our DSL, namely:

- **?->** – defines a conditional link between tasks (a special type of control flow)
After the definition of a conditional link, there should be a condition defined (in a case, the condition can be deduced from another conditional link starting in the same task, then the condition can be omitted).
- **!->** – defines an exceptional link between tasks (a special type of control flow)
After the definition of an exceptional link, an exception must be defined.
- **PARALLEL_x**, **PARALLEL_END_x**, **EXCLUSIVE_x**, **EXCLUSIVE_END_x**, **INCLUSIVE_x**, **INCLUSIVE_END_x**, **COMPLEX_x**, **COMPLEX_END_x** – define operators (as they are defined in the meta-model)

The operators can be considered in the scope of DSL as specialized tasks with predefined semantics. The “**x**” is a number to distinguish a particular instance of the operator. The “**_END_x**” variant is the corresponding end of a particular operator.

- **group** – defines a group of several tasks.

Similar to Listing 2, Listing 4 defines an assembled workflow for UC4 which configures two abstract tasks by setting their implementation to concrete (sub)workflows (instead of paths executables used in Listing 2).

Listing 4. An example of an UC4 assembled workflow definition in DSL.

```
package moby;

workflow MobyWorkflowAssembly1 from MobyWorkflow {
  configure task ModeDetectionClassifierTraining {
    implementation ModeDetectionClassifierTrainingImpl;
  }

  configure task ActivityDetectionClassifierTraining {
    implementation ActivityDetectionClassifierTrainingImpl;
  }
}
```

2.2.3 DSL for modeling experiments

The DSL shown in the previous section describes a workflow of an experiment. But to run an experiment, it is necessary to run the workflow with different settings (different values for parameters, etc.), i.e., it is necessary to define an experimentation space. The meta-model of the experimentation space is also defined in Deliverable D2.1. As with the workflow specifications, it is envisioned that the experimentation space will be defined visually in an IDE-like tool. Nevertheless, we created a textual-based DSL for the experimental space definition to speed up the development.

Listing 5. An example in Experimentation space DSL for UC5.

```
espace NNExpSpace of BinaryClassification_NN {

  configure SELF {
    define method object G as gridsearch;
    G["epochs_vp"] = G.values(50, 100);
    G["batch_size_vp"] = G.values(64, 128);
  }

  task ReadData{
  }
  task AddPadding{
  }
  task SplitData{
  }
  task TrainModel{
    param epochs = G["epochs_vp"];
    param batch_size_vp = G["batch_size_vp"];
  }
}
```

Listing 5 shows an example of the experimentation space (espace). The overall syntax is based on the same style as in the workflow DSL (blocks in curly braces, etc.).

A template for the space is generated from a particular assembled workflow (see tools in Section 2.3); the end user only fills in values for parameters, etc. The assembled workflow for which the space is generated is referenced after the **of** keyword. In this example, the assembled workflow is “BinaryClassification_NN” depicted in Listing 2. A similar space could be specified also for the “BinaryClassification_RNN” of Listing 2.

As the set values depend on the method by which the parameter space will be searched (and thus individual workflows executed), the **method object** has to be defined with the space. It is done in the configure SELF block, which provides configuration for the whole space. We expect the existence of multiple method objects providing implementations of different kinds of searches, e.g., gridsearch, randomsearch, Bayesian optimisation. As the parameter value specifications directly depend on the chosen method, the method objects provide appropriate means. In the listing, the chosen method is grid search, and the values of parameters are provided as lists. In the case of random search, the values would be provided as intervals. On the implementation level, the method object will be a Python (or another programming language) object.

Listing 6 provides another example of an espace, now in UC4.

Listing 6. An example in Experimentation space DSL for UC4.

```
espace MobyWorkflowExpSpace of MobyWorkflowAssembly1 {

  configure SELF {
    define method object M as gridsearch; // randomsearch, bayesianoptimization,...
    M["samplingFrequency"] = M.values(1, 5, 10);
    M["period"] = M.values(1, 5, 10);
  }

  task SensorReading {
    param samplingFrequency = M["samplingFrequency"]
    param timeToStopCollectingData = 200;
  }
  task TripDetection {
  }
  task TripDetectionUserValidation {
  }
  task ModeDetection {
  }
  task ModeDetectionUserValidation {
  }
  task ActivityDetection {
  }
  task ActivityDetectionUserValidation {
  }

  task ModeDetectionClassifierTraining {
    param period = M["period"]

    task DataPreprocessing {
    }
    task Labeling {
    }
    task FeatureExtraction {
    }
    task ModelTrainingAndEvaluation {
    }
  }

  task ActivityDetectionClassifierTraining {
    param period = M["period"]
  }
}
```

```
task DataPreprocessing {  
}  
task Labeling {  
}  
task FeatureExtraction {  
}  
task ModelTrainingAndEvaluation {  
}  
}  
}
```

2.3 Command-line tools

Together with the textual DSLs shown in the previous section, we have developed corresponding tools to process them. At this point, the tools are created as command-line ones; the same functionality is being integrated into web-based IDE. The currently available and planned tools are as follows.

2.3.1 Workflow DSL visualizer

The first version of the visualizer is available³. It internally uses the dot language (a part of Graphviz⁴). It takes the workflow in DSL and generates an SVG⁵ image. Figure 5. Visualization of Use Case 5 shows the generated visualization of the workflow in Listing 1.

³ <https://colab-repo.intracom-telecom.com/colab-projects/extremexp/experiment-modelling/experiment-dsl-cmdline-tools>

⁴ <https://graphviz.org/>

⁵ <https://www.w3.org/Graphics/SVG/>

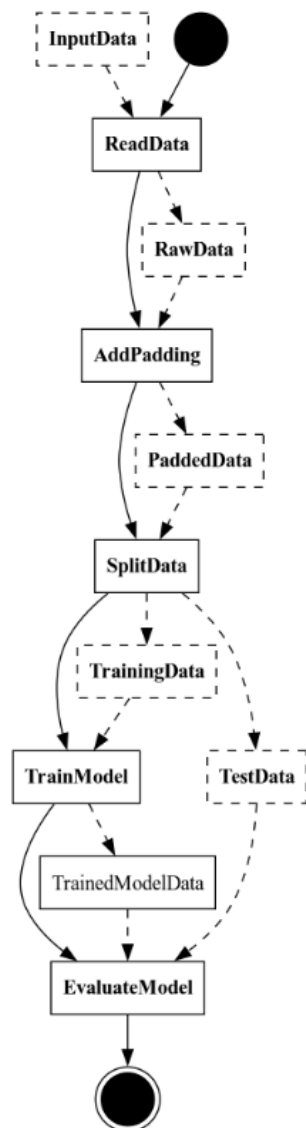


Figure 5. Visualization of Use Case 5

Figure 6 shows the generated visualization of the workflow in Listing 3.

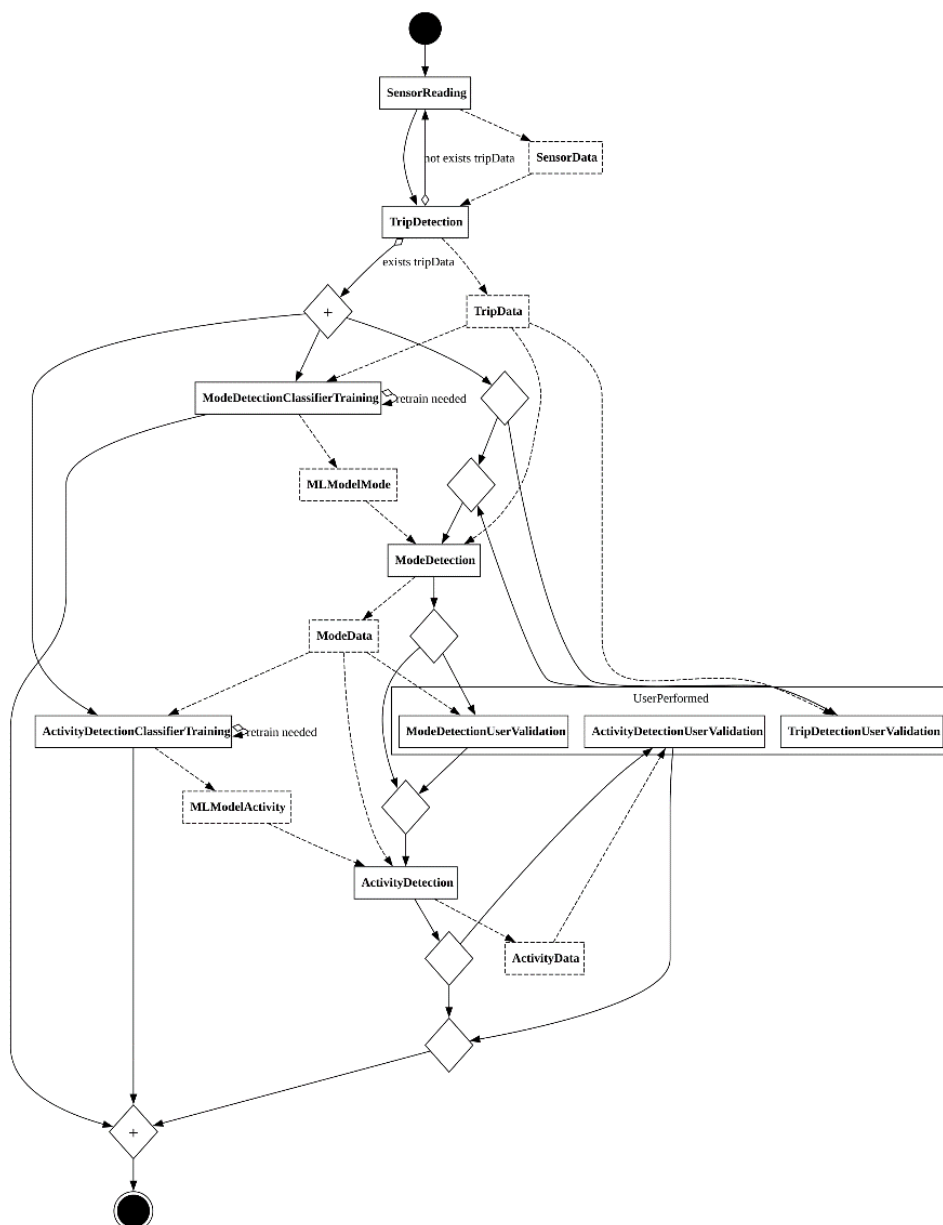


Figure 6. Visualization of the workflow (Use Case 4)

2.3.2 Validator of workflows

The validator is currently under development. It validates that the workflow is correctly designed and corresponds to the meta-model and its semantics (as it is defined in Deliverable D2.1).

2.3.3 Generator of espace

The generator is currently under development. It takes an assembled workflow and generates a template for espace.

2.4 Design Model Storage (DMS)

The tools above (and also future tools and visual tools) store the workflows, espaces, etc. in DMS and also can reuse entities already stored there. DMS is a file-based database. The entities stored there are in a path corresponding to their full name (package and name). For example, the full name of the workflow in Listing 3 is **moby.RunningExample**. Within DMS, this workflow is stored in the **moby/RunningExample/** directory. Within this directory, multiple files can be stored corresponding to different representations of the workflow (directly DSL, Ecore file generated from DSL, etc.). For tools, DMS will be accessible via a URL. For Java-based tools, a straightforward implementation is already available⁶.

2.5 Graphical editor for experiment specification

Apart from the textual DSLs and related command-line tools reported in the previous subsections, we have been developing a graphical editor for specifying workflows and experiments which takes the form of a web application (built using React). The web application comprises four primary pages: the user login page, dashboard page, editor page, and execution page, as depicted in Figure 7. The structure of the graphical editor.

1. The user login view facilitates user registration and authentication, enabling access to the graphical editor.
2. The dashboard functions as an artifact repository, permitting users to manage collections of specifications for experiments and tasks, as well as configure user settings.
3. The editor serves as the central component of the web application, allowing users to create and modify the layout of an experiment or task workflow in edit mode, and configure parameters and variability points of configurable nodes (e.g., task nodes, data nodes, operators) in configuration mode.
4. The execution view presents validation results of the workflow and execution properties.

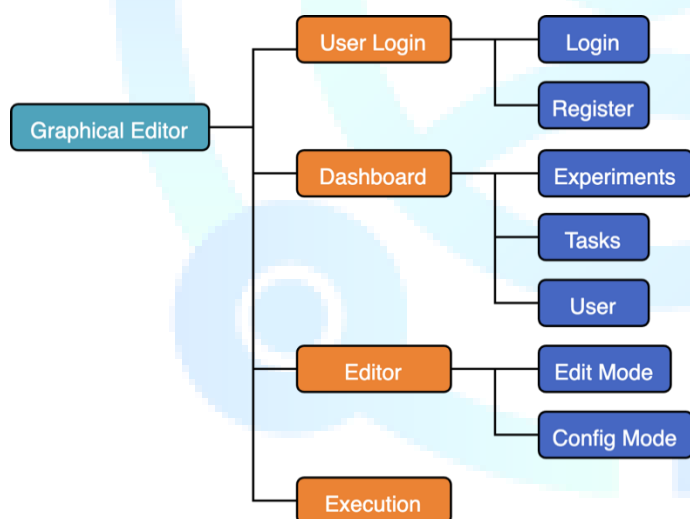


Figure 7. The structure of the graphical editor.

Figure 8. The user flow of the graphical editor. illustrates the user flow depicting navigation between various primary and sub-pages. Upon entering the web application, the system verifies the login

⁶ <https://colab-repo.intracom-telecom.com/colab-projects/extremexp/experiment-modelling/design-model-storage>

status. If the user is already authenticated, they are directed to the dashboard; otherwise, they must either log in via the login page or register for a new account through the registration page. Once logged in, users cannot reaccess the user login page unless they log out by clicking the sign-out button on the dashboard.

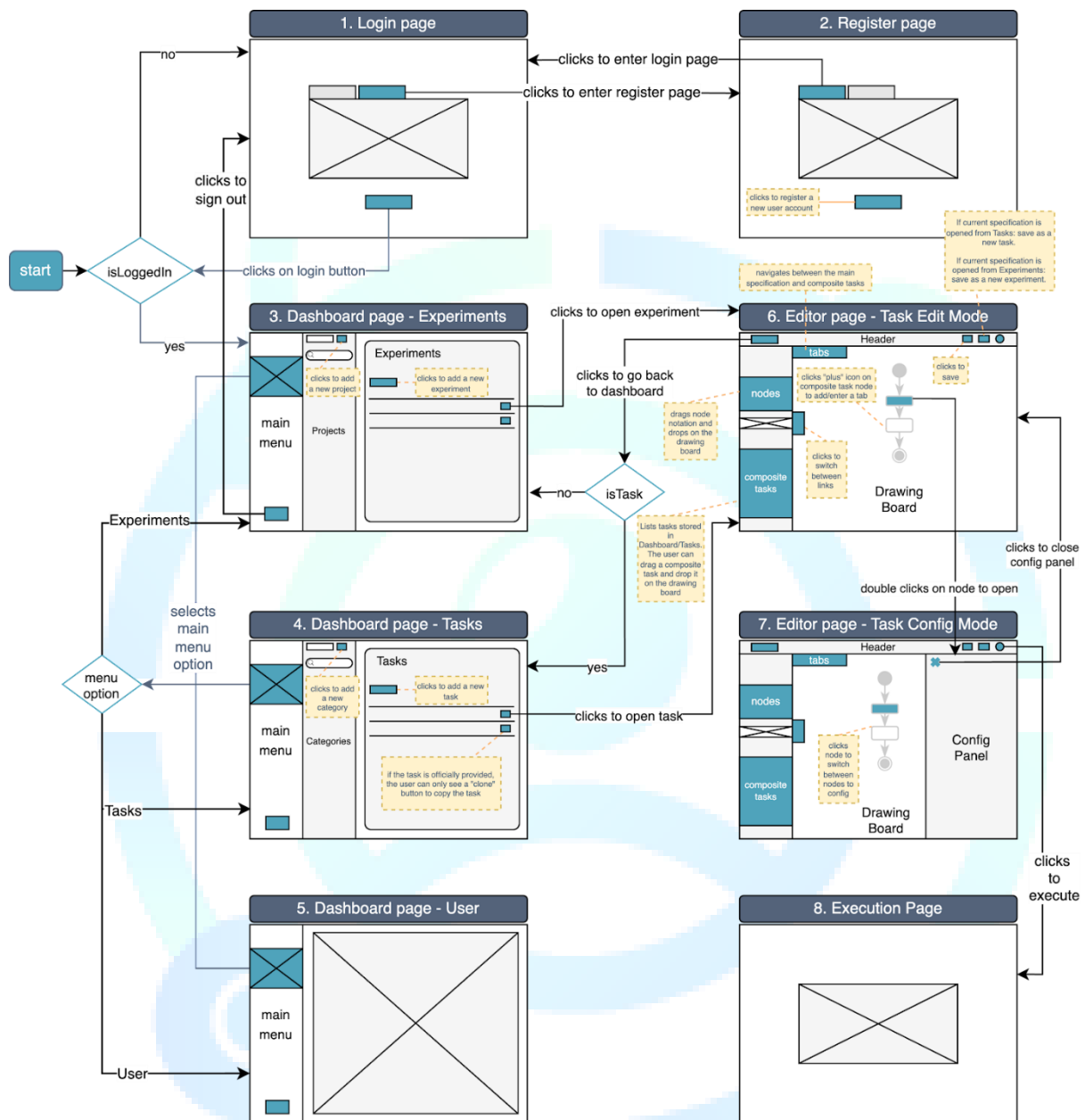


Figure 8. The user flow of the graphical editor.

Within the dashboard, users navigate between subpages using the main menu options. Users can open to edit an experiment or task specifications from the dashboard, leading them to the editor page. The editor page distinguishes between experiment and task specifications. If an experiment specification is opened, clicking the "back to dashboard" button returns the user to the experiments subpage of the dashboard. For task specifications opened in the editor, the "go back" button directs the user back to the tasks subpage of the dashboard.

The editor page offers two modes: the edit mode and the config mode, designed for specification workflow modelling and configuration, respectively. In the current implementation, users can access the configuration panel by double-clicking on a configurable node and closing it by clicking the close button within the panel.

Execution can be initiated from the editor page by clicking the execution button, although the execution page itself has not been implemented at this stage.

In the following, we showcase the functionalities offered so far by the graphical editor.

2.5.1 Login

The login page (Figure 9. The login page.) validates user input for the necessary login information. To create a new account, users are required to provide only a username and password. Attempts to create a new user account using an existing username will be detected and rejected.

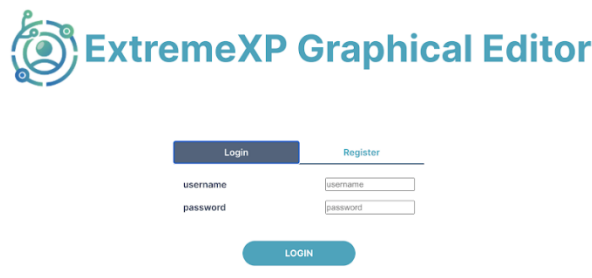


Figure 9. The login page.

2.5.2 Dashboard

2.5.2.1 Experiments

The user can manage experiment projects by creating, deleting, and editing project information, as well as related experiment specifications, on the experiments subpage (refer to Figure 10. Dashboard experiments page.). As the user flow section mentions, users can navigate to other subpages by selecting different options on the main menu.

On the secondary menu on the right-hand side of the main menu, users can create a new project (i.e. a collection of experiments) by entering the project name and clicking the "create" button. The search bar beneath enables users to search for projects based on their names. Following the search bar is a list of projects for users to select and review experiments associated with a specific project.

Within the project content board, users can edit project information, including the project name and description, and delete the entire project. To create a new experiment within the project, users have two options: 1) click the "new experiment" button to create an empty experiment; 2) import an experiment from a JSON file stored in the local file system. For each experiment, users can edit the experiment name, delete the experiment, and download it. The downloaded experiment contains only the graphical model information, which can subsequently be imported as a new experiment.

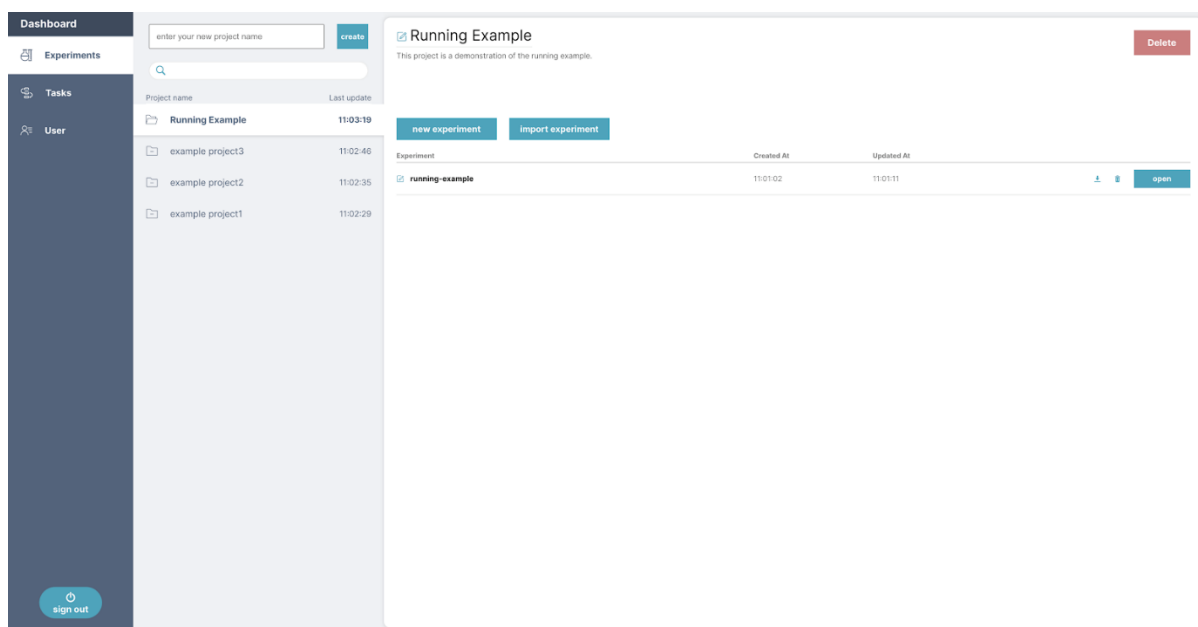


Figure 10. Dashboard experiments page.

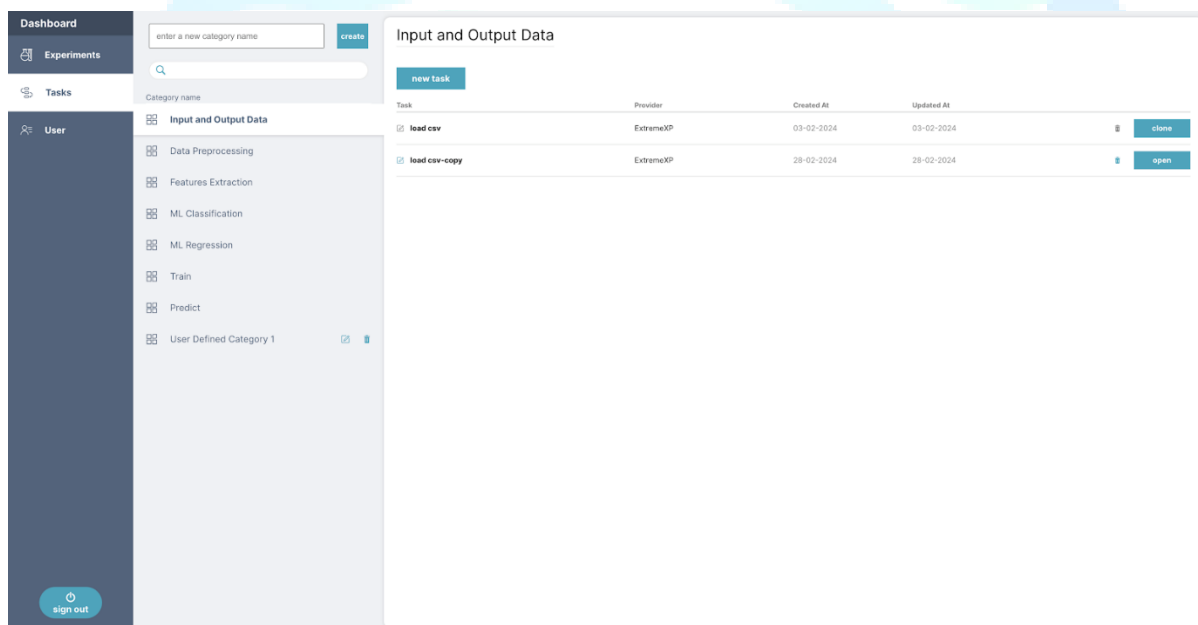


Figure 11. Dashboard tasks page.

2.5.2.2 Tasks

In contrast to experiments, tasks exhibit reusability and can be pre-defined. From the standpoint of task providers, two general types of tasks are discernible: 1) user-defined tasks; and 2) non-user-defined tasks.

On the secondary menu within the tasks sub-page (Figure 11), tasks are categorized based on their utility. Certain categories are pre-established, making them immutable to user modification. However, users retain the liberty to define their own categories, which are fully customizable. Within each predefined category, predefined tasks are presented, and supplied by the graphical editor (and potentially other third-party providers in the future).

Direct modification of predefined tasks is prohibited for users; instead, they can clone a task to render it editable in terms of name modification, and subsequently access the specification graphical model on the editor page.

2.5.3 Editor

2.5.3.1 Editor Overview

The editor, depicted in Figure 12, comprises a header at the top, an entity panel on the left, and a drawing board in the center. Additionally, there is a configuration panel on the right, which is hidden by default.

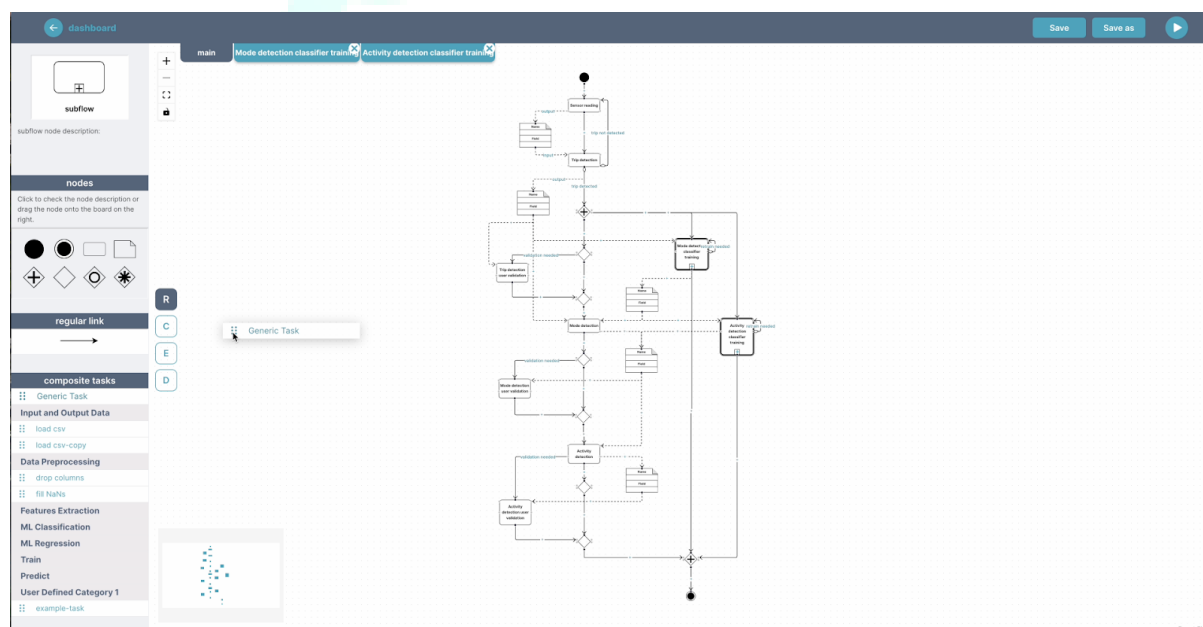


Figure 12. Editor overview.

The header facilitates navigation with a "go back to dashboard" button, along with options to save, save as, and execute. Users can preserve the current state of the graphical model of an experiment or task specification displayed on the drawing board by clicking the save button or utilizing the shortcut "Command + S" for MacOS, or "Ctrl + S" for Windows and Linux. Upon successful completion of the save process, a "Saved" message will appear in the center of the window (Figure 13).

To save the model as a new experiment or task, users can utilize the save as button. In the ensuing popup dialog window, users can modify the name of the duplicated specification (Figure 14).

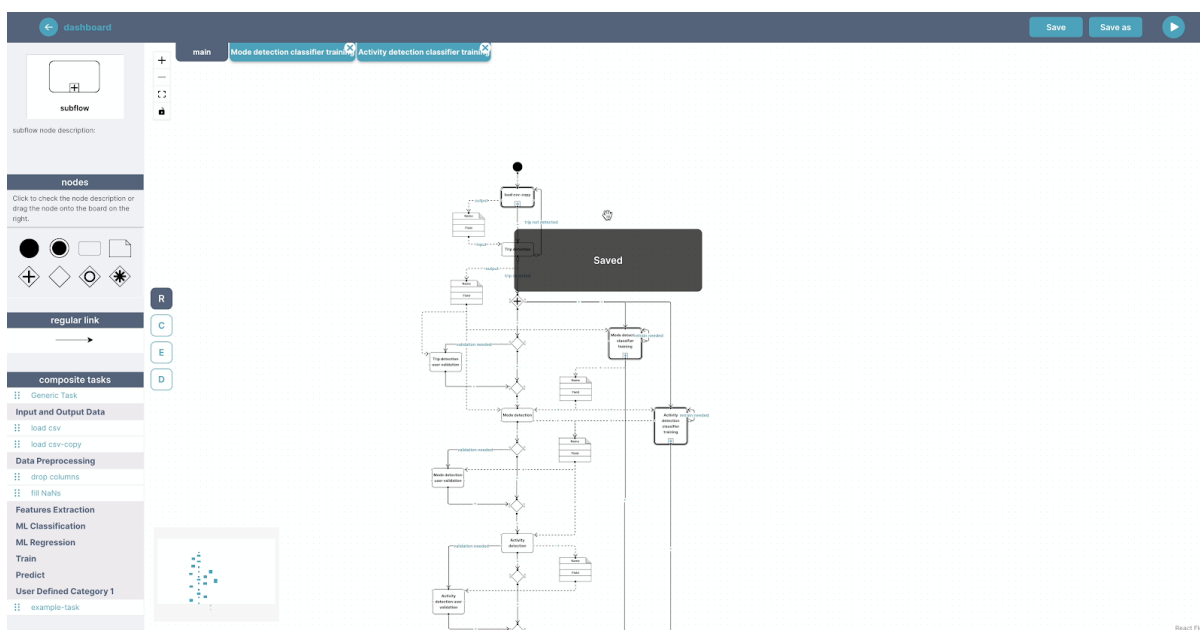


Figure 13. "Save model" functionality.

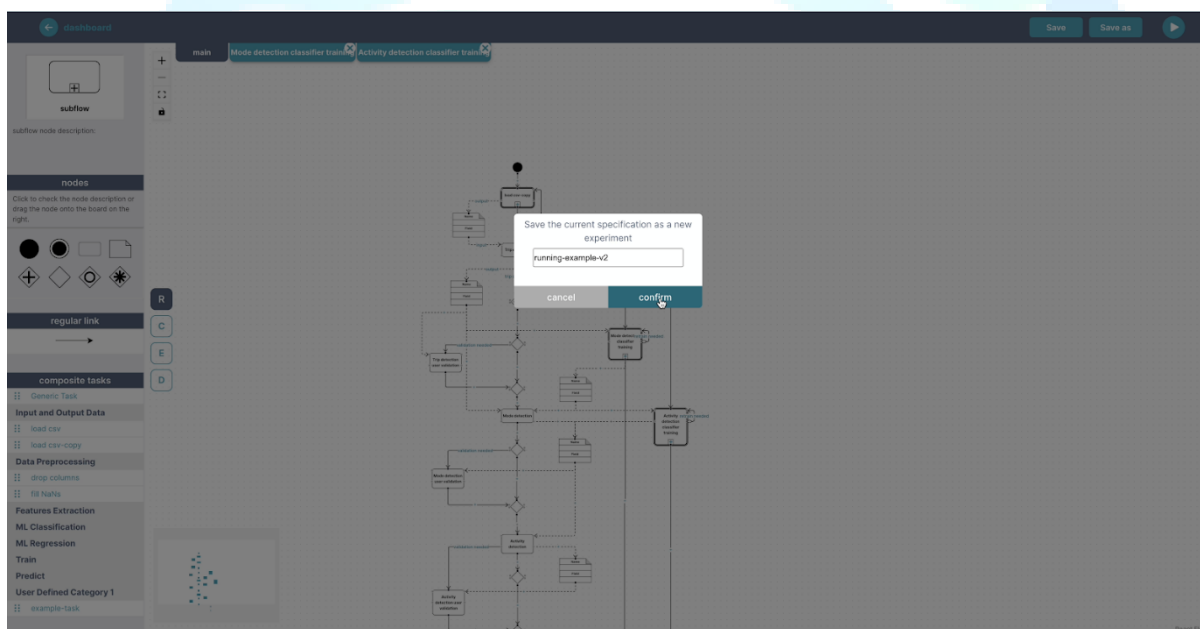


Figure 14. "Save as" functionality.

The entity panel situated on the left provides node notations, allowing users to drag and drop them onto the drawing board to add specific nodes. At the top of the entity panel resides a node window, which displays the currently selected node, along with its name and description. Beneath the node's notation list is the current selected link window, while the link selection panel is positioned on the right-hand side. Users can switch between links used to establish connections between nodes by selecting different options on the link selection panel.

Beneath the current link window lies the composite tasks list, comprising tasks containing a nested task workflow. All tasks stored in the Tasks sub-page from the dashboard are available here. Additionally, there is a "generic task" option, housing a composite task featuring one start node, one

task node, and one end node. Users can drag a composite task from the list and drop it onto the drawing board to add it as a composite task.

2.5.3.2 Editor Tabs

Below the header, a list of navigable tabs is presented. The main tab, always present, defaults to displaying the entire graphical model. To edit the graphical model associated with a composite task, users can click the plus icon on the composite task node, thereby creating a new tab corresponding to that composite task. Subsequently, users may access this tab by either clicking on it directly or selecting the plus icon on the respective task node once more. Figure 15. Composite task editing, illustrates an example of editing a composite task within the tab titled "generic task". The sub-tabs share the same functionalities offered in the main tab.

Upon closing a tab, users are redirected to the main tab. Furthermore, if the node associated with a tab is deleted, the tab is automatically removed.

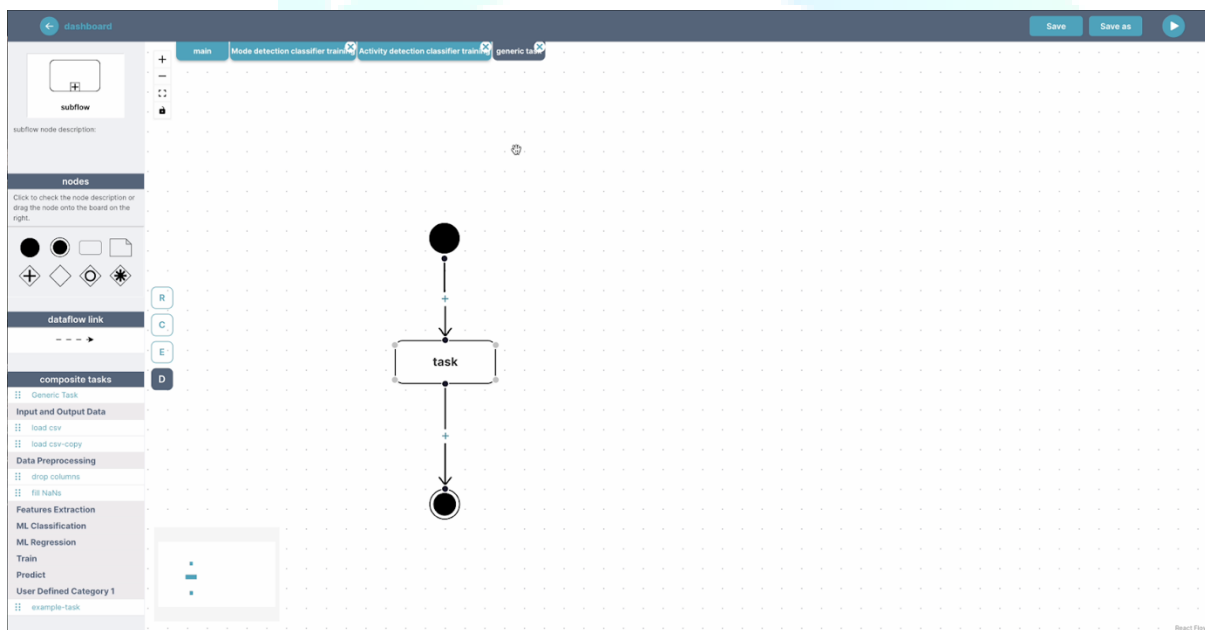


Figure 15. Composite task editing.

2.5.3.3 Configuration

The user can append a text label to each link by clicking the plus icon situated at the midpoint of the link's path. An illustration of label editing is depicted in Figure 16.

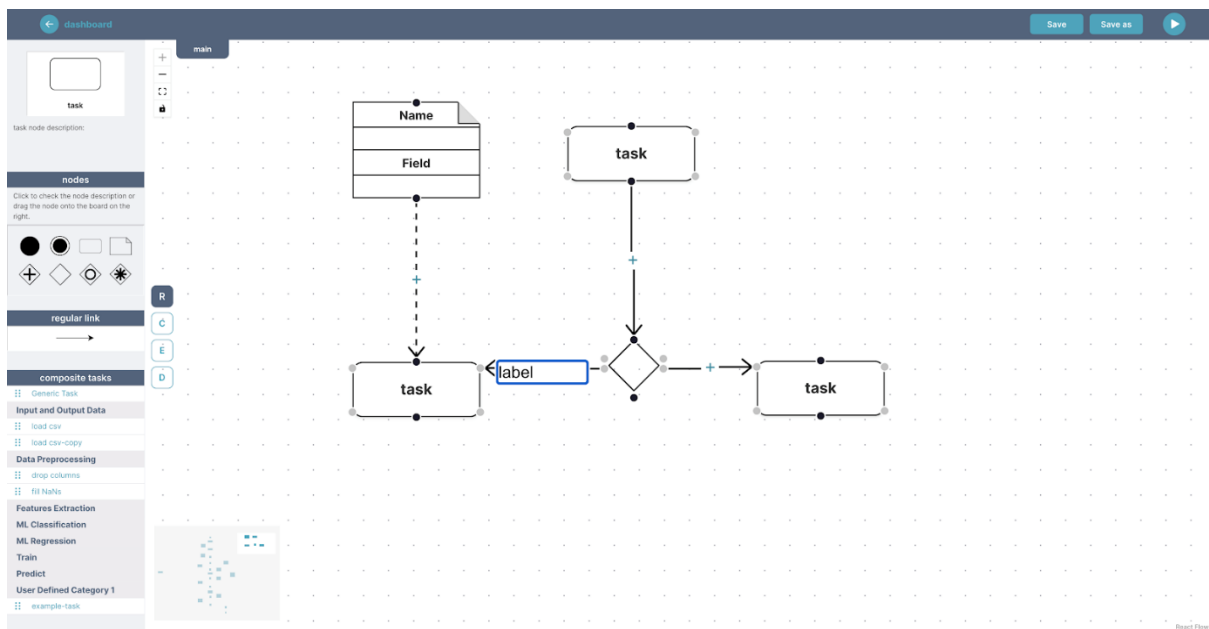


Figure 16. Label editing.

Upon double-clicking on a task node (and likewise for data nodes and operators), the configuration panel will be presented. Within this panel, users can modify the generic properties of a task node, including its name, description, isAbstract, category, and so forth. Parameters can be added and set as variability points and configured concerning various value types, such as ranges, strings, integers, and arrays. Figure 17 provides an example of name editing for the task node "Sensor reading". Users can seamlessly switch between configurable nodes while maintaining the configuration panel open by single-clicking on different nodes. The currently selected node will be distinguished by a green background color.

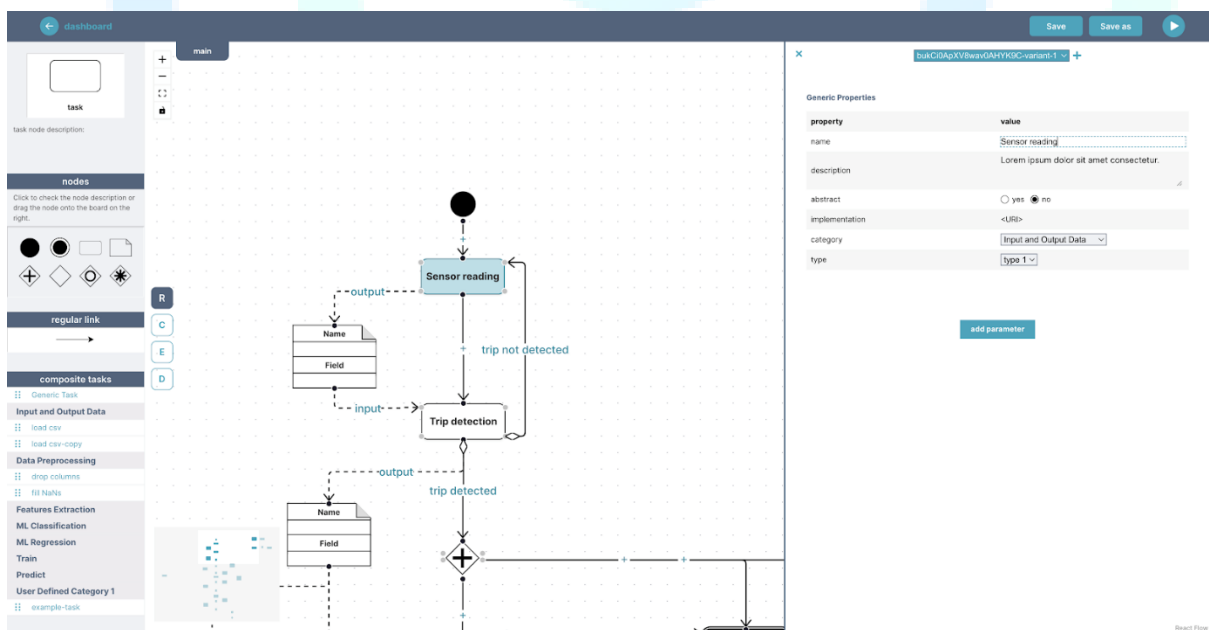


Figure 17. Task configuration panel - name editing.

A task can be conceptualized as a variability point as well. Within the configuration panel, users are presented with the capability to add a task variant (Figure 18). This variant task may be either a

standard task or a composite task. The composite task options encompass tasks sourced from the Tasks repository. Subsequently, upon adding a task variant, users can designate the variant task for display on the drawing board (Figure 19). In the event that the selected variant is a composite task, users have the opportunity to create a new tab and access it for editing purposes.

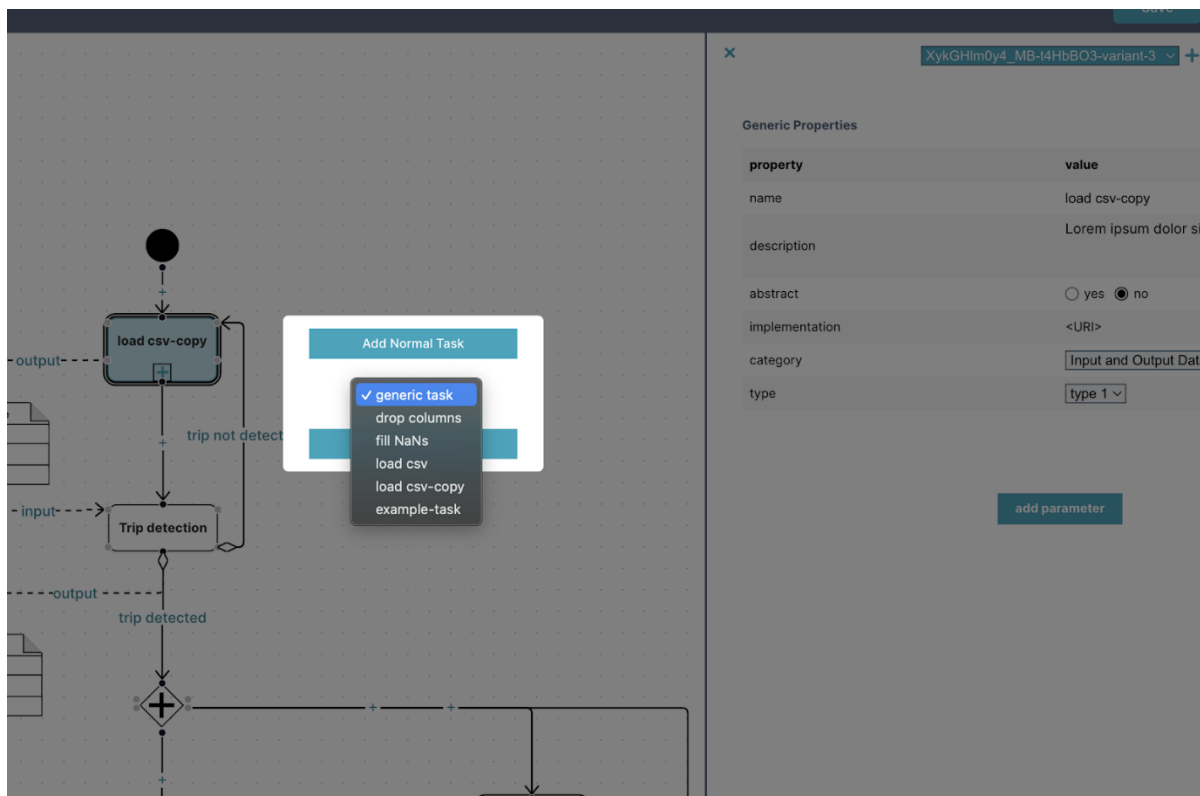


Figure 18. Add task variant.

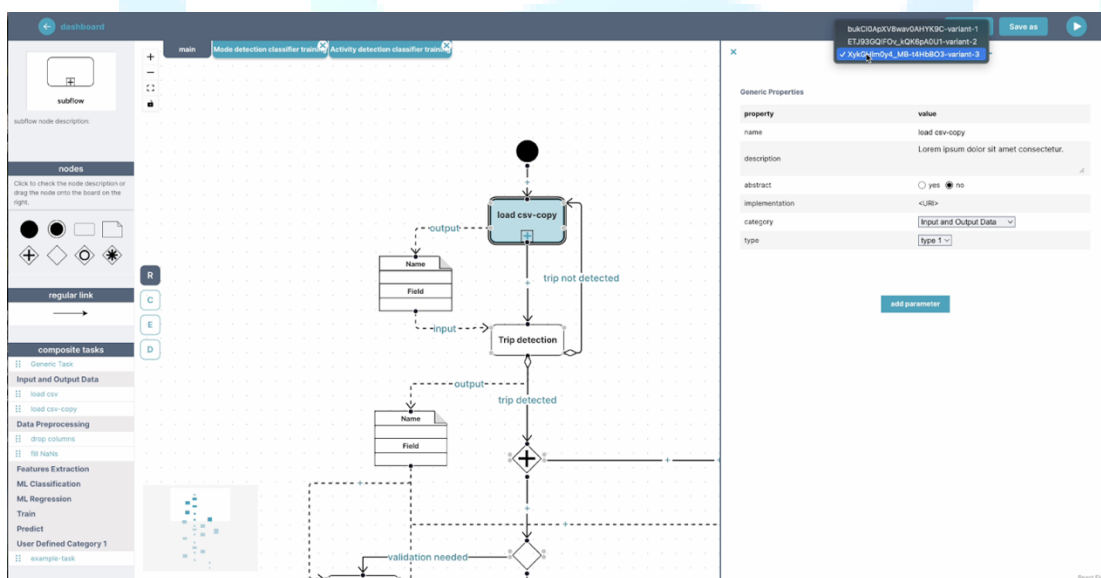


Figure 19. Task variant selection.

3 Continuous Adaptive Experiment Planning

Once an ExtremeXP experiment has been specified using the DSLs and corresponding tools described in Section 2, it is submitted to the Experimentation Engine, the “heart” of the Continuous Adaptive Experiment Planning module of the ExtremeXP framework. This module interprets the experiment specifications and creates concrete workflows to run for each experiment. It is responsible for instantiating experiments, submitting workflows to the framework’s executionware (Activeeon’s Proactive, Section 5), retrieving results, and comparing results from different workflow executions.

The following sections explain the code responsible for parsing the DSL, reading parsed workflows, creating assembled workflows, generating deployed workflows, and subsequently executing them using the Executionware.

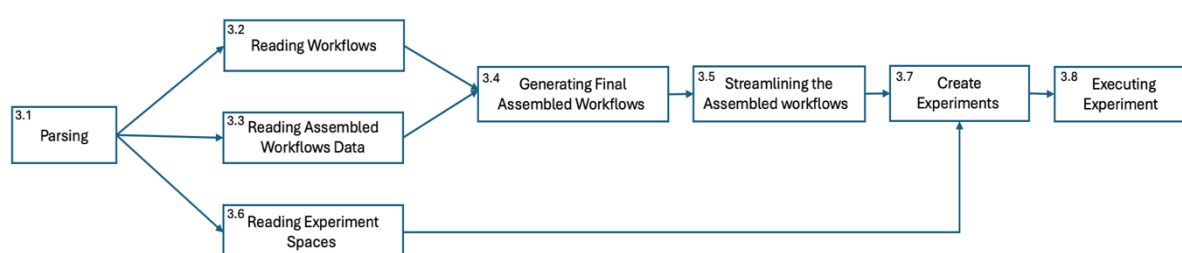


Figure 20. Overview of Continuous Adaptive Experiment Planning Module

3.1 Parsing

The Python library textX⁷ is used to construct the grammar and domain-specific language (DSL). The code reads the grammar rules from the file 'workflow_grammar.tx' to generate a metamodel (Listing 7). This metamodel acts as a blueprint for parsing text conforming to the DSL defined by those rules.

In particular, 'IDEKO-simple.dsl' is depicted in Listings 1, 2, and 5 in Section 2, while 'workflow_grammar.tx' is depicted in Appendix 2.

Listing 7. Code segment to read DSL file and create a meta model

```

with open('dsl/IDEKO-simple.dsl', 'r') as file:
    workflow_code = file.read()

workflow_metamodel = textx.metamodel_from_file('dsl/workflow_grammar.tx')
workflow_model = workflow_metamodel.model_from_str(workflow_code)
  
```

3.2 Reading Workflows

This code segment depicted in Listing 8 is responsible for parsing workflows and their elements from the DSL definition. An empty list named 'parsed_workflows' is created to store parsed workflow instances. The code iterates over each workflow defined in 'workflow_model', creating a 'Workflow' instance for each and appending it to the 'parsed_workflows' list. The code examines various elements

⁷ <https://textx.github.io/textX>

within each workflow, such as tasks, data, task configurations, events, and links, and handles them accordingly.

In addition, the code applies task dependencies and sets the order of tasks within each workflow, ensuring proper sequence in the workflow execution. The function 'apply_task_dependencies_and_set_order' is responsible for applying task dependencies and setting the execution order for tasks within a workflow. This function iterates over tasks within a workflow, checks if there are any dependencies associated with each task, and assigns them accordingly. Finally, it rearranges tasks within the workflow based on their dependencies, ensuring tasks are executed in the correct order.

Listing 8. Code segment to read workflows from the DSL specifications.

```
parsed_workflows = []
for workflow in workflow_model.workflows:
    wf = Workflow(workflow.name)
    parsed_workflows.append(wf)

    task_dependencies = {}

    for e in workflow.elements:
        if e.__class__.__name__ == "DefineTask":
            task = WorkflowTask(e.name)
            wf.add_task(task)

        if e.__class__.__name__ == "DefineData":
            ds = WorkflowDataset(e.name)
            wf.add_dataset(ds)

        if e.__class__.__name__ == "ConfigureTask":
            task = wf.get_task(e.alias.name)
            if e.workflow:
                task.add_sub_workflow_name(e.workflow.name)
            elif e.filename:
                if not os.path.exists(e.filename):
                    raise dsl_exceptions.ImplementationFileNotFound(f"{e.filename} in
task {e.alias.name}")
                task.add_implementation_file(e.filename)
            if e.dependency:
                task.input_files.append(e.dependency)

        if e.__class__.__name__ == "ConfigureData":
            ds = wf.get_dataset(e.alias.name)
            ds.add_path(e.path)

        if e.__class__.__name__ == "StartAndEndEvent":
            process_dependencies(task_dependencies, e.nodes, "StartAndEndEvent")

        if e.__class__.__name__ == "StartEvent":
            process_dependencies(task_dependencies, e.nodes, "StartEvent")

        if e.__class__.__name__ == "EndEvent":
            process_dependencies(task_dependencies, e.nodes, "EndEvent")

        if e.__class__.__name__ == "TaskLink":
            process_dependencies(task_dependencies, [e.initial_node] + e.nodes,
"TaskLink")

        if e.__class__.__name__ == "DataLink":
            add_input_output_data(wf, [e.initial] + e.rest)

    apply_task_dependencies_and_set_order(wf, task_dependencies)
```



```
def apply_task_dependencies_and_set_order(wf, task_dependencies):
    for t in wf.tasks:
        if t.name in task_dependencies.keys():
            t.add_dependencies(task_dependencies[t.name])
    re_order_tasks_in_workflow(wf)
```

3.3 Reading Assembled Workflows Data

Listing 9 shows the code segment responsible for assembling data structures representing workflows and their tasks from the DSL definition. An empty list 'assembled_workflows_data' is created to store data related to assembled workflows. The code then iterates over each assembled workflow defined in 'workflow_model', extracting relevant information such as the workflow's name, its parent workflow (if any), and its associated tasks. For each assembled workflow, a dictionary named 'assembled_workflow_data' is created to hold this information, and then appended to the 'assembled_workflows_data' list.

A nested dictionary named 'assembled_workflow_tasks' is created to hold task-specific information. The code then iterates over configurations (tasks) within each assembled workflow, extracting details such as workflow references or file implementations for each task.

Listing 9. Code segment to read assembled workflows data from the DSL specifications

```
assembled_workflows_data = []
for assembled_workflow in workflow_model.assembledWorkflows:
    assembled_workflow_data = {}
    assembled_workflows_data.append(assembled_workflow_data)
    assembled_workflow_data["name"] = assembled_workflow.name
    assembled_workflow_data["parent"] = assembled_workflow.parent_workflow.name
    assembled_workflow_tasks = {}
    assembled_workflow_data["tasks"] = assembled_workflow_tasks

    configurations = assembled_workflow.tasks
    while configurations:
        for config in assembled_workflow.tasks:
            assembled_workflow_task = {}
            if config.workflow:
                assembled_workflow_task["workflow"] = config.workflow
                assembled_workflow_tasks[config.alias.name] =
assembled_workflow_task
            elif config.filename:
                if not os.path.exists(config.filename):
                    raise
dsl_exceptions.ImplementationFileNotFound(f"{config.filename} in task
{config.alias.name}")
                assembled_workflow_task["implementation"] = config.filename
                assembled_workflow_tasks[config.alias.name] =
assembled_workflow_task
            configurations.remove(config)
            configurations += config.subtasks
```

3.4 Generating Assembled Workflows

The function 'generate_assembled_workflows' (Listing 10) is defined to generate the assembled workflows. The function takes two parameters: 'wfs' representing a list of parsed workflows and 'assembled_wfs_data' representing assembled workflow data. The function iterates over each assembled workflow's data in 'assembled_wfs_data', and finds the corresponding parent workflow

from the list of 'wfs' based on the parent's name and clones it. Then, it updates the cloned workflow's name with the assembled workflow's name and appends it to a new list named 'new_wfs'.

Subsequently, the function iterates over each task in the cloned workflow. If the task name matches a key in the tasks dictionary of the assembled workflow's data, it indicates that configuration is required for this task. The function proceeds to extract task-specific data from the assembled workflow's data. If an "implementation" key exists in this task data, it updates the task's implementation file accordingly.

If the task has a sub-workflow, the function recursively calls 'configure_wf' to configure the sub-workflow using the same assembled workflow data. Finally, the function returns the list of newly assembled workflows after configuration. This function ensures that the provided workflows will be executed based on the specifications.

Listing 10. Code segment to generate final assembled workflows

```
def generate_assembled_workflows(wfs, assembled_wfs_data):
    new_wfs = []
    for assembled_wf_data in assembled_wfs_data:
        wf = next(w for w in wfs if w.name == assembled_wf_data["parent"]).clone()
        wf.name = assembled_wf_data["name"]
        new_wfs.append(wf)
        print(wf.name)
        for task in wf.tasks:
            if task.name in assembled_wf_data["tasks"].keys():
                print(f"Need to configure task '{task.name}'")
                task_data = assembled_wf_data["tasks"][task.name]
                if "implementation" in task_data:
                    print(f"Changing implementation of task '{task.name}' to
                        '{task_data['implementation']}'")
                    task.add_implementation_file(task_data["implementation"])
                else:
                    print(f"Do not need to configure task '{task.name}'")
            if task.sub_workflow:
                configure_wf(task.sub_workflow, assembled_wf_data)
        print("-----")
    return new_wfs
```

3.5 Streamlining the assembled workflows

The set of functions given in Listing 11 is dedicated to flattening assembled workflows by simplifying them by integrating sub-workflows into the main workflow structure. The main function, 'flatten_workflows', takes an assembled workflow as input, iterates over its tasks, and checks if each task has a sub-workflow. If a task does have a sub-workflow, the function recursively calls 'get_underlying_tasks' to extract its constituent tasks and add them to the main workflow.

Listing 11. Code segment to streamline the assembled workflows

```
def flatten_workflows(assembled_wf):
    print(f"Flattening assembled workflow with name {assembled_wf.name}")
    new_wf = Workflow(assembled_wf.name)
    for t in assembled_wf.tasks:
        if t.sub_workflow:
            tasks_to_add = get_underlying_tasks(t, assembled_wf, [])
            for t in tasks_to_add:
                new_wf.add_task(t)
```

```

    else:
        new_wf.add_task(t)
    re_order_tasks_in_workflow(new_wf)
    new_wf.set_is_main(True)
    return new_wf

```

The 'get_underlying_tasks' function (Listing 12) traverses through sub-workflows to retrieve all underlying tasks. It ensures that dependencies between tasks are maintained correctly. If a task within a sub-workflow has dependencies on tasks outside the sub-workflow, it adjusts these dependencies accordingly.

The 're_order_tasks_in_workflow' function (Listing 13) is responsible for setting the execution order of tasks within the workflow. It starts by identifying the first task in the workflow (i.e., the task with no dependencies) and assigns it an order of 0. Then, it iteratively assigns increasing orders to dependent tasks, ensuring that tasks are executed in the correct sequence.

These functions are responsible for streamlining the structure of assembled workflows by incorporating sub-workflows into the main workflow and organizing tasks in the desired execution order.

Listing 12. Code segment to find the tasks within a subworkflow

```

def get_underlying_tasks(t, assembled_wf, tasks_to_add):
    i = 0
    for task in sorted(t.sub_workflow.tasks, key=lambda t: t.order):
        if not task.sub_workflow:
            if i==0:
                task.add_dependencies(t.dependencies)
            if i==len(t.sub_workflow.tasks)-1:
                dependent_tasks = find_dependent_tasks(assembled_wf, t, [])
                dep = [t.name for t in dependent_tasks]
                print(f"{t.name} --> {dep} becomes {task.name} --> {dep}")
                for dependent_task in dependent_tasks:
                    dependent_task.remove_dependency(t.name)
                    dependent_task.add_dependencies([task.name])
                tasks_to_add.append(task)
            else:
                get_underlying_tasks(task, assembled_wf, tasks_to_add)
        i += 1
    return tasks_to_add

```

Listing 13. Code segment to order the tasks in sequence

```

def re_order_tasks_in_workflow(wf):
    first_task = [t for t in wf.tasks if not t.dependencies][0]
    order = 0
    first_task.set_order(order)
    dependent_tasks = [t for t in wf.tasks if first_task.name in t.dependencies]
    while dependent_tasks:
        order += 1
        new_dependent_tasks = []
        for dependent_task in dependent_tasks:
            dependent_task.set_order(order)
            new_dependent_tasks += [t for t in wf.tasks if dependent_task.name in
t.dependencies]
        dependent_tasks = new_dependent_tasks

```

3.6 Reading Experiment Spaces

In Listing 14, the code segment processes experiment spaces defined within the 'workflow_model'. It iterates over each experiment space ('espace') defined within the model. For each experiment space, it initializes an empty list named 'vp_methods' to store information about its name, assembled workflows it belongs to, different variability points (VPs), and associated tasks.

It iterates over methods configured within the experiment space, extracting details such as method name, type, runs, etc., and storing them in 'vp_methods'. Subsequently, it processes VPs defined within the experiment space, categorizing them based on the method type (grid search or random search) and storing their values accordingly.

Additionally, it iterates over tasks defined within the experiment space and their configurations, extracting parameters and associated VP values. It organizes this information within 'vp_methods' under the corresponding method and task names.

Listing 14. Code segment to read experiment spaces

```
for espace in workflow_model.espaces:
    vp_methods = []

    print(f"Experiment Space '{espace.name}' of '{espace.assembled_workflow.name}'")

    for m in espace.configure.methods:
        vp_method = {}
        vp_methods.append(vp_method)
        vp_method["espace"] = espace.name
        vp_method["assembled_workflow"] = espace.assembled_workflow.name
        vp_method["name"] = m.name
        vp_method["type"] = m.type
        vp_method["vps"] = {}
        vp_method["tasks"] = {}
        if m.runs:
            vp_method["runs"] = m.runs
        print(f"Method with name '{m.name}' of type '{m.type}'")

    for vp in espace.configure.vps:
        method_name = vp.method.name
        vp_method = next(m for m in vp_methods if m["name"] == method_name)
        if vp_method["type"] == "gridsearch":
            vp_method["vps"][vp.name] = vp.vp_values.values
        if vp_method["type"] == "randomsearch":
            vp_method["vps"][vp.name] = {"min": vp.vp_values.min, "max":
vp.vp_values.max}

    for task in espace.tasks:
        for c in task.config:
            param = {c.name: c.vp}
            vp_method = next(v for v in vp_methods if v["name"] == c.method.name)
            tasks = vp_method["tasks"]
            if task.alias.name not in tasks:
                tasks[task.alias.name] = {c.name: c.vp}
            else:
                tasks[task.alias.name][c.name] = c.vp
```

3.7 Creating Experiments

Finally, for each method defined within 'vp_methods', it calls the 'run_experiment' function depicted in Listing 15 to execute experiments based on the configured parameters. The function extracts the method type from the 'vp_method' dictionary and runs the experiments accordingly.

Listing 15. Code segment to create experiments according to the experiment method

```

for vp_method in vp_methods:
    run_experiment(vp_method)

def run_experiment(vp_method):
    print(f"Running experiment of espace '{vp_method['espace']}' of type '{vp_method['type']}'")
    method_type = vp_method["type"]
    if method_type == "gridsearch":
        run_grid_search_exp(vp_method)
    if method_type == "randomsearch":
        run_random_search_exp(vp_method)

def run_grid_search_exp(vp_method):
    combinations = generate_parameter_combinations(vp_method["vps"])
    print(f"Grid search generated {len(combinations)} configurations to run.")
    pp.pprint(combinations)
    run_count = 1
    for c in combinations:
        print(f"Run {run_count}")
        workflow_to_run = configure_final_workflow(vp_method, c)
        execute_wf(workflow_to_run)
        print(".....")
        run_count += 1

def run_random_search_exp(vp_method):
    for i in range(vp_method["runs"]):
        print(f"Run {i+1}")
        c = generate_random_config(vp_method["vps"])
        workflow_to_run = configure_final_workflow(vp_method, c)
        execute_wf(workflow_to_run)
        print(".....")

```

3.8 Executing Experiments

Listing 16 shows the function that orchestrates the execution of workflows using the Proactive platform from Activeeon as the executionware. The function establishes a connection with the Proactive platform, creates a job for the workflow, and sets up an environment for parallel execution. It iterates through each task within the workflow, creating corresponding Python tasks within the job, and configuring them with necessary input files and parameters. Task dependencies are maintained to ensure the correct execution sequence. Once all tasks are added to the job, it is submitted for execution on the Proactive platform (Section 5).

Listing 16. Code segment to submit experiment workflows on Proactive

```

gateway = create_gateway_and_connect_to_it(credentials.proactive_username,
credentials.proactive_password)
job = create_job(gateway, w.name)
fork_env = create_fork_env(gateway, job)

previous_tasks = []
for t in w.tasks:

```

```
task_to_execute = create_python_task(gateway, t.name, fork_env, t.impl_file,
t.input_files, previous_tasks)
if len(t.params)>0:
    configure_task(task_to_execute, t.params)
    job.addTask(task_to_execute)
    previous_tasks = [task_to_execute]
print("Tasks added.")

job_id, job_result, job_outputs =
submit_job_and_retrieve_results_and_outputs(gateway, job)
teardown(gateway)
```



4 Data Abstraction Layer

4.1 Strategy and process for traceability and repeatability of experiments and their results

The key attribute of the ExtremeXP experimentation process is the support for traceability and repeatability. Any dataset or metric created as a result of the experimentation comes with metadata that point to the complete workflow that generated them. This allows us to repeat experiments, but also to be able to perform meta-analyses over data collected (even in an unsystematic way) over various different experiments.

The difficulty in establishing such traceability is how to align the design and prototyping, which needs flexibility to freely change a workflow and its parameters without having to worry about versioning all artifacts and the need to remember every execution of an experiment.

In the ExtremeXP approach, we marry these two concerns by having two model repositories. One lies at the design level (DMS in Figure 1, for implementation details see Section 2.4) containing reusable workflows and tasks. This repository is freely modifiable and does not require systematic versioning.

Another is the Model repository (see Figure 1) and is on the level of the experiment execution (forming the Data Abstraction Layer – as described in Section 4.2). This repository stores data about every executed experiment – in particular, it stores the serialized workflow, the pointers to input datasets (which we consider as versioned and thus immutable), parameters used to parameterize the workflow, and human-in-the-loop inputs during the experiment. This model repository has the write-once semantics. Once the workflow is stored in the repository, it cannot be changed or deleted. A permanent URI exists for the workflow in the repository, which is recorded in the metadata of datasets and metrics created by the respective workflow.

To allow for additional flexibility in design and prototyping, we allow early versions of experiments to be run without being recorded in the Data Abstraction Layer repository. However, it is strongly recommended that once the experiments are run to provide any data, the recording is enabled.

4.2 Data model used by the Data Abstraction Layer

The **Data Abstraction Layer** (DAL) serves as a storage of “snapshots” of executed experiments together with a whole environment (inputs and outputs). This section presents a data model (defined via a meta-model) used in DAL. The implementation of DAL is discussed in section 4.3.

The meta-model of the data model used in DAL is shown in Figure 21. Importantly, the experiments are stored in DAL unstructured, i.e., as BLOBs, but the actual structure can be obtained from the Design Model Storage (a part of the framework architecture and defined in Deliverable D2.1).

To allow for searching the executed experiments by their various characteristics (e.g., datasets used, algorithms used, types of experiment, user involved, etc, the executed experiment blob is annotated by key-value pairs, each of which represents one of these characteristics. This is unstructured but allows for indexing in no-SQL databases. These characteristics are derived from the experiment workflow (which is stored as the BLOB).

In detail, the structure of DAL is as follows (in the meta-model description, for better readability of the text, we omit the “meta” prefix when describing the meta-model elements, i.e., meta-class is referred

to as a class, meta-attribute as an attribute, etc.) To further enhance the readability, white classes in Figure 21 are abstract, while colored ones are non-abstract.

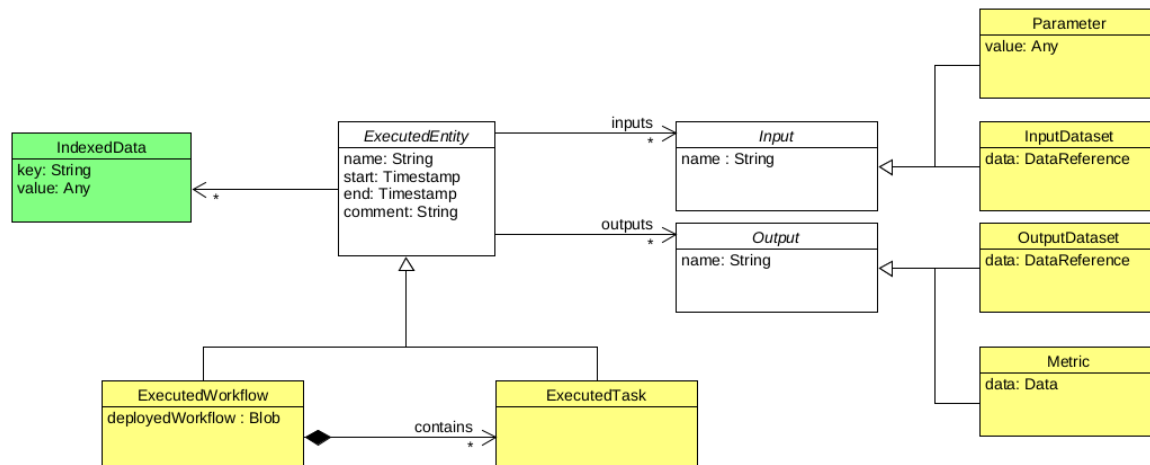


Figure 21. Data layer meta-model

The **ExecutedEntity** is an abstract class that represents an executed experiment (either the whole one or its part). Its attributes are as follows:

- **name** – refers to the name of the executed entity. Via the **name**, the entity may be searched in the Design Model Storage.
- **start, end** – refer to the timestamps of the start of the execution of the entity and the end of its execution. The two are necessary as it is expected that the same workflow might be executed many times.
- **comment** – contains any textual comments a user wants to add.

Concrete subclasses of **ExecutedEntity** are the **ExecutedWorkflow** and **ExecutedTask**. The **ExecutedWorkflow** represents a whole executed workflow. It adds the **deployedWorkflow** attribute, which contains a serialized workflow definition. Also, it may refer to a number of the **ExecutedTasks**. The **ExecutedTask** represents a single executed *primitive* task (primitive in this context means a task that is not composed of any subtask and is directly implemented, e.g., as an executables file or a service) that is executed as a part of a workflow. Within the **ExecutedWorkflow**, the **ExecutedTasks** are not held in any particular order; their ordering can be achieved via the **start** and **end** timestamps.

The **ExecutedEntity** (workflow or task) is associated with inputs and outputs. The **Input** and **Output** classes are abstract and serve as a common input/output representation. The particular kinds are **Parameter** and **InputDataset** for inputs and **Metric** and **OutputDataset** for outputs. As the names suggest, the **InputDataset** and **OutputDataset** represent datasets. Datasets are not stored directly in DAL but only referenced to the Dataset Storage (see the framework architecture in Deliverable D2.1) via the **data** attribute of the **InputDataset/OutputDataset**. The **Parameter** is used for values that are passed to the parameters of workflows/tasks (see the Workflows meta-model in Deliverable D2.1). The **Metric** is a value generated from a workflow/task as a Metric (see the Workflows meta-model in Deliverable D2.1). Finally, the **IndexedData** are values that are “drawn out” from a workflow/task to be indexed and further used by tools (e.g., visualizer). The characteristics used for indexing the workflow are kept in the **IndexedData** class which is attached with 1-to-many cardinality to the **ExecutedEntity**.

Figure 22 shows part of the DAL meta-model used to represent values in the **Metric**. The meaning of the individual classes is obvious from their names.

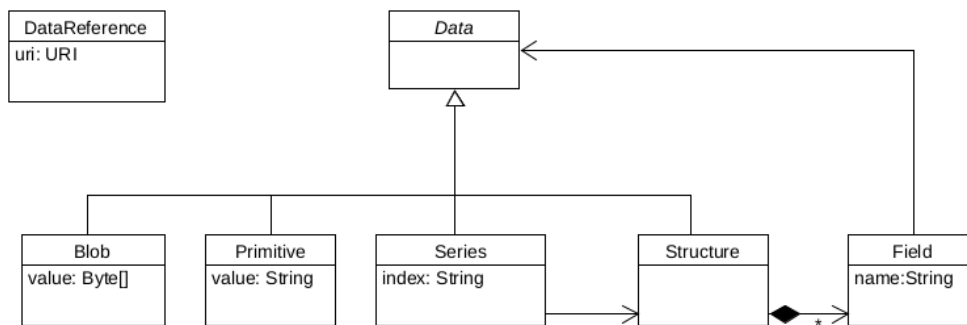


Figure 22. Meta-model of data used in the Data abstraction layer

4.3 Reference architecture of the Data Abstraction Layer

DAL will be implemented with the help of the IVIS platform [1]. The high-level view of the IVIS architecture is shown in Figure 23. IVIS architecture and shows that IVIS is a client-server application with the client frontend running in the web-browser and the server backend running in a cloud.

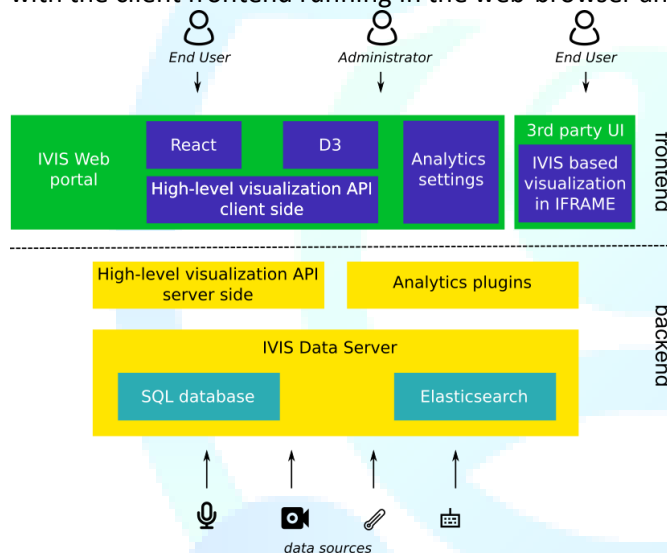


Figure 23. IVIS architecture

The backend is responsible for data management and offers an API to the frontend. It serves as the executor and coordinator for plugins that carry out data processing and analysis tasks. It collects data from different sources, such as sensors, either by actively retrieving it from these sensors and their gateways or by passively receiving it through an API that allows these devices to push data to it. Once the data is collected, it is organized within Elasticsearch⁸, a distributed search engine that manages data searches and real-time data aggregations needed by the frontend.

The web-based frontend features both an administrative platform and data visualizations through user dashboards, referred to as panels. This administrative section offers management options for settings like tenants, users, and permissions. Furthermore, it includes a streamlined development environment for creating and developing new visualizations and data processing activities. Access to

⁸ <https://www.elastic.co/elasticsearch>

these visualizations is available directly through the integrated web portal or indirectly via a third-party interface that incorporates the visualizations from the system as an HTML IFRAME.

The figure also shows the data server, which will be used to store data from the data model described in Section 3.2. Data will be accessible via designed API. In order to propose robust and flexible API, scenarios of data usages need to be established first.

During designing the API scenarios of data retrieval were gathered and this led to creation of following scenarios categories.

- Experiments overview
- Data overview
- Single experiment analysis
- Single experiment description
- Checking reliability and reproducibility of an approach
- Comparison of multiple approaches on single dataset
- Traceability

Appendix 1 contains a listing of Data Abstraction Layer API calls and their expected parameters.

The data abstraction layer will be implemented as a plugin to IVIS. It will exploit IVIS extension mechanism, which allows plugins to hook to internal events. It will be hooked to the API layer, which it will extend by the API calls listed in Appendix 1. These API calls will be connected to the data store layer of IVIS, which takes care of indexing data (in this case experiment descriptors and metrics) in the Elasticsearch storage. To this end, IVIS will be extended to support the storage of the serialized ExecutedWorkflow (as a blob) and for indexing features extracted from the executed workflow as in Figure 21. Data layer meta-model (represented as IndexData instances).

5 Runtime for Scheduling Complex Workflows

5.1 Introduction

As already mentioned in previous sections, an ExtremeXP experiment essentially consists of potentially several complex analytics workflows that need to be run and compared to each other. The runtime component for scheduling complex workflows is part of the Experiment Execution block of the ExtremeXP architecture shown in Figure 1 and implements the scheduling, execution, and monitoring of workflows that can be run on various types of computing resources, ranging from local machines to large-scale cloud infrastructures. The component is based on ProActive Workflows & Scheduling (PWS)⁹, a tool designed for orchestrating and automating tasks across distributed computing environments. PWS is particularly well-suited for applications in big data analysis, scientific research, and machine learning, where tasks can be computationally intensive and may need to be executed in a specific order or in parallel to optimize performance and resource utilization. It offers a comprehensive suite of features designed to improve operational efficiency, reduce costs, and increase agility for businesses.

5.1.1 Overall Architecture of the PWS

PWS is composed of a central server (the Scheduler) that manages the task queues and orchestrates the execution across available resources, which are managed by the Resource Manager. The user interacts with the system through a client interface, which could be a web portal, a command-line interface, or an API, to submit workflows and monitor their execution. This architecture (Figure 24) ensures that ProActive can efficiently manage and execute a large number of tasks across diverse computing environments, making it a powerful tool for automating complex workflows in various domains.

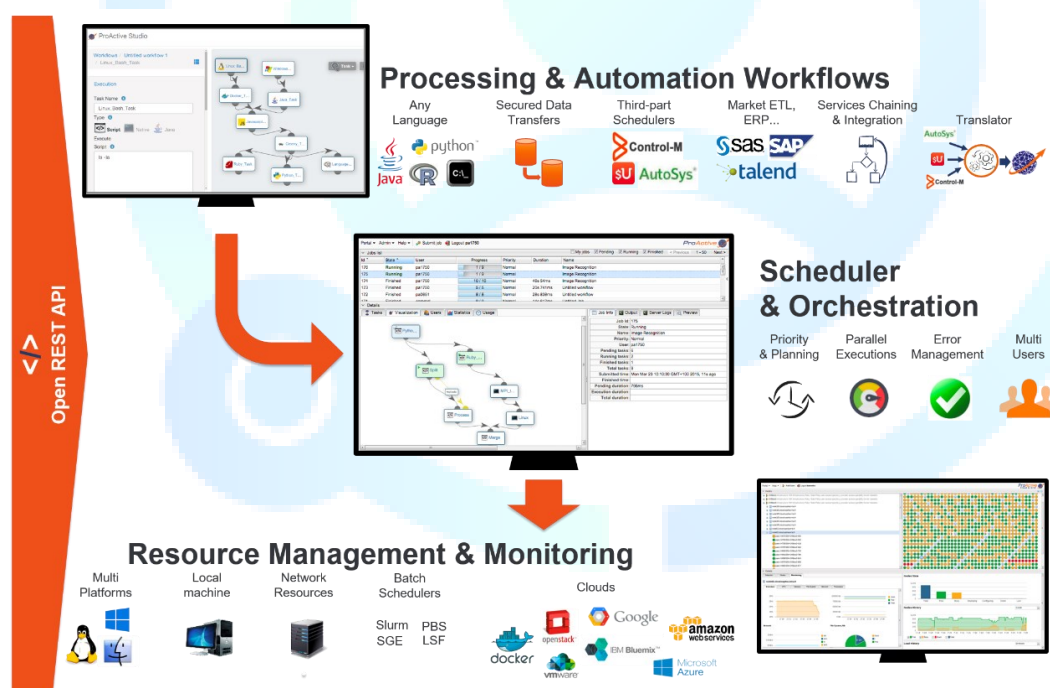


Figure 24. Architecture of ProActive Workflows & Scheduling System

⁹ <https://www.activeeon.com/products/workflows-scheduling/>

The key components of the PWS architecture are:

- 1. Workflow Studio:** This visual interface allows users to design and configure workflows by defining tasks, dependencies, and execution logic. It supports various programming languages and includes drag-and-drop functionality for ease of use.
- 2. Resource Manager:** It provides centralized control over all resources, including physical machines, virtual machines, containers, and cloud resources. It allows for the dynamic allocation and deallocation of resources based on the needs of the executing workflows, ensuring efficient use of available computing power.
- 3. Scheduling Engine:** It is responsible for managing the execution of tasks defined in workflows. It ensures that tasks are executed in the correct order, respecting their dependencies, and optimally utilizing available computing resources. The scheduler is capable of handling dynamic changes in resource availability and can adapt the execution strategy in real time to optimize performance.
- 4. Execution Environment (Agent):** It is where tasks are run. ProActive supports a diverse set of execution environments, allowing tasks to be executed on anything from a single laptop to a distributed cloud infrastructure. The system abstracts the complexities of the computing environment, providing a consistent interface for task execution.
- 5. Monitoring and Control:** Offers comprehensive monitoring and control capabilities, allowing users to track the progress of workflows, inspect the output of tasks, and intervene manually if necessary. This is crucial for long running or complex workflows where issues may need to be addressed in real time.
- 6. Security:** Protects sensitive data and computing resources. The system supports authentication, authorization, and secure communication, ensuring that only authorized users can access and execute workflows.
- 7. Extensibility and Integration:** Designed to be extensible and can be integrated with other systems and software. This allows users to incorporate custom tasks, use ProActive in conjunction with existing tools, and extend the system's capabilities to meet specific requirements.

5.2 ProActive AI Orchestration

ProActive AI Orchestration (PAIO)¹⁰ is a platform designed to facilitate the orchestration and automation of AI workflows across distributed computing environments. It is part of the broader PWS ecosystem, which specializes in managing complex job executions, resource management, and task scheduling on various infrastructures, including clouds, clusters, and hybrid environments.

Integrating artificial intelligence (AI) automation within the PWS framework enhances the management of complex, data-driven processes. This integration leverages AI's capabilities to analyze data, make predictions, and recognize patterns, thereby enriching the ecosystem with intelligent decision-making. The orchestration capability of ProActive ensures the efficient coordination of diverse tasks and processes across multiple systems and environments. It orchestrates workflows to ensure tasks are executed in an optimal sequence, considering dependencies and the availability of resources. This seamless orchestration and AI-driven automation streamline the execution of workflows, making complex task management more efficient and responsive to dynamic conditions.

¹⁰ <https://www.activeeon.com/products/proactive-ai-orchestration/>

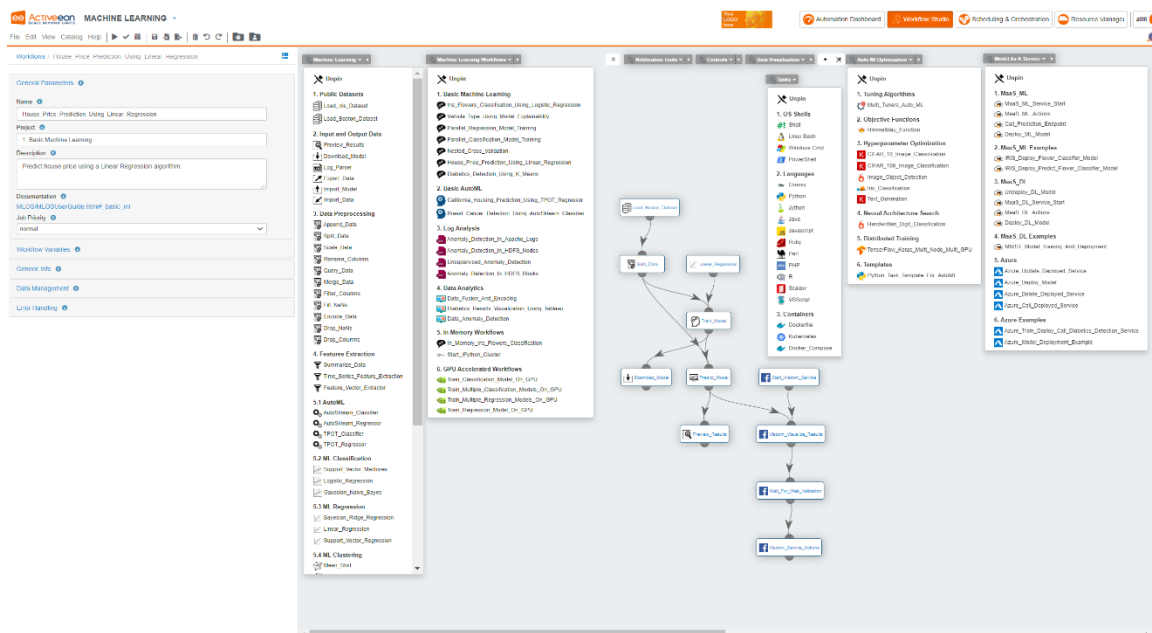


Figure 25. Designing Predictive Models with ProActive's Machine Learning Studio

PAIO is highly adaptable, making it suitable for a wide range of industries and applications, from IT operations and DevOps to data analytics and beyond. Its scalable nature allows it to handle workflows of varying complexities, from simple task automation to orchestrating large-scale, data-intensive processes. PAIO supports the following use cases:

- **Automated Machine Learning (AutoML):** Automating the process of applying machine learning to real-world problems, including automatic model selection, hyperparameter tuning, and model validation.
- **Data Pipeline Automation:** Orchestrating complex data pipelines that involve data collection, cleaning, transformation, and feature engineering to prepare data for AI/ML models.
- **Distributed Model Training:** Managing the distributed training of AI models across multiple computing nodes to handle large datasets and complex models efficiently.
- **Model Serving and Deployment:** Automating the deployment of AI models into production environments, including the management of model versioning, scaling, and updates.

5.3 Scheduling Abstraction Layer and Proactive Python SDK

ProActive Python SDK¹¹ is an open-source library that empowers users to interact with and manage PWS and PAIO directly from their scripts. This integration unlocks seamless programmatic control over complex workflows, boosting efficiency and flexibility in your data-driven processes. Key features of the SDK include:

- **Submit and Manage Workflows:** Easily submit workflows to the ProActive Scheduler for execution and monitor their progress in real-time. Gain valuable insights into execution status, resource utilization, and task completion.
- **Resource Control:** Take command of resources managed by the ProActive Resource Manager. Allocate, release, and monitor resources programmatically, ensuring optimal utilization and performance.

¹¹ <https://github.com/ow2-proactive/proactive-python-client>

- **Extensibility:** Develop custom Python modules to leverage the full potential of ProActive within your applications. Integrate automation logic, custom task implementations, and tailored communication channels.
- **Simplified Development:** Enjoy the convenience of Python, a widely adopted and versatile language, for your workflow management tasks. Eliminate the need for complex low-level API interactions.

5.3.1 Applications and Examples:

- **Automating scientific simulations:** Submit and manage large-scale simulations remotely through Python scripts, optimizing resource utilization and reducing manual intervention.
- **Dynamic resource allocation:** Implement real-time resource allocation based on data-driven insights, ensuring your workloads have the necessary resources for optimal performance.

5.3.2 Getting Started with the ProActive Python SDK

Getting started with the ProActive Python SDK involves several steps to set up your environment and run your first tasks or workflows on the ProActive Scheduler. Here is a guide to help you get started:

1. Prerequisites

- Have access to the ProActive Scheduler (trial or private)
- Java SDK LTS (works with Java 8, 11 and 13)
- Python 3.5 or later

2. Create a Virtual Environment (Optional)

It is a good practice to create a virtual environment for your Python projects. This keeps dependencies required by different projects separate by creating isolated Python environments for them.

To create a virtual environment, run the following commands in your terminal:

Listing 17. Creating a virtual environment

```
python3 -m venv env

# On Mac/Linux
source env/bin/activate

# On Windows
env\Scripts\activate
```

3. Upgrade Required Packages

Once the virtual environment is activated, upgrade **pip**, **setuptools**, and **python-dotenv** to the latest versions:

```
python3 -m pip install --upgrade pip setuptools python-dotenv
```

4. Installing the ProActive Python SDK

To install the latest pre-release or development version of the ProActive Python SDK, which includes the most recent features and fixes:

```
python3 -m pip install --upgrade --pre proactive
```

This command uses the `--pre` flag to allow pip to find and install pre-releases.

5. Usage Example

This simple example demonstrates connecting to a ProActive server, creating a job, adding a Python task, and submitting the job:

Listing 18. Connecting to a ProActive server

```
import getpass
from proactive import ProActiveGateway

proactive_url = "https://try.activeeon.com:8443"

print(f"Connecting to {proactive_url}...")
gateway = ProActiveGateway(proactive_url)

# Securely input your credentials
gateway.connect(username=input("Username: "), password=getpass.getpass("Password: "))
assert gateway.isConnected(), "Failed to connect to the ProActive server!"

# Create a ProActive job
print("Creating a ProActive job...")
job = gateway.createJob("SimpleJob")

# Create a ProActive task
print("Creating a ProActive Python task...")
task = gateway.createPythonTask("SimplePythonTask")
task.setTaskImplementation('print("Hello from ProActive!")')
task.addGenericInformation("PYTHON_COMMAND", "python3")

# Add the Python task to the job
job.addTask(task)

# Job submission
job_id = gateway.submitJob(job)
print(f"Job submitted with ID: {job_id}")

# Retrieve job output
print("Job output:")
print(gateway.getJobOutput(job_id))

# Cleanup
gateway.close()
print("Disconnected and finished.")
```

5.3.3 Supported programming languages

The ProActive Python SDK supports a wide range of programming languages for task execution within the ProActive Scheduler environment. These supported languages include:

- **bash** : Linux Bash
- **cmd** : Windows Cmd
- **docker-compose** : Docker Compose
- **scalaw** : Scalaw
- **groovy** : Groovy
- **javascript** : JavaScript
- **python** : Jython (implementation of Python in Java)
- **cpython** : Python (implementation of Python in C – original Python implementation)
- **ruby** : Ruby

- **perl** : Perl
- **powershell** : PowerShell
- **R** : R Language

This comprehensive support enables you to integrate a variety of programming languages into your workflows, allowing for a flexible and versatile development experience within the ProActive Scheduler.

To print the list of supported programming languages in the ProActive Python SDK, you can use the following command:

```
print(gateway.getProactiveScriptLanguage().get_supported_languages())
```

This command fetches the supported languages from the ProActive server via the Python SDK, ensuring you have access to the most up-to-date list directly from your Python script.

Example: Creating a Groovy Task

To create a Groovy task using the ProActive Python SDK, you can follow these steps, which include creating a job, and adding a Groovy task to this job with a specific task implementation. Below is an example that demonstrates these steps:

Listing 19. Creating a task

```
...
# Create a new ProActive job
print("Creating a new ProActive job...")
job = gateway.createJob("Groovy_Job_Example")

# Create a new task using Groovy
print("Creating a Groovy task...")
groovy_task = gateway.createTask(language="groovy")
groovy_task.setTaskName("Groovy_Task_Example")
groovy_task.setTaskImplementation("""
println "Hello from Groovy task!"
""")

# Add the Groovy task to the job
job.addTask(groovy_task)
...
```

5.3.4 Task dependencies

To manage task dependencies in your ProActive Python SDK scripts, you can use the **addDependency** method. This method allows you to specify that a task depends on the completion of another task before it can begin execution. Here is a simplified example to illustrate how you can manage task dependencies using the ProActive Python SDK:

Listing 20. Creating a task dependency

```
...
# Create a ProActive job
job = gateway.createJob("Simple_Dependency_Demo")

# Create the first task (Task A)
task_A = gateway.createPythonTask("Task_A")
task_A.setTaskImplementation('print("Task A is running")')

# Create the second task (Task B) which depends on Task A
```

```
task_B = gateway.createPythonTask("Task_B")
task_B.setTaskImplementation('print("Task B is running")')
task_B.addDependency(task_A)

# Add tasks to the job
job.addTask(task_A)
job.addTask(task_B)

# Submit the job to the ProActive Scheduler
job_id = gateway.submitJob(job)
print(f"Job submitted with ID: {job_id}")
...
```

In this example:

- **Task A** is a simple Python task that prints "Task A is running".
- **Task B** is another Python task that prints "Task B is running".
Task B has a dependency on Task A, meaning it will only start after Task A has successfully completed. This dependency is established using the **addDependency(task_A)** method.

After both tasks are created and configured, they are added to the job, which is then submitted to the ProActive Scheduler. Task B will wait for Task A to finish before executing.

5.3.5 Job and task variables

In the ProActive Scheduler, managing data flow and configuration across jobs and their constituent tasks is streamlined through the use of variables. These variables can be defined both at the job level, affecting all tasks within the job, and at the individual task level, for task-specific configurations. The following example demonstrates how to set and utilize these variables using the ProActive Python SDK, showcasing a simple yet effective way to pass and access data within your ProActive workflows.

Listing 21. Creating job and task variables

```
...
# Create a new ProActive job
job = gateway.createJob("Example_Job")

# Define job-level variables
job.addVariable("jobVar", "jobValue")

# Create the first Python task
task = gateway.createPythonTask()
task.setTaskName("task")
task.setTaskImplementation("""
print("Job variable: ", variables.get("jobVar"))
print("Task variable: ", variables.get("taskVar"))
""")

# Define task-level variables
task.addVariable("taskVar", "taskValue")

# Add the tasks to the job
job.addTask(task)
...
```

This example illustrates the flexibility of the ProActive Python SDK in managing data flow between jobs and tasks through the use of variables. Job-level variables are useful for defining parameters that are common across all tasks in a job, while task-level variables allow for task-specific configurations.

5.3.5.1 Global variables

To transfer data (variables) between **TaskA** and **TaskB**, we can use the mechanism of global variables, where the **TaskA** creates a global variable that is visible by the **TaskB** and any other tasks created on the same job.

TaskA: Producing a global variable

In **TaskA**, you can define a variable within the task implementation and set its value. After the task execution, you mark this variable as a global variable.

Listing 22. Producing a global variable

```
...
# TaskA Implementation
taskA = gateway.createPythonTask("TaskA")
taskA.setTaskImplementation('''
# Task logic
variableA = "Hello from TaskA"
# Setting a resulting variable
variables.put("variableFromA", variableA)
''')
...
```

TaskB: Consuming the global variable

In **TaskB**, you access the global variable produced by **TaskA** using the **variables.get()** method. This requires **TaskB** to have a dependency on **TaskA**, ensuring **TaskA** executes before **TaskB** and the variable is available.

Listing 23. Consuming a global variable

```
...
# TaskB Implementation
taskB = gateway.createPythonTask("TaskB")
taskB.addDependency(taskA)
taskB.setTaskImplementation('''
# Accessing the variable from TaskA
variableFromA = variables.get("variableFromA")
print("Received in TaskB:", variableFromA)
''')
...
```

5.3.5.2 Result variable

Using **result** variables is a powerful method to transfer data between tasks within the same job. This approach allows a task (e.g., **TaskA**) to produce a result that can be accessed by subsequent tasks (e.g., **TaskB**) that depend on it. Here is how you can work with result variables in ProActive workflows:

TaskA: Producing a result variable

TaskA executes its business logic and generates a result. This result is then implicitly available to any tasks that are defined as its successors, making it an effective way to pass data forward in a workflow.

Listing 24. Producing a result variable

```
...
# Create a Python task A
print("Creating a Python task...")
taskA = gateway.createPythonTask("PythonTaskA")
```

```
taskA.setTaskImplementation("""
print("Hello")
result = "World"
""")
...
```

TaskB: Consuming the result variable

TaskB, which has a dependency on **TaskA**, can access the result produced by **TaskA**. This is done by iterating over the results object, which contains the outcomes of all predecessor tasks **TaskB** is dependent on.

Listing 25. Consuming the results variable

```
...
# Create a Python task B
print("Creating a Python task...")
taskB = gateway.createPythonTask("PythonTaskB")
taskB.addDependency(taskA)
taskB.setTaskImplementation("""
for res in results:
    print(str(res))
""")
...
```

To ensure that **TaskB** correctly consumes the result variable produced by **TaskA**, you must explicitly declare **TaskA** as a dependency of **TaskB**. This setup guarantees the sequential execution order where **TaskA** completes before **TaskB** starts, allowing **TaskB** to access the results produced by **TaskA**.

5.3.6 Workflow Data management

Efficient data management is pivotal in executing distributed tasks with the ProActive Scheduler. This encompasses handling input and output files, managing data flow between tasks, and storing or sharing resources across different executions. The ProActive Python Client facilitates robust data management through various mechanisms, enabling you to manipulate data spaces and directly transfer files to/from the task's execution environment. This section elucidates two primary aspects of data management: "Data Spaces" and "Uploading and Downloading Files," highlighting their applications and differences to cater to diverse workflow requirements.

5.3.6.1 Workflow Data Spaces

Workflow Data Spaces refer to the designated storage areas within the ProActive Scheduler's environment, specifically the "User Space" and the "Global Space." These spaces are intended for storing files that can be accessed by jobs and tasks, allowing for resource sharing and data persistence across different executions. Utilizing data spaces is essential for workflows that require access to shared configurations, libraries, or datasets, or need to persist output for future use or analysis.

- **User Space:** A private storage area accessible only to the user's jobs, ideal for personal or sensitive data.
- **Global Space:** A shared storage space accessible by all users, suitable for commonly used data or resources.

Data spaces are particularly useful for workflows that rely on a centralized repository of files or for those that need to share data between different users or jobs.

Example: Managing Data Transfers with User and Global Spaces

This example demonstrates transferring files between the task's local execution space and the ProActive Scheduler's user or global data spaces, facilitating the use of shared or persistent data in workflows.

Scenario:

1. A text file named `hello_world.txt` containing "Hello World" is created in the local space.
2. The file is then transferred to the user space using **userspaceapi** for demonstration purposes.
3. Instructions are provided to modify the code to utilize the global space instead, using **globalspaceapi**.

TaskA: Transferring data to the user space

To transfer files from the local machine to the user space in TaskA, you can use the following approach:

Listing 26. Transferring data to the user space

```
...
# Task A: Creating and transferring a file to the user space
taskA = gateway.createPythonTask("TaskA")
taskA.setTaskImplementation("""
import os

file_name = 'hello_world.txt'
with open(file_name, 'w') as file:
    file.write("Hello World")
print("File created: " + file_name)

# Define the data space path
dataspace_path = 'path/in/user/space/' + file_name

# Transferring file to the user space
print("Transferring file to the user space")
userspaceapi.connect()
userspaceapi.pushFile(gateway.jvm.java.io.File(file_name), dataspace_path)
print("Transfer complete")

# Transfer the file info to the next task
variables.put("TASK_A_FILE_NAME", file_name)
variables.put("TASK_A_DATASPACE_PATH", dataspace_path)
""")
...
```

TaskB: Transferring data from the user space

In TaskB, to import files from the user space back to the local space, follow this method:

Listing 27. Transferring data from the user space

```
...
# Task B: Importing and displaying the file from the user space
taskB = gateway.createPythonTask("TaskB")
taskB.addDependency(taskA)
taskB.setTaskImplementation("""
import os

# Get the file info from the previous task
file_name = variables.get("TASK_A_FILE_NAME")
dataspace_path = variables.get("TASK_A_DATASPACE_PATH")

# Transfer file from the user space to the local space
print("Importing file from the user space")
...

```

```

userspaceapi.connect()
userspaceapi.pullFile(dataspace_path, gateway.jvm.java.io.File(file_name))
if os.path.exists(file_name):
    with open(file_name, 'r') as file:
        print("File contents: " + file.read())
else:
    print("File does not exist.")
"""
...

```

Modifying for Global Space Use

To adapt the above example for transferring data to and from the global space, replace the userspaceapi calls with globalspaceapi as demonstrated below:

For Transferring to Global Space in TaskA:

Listing 28. Transferring to global space

```

...
print("Transferring file to the global space")
globalspaceapi.connect()
globalspaceapi.pushFile(gateway.jvm.java.io.File(file_name), dataspace_path)
print("Transfer complete")
...

```

For Importing from Global Space in TaskB:

Listing 29. Importing from global space

```

...
print("Importing file from the global space")
globalspaceapi.connect()
globalspaceapi.pullFile(dataspace_path, gateway.jvm.java.io.File(file_name))
print("Import complete")
...

```

5.3.6.2 Uploading and Downloading files

In contrast to 'Data Spaces', you may need to upload files from your local machine to the execution environment where your tasks run (the "local space" of the task) and download the results back to your machine after the task execution. This process is facilitated by two key methods in the ProActive Python Client: **addInputFile** for uploading and **addOutputFile** for downloading files.

Uploading Files to the Task's Local Space

Before task execution, files from the user's local machine can be specified for upload to the task's local execution environment, ensuring necessary data or scripts are available for processing.

Example: Uploading Files for Task Execution

Listing 30. Uploading files

```

...
# Assuming 'task' is your task object
# Upload files located in 'local_directory_path/' on your local machine
# to the task's local space before execution
task.addInputFile('local_directory_path/**')
...

```

This code snippet demonstrates how to upload all files from a local directory (**local_directory_path/**) to the task's local space. The ****** pattern is used to include all files and subdirectories within that directory.

Downloading Files from the Task's Local Space

After the task execution, you can specify files or directories to be downloaded from the task's local space back to your local machine using the **addOutputFile** method. This allows you to easily retrieve results, logs, or any other files generated by your task.

Example: Downloading Task Results

Listing 31. Downloading task results

```
...
# Assuming 'task' is your task object
# Download files from the task's local space to your local machine
# after task execution
task.addOutputFile('path/in/task/local/space/**')
...
```

In this example, **path/in/task/local/space/**** should be replaced with the specific path in the task's local space where the output files are located. The ****** pattern ensures that all files and subdirectories under this path are downloaded.

The **addInputFile** and **addOutputFile** methods provide a straightforward approach to manage file transfers directly between your local machine and the ProActive task's execution environment. This mechanism simplifies the process of providing input data to your tasks and retrieving the results, enhancing the efficiency and flexibility of your ProActive workflows. Note that this process is distinct from using the ProActive Scheduler's user or global data spaces, which involve transferring files within the server's data space rather than between the user's local machine and the task's local space.

5.3.7 IDEKO Use Case (UC5)

The implementation described here serves as an exemplary application of Use Case 5, aimed at binary classification for failure prediction within a manufacturing setup. The specified workflow encapsulates a sequence of tasks integral to the ML training pipeline, which includes data preprocessing, model training, and validation. The primary objective is to forecast potential failures, thereby facilitating preemptive actions to mitigate downtime and enhance production efficiency.

5.3.7.1 Implementation Overview

This example delineates the workflow for a specific subset within the manufacturing domain, focusing on IDEKO's scenario. The implementation journey begins with the setup of the ProActive Python SDK environment, ensuring all prerequisites are met. Following this, the ProActive gateway is established, creating a conduit for executing the predefined ML workflow on the ProActive Scheduler.

Listing 32. UC5 example setup

```
from util import *
import credentials

gateway = create_gateway_and_connect_to_it(credentials.proactive_username,
credentials.proactive_password)

job = create_job(gateway, "IDEKO")
fork_env = create_fork_env(gateway, job)
```

```

input_data_folder = "datasets/ideko-subset"
tasks_folder = "tasks/IDEKO/"
tasks_folder_src_all = tasks_folder + "src/**"

read_data_task = create_python_task(gateway, "read_data", fork_env, tasks_folder +
'read_data.py', [input_data_folder+"/**", tasks_folder_src_all])
add_padding_task = create_python_task(gateway, "add_padding", fork_env,
tasks_folder + 'add_padding.py', [tasks_folder_src_all], [read_data_task])
split_data_task = create_python_task(gateway, "split_data", fork_env, tasks_folder
+ 'split_data.py', [tasks_folder_src_all], [add_padding_task])
train_nn = create_python_task(gateway, "train_nn", fork_env, tasks_folder +
'train_nn.py', [tasks_folder_src_all], [split_data_task])

job.addTask(read_data_task)
job.addTask(add_padding_task)
job.addTask(split_data_task)
job.addTask(train_nn)

job_id = gateway.submitJobWithInputsAndOutputsPaths(job, debug=False)

print("Getting job results...")
job_result = gateway.getJobResult(job_id)
print(job_result)

print("Getting job outputs...")
job_outputs = gateway.printJobOutput(job_id)
print(job_outputs)

teardown(gateway)

```

Workflow Specification

The workflow is articulated through a series of sequential tasks:

- **Data Reading (`read_data`)**: This task is responsible for ingesting the dataset, which could be a comprehensive dataset or a subset thereof, depending on the scope and scale of the experiment.
- **Data Preprocessing (`add_padding`)**: Post data ingestion, this step applies necessary preprocessing techniques such as padding, normalization, or any other transformation required to make the data conducive for model training.
- **Data Splitting (`split_data`)**: It partitions the pre-processed data into training and test sets, ensuring a robust framework for training and validating the ML model.
- **Model Training (`train_nn`)**: The core of the workflow, this task entails the training of a neural network (or potentially a recurrent neural network, based on the configuration) on the processed data.

Each task is meticulously crafted and configured, ensuring dependencies are managed to maintain the correct execution sequence. The tasks are then collectively submitted as a job to the ProActive Scheduler for execution.

The execution script not only orchestrates the workflow but also provides mechanisms for job submission, results retrieval, and output analysis. It showcases the integration of the ProActive Python SDK with the machine learning pipeline, illustrating a streamlined process from data ingestion to model training. This approach not only accelerates the development and deployment of ML models but also enhances the scalability and manageability of manufacturing analytics projects.

This use case implementation exemplifies the application of the ProActive Python SDK in operationalizing machine learning workflows within the manufacturing sector. By leveraging this powerful toolkit, industries can harness the full potential of predictive analytics, driving significant

improvements in reliability, efficiency, and overall operational excellence. The outlined methodology paves the way for a future where advanced analytics and machine learning are integral components of manufacturing processes, ensuring competitiveness and innovation in the industrial landscape.



6 Context-aware Access Control for Experiment-driven Analytics

A primary objective of the ExtremeXP framework is to provide the means to data analysts and data engineers to run their experiments with complex analytics workflows in a secure way that does not compromise the privacy and confidentiality requirements of both experiment inputs (datasets) and outputs (knowledge synthesized from experiments). To this end, a dedicated component focusing on context-aware access control has been designed and developed.

In this section, we overview the advancements achieved within task 5.4 concerning context-aware access control for experiment-driven analytics. We begin by exploring the relevant technologies used in our access control system, delving into the XML-based model language for modelling of access control policies. Furthermore, we discuss the integration of Hyperledger Besu and smart contracts, emphasising its connection to the project's deliverable D2.1. Additionally, we introduce Keycloak for authentication and JWTokens, a new component of our access control system. Following this, we present the architecture of our access control system and functionalities across modelling access control policies, context handling, and access control enforcement flow. By presenting our system's design, we aim to provide an understanding of its operational mechanisms and design principles that will be applied in the uses cases of the project.

We present a practical demonstration of our context-aware access control framework within the specific use case of UC5 at IDEKO. This demonstration showcases specific access control policies tailored to UC5, along with the contextual attributes and handlers utilised to enforce them effectively. Furthermore, we detail the implementation of Hyperledger Besu network, including nodes, users, and actions, elucidating the access control mechanisms both within and outside the blockchain context.

Finally, we outline the next steps, anticipating advancements and milestones of achieving our KPIs, including further refinement of our access control framework, integration with the other use cases, and potential enhancements based on feedback and evaluation, are discussed.

6.1 Relevant technologies

6.1.1 Attribute-Based Access Control Model language

XML (eXtensible Markup Language) is frequently utilized in Attribute-Based Access Control (ABAC) systems to define security policies. ABAC is an access control model where rights are assigned based on policies that evaluate attributes of users, resources, and environmental conditions. This model is particularly suited for complex and dynamic settings, such as cloud computing and large enterprises, due to its flexibility and scalability.

In ABAC, XML-based policies are essential for detailing the conditions under which access should be granted or denied. These policies typically consist of rules that specify subjects (usually users), resources (the objects needing protection), actions (operations users wish to perform), and conditions (additional requirements for access). XACML (eXtensible Access Control Markup Language) is a prominent XML-based language used in ABAC systems for crafting these detailed security rules.

To manage the complexity of XML-based policies, specialized editors are employed. These tools aid administrators in creating, validating, and managing policies through features like syntax highlighting and validation tools, enhancing the ease and accuracy of policy management.

The use of XML in ABAC systems offers standardized, flexible, and interoperable methods for defining security policies, which supports fine-grained and dynamic access control across various data providers and platforms. In the ExtremeXP framework, XML in ABAC will enable a standard definition and management of complex access control policies for external distributed data and the execution tasks within the framework components.

6.1.2 Hyperledger Besu and Smart contracts

Smart contracts-based ABAC (Attribute-Based Access Control) represents an innovative approach to access control in decentralized environments, leveraging blockchain technology. In this model, a peer-to-peer network of nodes uses smart contracts to manage and enforce access control policies. These contracts facilitate the request and verification processes to determine whether access to a resource should be permitted.

Within this framework, a "resource" is defined according to the XML standard as any data that can be accessed through smart contracts-based ABAC. A "target" comprises the context expression, action, and the specific resource requested. The blockchain's inherent transparency allows any node within the network to audit access requests, targets, decision-making, and policy enforcement logs, ensuring a high level of accountability and traceability.

The smart contracts to implement the ABAC system are Policy Decision Point Smart Contract (PDPSC), Policy Administration Point Smart Contract (PAPSC) and Point Information Point (PIPSC), which are defined in D2.1 section 5.3.2. The Point Enforcement Point (PEP) is implemented outside of the blockchain as a middleware. By operating off-chain, the PEP can handle requests and enforce policies without facing the latency associated with blockchain transactions. It can also provide us with more flexibility in integration with the authentication service.

The smart contracts are deployed on a Hyperledger Besu network. Hyperledger Besu is an open-source Ethereum client developed under the Hyperledger project that supports our need for a private blockchain network and supports Ethereum's standard features, such as smart contracts and NFTs. Hyperledger Besu supports different consensus mechanisms based on different needs, including Proof of Authority (PoA) protocols IBFT 2.0, QBFT and Clique and Proof of Stake to use in Ethereum mainnet and public testnets.

6.1.3 Keycloak for authentication and JWTokens

Keycloak is an open-source access management solution (including authentication and authorisation services), offering a vast community for support and extensive documentation that supports fine-grained authorisation policies and the Attribute-Based Access Control mechanism. It can be extended with custom plugins with the support of Web3j, which is a Java library that provides integration between Keycloak and Ethereum, enabling the interaction of our smart contracts with Keycloak. Keycloak also supports current security protocols such as OAuth 2.0 and OpenID connect standards, which ensures integration with other systems and components of the ExtremeXP framework. These features make Keycloak a suitable choice to build the ExtremeXP access control system with.

In Keycloak a "Policy" is the rules and conditions that need to be met to grant access to a "Resource". A "Resource" is an object protected by access control policies. The resources are being hosted on a "resource server" that can respond to resource requests. "Permission" evaluates the policies for the corresponding resource, determining granting access or denial.

In Keycloak, the token endpoint is a server function that issues access tokens to OAuth2.0¹² clients, such as front-end applications, enabling them to access protected resources on a resource server, like back-end services, based on evaluated access policies.

Keycloak token endpoint generates JSON Web Tokens (JWTs) to handle authentication and authorization which is based on the OAuth 2.0 standard:

Access Tokens: In Keycloak, access tokens are JSON Web Tokens (JWTs) that contain claims about the user (such as user identity, roles, and additional attributes). Clients (applications or services) present these tokens when making requests to resources that require authenticated access.

Resource Access Tokens (RPTs): RPTs are specific to Keycloak's authorisation services and are used in the context of fine-grained access control, particularly when implementing the User-Managed Access (UMA) protocol. An RPT is issued by Keycloak when a user requests access to a protected resource, and it contains permissions that specify what actions the user can perform on that resource. Unlike regular access tokens that grant access based on roles or scopes, RPTs are permission-based and are evaluated against the resource's policies and the user's attributes.

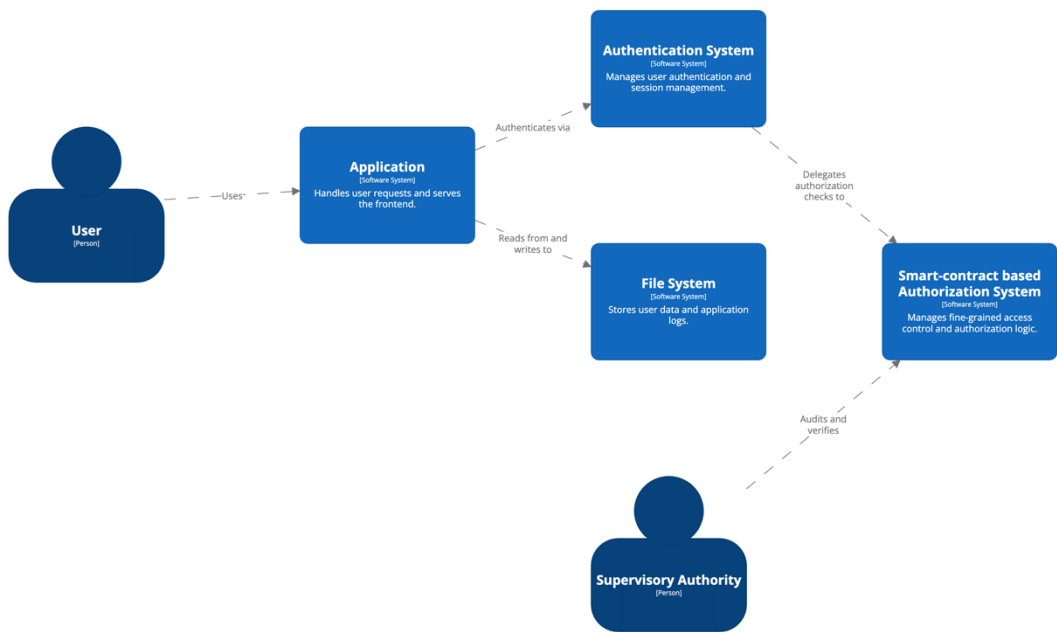
Refresh Tokens: In Keycloak, an optional refresh token is a type of token that applications can use to obtain a new access token when the current access token expires or becomes invalid. Unlike access tokens, which are used to access protected resources, refresh tokens are used to securely obtain new access tokens without requiring the user to go through the authentication process again. Refresh tokens are "optional". Using refresh tokens enhances security by allowing access tokens to have shorter lifespans, thereby reducing the risk if an access token is compromised.

6.2 Access Control System Architecture

In this section, we introduce the architecture of our integrated access control system, using the C4 model. The C4 model is a framework for visualising the architecture of software systems, breaking them down into four levels: Context, Containers, Components, and Code. The C4 Model's notation for mapping software architecture shows three entities: a "Person" representing the user or actor that interacts with the system; a "Container" in light blue, denoting an application or a data store; and a "Software System" in dark blue, indicating the broader system or service. The dashed arrow labelled "Relationship" illustrates the interactions between these entities. This model is chosen for its clarity and ability to turn complex software architectures into understandable components and interactions, making it a popular choice for conveying system designs to both technical and non-technical stakeholders.

As shown in Figure 26, users interact with an application that processes requests and presents information via a GUI. For authentication, the application defers to an Authentication System, managing login sessions and user credentials. Once authenticated, user actions are checked against a Smart-contract-based Authorization System, which handles fine-grained access control and logs these events for audit purposes. The File System is utilized by the application for reading and writing user data as well as storing logs. Meanwhile, a Supervisory Authority oversees the system, auditing and verifying compliance through the smart contracts' immutable logs.

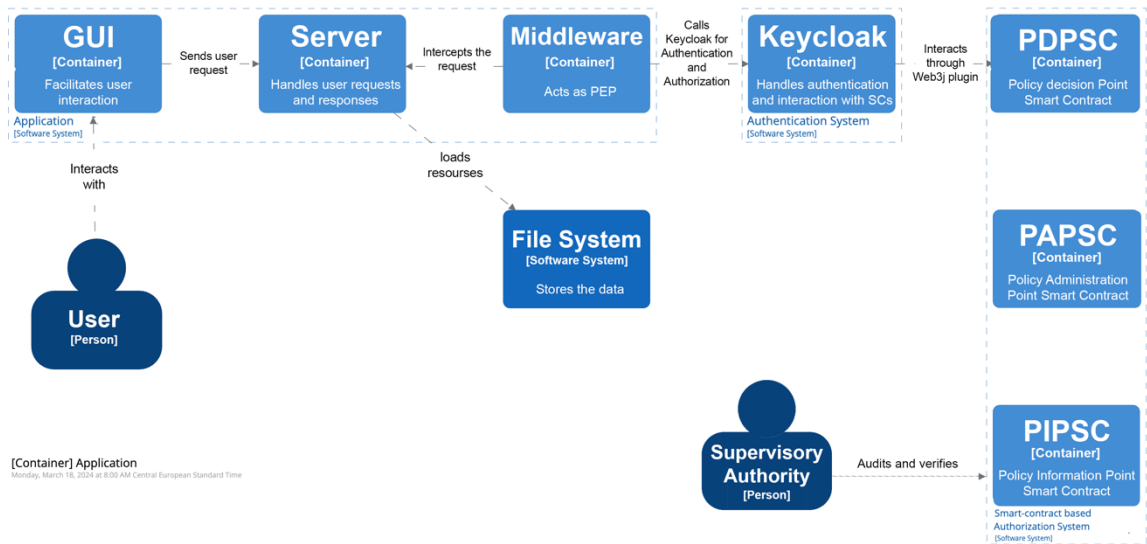
¹² https://openid.net/specs/openid-connect-core-1_0.html#TokenEndpoint



[System Landscape]
Sunday, March 17, 2024 at 8:13 PM Central European Standard Time

Figure 26. IAM system landscape

With more details at the container level, as depicted in Figure 27, users interact with the application's GUI, which channels user requests to a Server. The Server processes these requests, coordinating with a Middleware that serves as a Policy Enforcement Point (PEP) for our ABAC system. To authenticate a user using login, the Server communicates with Keycloak, which checks the credentials and issues tokens. After authentication, Keycloak facilitates authorization decisions via Web3j, which interacts with three smart contract components: PDPSC for policy decisions, PAPSC for policy administration, and PIPSC for attribute information.



[Container] Application
Monday, March 18, 2024 at 9:00 AM Central European Standard Time

Figure 27. IAM container view

6.2.1 Users roles

Based on GDPR and what we defined in D2.1 – section 5.3.2, we have three key roles in the system. These roles are crucial for ensuring compliance and safeguarding data within the system.

Data Controller: This is a user role that determines the purposes and means of processing personal data. Data controller is a person (or on behalf of an organisation) who decides how and why the data is collected and used, defining the access control policies that protect their data. The data controller evaluates and logs the authentication, authorisation, and access requests.

Data Processor: This is the User role in processing data on behalf of the data controller. The data processor is the User who sends requests to access a resource or to perform a task using the ExtremeXP framework.

Supervisory Authority: The supervisory authority is the auditor who checks compliance with the policies, attributes, and events captured from the access control logs.

Data Subject: A Data Subject refers to an individual who can be identified, directly or indirectly, whose personal data undergoes processing. They are at the centre of GDPR's protective measures. In our system, the Data Subject's personal information is protected by the policies established by Data Controllers and executed by Data Processors.

Data storage: A Data Storage, in the context of GDPR, refers to any file system or database where personal data of Data Subjects is held. It must be designed and operated in compliance with GDPR principles, ensuring data is processed lawfully, transparently, and securely.

6.2.2 Application

This is the web application that connects the user to the access control system.

GUI: A Graphical User Interface (GUI). The Graphical User Interface is the Front-end of the application that the user interacts with.

Server: A server that processes the requests, which is the back end of the application. It processes the access requests received by GUI from the user and calls the authentication system through PEP.

PEP: A middleware to map and relay the request to the Authentication system. The middleware connects the Keycloak to the application's back-end server. It serves as the Policy Enforcement Point of the ABAC protocol.

6.2.3 Smart contracts

Smart contracts are used to automate and enforce access control decisions transparently and securely, using blockchain technology's immutable and decentralised nature for transparency and accountability. Smart contracts and their functions are denoted as <smart contract name>:<function name>. A sender is a node or smart contract allowed to emit the transaction to run the function. There is a checking condition inside each function to verify if the sender is allowed to run the function, and if the required condition is true, the node is allowed to run the function. The SCs contain the function code and a relational database where each argument is saved and managed according to the execution of each function. The functions can add, remove, load, change, and save a variable in the relational database of the smart contract. For example, the database contained in PAPSC stores the policies for each the target in the following format (ContextExp, Actions, ResourceID).

Every transaction that executes a function is validated and mined in a new block added to the blockchain. The transaction payload logs the arguments input, such as the target, attributes, policies, and decisions used when executing the corresponding function. Any network node can search for an argument and retrieve all the transactions containing the argument.

PAPSC (Policy Administration Point Smart Contract): The contract implements the PAP functionality in the ABAC XACML model, where the policies are defined, managed, and stored. A policy is related to the PolicyID, ContextExp, Actions, ResourceID and WorkflowID. It contains four functions: PAPSC:CreatePolicy, PAPSC:ChangePolicy, PAPSC:RemovePolicy and PAPSC:LoadPolicy. The policies get implemented on this smart contract so that all the blockchain nodes can agree on them.

PIPSC (Policy Information Point Smart contract): The smart contract implements the PIP functionality in the ABAC XACML model, gathering the information necessary to evaluate the requests. PIPSC stores the addresses of all nodes. The attributes get implemented on this smart contract so that all the blockchain nodes can have an agreement on them. The PIPSC is connected to a set of Decentralized Oracles that provide the contextual attributes. These Oracles send attribute information on chain which is saved on the PIPSC memory table of attributes. The relational database used to store the ContextAttr may differ for each ContextExp supported in the system. The functions are PIPSC:AddContextAttr, PIPSC:GetContextAttr, PIPSC:RevokeContextAttr.

PDPSC (Policy Decision Point Smart Contract): The contract implements the PDP functionality in the ABAC XACML model, being the core decision point for any incoming access request. It contains only the PDPSC:EvaluateRequest function, which evaluates the access requests. This function is called by PEPSC:RequestAccess. The PDPSC takes the request from Keycloak and loads the corresponding policies from PAPSC and contextual attributes from PIPSC to take a grant/deny decision, based on the access request. The needed attributes get defined by PIPSC and then based on those attributes PDPSC calls the dataset of the attribute values provided by the Oracles, ensuring the validity of the values by a consensus algorithm.

KeycloakLoginEvents: The KeycloakLoginEvents contract introduces a mechanism for logging events related to logins to Keycloak. It defines an Ethereum event, encapsulating various data points, including the event type, timestamp, and client. The contract provides a function, KeycloakLoginEvents:saveKeycloakEvent, which is responsible for emitting the event. This function takes the parameters corresponding to the event's details and triggers the emission that records the login-related activities on the blockchain.

KeycloakAdminEvents: This contract is designed to log administrative events within the Keycloak system. It specifies an Ethereum event, structured to hold information about administrative actions. The KeycloakAdminEvents:saveKeycloakEvent function enables the logging of such events. By accepting detailed parameters about the administrative activity and emitting the Event, it serves as a record of Keycloak administration.

6.2.4 Context handling

The Policy Information Point Smart Contract (PIPSC) plays a crucial role in managing and recording attribute data within the system. It aggregates contextual information through Oracles, which serve as bridges fetching real-time data into the blockchain environment. This data can come from Hardware Oracles, such as biometric sensors like fingerprint readers, or Software Oracles that retrieve data via APIs. When evaluating a contextual attribute, PIPSC either confirms a unanimous value

reported by all Oracles or engages in a consensus evaluation to find an agreement on the attribute value and ensure the integrity and accuracy of the information.

6.2.5 Modelling the access policies

We have used a template in Figure 28 to register access control policies and the respective stakeholders for each use case scenario. The template involves a short description of the objectives and resources that must be protected. Moreover, it provides placeholders for expressing context-driven access control rules through its tabular format, i.e., to list the requester, action, resource, environment, and logical operators that combine rules and the desired access control decision.

Each use case needs to fill the template with all the relevant procedures specified concerning the need to access the data. Thus, the template constitutes the base for extracting the appropriate contextual information that should bind the access control decisions.

Policy Id & Name								
Objectives								
Resources/Scope								
Scenario								
Overview: Constraints: Definitions used below: Obligations:								
Explicit Access Attempt/Request info				Context Conditions				
#	Requester	Action	Resource	Contextual Attributes	Operator	Parameters	AND/OR	Permit / Deny
1.				Connection Protocol =				
2.				Connection Protocol =				
Rule Combining Algorithm								

Figure 28. Policy definition template

6.2.6 Access control enforcement flow

Figure 29 illustrates the access control flow, divided into two phases, Authentication and Authorisation:

Authentication Phase: The Data Processor logs in with their credentials (step 1.1), which the application's GUI sends to the Keycloak authentication server (steps 1.1 - 1.4). Upon successful verification, the server issues an access token back to the application (step 1.5), and the result is displayed by the GUI (step 1.6), certifying the client's identity. This authentication leverages the OpenID Connect protocol. The token then returns to the server (step 1.5), with the authentication outcome shown to the user (steps 1.7 - 1.8).

Authorisation Phase: The sequence begins with the user's action request (step 2.1), attaching the access token server-side (step 2.2). The PEP, functioning as middleware, intercepts this request (step

2.3) and requests a resource token from Keycloak (step 2.4), initiating policy evaluation via the Web3j plugin (step 2.5). The PDPSC compiles context from the PIPSC, relaying it to the PAPSC for policy assessment (steps 2.6 - 2.9), which either validates or denies the request (step 2.10). Depending on this decision, the PEP communicates a denial or permits the application to provide the requested resource to the user (steps 2.11 - 2.15).

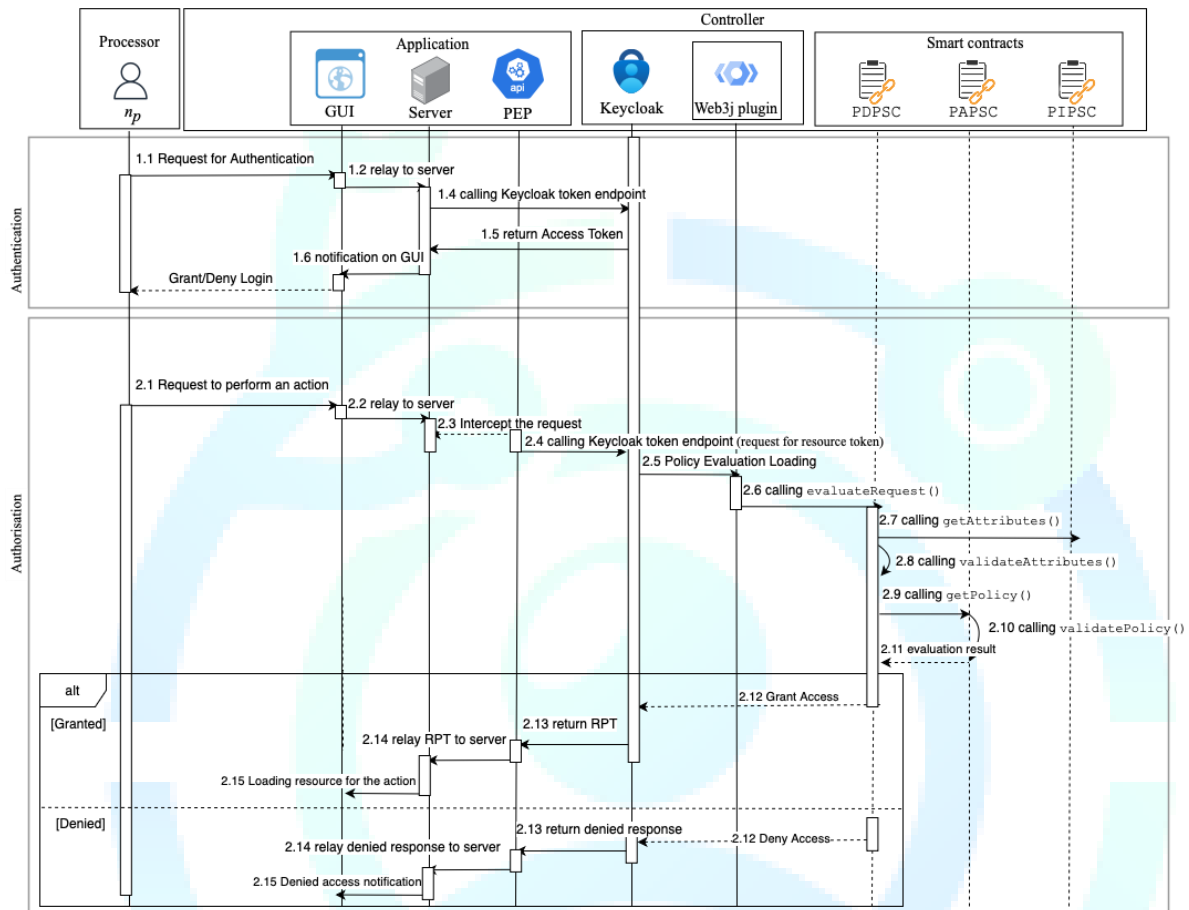


Figure 29. Flow of Authentication and Authorization (for UC5 IDEKO)

Now that we have designed our access control system and established its flow, the critical question arises: how do we dynamically manage and implement the access policies on PAPSC Figure 30 shows the flow that we envision for processing the policies, from mapping them into XACML standard using AMPLE to then translating them into Smart Contracts using a middleware, which will be developed in the second part of the project. Then, these policies can be loaded to the PDPSC to yield grant/deny decisions, which Keycloak, through the Web3j plugin, can read.



Figure 30. Flow of processing the policies

6.3 Demonstration of a Context-aware Access Control for Experiment-driven Analytics - UC5 IDEKO

6.3.1 User roles

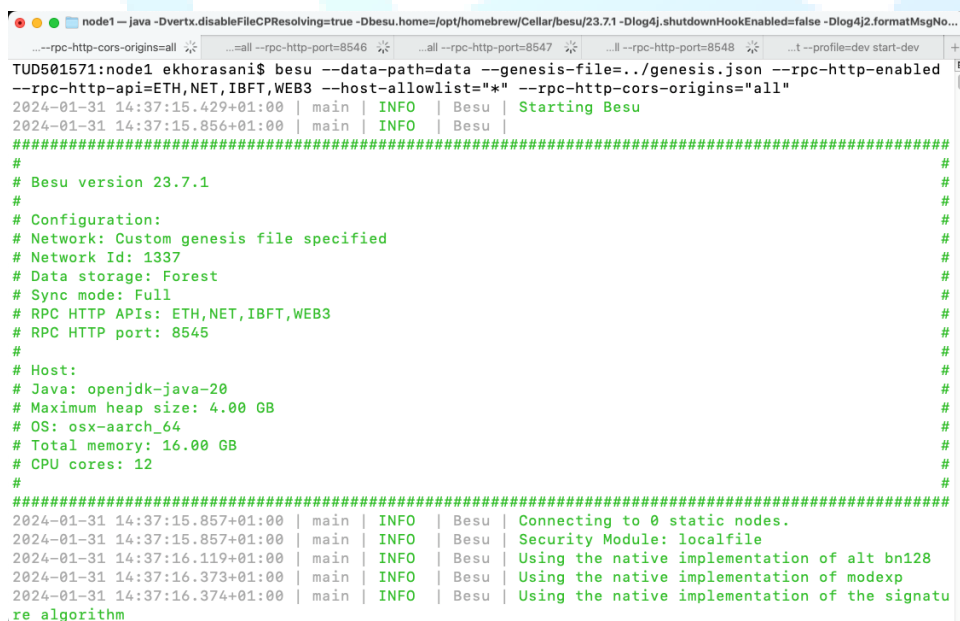
For our IDEKO use case (UC5), we have tailored an access control system that orchestrates consensus among multiple data controllers to manage the permissions of two user groups: Data Analysts and Machine Operators. IDEKO Data analysts have expertise in AI and manufacturing processes and are authorised to initiate and manage tasks for model development. Machine Operators, on the other hand, work in organisations that have bought machines and have permission to view results and perform validations on detected anomalies, thereby improving decision-making accuracy and reducing manufacturing downtimes.

Access control within the system is governed by three primary contextual attributes: Role, IP Address, and Location.

- **Role:** Users are assigned the roles "Data Analyst" or "Machine Operator", which dictate their access rights based on their job responsibilities.
- **IP Address:** This attribute validates the authenticity of access requests by checking if they originate from a trusted computer.
- **Location:** Access is controlled by the requestor's physical location, identified through latitude and longitude. This enables access permissions to be adjusted based on geolocation. Since location is a real-time dynamic attribute, it needs a context handler to verify, which will be developed and presented in the D5.1 deliverable.

6.3.2 Network and Nodes

We have implemented four nodes in our Hyperledger Besu network that operate under the IBFT consensus mechanism for two reasons. First, its simplicity and second, since it is a PoA-based algorithm and PoA protocol is better suited for permissioned private networks as we have in our system, it is easier to implement than PoS, which is more suited for public blockchains. The configuration and the command to run these nodes are shown in Figure 31 for the first node and Figure 32 for the fourth node.



```
node1 ~ java -Dvertx.disableFileCPResolving=true -Dbesu.home=/opt/homebrew/Cellar/besu/23.7.1 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.formatMsgNo...
...--rpc-http-cors-origins=all ...all --rpc-http-port=8546 ...all --rpc-http-port=8547 ...ll --rpc-http-port=8548 ...t --profile=dev start-dev +
TUD501571:node1 ekhorasani$ besu --data-path=data --genesis-file=./genesis.json --rpc-http-enabled
--rpc-http-api=ETH,NET,IBFT,WEB3 --host-allowlist="*" --rpc-http-cors-origins="all"
2024-01-31 14:37:15.429+01:00 | main | INFO | Besu | Starting Besu
2024-01-31 14:37:15.856+01:00 | main | INFO | Besu |
#####
# Besu version 23.7.1
#
# Configuration:
# Network: Custom genesis file specified
# Network Id: 1337
# Data storage: Forest
# Sync mode: Full
# RPC HTTP APIs: ETH,NET,IBFT,WEB3
# RPC HTTP port: 8545
#
# Host:
# Java: openjdk-java-20
# Maximum heap size: 4.00 GB
# OS: osx-aarch_64
# Total memory: 16.00 GB
# CPU cores: 12
#
#####
2024-01-31 14:37:15.857+01:00 | main | INFO | Besu | Connecting to 0 static nodes.
2024-01-31 14:37:15.857+01:00 | main | INFO | Besu | Security Module: localfile
2024-01-31 14:37:16.119+01:00 | main | INFO | Besu | Using the native implementation of alt bn128
2024-01-31 14:37:16.373+01:00 | main | INFO | Besu | Using the native implementation of modexp
2024-01-31 14:37:16.374+01:00 | main | INFO | Besu | Using the native implementation of the signatu
re algorithm
```

Figure 31. First Node in the Hyperledger Besu Network


```
node4 — java -Dvertx.disableFileCPResolving=true -Dbesu.home=/opt/homebrew/Cellar/besu/23.7.1 -Dlog4j.shutdownHookEnabled=false -Dlog4j2.formatMsgNo...
...--rpc-http-cors-origins=all ...--rpc-http-port=8546 ...all --rpc-http-port=8547 ...ll --rpc-http-port=8548 ...t --profile=dev start-dev +
TUD501571:node4 ekhorasani$ besu --data-path=data --genesis-file=./genesis.json --bootnodes=enode://206668839f47ebce721e878289c8349f9de40586daf35d0398ab96636a62432c6ece9d64a685978d6585c9cf52376c47f15
23c1b5bde69acbd6445defd086fcc@127.0.0.1:30303 --p2p-port=30306 --rpc-http-enabled --rpc-http-api=ETH
,NET,IBFT,WEB3 --host-allowlist="*" --rpc-http-cors-origins="all" --rpc-http-port=8548
2024-01-31 14:37:23.608+01:00 | main | INFO | Besu | Starting Besu
2024-01-31 14:37:23.890+01:00 | main | INFO | Besu |
#####
# Besu version 23.7.1
#
# Configuration:
# Network: Custom genesis file specified
# Network Id: 1337
# Data storage: Forest
# Sync mode: Full
# RPC HTTP APIs: ETH,NET,IBFT,WEB3
# RPC HTTP port: 8548
#
# Host:
# Java: openjdk-java-20
# Maximum heap size: 4.00 GB
# OS: osx-aarch_64
# Total memory: 16.00 GB
# CPU cores: 12
#
#####
2024-01-31 14:37:23.891+01:00 | main | INFO | Besu | Connecting to 0 static nodes.
2024-01-31 14:37:23.891+01:00 | main | INFO | Besu | Security Module: localfile
2024-01-31 14:37:23.971+01:00 | main | INFO | Besu | Using the native implementation of alt bn128
2024-01-31 14:37:24.033+01:00 | main | INFO | Besu | Using the native implementation of modexp
```

Figure 32. Last (Fourth) Node in the Hyperledger Besu Network

6.3.3 Modelling access control policies

We have initially defined four policies for IDEKO (UC5) that control the access to four actions in the framework: 1. Developing the model; 2. Showing the results; 3. Validating the results; and 4. Validating the input. Figure 33 illustrates these policies using a Policy definition template by AC-ABAC¹³. To showcase the proof of concept and build the first prototype, we have considered the first action, developing the model, as the policy to develop. This policy controls the access to the client data files. It gives permission only to Data analysts of IDEKO who are either present on the premises of IDEKO or using a computer with an IP address that is present in a whitelist of IP addresses. The Data analyst should also belong to the predefined list of the IDEKO data analysts team.

#	Explicit Access Attempt/Request info			Context Conditions	Permit/ Deny
	Requester	Action	Resource	Contextual Attributes Operator Parameters AND/OR	
1	Data Analyst	Developing the model	All the client data files	[Data_Analyst Location IN premises [Computer ip address is IN whitelist of IP addresses Data_Analyst belongs to the IDEKO data analyst team]	OR AND Permit
2	Machine Operator (Domain expert)	Showing results	Prediction results	[Machine_Operator Location IN premises [Computer ip address is IN whitelist of IP addresses Machine_Operator belongs to the Machine Operators team]	OR AND Permit
3	Machine Operator (Domain expert)	Validating the results	Validation files	[Machine_Operator Location IN premises [Computer ip address is IN whitelist of IP addresses Machine_Operator belongs to the Machine Operators team]	OR AND Permit
4	Data Analyst	Validating the input of the Machine Operator	Validation files	[Data_Analyst Location IN premises [Computer ip address is IN whitelist of IP addresses Data_Analyst belongs to the IDEKO data analyst team]	OR AND Permit
Rule Combining Algorithm			Deny unless permit		

Figure 33. Policy definition for IDEKO

¹³ <https://doi.org/10.1016/j.eswa.2022.119271>

6.3.4 Example tokens issued for IDEKO

Keycloak tokens are encoded in the JWT format, which means they are composed of three parts: header, Payload, and Signature. The payload section of the JWT contains the claims, which include information about the user, the token's expiration, issued time, and specific permissions (in the case of an RPT).

Figure 34 illustrates the authentication request and response generated by the Keycloak token endpoint. The authentication request is sent as a POST HTTP method to the token endpoint address, which contains the user's username, password, client_id, client_secret, and scopes, as defined on Keycloak. In the Preview pane, the response from the token endpoint is shown, including the Access Token generated as an encoded JWT, along with the expiry time of the token, the Refresh Token also as an encoded JWT and the expiry time of the Refresh Token.

Keycloak can authenticate in different ways. For simplicity, the *password* grant type is used here, which requires a *client_id* and a *client_secret*.

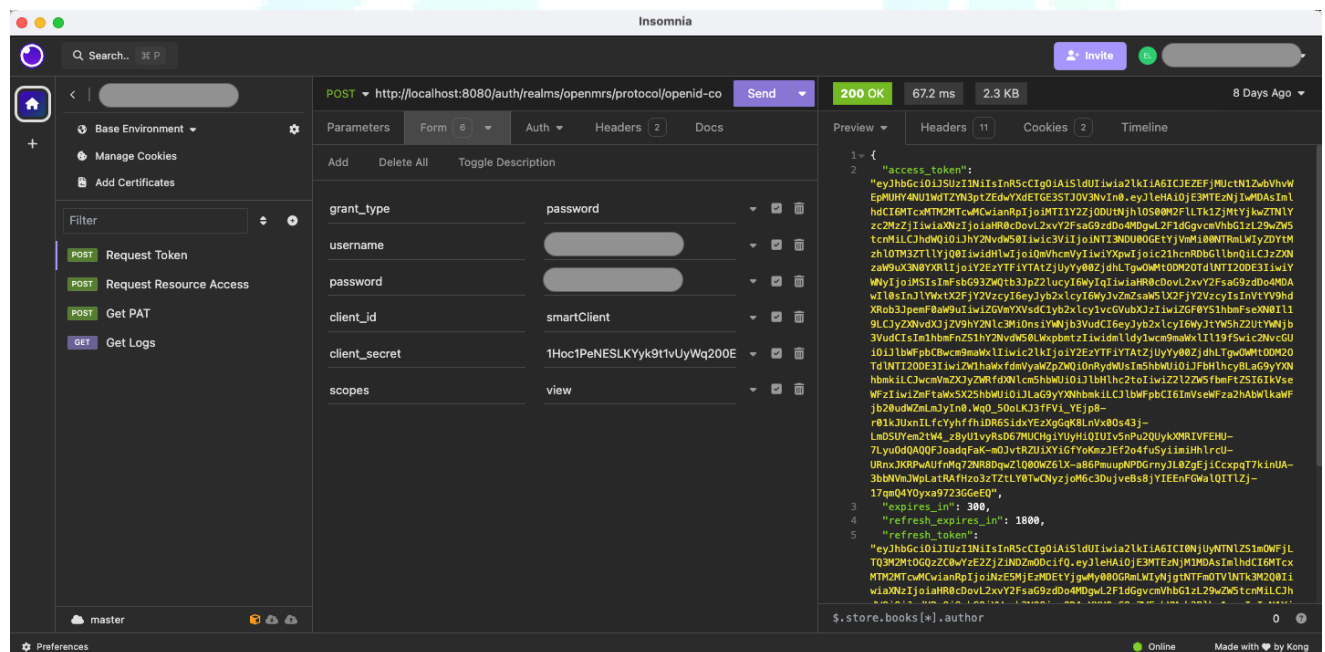


Figure 34. Access token generation for authentication

The access token is then sent to the Keycloak end point as an authorisation code in the header of the HTTP POST method, as seen in Figure 35 and 36. The request also consists of the following parameters:

- `grant_type`: Should be `urn:ietf:params:oauth:grant-type:uma-ticket` based on the Keycloak documentation¹⁴.
- `permission`: This is an optional parameter. A string that identifies the set of resources and scopes to which the client is requesting access.
- `audience`: This is an optional parameter. The resource server's client identifier that the client is attempting to access. If the permission parameter is defined, then this parameter must be provided. It acts as a hint to Keycloak about the situation in which permissions must be assessed.

¹⁴ https://www.keycloak.org/docs/latest/authorization_services/index.html

[illegible][illegible]

¹⁵ <https://www.rfc-editor.org/info/rfc6750>

6.3.5 The Middleware (PEP)

The **BlockchainValidator** class is an integral part of the access control system, interfacing with Keycloak to authenticate and authorise user requests based on blockchain-validated policies. It serves as a bridge between the application's user interface and Keycloak's authentication mechanisms, logging essential data for audit trails and implementing a policy check against user credentials and geolocation data. The class encapsulates the logic necessary to interact with blockchain smart contracts to ensure access decisions are consistent with established policies.

Listing 33. Validating access permissions

```
...
package nl.tudelft.tpm.extremexp.policy;

import org.keycloak.authorization.AuthorizationProvider;
import org.keycloak.authorization.model.Resource;
import org.keycloak.authorization.permission.ResourcePermission;
import org.keycloak.authorization.policy.evaluation.EvaluationContext;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.math.BigInteger;

public class BlockchainValidator {

    // Logger for this class to output information
    private static final org.slf4j.Logger log =
        LoggerFactory.getLogger(BlockchainValidator.class);

    // Map to hold policy identifiers and their descriptions
    private static final Map<String, String> policies = new HashMap<String,
String>() {{
        put("logs", "simple_policy");
    }};

    // Method to run a policy check, returns a Boolean
    private static Boolean runSimplePolicy(String role, String ipAddress,
BigInteger latitude, BigInteger longitude) {
        SimplePolicy policy = new SimplePolicy(); // Instance of SimplePolicy class
        try {
            // Validates the policy using the given parameters
            return policy.validate(role, ipAddress, latitude, longitude);
        } catch (Exception e) {
            // Rethrows an exception as a RuntimeException
            throw new RuntimeException(e);
        }
    }

    // Validates access to a resource based on permissions, context, and
    authorization
    public static boolean validate(ResourcePermission resourcePermission,
EvaluationContext context, AuthorizationProvider authorizationProvider) {
        // Get the associated resource from the resource permission
        Resource resource = resourcePermission.getResource();
        String role = null; // Placeholder for the user's role

        if (resource != null) {
            // Log all identity attributes
            for (Map.Entry<String, Collection<String>> entry :
context.getIdentity().getAttributes().toMap().entrySet()) {
                log.info(entry.getKey() + ":" + entry.getValue().toString());
            }
        }
    }
}
```

```

    };

    Collection<String> roles = new ArrayList<>(); // Stores roles from the
identity attributes
    // Check if realm roles exist and assign them to 'roles'
    if
(context.getIdentity().getAttributes().toMap().containsKey("kc.realm.roles")) {
        roles =
context.getIdentity().getAttributes().toMap().get("kc.realm.roles");
    }

    // Loop through roles to find 'data-analyst'
    String desiredRole = "data-analyst";
    for (String r : roles) {
        if (r.equals(desiredRole)) {
            role = r;
            break; // Exit loop once role is found
        }
    }

    // Default to the first role if no specific role is found
    if (role == null && !roles.isEmpty()) {
        role = roles.iterator().next();
    }

    // Log the chosen role
    log.info("Chosen role: " + role);

    // Get IP address from the authorization provider
    String ipAddress =
authorizationProvider.getKeycloakSession().getContext().getConnection().getLocalAdd
r();

    // Parse and log latitude and longitude from 'location' attribute
    String[] location =
context.getIdentity().getAttributes().getValue("location").asString().split(",");
    double latitude = Double.parseDouble(location[0]);
    double longitude = Double.parseDouble(location[1]);
    log.info("Location: Latitude = " + latitude + ", Longitude = " +
longitude);

    // Convert latitude and longitude to BigInteger and scale
    BigInteger scaledLatitude = BigInteger.valueOf((long) (latitude *
1_000_000));
    BigInteger scaledLongitude = BigInteger.valueOf((long) (longitude *
1_000_000));

    // Execute policy validation and return the result
    return runSimplePolicy(role, ipAddress, scaledLatitude,
scaledLongitude);
    }
    return false; // Default deny if no resource is linked
    }
}
...

```

The **SimplePolicy** class is a representation of a policy enforcement module that interacts with Ethereum-based smart contracts using Web3j, a lightweight Java library. Its primary function is to validate user requests against smart contract-defined policies, determining whether actions are permitted or denied based on user attributes such as role and location.

Listing 34. Enforcing Smart contract-based Authorizations

```
package nl.tudelft.tpm.extremexp.policy;

import nl.tudelft.tpm.extremexp.handlers.ContractLoader;
import nl.tudelft.tpm.extremexp.policy.contracts.PolicyEvaluationPoint;
import org.web3j.protocol.Web3j;
import org.web3j.protocol.core.methods.response.TransactionReceipt;
import org.web3j.protocol.http.HttpService;

import java.util.concurrent.atomic.AtomicBoolean;
import java.math.BigInteger;

public class SimplePolicy {
    // Initialize Web3j to interact with the blockchain
    private final Web3j web3j = Web3j.build(new
    HttpService(System.getenv("SM_CHAIN_ADDRESS")));
    // Address of the PolicyEvaluationPoint contract on the blockchain
    private static final String contractAddress =
    System.getenv("SM_EVALUATION_POINT_ADDRESS");
    // Helper to load contracts
    private final ContractLoader contractLoader;
    // Hex string to compare with a log topic for permission
    public static final String PERMITTED =
    "0xb656509e8ebb6eb7a22abef51ed8597f89bae6856ac48c7e7ee6fb33902ecb74";

    public SimplePolicy() {
        // Retrieve a single instance of ContractLoader
        contractLoader = ContractLoader.getInstance();
    }

    // Validates a policy by sending a request to the PolicyEvaluationPoint
    contract
    public boolean validate(String clientRole, String ipAddress, BigInteger
    latitude, BigInteger longitude) throws Exception {
        // Check if credentials are loaded
        if (contractLoader.getCredentials() != null) {
            // Load the PDPSC contract
            PDPSC contract = (PDPSC)
            contractLoader.getContract(contractAddress);
            // Send the policy evaluation request to the contract
            TransactionReceipt transactionReceipt =
            contract.evaluateRequest(clientRole, ipAddress, latitude, longitude).send();
            // Default permission to false
            AtomicBoolean permitted = new AtomicBoolean(false);
            // Process the transaction receipt logs

            TransactionReceipt transactionReceipt = contract.evaluateRequest(clientRole,
            ipAddress, latitude, longitude).send();
            System.out.println(TAG + "[publish] - Transaction complete¹: " +
            transactionReceipt.getTransactionHash());
            System.out.println(TAG + "[publish] - Transaction complete²: " +
            transactionReceipt.getLogs().toString());
            AtomicBoolean permitted = new AtomicBoolean(false);

            // Simulate checking the validation of each attribute
            if (!clientRole.equals(validRole)) {
                System.out.println(TAG + " - Role not validated.");
            }
            if (!ipAddress.equals(validIpAddress)) {
                System.out.println(TAG + " - IP Address not validated.");
            }
            if (!latitude.equals(validLatitude) ||
            !longitude.equals(validLongitude)) {
                System.out.println(TAG + " - Location not validated. You are not in
            the premises of IDEKO.");
            }
        }
    }
}
```



```

    }

    transactionReceipt.getLogs().forEach(transactionLog ->
transactionLog.getTopics().forEach(topic -> {
        System.out.println(TAG + "[publish] - Topic: " + topic);
        if (topic.equals(PERMITTED)) {
            permitted.set(true);
        }
    }));
    return permitted.get();
}
// Return false if credentials are not available
return false;
}
}

```

6.3.6 The Smart Contracts

Smart contracts are used to automate and enforce access control decisions transparently and securely, as outlined in section 6.2.3. The listing 35, shows the implementation of the smart contracts using solidity language. The policies for IDEKO (UC5) are hardcoded in PAPSC.

Listing 35. Implementing Access Control and Event Logging in Smart Contracts

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

// Ownable contract to restrict access to certain functions to the owner only
contract Ownable {
    // State variable to store owner's address
    address public owner;

    // Modifier to restrict function access to the owner
    modifier onlyOwner {
        require(msg.sender == owner, "Denied: You are not the owner.");
        _;
    }

    // Constructor sets the deployer as the owner
    constructor () {
        owner = msg.sender;
    }
}

// Contract to log Keycloak login events, inherits Ownable for ownership control
contract KeycloakLoginEvents is Ownable {

    // Event declaration for logging login events
    event Event(
        string _type,
        uint8 _timestamp,
        string _client,
        string _user,
        string _role,
        string _location,
        string _ip_address,
        string _details
    );

    // Function to emit a login event, callable by anyone
    function saveKeycloakEvent(
        string memory _type,
        uint8 _timestamp,
        string memory _client,

```

```

        string memory _user,
        string memory _role,
        string memory _location,
        string memory _ip_address,
        string memory _details
    ) public {
        emit Event(
            _type,
            _timestamp,
            _client,
            _user,
            _role,
            _location,
            _ip_address,
            _details
        );
    }
}

// Similar to KeycloakLoginEvents but for admin activities
contract KeycloakAdminEvents is Ownable {

    // Event declaration for logging admin activities
    event Event(
        string _type,
        uint8 _timestamp,
        string _operation_type,
        string _resource_type,
        string _realm,
        string _client,
        string _user,
        string _role,
        string _location,
        string _ip_address
    );

    // Function to emit an admin activity event, callable by anyone
    function saveKeycloakEvent(
        string memory _type,
        uint8 _timestamp,
        string memory _operation_type,
        string memory _resource_type,
        string memory _realm,
        string memory _client,
        string memory _user,
        string memory _role,
        string memory _location,
        string memory _ip_address
    ) public {
        emit Event(
            _type,
            _timestamp,
            _operation_type,
            _resource_type,
            _realm,
            _client,
            _user,
            _role,
            _location,
            _ip_address
        );
    }
}

// Contract for logging ABAC events, inherits Ownable for ownership control

```



```

contract KeycloakABACEvents is Ownable {
    // Event declaration for ABAC-specific activities
    event ABACEvent (
        string _event
    );

    // Function to emit an ABAC event, callable by anyone
    function saveKeycloakABACEvent(string memory _event) public{
        emit ABACEvent(
            _event
        );
    }
}

// Contract for the policy administration point, manages policy validation
contract PAPSC {
    // Events for evaluating roles, IPs, and locations
    event EvaluatedRole(string role, bool isValid);
    event EvaluatedIP(string ip, bool isValid);
    event EvaluatedLocation(int256 latitude, int256 longitude, bool isValid);

    // Functions to validate roles, IPs, and locations
    function validateRole(string memory role) public returns (bool) {
        // Simplified validation logic based on hardcoded values
        bool isValid = keccak256(abi.encodePacked(role)) ==
keccak256(abi.encodePacked("Data Analyst"));
        emit EvaluatedRole(role, isValid);
        return isValid;
    }

    function validateIP(string memory ip) public returns (bool) {
        // Simplified validation logic based on hardcoded values
        bool isValid = (keccak256(abi.encodePacked(ip)) ==
keccak256(abi.encodePacked("192.168.1.1"))) || keccak256(abi.encodePacked(ip)) ==
keccak256(abi.encodePacked("10.0.0.2")));
        emit EvaluatedIP(ip, isValid);
        return isValid;
    }

    function validateLocation(int256 latitude, int256 longitude) public returns
(bool) {
        // Simplified validation logic based on hardcoded values
        bool isValid = (latitude == 40712800 && longitude == -74006000); // Example
values scaled
        emit EvaluatedLocation(latitude, longitude, isValid);
        return isValid;
    }
}

// Contract for the policy evaluation point, uses PAPSC contract for validation
contract PDPSC {
    PAPSC public papContract;

    // Constructor to set the PAPSC contract address and initializes contract
interaction
    constructor(address _papContractAddr) {
        papContract = PAPSC(_papContractAddr);
    }

    // Event to log the outcome of a request evaluation
    event RequestEvaluated(bool isPermitted);

    // Main function to evaluate an access request based on role, IP, and location
    function evaluateRequest(string memory role, string memory ip, int256 latitude,
int256 longitude) public returns (bool) {

```

```

// Validate each attribute against policies defined in PAP
bool isRoleValid = papContract.validateRole(role);
bool isIPValid = papContract.validateIP(ip);
bool isLocationValid = papContract.validateLocation(latitude, longitude);

// Combine individual validations to determine overall access permission
bool isPermitted = isRoleValid && isIPValid && isLocationValid;
// Emit the result of the evaluation
emit RequestEvaluated(isPermitted);
// Return the permission status
return isPermitted;
}
}

```

6.4 Next steps

As we progress into the next phase of our project, several key initiatives are planned to advance our context-aware access control framework further. These initiatives are aligned with our project's key performance indicators (KPIs) and aim to enhance the efficiency, scalability, and functionality of our system.

6.4.1 Time complexity test and performance evaluation

One of our primary objectives is to assess the time overhead of the Smart contract-based Attribute-Based Access Control (ABAC) system compared to a traditional centralised access control system. In pursuit of KPI 6.3, a comprehensive time complexity test is planned to ensure that access decisions are delivered with less than 10% time overhead. This test will be conducted using a 4-node private Besu Network, employing the IBFT consensus algorithm. The complexity test scenarios will include traditional Keycloak ABAC flow and Smart contract-based ABAC flow, with a focus on measuring authorization time efficiency. Additionally, performance tests will be conducted on the proof of concept of our smart contracts using Hyperledger Caliper to evaluate transaction throughput, latency, and resource usage, ensuring their readiness for deployment in the ExtremeXP framework.

6.4.2 Implementation of additional context handlers for UC5

To further enhance the context-awareness of our access control system, we plan to implement at least two more context handlers specifically tailored for UC5. This initiative is aimed at meeting KPI 6.1, which focuses on refining the modelling of access control policies. By expanding the repertoire of context handlers, we aim to enrich the contextual information available for access control decision-making, thereby improving the precision and granularity of our policies.

6.4.3 Policy Modeling and Context Handling for Other Use Cases (UC1-UC4)

Expanding beyond UC5, our next steps involve extending policy modelling and context handling to cover other use cases, including UC1 to UC4. This initiative is aligned with KPIs 6.1 and 6.2, which emphasise the importance of comprehensive policy modelling and context awareness across all use cases. By applying our framework to a broader spectrum of scenarios, we aim to ensure its versatility and applicability across diverse contexts within the ExtremeXP framework.

6.4.4 Designing the Translator Middleware for Smart Contract Policy Mapping

Our current activities involve designing the translator middleware to map the policies in the XML for the ABAC web editor onto Smart Contracts policies. By developing this middleware translator, we aim to establish a bridge between policy specifications and their implementation execution on smart contracts, thereby facilitating the deployment and management of access control policies in a distributed environment.

7 Management of Decentralized Data and Knowledge for and from Experiments

7.1 Approach

The current focus of the data & knowledge management module is on the management of data pertaining to the execution of experiments and for providing secure access to the datasets used or produced by the experiments. The module will evolve to address additional knowledge management requirements, as these emerge through the use of ExtremeXP. We envisage to extend the focus of the component to capture, represent and provide access to insights, lessons learnt and good practices from experiments, which are usable for future users and future experiments.

Managed data refer to the specification of running or already run experiments, calculated metrics for each experiment but also for each CAW variant within an experiment, metadata pertaining to the execution and the progress of experiments (e.g., duration), as well as metadata pertaining to the outputs of CAW variants (e.g., the location of the produced datasets). Datasets are provided by data creators and are made available via a distributed file system. Data engineers are responsible for specifying data access policies, which are hosted by Knowledge Management. Experiment Modelling, Experiment Execution and User Interaction modules must go through the Knowledge Management module whenever they need to obtain access to datasets. This module supports framework-wide trustworthiness and traceability, as explained in more detail in 7.5.6

Decentralized technologies present a compelling opportunity to revolutionize the contemporary data landscape across various industries and domains. Within the framework of this work package, it is imperative to delineate a layered approach and a collaborative scheme among the involved technologies to attain the objectives of a Decentralized Architecture supporting Data Management for the respective elements participating in one or more swarms within the ecosystem.

The transition toward a decentralized data management ecosystem necessitates a departure from centralized data storage and management systems toward a distributed yet disparate model. In this model, data is dispersed, processed, and administered across multiple network nodes. Such a transition holds the potential to bolster data security, privacy, and accessibility, while concurrently fortifying resilience against data breaches and service interruptions.

7.2 Conceptual architecture

The term of a Decentralized Data Management (hereinafter DDM) refers to a data management approach that distributes data processing and storage demands across multiple nodes participating in the network. The decentralized aspect is driven by the encapsulation of a Pub/Sub/Query protocol that unifies data in motion, data at rest and computations that may occur from the core up to the edge of the network.

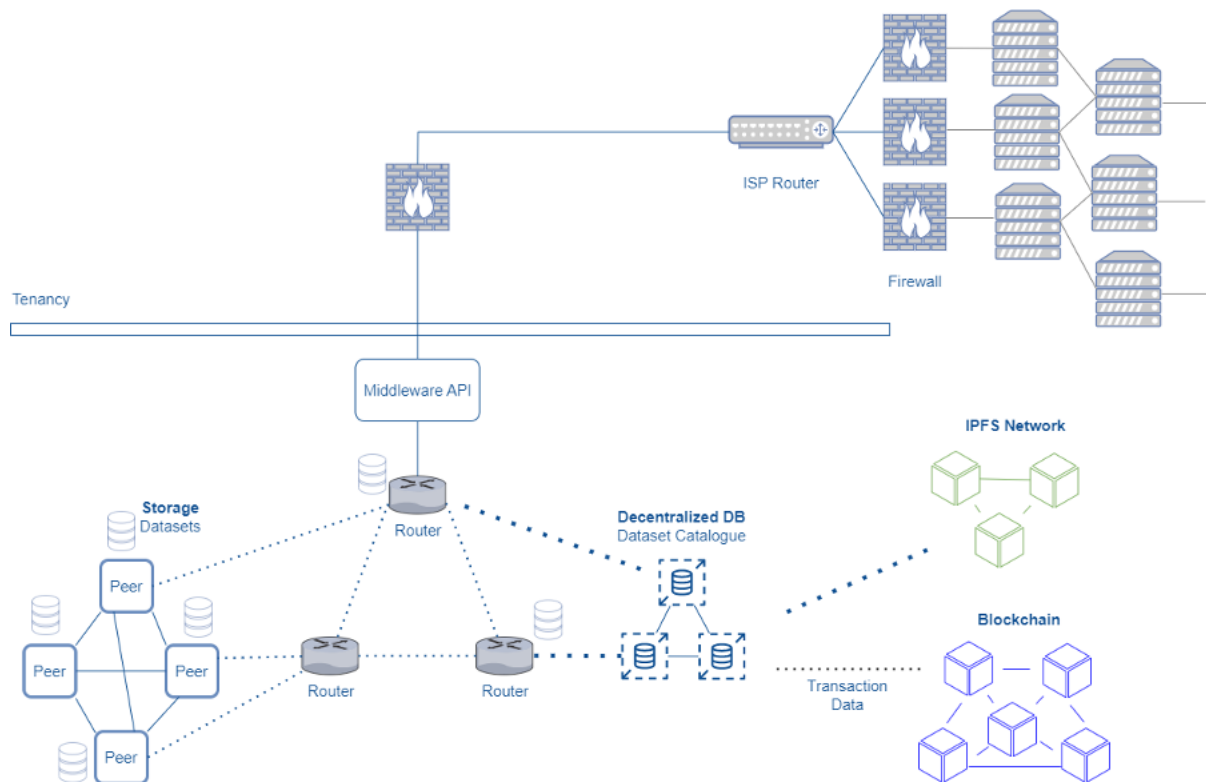


Figure 37. Decentralised data management conceptual architecture

The proposed architecture aims to provide location-transparent abstraction for high performance pub/sub and distributed queries across heterogeneous systems enabling the aspects and functionality that a Decentralized Data Management requires (DDM).

A DDM, offers to the entailed entities of the ecosystem, the ability to scale their data processing and storage capabilities flexibly, by adding or removing nodes as necessary (demand-wise), accommodating large volumes of data, and supporting the growing processing needs. This decentralized architecture enhances resilience against system failures, as the failure of a single node does not disrupt the network's overall functionality, ensuring continuous data availability.

Additionally, DDM can significantly improve data processing and analytics performance through distributed processing across multiple nodes, leading to quicker data queries and analysis. Moreover, the distributed nature of DDM supports enhanced data privacy and compliance, as the risk of data breaches is reduced through data encryption and storage across various nodes, thereby minimizing the consequences of any single point of failure.

In this conceptual architecture, datasets are distributed across multiple nodes, datasets can be stored in various or multiple nodes with local file systems or databases according to the Zenoh configuration satisfying the extreme XP framework requirements in terms of safe persistent replicated storage and fast and high availability while keeping the data stored in individual nodes if needed.

Metadata associated with datasets resides within a decentralized database, enabling sophisticated queries and supporting the extreme XP framework's dynamic dataset discovery capabilities. Dataset paths and related metadata are accessible through this decentralized database, which also manages the dataset catalog and metadata. Additionally, it handles smart contract transaction and IPFS data for NFT provenance mechanisms, ensuring immutable tracking of user data.

The middleware API provides a unified approach for interacting with the Zenoh network and managing metadata within the decentralized database. It offers the opportunity to implement Attribute-Based Access Control (ABAC), enhancing security and data governance.

7.3 Components Functionality

At the heart of the component lies Zenoh¹⁶, which facilitates efficient data access, processing, and distribution in real-time. Zenoh is a Pub/Sub/Query protocol that provides a set of unified abstractions to deal with data in motion, data at rest and computations at Internet Scale. Zenoh is capable of running above a Data Link, Network, or Transport Layer as shown in Figure 38. This capability is inherent in the protocol design, which establishes and expands upon a Zenoh Session Protocol. This protocol offers abstractions for ordered best effort and reliable channels, accommodates various priorities, and supports an unlimited Maximum Transmission Unit (MTU). The mission of Zenoh in the foreseen architecture is to offer a dynamic set of components hosted on the nodes supporting real-time data distribution, for interdisciplinary research in IoT technologies, environmental science, and sustainable urban planning according to use cases.

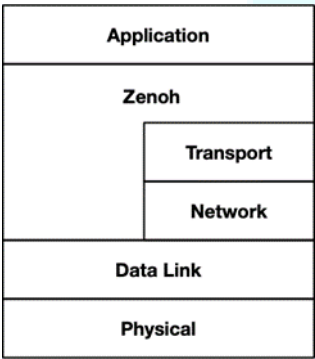


Figure 38. Zenoh positioning on ISO/OSI Model

Zenoh operates within a network of interconnected peers, clients, and routers, each playing a distinct role in facilitating efficient data communication and management. Peers in Zenoh represent individual nodes or entities within a distributed network. Each peer acts autonomously and collaboratively to share, process, and manage data in a decentralized manner. Peers can host various applications and services, making them capable of both producing and consuming data. Through peer-to-peer communication, Zenoh enables seamless data exchange and collaboration across distributed environments, enhancing scalability, resilience, and efficiency.

Clients in Zenoh are entities or applications that interact with the Zenoh middleware to access and manipulate data. Clients can be producers, consumers, or both, depending on their role in the data communication process. Producers generate data and publish it to the Zenoh network, while consumers subscribe to specific data streams and receive updates. Clients leverage Zenoh's data-centric communication model to exchange data with other peers and clients, enabling real-time data streaming, event processing, and distributed computing.

Routers serve as intermediaries within the Zenoh network, facilitating data routing and forwarding between peers and clients. They manage the dissemination of data across the network, ensuring efficient delivery and optimal utilization of network resources. Moreover, routers play a critical role in maintaining network connectivity, resolving communication paths, and optimizing data transmission paths based on network topology and conditions. By orchestrating data flows and

¹⁶ Zero Overhead Network Protocol <https://zenoh.io/>

managing network traffic, routers enable scalable and resilient communication within distributed Zenoh deployments.

By default, Zenoh applications operate in peer-to-peer mode, facilitating direct communication among all applications within the local network. This mode fosters seamless interaction without relying on centralized infrastructure, promoting agility and flexibility in data exchange.

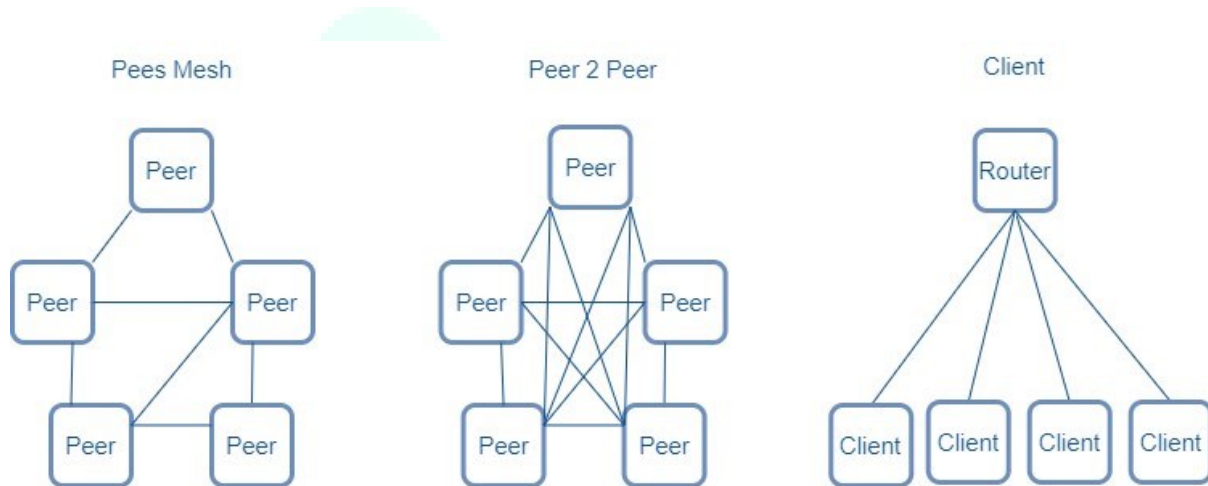


Figure 39. Zenoh peers and routes

Zenoh applications in peer mode (Figure 39) employ both multicast and gossip scouting techniques to discover other applications or routers within the network. Multicast scouting utilizes multicast communication to discover local peers, while gossip scouting forwards discovered applications to newly scouted peers. These mechanisms ensure robust connectivity and dynamic network discovery.

In scenarios where maintaining multiple sessions with peers is impractical for scalability or resource constraints, Zenoh applications can operate in client mode. In this mode, applications maintain a single session with a designated router, providing connectivity to the broader system while minimizing session overhead.

In a mesh network where direct peer-to-peer connections are unfeasible, Zenoh applications in peer mode can utilize a link-state protocol to communicate within the mesh. This approach enables efficient data exchange even in complex network topologies, ensuring connectivity across distributed environments.

The Zenoh router, executable as `zenohd`, offers support for loading plugins either during startup or at runtime, contingent upon write permission being configured on its admin space.

Zenoh routers play a pivotal role in routing data between clients and local subnetworks of peers. Deployable in various topologies, routers must be manually configured with the endpoints of other routers to establish interconnections, enabling seamless data transmission across distributed Zenoh networks.

7.3.1 Zenoh Backends and Volumes

The storage architecture of Zenoh nodes relies on a flexible system consisting of a storage manager plugin and dynamically loaded backends. These backends, which include options like S3, InfluxDB, RocksDB, and File System, serve as interfaces to various storage technologies such as databases and local file systems. Each backend is associated with a GitHub repository and documentation for easy reference and implementation.

Given the diverse storage requirements of Zenoh nodes, the storage manager plugin relies on dynamically loaded "backends" to facilitate storage functionalities. Typically, these backends utilize third-party technologies, such as databases, to manage storage effectively. An added benefit of utilizing databases as backends is the potential for them to serve as an interface between Zenoh infrastructure and external systems that interact with the database independently. For instance, multiple instances of the same backend can be loaded with varying configurations, known as "volumes," allowing for flexible storage management. These volumes, akin to storing multiple files on a filesystem volume, can be utilized by numerous storages as needed.

During runtime, Zenoh routers can manage volumes and storages dynamically via their admin space, provided it is configured to be writable. This management can be performed either through the configuration file or the zenohd command line option. The admin space also offers a convenient method for editing configurations via the REST API, allowing for runtime configuration operations. This approach provides flexibility and ease of management, enabling users to adapt their storage configurations according to evolving requirements.

7.3.2 Zenoh taxonomy of components

Resources, Key Expression and Selectors. Zenoh operates over resources. A resource is a *(key,value)* tuple, where the key is an array of arrays of characters. A set of keys can be expressed by means of a key selector, which may include * or ** which expand respectively to an arbitrary array of characters not including the separator, and an array of arrays of characters. The value can vary widely, ranging from a single integer to a complex entity like a PDF document or even an entire folder containing multiple files and directories. The aim is to let Zenoh, based on necessary business logic on the published data, to select a set of resources by using a queryable interface referred as a selector¹⁷. The syntax supported by Zenoh's selector is **keyexpr?arg1=val1&arg2=value** – where keyexpr is a key expression as defined above. Some args, such as those for indicating filters, projections and time intervals are built-in application-specific semantics can be added by defining additional arguments. An example of the selector filtering information across the Routers of the ecosystem follows:

home/*/sensor/temperature?_filter="temp>25"&_project="hum", among all the temperature sensors in a location it would select those whose value is greater than 25 and project their humidity.

¹⁷ Zenoh Selector: <https://github.com/eclipse-zenoh/zenoh-java/blob/main/examples/src/main/java/io/zenoh/ZQueryable.java>

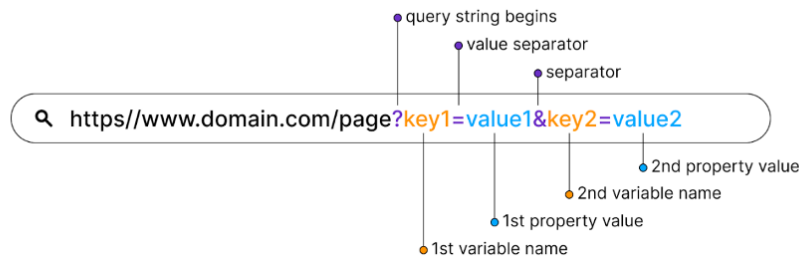


Figure 40. Zenoh url filtering leveraging selectors (Key1: First variable name; Key2: Second variable name; Value1: First property value; Value2: Second property value;?: Query string begins; =: Value separator; &: Parameter separator)

Furthermore, Zenoh employs a Publishers and Subscribers model, where Publishers serve as sources of values for a key expression, pushing data, while Subscribers act as receivers, pulling values for a key expression.

7.4 Prototype Zenoh Configuration & Rest API

To demonstrate the intended purpose and functionalities of the components described in Components Functionality, we employ straightforward examples from Use Case 5 of the project titled "Failure Prediction in Manufacturing." This particular scenario revolves around a factory machine following a predefined path, gathering high-frequency data concerning its position, expected position, consumed current, and the commanded movement. Various types of machine failures may occur, including issues with the machine's screw, bearings, or motor. The objectives of this use case are as follows:

- i. Efficiently gather and analyze high-frequency sensor data from the machine to identify anomalies.
- ii. Develop and maintain an anomaly detection classifier capable of accurately recognizing machine anomalies.
- iii. Enable user validation and feedback to continually enhance the anomaly detection model.

In order to support the above objectives, we configure and deploy a network of Zenoh nodes, operating in peer-to-peer mode, as well as a Flask-based API on top of the Zenoh network to provide a unified interface for external clients to interact with the components of the Zenoh Network. The Zenoh topology consists of two Filesystem storage nodes, one InfluxDB node and an S3-Object Storage node (MinIO). Each node is assigned a different IP address and is accessible via different ports for Zenoh's core functionality as well as for administrative purposes. Each node is configured via a Zenoh specific configuration json5 file. These configuration files are specific to the type of nodes involved in the Zenoh topology. For the given described topology, we make use of:

- zenoh-fs.json5 for the two Filesystem storage nodes
- zenoh-influxdb2.json5 for the InfluxDB node
- zenoh-s2.json5 for the an S3-Object Storage node

The provided configurations specify different Zenoh settings and plugins, such as the storage manager plugin for handling data storage and retrieval, along with configuration details for different types of storage volumes (filesystem, InfluxDB, S3-Object Storage). Additionally, the configurations include settings for gossip protocol, auto-connect options, replication options and REST plugin settings.

This topology showcases a distributed system architecture leveraging Zenoh instances for data management and communication, alongside supporting services such as InfluxDB and MinIO for data storage and Flask API for application layer interaction.

The chosen configuration for this prototype application¹⁸ enables us to demonstrate alternative approaches to data propagation for Use Case 5. In this scenario, the input data specified in Use Case 5 comprises files organized into three distinct folders: mechanical anomalies, electrical anomalies, and non-anomalous data. We configure the Zenoh network to support methods for handling file CRUD (Create, Read, Update, Delete) operations under predefined paths, utilizing publish-subscribe patterns. Additionally, the Zenoh network is configured to receive sensor data (time-series data) emitted from various nodes.

Regarding the handling of files, the dataset owner can upload files to geographically distributed filesystem nodes. We leverage Zenoh network replication options and designate Zenoh nodes 1 and 4 (filesystem nodes) to replicate data uploaded between them. For the management of time-series data, we assume that these emissions are stored in the local temporary storage of a node and are replicated to the InfluxDB of node 2 within the Zenoh network.

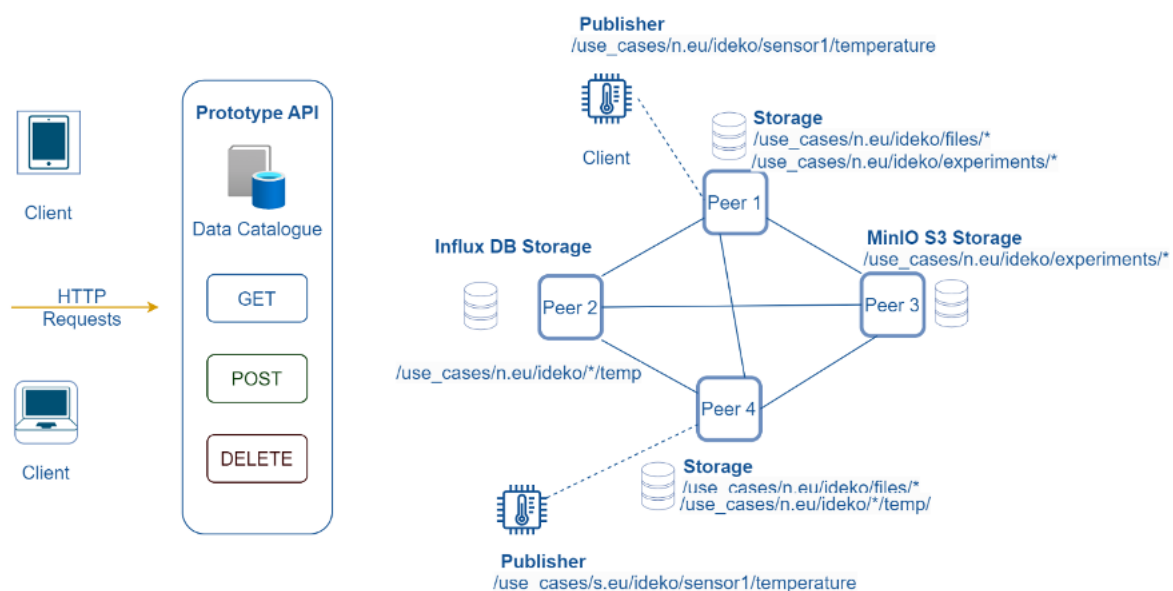


Figure 41. Prototype Rest API for file management

We developed and deployed a REST API (Figure 41), tailored for managing files within a distributed system, specifically designed to address the parameters outlined in Use Case 5 - "Failure Prediction in Manufacturing." Leveraging Flask and its extensions, including flask-restx, the application offers a structured framework for handling various file operations.

Central to the application are its endpoints, meticulously crafted to execute distinct functionalities pertinent to file management. The supported functionalities allow actions such as file listing, uploading, downloading, and deletion, all neatly encapsulated within Flask-REST resources, designated as FileListResource, and FileResource. Accessible via HTTP methods such as GET, POST, and

¹⁸https://colab-repo.intracom-telecom.com/colab-projects/extremexp/knowledge-management/zenoh_tests/-/tree/master?ref_type=heads

DELETE, these endpoints orchestrate the files-in-motion related tasks, facilitating seamless integration with the proposed attribute based access control mechanism.

Currently, the application uses a Postgres Database for user data and dataset-catalogue management, as well as a simple jwt-authentication mechanism which are expected to be replaced by the DecentralizedDB described in management and dataset catalog.

The data within Zenoh storages are organized in a hierarchical tree format, following the structure: "use_cases/<use_case>/<folder>/<subfolder>." Here, the "use_case" segment ranges from one to five, while the "folder" segment includes categories such as "input data", "output data", "experiments", and "metrics". For instance, under "use_case" 5, under "input_data" which pertains to "electrical anomalies", "mechanical anomalies", "and non-anomalies" as noted in the uploaded GitLab repository by use case 5, the data is stored accordingly. This basic structured approach ensures that data are stored with meaningful organization, facilitating efficient access and management within the Zenoh framework.

While currently following this basic structured approach, we have established these endpoints with base parameters in the path. However, as our system evolves, we anticipate incorporating structured data to enable querying based on a schema, thereby enhancing flexibility and adaptability within the Zenoh framework. Additionally, with the implementation of decentralized databases, we envision having a full schema, further enriching our querying capabilities, and bolstering our system's scalability and efficiency.

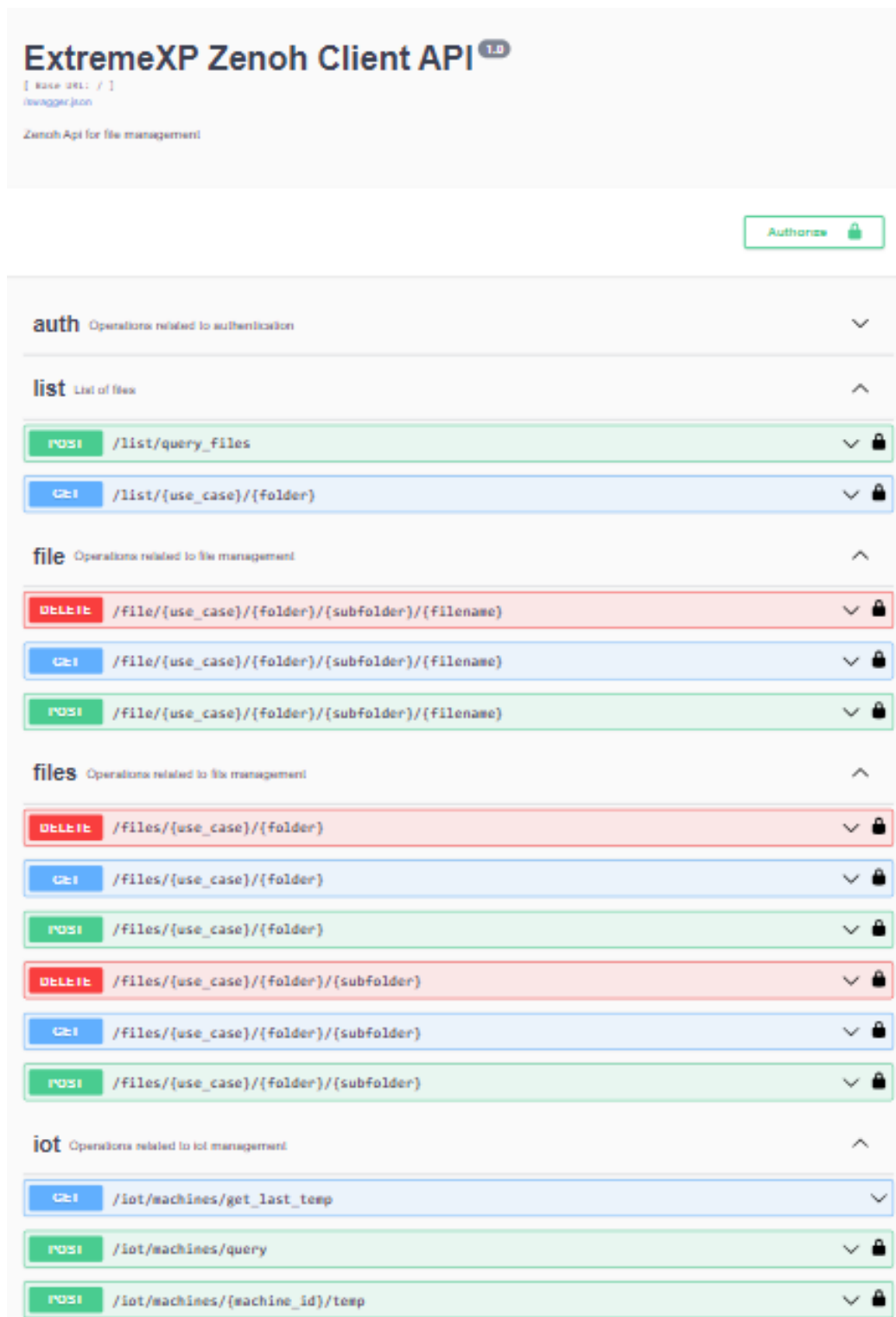


Figure 42. Zenoh API Client

The application utilizes Swagger documentation principles, generating detailed API documentation with the flask-restx extension. This documentation offers clear insights into available endpoints and their functions, making it easier for users to understand and navigate the application (Figure 42). Detailed documentation can be found in the appendix for further reference.

Finally, a demo interface has been developed in order to interact with the available endpoints in a user-friendly way, as shown in Figure 43.

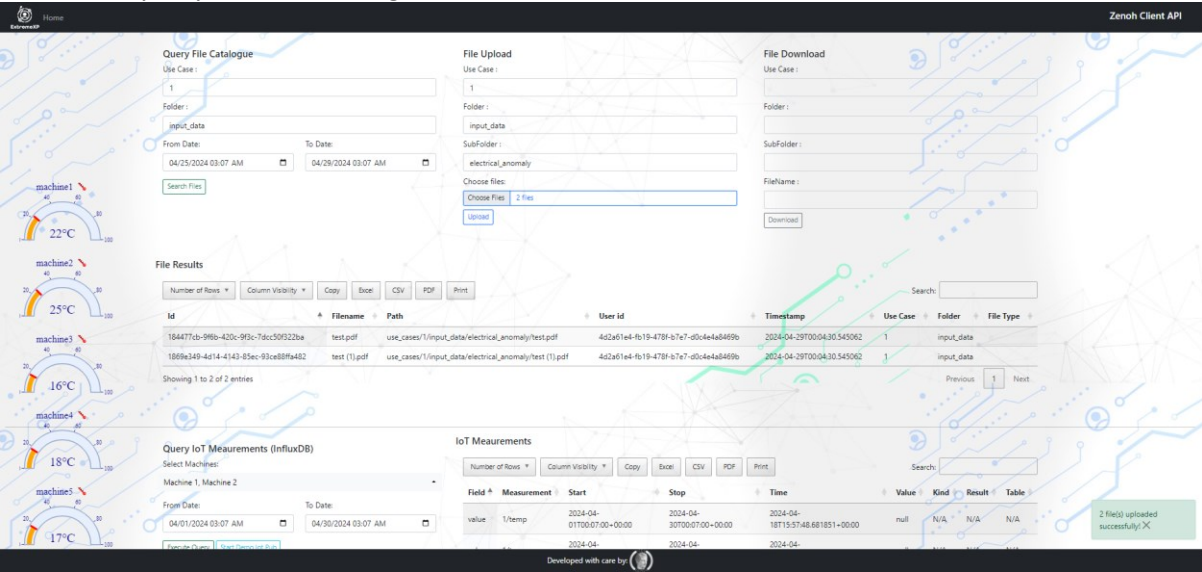


Figure 43. Zenoh API Client UI

7.5 Relevant Technologies and Investigations

In addition to the Zenoh system, we have explored the capabilities and applicability of the technologies described in this section.

7.5.1 Interplanetary File System (IPFS)

Apart from Zenoh persistent storages, we are also exploring the use of the Interplanetary File System (IPFS). To evaluate its efficiency and integration potential within our Zenoh-based architecture, we have deployed a private IPFS cluster consisting of four nodes in different Virtual Machines. We conduct experiments that assess both the performance of IPFS and its compatibility with our architecture. Through these experiments, we aim to optimize our system for decentralized data storage and retrieval while ensuring seamless integration between Zenoh and IPFS components. We also aim to utilize it for NFT provenance smart contracts, relying on its decentralized and immutable storage capabilities to securely and transparently track digital asset ownership and provenance.

Sharing data on a private IPFS network involves replicating copies across multiple nodes, a strategy that not only enhances resilience against node failures but also boosts accessibility regardless of specific node availability. This capability is particularly advantageous for large networks, where node loss or temporary unavailability could disrupt operations. Therefore, ensuring consistent and reliable data replication becomes crucial for the operational efficiency of an IPFS private network. By reducing dependencies on single origin servers, file distribution becomes more effective across networks, while also diminishing latency and enhancing speed. The collective participation of nodes further bolsters safety, as it distributes the responsibility of data storage and retrieval. Moreover, an IPFS private network provides an additional layer of control over content access by restricting network interaction solely to trusted nodes, thereby offering heightened security measures.

Every object added to an IPFS network is assigned a unique cryptographic hash, functioning as a digital fingerprint that precisely represents its contents. These identifiers are derived directly from the content itself rather than its location, meaning that duplicate files result in identical hashes, ensuring

that each hash points to a distinct piece of information. This feature guarantees data integrity, as even a minor modification to stored content results in a completely different hash. Thus, cryptographic hashes not only facilitate efficient data storage but also serve as dependable validators when retrieving shared digital assets, an extremely useful feature for smart contract operations within blockchain networks.

7.5.2 Decentralized DB

While Zenoh offers valuable capabilities for managing data flow within our applications, it becomes evident that it may not fully meet our requirements in terms of storing and querying metadata related to data assets. Recognizing the importance of robust metadata management in our framework, the presence of a decentralized database emerges as crucial.

7.5.3 Dataset Catalog

To maintain a resilient system architecture with no single point of failure and provide a decentralized dataset catalog with metadata for heterogeneous datasets stored across different locations, decentralized databases such as OrbitDB and GunDB stand out as promising options. OrbitDB, built on top of IPFS, offers distributed, peer-to-peer database capabilities, ensuring data integrity and availability across the network. Similarly, GunDB provides decentralized data synchronization and storage, facilitating real-time updates and collaboration in a distributed environment. Alternative options are included in Appendix. By integrating these decentralized databases into our architecture, we aim to establish a reliable mechanism for storing and querying metadata related to datasets.

7.5.4 Distributed Ledger Data

Regarding the transaction data produced in the Hyperledger Besu, we shall offer a store and query mechanism. Besu offers APIs and interfaces that allow developers to interact with the blockchain and query transaction data programmatically. These include JSON-RPC APIs and GraphQL APIs, which can be used to retrieve transaction information based on specific criteria such as block number, transaction hash, or contract address. We intent to incorporating transaction data from the Hyperledger Besu network into these databases. We aim to incorporate transaction data from the Hyperledger Besu network into these databases, enhancing querying capabilities to enable seamless access to transactional information alongside other metadata. This approach establishes a robust foundation for our applications, ensuring effective management and utilization of metadata resources within a decentralized ecosystem.

7.5.5 API

For seamless integration and enhanced accessibility, an API-driven layer will be exposed to provide a unified interface for external clients to interact with the knowledge base. This layer will abstract the underlying complexity of the decentralized network, offering simplified access to the data and functionalities within the framework. Moreover, to ensure security and controlled access, the API-driven layer will be protected by an Attribute-Based Access Control (ABAC) mechanism, currently under development. ABAC will provide granular control over user access based on specific attributes, such as roles, permissions, and contextual information, enhancing the overall security posture of the system while enabling fine-grained access control policies.

7.5.6 NFT Provenance and Traceability

Safeguarding the ownership rights of AI assets and honoring the contributions of their creators and collaborators is crucial to prevent misappropriation. Non-fungible Tokens (NFTs) offer a robust solution by leveraging blockchain technology to uniquely represent assets and securely associate them with their rightful owners. NFTs were introduced with Ethereum's improvement proposals and

became a standard under ERC-721 (Ethereum Request for Comments 721). Unlike the fungible tokens standard (ERC-20) commonly used for cryptocurrencies, NFTs are unique and non-interchangeable. At the same time, NFTs reserve the right to use and distribute assets and manage them in public records with indisputable proof of ownership to ensure appropriate remuneration. This is because the unique token can be transferred while keeping an immutable history of a digital asset's information.

We are endeavoring to establish a robust Non-Fungible Token (NFT)-based mechanism to track the provenance of datasets. This initiative aims to revolutionize how metadata is associated with datasets, particularly for experiment-driven analytics. By attaching metadata to datasets, we strive to achieve transparency, immutability, and uniqueness in characterizing data resources. This approach enables us to transparently track the origin, evolution, and usage of datasets, ensuring their integrity and reliability throughout their lifecycle.

Metadata plays a pivotal role in unlocking the meaning embedded within vast datasets. By providing contextual information about the data, metadata empowers analysts to derive insights, make informed decisions, and extract maximum value from the available resources. The transparent tagging of datasets with metadata not only facilitates their identification and discovery but also enables seamless association and integration across various domains and applications within an organization. This holistic view of data enhances collaboration, accelerates innovation, and fosters data-driven decision-making processes.

In our pursuit of establishing a comprehensive dataset provenance mechanism, we are considering the InterPlanetary File System (IPFS), which offers a decentralized and resilient platform for storing data in an immutable manner. By leveraging IPFS, we aim to create a reliable infrastructure for preserving metadata, thereby safeguarding the authenticity and integrity of dataset provenance records.

In addition to our past and ongoing experiments regarding NFT-based dataset provenance mechanisms on Ethereum's testnets (such as Sepolia) and the Besu network, we are actively researching other protocols, including those specifically designed for managing datasets such as the Ocean Protocol and the T-REX Protocol (Recognized as ERC3643).

In summary, our endeavor to establish a Non-Fungible Token-based dataset provenance mechanism underscores our commitment to promoting transparency, accountability, and trustworthiness in data analytics. By harnessing the power of blockchain technology and innovative storage solutions, we aim to explore how datasets can be characterized, tracked, and utilized, thereby unlocking new possibilities for data-driven innovation and collaboration across diverse domains.

8 Conclusion and Future works

This deliverable describes the current, intermediate status of key components of the ExtremeXP framework, as well as a demonstration of their intended functionalities and capabilities.

The main achievements of this work are the following:

- Design of the ExtremeXP framework architecture and definition of domain description languages (DSLs) and tools for the preparation and execution of workflows.
- Design and initial development of graphical editor for experiment specification.
- Design and initial development of the continuous adaptive experiment planning module, aimed at streamlining experiment planning processes. The module interprets experiment specifications outlined in the DSL and creates concrete workflows for each experiment.
- Analysis and further development and tailoring of the framework's workflow executionware.
- Design and initial development of the blockchain-based attribute-based access control (ABAC) approach for the ExtremeXP framework.
- Design and initial development of the distributed metadata and knowledge management component.

These achievements have (up to M16) produced the following results in reference to the target WP5 KPIs:

- KPI-1.1: Meta-models and language specification for experiment-driven analytics covering 100% the ExtremeXP use cases' needs
 - We have a first version of the DSLs for modeling all the aspects pertinent to experimentation of CAWs (workflows, experimentation spaces)
 - We have used it in these use cases so far: UC2, UC4, and UC5.
- KPI-6.1: Develop a minimum of 3 context handlers for each use case
 - 3 context handlers are already defined for UC5 on the smart contract, experimenting with them.
- KPI-6.2: Publish access policy model including at least 3 detailed policies for each use case
 - 4 policies (only 1 described) for UC5
- KPI-4.1 Running at least 10 different experiments in each ExtremeXP use case using the ExtremeXP framework.
 - 2 experiments specified and run for UC5 so far.
- Related Dissemination KPIs
 - Open-source software: 5 released open-source software tools
 - Papers
 - "AuthAccess: Authenticated attribute-based Access control system for Data-driven decision-making systems" to be submitted (related to 5.4)

Following the delivery of the initial architecture, language models and core functionalities, WP5 technologies and tools will be checked against the existing use case requirements and considered for testing deployments in selected use case sites. The components will continue to evolve by considering the results of other technical WPs as well as the evolving use case requirements and constraints. The final experimentation framework and its constituent components will be delivered in M30 as part of D5.1.

9 References

- [1] L. Bulej, T. Bureš, P. Hnětynka, V. Čamra, P. Siegl, M. Töpfer: IVIS: Highly customizable framework for visualization and processing of IoT data, in Proceedings of EUROMICRO SEAA 2020, Portorož, Slovenia, 2020



10 Appendices

10.1 Appendix 1: API of the Data Abstraction Layer

10.1.1 Retrieval of executed workflows overview

Retrieves list of executed workflows, which can be specified by a period or an interval of allowed ids.

10.1.1.1 Request

GET <https://address/executedWorkflowsOverview>

10.1.1.2 Parameters

- Auth - Authentication token
- DateFrom – Lower bound for date (format YYYY-MM-DD HH:mm:ss)
- DateTo – Upper bound for date (format YYYY-MM-DD HH:mm:ss)
- IdFrom – Lower bound for ID
- IdTo – Upper bound for ID

10.1.2 Addition of an executed workflow to the overview

Adds executed workflow with specified data to the database.

10.1.2.1 Request

POST <https://address/executedWorkflowsOverview>

10.1.2.2 Payload

- StartTime – Timestamp of workflow execution
- EndTime – Timestamp of workflow completion
- Model – Serialized workflow model
- WorkflowId – ID of workflow model
- ChildrenTasks – List of IDs of children executed tasks
- ...

10.1.2.3 Parameters

- Auth - Authentication token

10.1.3 Modification of an executed workflow in the overview

Modifies information about an executed workflow in the database.

10.1.3.1 Request

PUT <https://address/executedWorkflowsOverview>

10.1.3.2 Payload

- ID – ID of the executed workflow
- StartTime – Timestamp of workflow execution

- EndTime – Timestamp of workflow completion
- Model – Serialized workflow model
- WorkflowId – ID of workflow model
- ChildrenTasks – List of IDs of children executed tasks
- ...

10.1.3.3 Parameters

- Auth - Authentication token

10.1.4 Retrieval of executed tasks overview

Retrieves list of executed tasks, which can be specified by a period, an interval of allowed ids or parent executed workflow id.

10.1.4.1 Request

GET [https://address /executedTasksOverview?](https://address/executedTasksOverview?)

10.1.4.2 Parameters

- Auth - Authentication token
- DateFrom – Lower bound for date (format YYYY-MM-DD HH:mm:ss)
- DateTo – Upper bound for date (format YYYY-MM-DD HH:mm:ss)
- IdFrom – Lower bound for ID
- IdTo – Upper bound for ID
- ExecutedWorkflowId – Identification of parent executed workflow

10.1.5 Addition of an executed task to the overview

Adds executed task with specified data to the database.

10.1.5.1 Request

POST <https://address/executedTasksOverview>

10.1.5.2 Payload

- StartTime – Timestamp of task execution
- EndTime – Timestamp of task completion
- ExecutedWorkflowId – ID of parent executed workflow
- ...

10.1.5.3 Parameters

- Auth - Authentication token

10.1.6 Modification of an executed task in the overview

Modifies information about an executed task in the database.

10.1.6.1 Request

PUT <https://address/executedTasksOverview>

10.1.6.2 Payload

- ID – ID of the executed task
- StartTime – Timestamp of task execution
- EndTime – Timestamp of task completion
- ExecutedWorkflowId – ID of parent executed workflow
- ...

10.1.6.3 Parameters

- Auth - Authentication token

10.1.7 Retrieval of input datasets overview

Retrieves list of input datasets, which can be specified by executed workflow id or executed task id.

10.1.7.1 Request

GET <https://address/inputDatasetsOverview>

10.1.7.2 Parameters

- Auth - Authentication token
- ExecutedWorkflowId – ID of an executed workflow using searched input dataset
- ExecutedTaskId – ID of an executed task using searched input dataset

10.1.8 Addition of an input dataset to the overview

Adds input dataset information to the database.

10.1.8.1 Request

POST <https://address/inputDatasetsOverview>

10.1.8.2 Payload

- Title – Title of the dataset
- Reference – URI reference to the data storage of the dataset
- Description – Description of the dataset
- Checksum – Checksum for ensuring validity of the dataset
- ExecutedWorkflowID – ID of an executed workflow using the dataset reference
- ExecutedTaskID – ID of an executed task using the dataset reference

10.1.8.3 Parameters

- Auth - Authentication token

10.1.9 Modification of an input dataset in the overview

Modifies input dataset information in the database.

10.1.9.1 Request

PUT <https://address/inputDatasetsOverview>

10.1.9.2 Payload

- ID – ID of the input dataset information
- Title – Title of the dataset
- Reference – URI reference to the data storage of the dataset
- Description – Description of the dataset
- Checksum – Checksum for ensuring validity of the dataset
- ExecutedWorkflowID – ID of an executed workflow using the dataset reference
- ExecutedTaskID – ID of an executed task using the dataset reference

10.1.9.3 Parameters

- Auth - Authentication token

10.1.10 Retrieval of output dataset overview

Retrieves list of output datasets, which can be specified by executed workflow id or executed task id.

10.1.10.1 Request

GET <https://address/outputDatasetsOverview>

10.1.10.2 Parameters

- Auth - Authentication token
- ExecutedWorkflowId – ID of an executed workflow which generated the dataset
- ExecutedTaskId – ID of an executed task which generated the dataset

10.1.11 Addition of an output dataset to the overview

Adds output dataset information to the database.

10.1.11.1 Request

POST <https://address/outputDatasetsOverview>

10.1.11.2 Payload

- Title – Title of the dataset
- Reference – URI reference to the data storage of the dataset
- Description – Description of the dataset
- Checksum – Checksum for ensuring validity of the dataset
- ExecutedWorkflowID – ID of an executed workflow, which generated the dataset
- ExecutedTaskID – ID of an executed task which generated the dataset

10.1.11.3 Parameters

- Auth - Authentication token

10.1.12 Modification of an output dataset in the overview

Modifies output dataset information in the database.

10.1.12.1 Request

PUT <https://address/outputDatasetsOverview>

10.1.12.2 Payload

- ID – ID of the output dataset information
- Title – Title of the dataset
- Reference – URI reference to the data storage of the dataset
- Description – Description of the dataset
- Checksum – Checksum for ensuring validity of the dataset
- ExecutedWorkflowID – ID of an executed workflow, which generated the dataset
- ExecutedTaskID – ID of an executed task which generated the dataset

10.1.12.3 Parameters

- Auth - Authentication token

10.1.13 Retrieval of metrics overview

Retrieves list of computed metrics, which can be specified by executed workflow id, executed task id or metric name.

10.1.13.1 Request

GET <https://address/metricsOverview>

10.1.13.2 Parameters

- Auth - Authentication token
- ExecutedWorkflowId – ID of an executed workflow which computed the metric
- ExecutedTaskId – ID of an executed task which computed the metric
- MetricName – Name of the metric

10.1.14 Addition of a metric to the overview

Adds metric information to the database.

10.1.14.1 Request

POST <https://address/metricsOverview>

10.1.14.2 Payload

- MetricName – Name of the metric

- Value – value or reference to the metric result
- ExecutedWorkflowID – ID of an executed workflow, which computed the metric
- ExecutedTaskID – ID of an executed task which computed the metric

10.1.14.3 Parameters

- Auth - Authentication token

10.1.15 Modification of a metric in the overview

Modifies metric information in the database.

10.1.15.1 Request

PUT <https://address/metricsOverview>

10.1.15.2 Payload

- ID – ID of the metric information
- MetricName – Name of the metric
- Value – value or reference to the metric result
- ExecutedWorkflowID – ID of an executed workflow, which computed the metric
- ExecutedTaskID – ID of an executed task which computed the metric

10.1.15.3 Parameters

- Auth - Authentication token

10.1.16 Retrieval of parameters overview

Retrieves list of used parameters, which can be specified by executed workflow id, executed task id or parameter name.

10.1.16.1 Request

GET <https://address/parametersOverview>

10.1.16.2 Parameters

- Auth - Authentication token
- ExecutedWorkflowId – ID of an executed workflow which generated the dataset
- ExecutedTaskId – ID of an executed task which generated the dataset
- ParameterName – Name of the parameter

10.1.17 Addition of a metric to the overview

Adds metric information to the database.

10.1.17.1 Request

POST <https://address/parametersOverview>

10.1.17.2 Payload

- ParameterName – Name of the parameter
- Value – Value or reference to data of the parameter
- ExecutedWorkflowID – ID of an executed workflow, which uses the parameter
- ExecutedTaskID – ID of an executed task which uses the parameter

10.1.17.3 Parameters

- Auth - Authentication token

10.1.18 Modification of a parameter in the overview

Modifies parameter information in the database.

10.1.18.1 Request

PUT <https://address/parametersOverview>

10.1.18.2 Payload

- ID – ID of the parameter information
- ParameterName - name of the parameter
- Value – value or reference to the data
- ExecutedWorkflowID – ID of an executed workflow, which used the parameter
- ExecutedTaskID – ID of an executed task, which used the parameter

10.1.18.3 Parameters

- Auth - Authentication token

10.2 Appendix 2: TextX Grammar

```

Root: workflows*=Workflow
      assembledWorkflows*=AssembledWorkflow
      espaces*=ESpace;

Workflow:
    'workflow' name=ID '{'
        elements*=Element
    '}'
;

// Note: StartAndEndEvent has to be come BEFORE StartEvent in the next line
Element: Node | DefineData | ConfigureTask | ConfigureData | StartAndEndEvent |
StartEvent | EndEvent | TaskLink | ConditionLink | DataLink | GroupTask | Comment;

Node:
    DefineTask | Operator
;

StartAndEndEvent:
    'START'
    '->'
    (nodes=[Node] '->')+
    'END'
    ';'
;

StartEvent:
    'START'
    ('->' nodes+=[Node])+
    ';'
;

EndEvent:
    (nodes=[Node] '->')+
    'END'
    ';'
;

TaskLink:
    initial_node=[Node]
    ('->' nodes+=[Node])+
    ';'
;

ConfigureTask:
    'configure task' alias=[DefineTask] '{'
        ('param' parameters=ID ('=' values=INT | STRING | FLOAT | BOOL | "null")*
        ',')*
        ('implementation' workflow=[Workflow]';' | 'implementation'
filename=STRING';')?
        ('dependency' dependency=STRING';')?
        (subtasks=ConfigureTask)*
    '}}';

DefineTask: 'define task' name=ID ';;';

ConfigureData:
    'configure data' alias=[DefineData] '{'
        'path' path=STRING';'
    '}}';

Operator: 'operator' name=ID ';;';

```

```

DefineData: 'define data' name=ID '[' data=' data*=[Value][',' ]';' | 'define
data' name=ID ';' ;

Value: STRING | INT | FLOAT | BOOL | "null";

DataLink:
    initial=[Node] ('-->' rest=[Data])* ';' |
    initial=[Data] ('-->' rest=[Node])* ';'
;

ConditionLink: node=[Node] '?->' node=[Node] '[condition=' STRING '];';

GroupTask: 'group' name=ID '{' node+=[Node]* '};' ;

AssembledWorkflow:
    'assembled workflow' name=ID 'from' parent_workflow=[Workflow] '{'
    (tasks+=ConfigureTask)*
    '};';

ESpace:
    'espace' name=ID 'of' assembled_workflow=[AssembledWorkflow] '{'
    configure=ConfigureBlock
    (tasks=ESpaceTaskConfiguration)*
    '};';

ConfigureBlock:
    'configure self {'
    (methods=Method | vps=VP)*
    '};';

Method: 'method' type=ID 'as' name=ID ('{' 'runs' '=' runs=INT '}')? '};';

VP:
    method=[Method] '.' name=ID '=' vp_values=VP_values
;

VP_values:
    'enum(' values+=INT [',' ] ')' ';' |
    'enum(' values+=STRING [',' ] ')' ';' |
    'range([' min=INT ',' max=INT '])' ';'
;

ParameterConfiguration: 'param' name=ID '=' method=[Method] '.' vp=ID '};';

ESpaceTaskConfiguration:
    'task' alias=[DefineTask] '{'
    (config=ParameterConfiguration)*
    '};';

Comment: /\ /\ /. *$/;

```

10.3 Appendix 3: Zenoh API endpoints

GET /list/<string:use_case>/<string:folder>

Description: Get list with all files from the specified use_case, folder

Parameters:

- use_case: The use_case where the file is located.(1,...,5)
- folder: The folder where the file is located
(input_data, metrics, experiments,output_data)

Response:

200 OK: File downloaded successfully.

404 : No use_case selected.

GET /file/<string:use_case>/<string:folder>/<subfolder>/<string:filename>

Description : Download a file from the specified use_case, folder, subfolder

Parameters:

- use_case: The use_case where the file is located.
(1,...,5)
- folder: The folder where the file is located.
(input_data, metrics, experiments,output_data)
- subfolder: The sub-folder where the file is located.
(electrical_anomaly, mechanical_anomaly, no_anomaly, for use_case 5)
- filename: The name of the file to download.

Response:

200 OK: File downloaded successfully.

400 : 'No file uploaded',

'No use_case selected'

'No folder selected'

'No subfolder selected'

POST file/<string:use_case>/<string:folder>/<subfolder>/<string:filename>

Description: Upload a file to the specified use_case, folder, subfolder

Parameters:

- use_case: The use_case where the file is located.
(1,...,5)
- folder: The folder where the file is located.
(input_data, metrics, experiments,output_data)
- subfolder: The sub-folder where the file is located.
(electrical_anomaly, mechanical_anomaly, no_anomaly, for use_case 5)
- filename: The name of the file to upload.

Response:

200 OK: File uploaded successfully.

400 : 'No file uploaded',

'No use_case selected'

'No folder selected'

'No subfolder selected'

DELETE /file/<string:use_case>/<string:folder>/<subfolder>/<string:filename>

Description: Download file with filename from the specified use_case, folder, subfolder

Parameters:

- use_case: The use_case where the file is located.

(1,...,5)

- folder: The folder where the file is located.

(input_data, metrics, experiments,output_data)

- subfolder: The sub-folder where the file is located.

(electrical_anomaly, mechanical_anomaly, no_anomaly, for use_case 5)

- filename: The name of the file to upload.

Response:

200 OK: File deleted successfully.

400: 'No file uploaded',

'No use_case selected'

'No folder selected'

'No subfolder selected'.

GET /files/<string:use_case>/<string:folder>/<subfolder>

Description: Download all files from the specified use_case, folder, subfolder

Parameters:

- use_case: The use_case where the file is located.

(1,...,5)

- folder: The folder where the file is located.

(input_data, metrics, experiments,output_data)

- subfolder: The sub-folder where the file is located.

(electrical_anomaly, mechanical_anomaly, no_anomaly, for use_case 5)

Response:

200 OK: File downloaded successfully.

400: 'No use_case selected'

'No folder selected'

'No subfolder selected'.

POST files/<string:use_case>/<string:folder>/<subfolder>

Description: Upload files to the specified use_case, folder, subfolder

Parameters:

- use_case: The use_case where the file is located.

(1,...,5)

- folder: The folder where the file is located.

(input_data, metrics, experiments,output_data)

- subfolder: The sub-folder where the file is located.

(electrical_anomaly, mechanical_anomaly, no_anomaly, for use_case 5)

Response:

200 OK: File uploaded successfully.

400: 'No use_case selected'

'No folder selected'

'No subfolder selected'.

DELETE /files/<string:use_case>/<string:folder>/<subfolder>

Description: Delete all files under the specified use_case, folder, subfolder

Parameters:

- use_case: The use_case where the file is located.

(1,...,5)

- folder: The folder where the file is located.

(input_data, metrics, experiments,output_data)



- subfolder: The sub-folder where the file is located.
(electrical_anomaly, mechanical_anomaly, no_anomaly, for use_case 5)

Response:

200 OK: File downloaded successfully.

400: 'No use_case selected'

'No folder selected'

GET /files/<string:use_case>/<string:folder>

Description: Download all files under the specified use_case folder

Parameters:

- use_case: The use_case where the file is located.
(1,...,5)

- folder: The folder where the file is located.
(input_data, metrics, experiments,)

Response:

200 OK: Files downloaded successfully.

400: 'No use_case selected'

'No folder selected'

POST files/<string:use_case>/<string:folder>/

Description: upload file to the specified use_case, folder

Parameters:

- use_case: The use_case where the file is located.
(1,...,5)

- folder: The folder where the file is located.
(input_data, metrics, experiments,)

Response:

200 OK: Files uploaded successfully.

400: 'No use_case selected'

'No folder selected'

DELETE /files/<string:use_case>/<string:folder>

Description: Delete all files under the specified use_case, folder

Parameters:

- use_case: The use_case where the file is located.
(1,...,5)

- folder: The folder where the file is located.
(input_data, metrics, experiments,)

Response:

200 OK: Files deleted successfully.

400: 'No use_case selected'

'No folder selected'