



# SmartEdge

## Semantic Low-code Programming Tools for Edge Intelligence

*This project is supported by the European Union's Horizon RIA research and innovation programme under grant agreement No. 101092908*

Deliverable D3.2

### First Implementation of Tools for Continuous Semantic Integration

**Editor** Mario Scrocca (CEF)

**Contributors** D. Anicic (SAG), B. Anuraj (HESSO), M. Bagheri (CONV), L. Bassbouss (FhG), D. Bowden (DELL), J. Calbimonte (HESSO), A. Carenini (CEF), K. Dorofeev (SAG), G. Gizzi (CEF), M. Grassi (CEF), K. Köhle (SAG), A. Thuluva (SAG), R. Wenning (ERCIM)

**Version** 1.0

**Date** December 08, 2024

**Distribution** PUBLIC (PU)

## DISCLAIMER

This document contains information which is proprietary to the SmartEdge (Semantic Low-code Programming Tools for Edge Intelligence) consortium members that is subject to the rights and obligations and to the terms and conditions applicable to the Grant Agreement number 101092908. The action of the SmartEdge consortium members is funded by the European Commission.

Neither this document nor the information contained herein shall be used, copied, duplicated, reproduced, modified, or communicated by any means to any third party, in whole or in parts, except with prior written consent of the SmartEdge consortium members. In such case, an acknowledgement of the authors of the document and all applicable portions of the copyright notice must be clearly referenced. In the event of infringement, the consortium members reserve the right to take any legal action it deems appropriate.

This document reflects only the authors' view and does not necessarily reflect the view of the European Commission. Neither the SmartEdge consortium members as a whole, nor a certain SmartEdge consortium member warrant that the information contained in this document is suitable for use, nor that the use of the information is accurate or free from risk, and accepts no liability for loss or damage suffered by any person using this information.

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

## REVISION HISTORY

<b>Revision</b>	<b>Date</b>	<b>Responsible</b>	<b>Comment</b>
0.1	15.04.2024	CEF	ToC
0.2	10.07.2024	CEF	Revised ToC with feedback from WP3 partners and content from MS3.1
0.3	15.07.2024	SAG	Revised the Concept of Continuous Semantic Integration and Integration Tools
0.4	13.09.2024	CEF	Add final design for DataOps Toolbox as artefacts 3.5, 3.6, 3.7.
0.5	20.09.2024	SAG	Finalization of Standardized Semantic Interfaces for SmartEdge
0.6	05.10.2024	SAG	First Implementation of Standardized Semantic Interfaces for SmartEdge
0.7	16.10.2024	CEF, SAG, HES-SO	Introduction to WP3 artefacts and how they enable Continuous Semantic Integration
0.8	30.10.2024	CEF	Revise final design and add first implementation for DataOps Toolbox
0.9	10.11.2024	HES-SO	Revised artefacts 3.8, 3.9, 3.10
0.10	18.11.2024	CEF, SAG, HES-SO	Document ready for internal review
0.11	25.11.2024	CEF, SAG, HES-SO	Version integrating comments from internal reviewers
0.12	27.11.2024	CEF, SAG, HES-SO	Document ready for quality review
1.0	6.12.2024	CEF, ERCIM	Version integrating comments from quality check and ready for submission

## LIST OF AUTHORS

<b>Partner</b>	<b>Name Surname</b>	<b>Contributions</b>
CEF	Mario Scrocca	Editor, Section 1, 3, 5
CEF	Marco Grassi	Section 3
CEF	Alessio Carenini	Section 3.1
CEF	Gianluca Gizzi	Section 3.1.2, 3.2.2, 3.3
SAG	Darko Anicic	Section 1, 2

<b>SAG</b>	<i>Kirill Dorofeev</i>	<i>Section 1.5, 2.1.4, 2.1.6, 2.2.3, 2.2.5</i>
<b>SAG</b>	<i>Aparna Saisree Thuluva</i>	<i>Section 2.1.1, 2.1.2, 2.2.1, 2.2.2</i>
<b>SAG</b>	<i>Kay Koehle</i>	<i>Section 2.1.4, 2.2.4</i>
<b>HESSO</b>	<i>Jean-Paul Calbimonte</i>	<i>Section 1, 4</i>
<b>HESSO</b>	<i>Banani Anuraj</i>	<i>Section 4</i>
<b>DELL</b>	<i>Dai Bowden</i>	<i>Section 1, 2.1.6, 2.2.5</i>
<b>FHG</b>	<i>Louay Bassbouss</i>	<i>Section 2.1.4, 2.2.3</i>
<b>CONV</b>	<i>Mehrdad Bagheri</i>	<i>Section 3.3.1</i>
<b>ERCIM</b>	<i>Rigo Wenning</i>	<i>Quality Check</i>

## ABBREVIATIONS

<b>Acronym</b>	<b>Description</b>
<b>ADAS</b>	<i>Advanced Driving Assistance System</i>
<b>AI</b>	<i>Artificial Intelligence</i>
<b>API</b>	<i>Application Programming Interface</i>
<b>BLE</b>	<i>Bluetooth</i>
<b>CAD</b>	<i>Computer-Aided Design</i>
<b>CPU</b>	<i>Central Processing Unit</i>
<b>CQLES</b>	<i>Continuous Query Evaluation over Linked Streams</i>
<b>CQLES-QL</b>	<i>CQLES-Query Language</i>
<b>CSI</b>	<i>Continuous Semantic Integration</i>
<b>DB</b>	<i>Database</i>
<b>DDS</b>	<i>Data Distribution Service</i>
<b>ETL</b>	<i>Extract-Transform-Load</i>
<b>HW</b>	<i>Hardware</i>
<b>ID</b>	<i>Identity</i>
<b>IDE</b>	<i>Integrated Development Environment</i>
<b>IoT</b>	<i>Internet of Things</i>
<b>IT</b>	<i>Information Technology</i>
<b>JSON</b>	<i>JavaScript Object Notation</i>
<b>JSON-LD</b>	<i>JSON-Linked Data</i>
<b>KB</b>	<i>Knowledge Base</i>
<b>JVM</b>	<i>Java Virtual Machine</i>
<b>KG</b>	<i>Knowledge Graph</i>
<b>KPI</b>	<i>Key Performance Indicator</i>
<b>LiDAR</b>	<i>Light Detection and Ranging</i>
<b>ML</b>	<i>Machine Learning</i>
<b>MQTT</b>	<i>Message Queuing Telemetry Transport</i>
<b>MX</b>	<i>Mendix</i>

<b>OCI</b>	<i>Open Container Initiative</i>
<b>OPC</b>	<i>Open Platform Communications</i>
<b>OPC-UA</b>	<i>OPC Unified Architecture</i>
<b>OS</b>	<i>Operating System</i>
<b>OT</b>	<i>Operations Technology</i>
<b>PACK-ML</b>	<i>Packaging ML</i>
<b>PLC</b>	<i>Programmable Logic Controller</i>
<b>R2RML</b>	<i>RDB to RDF Mapping Language</i>
<b>RDF</b>	<i>Resource Description Language</i>
<b>RML</b>	<i>RDF Mapping Language</i>
<b>ROS</b>	<i>Robot Operating System</i>
<b>SAREF</b>	<i>Smart Applications Reference</i>
<b>SHACL</b>	<i>Shapes Constraint Language</i>
<b>SotA</b>	<i>State of the Art</i>
<b>SotP</b>	<i>State of the Practice</i>
<b>SPARQL</b>	<i>SPARQL Protocol and RDF Query Language</i>
<b>SQL</b>	<i>Structured Query Language</i>
<b>SOSA</b>	<i>Sensor, Observation, Sample, and Actuator ontology</i>
<b>SW</b>	<i>Software</i>
<b>TDD</b>	<i>Thing Description Directory</i>
<b>TRL</b>	<i>Technical Readiness Level</i>
<b>UC</b>	<i>Use case</i>
<b>URDF</b>	<i>Unified Robot Description Format</i>
<b>WoT</b>	<i>Web of Things</i>
<b>WoT TD</b>	<i>WoT Thing Description</i>

## EXECUTIVE SUMMARY

Deliverable D3.2 details the final design of the tools for Continuous Semantic Integration (CSI) and their first implementation defined by the SmartEdge project within WP3. Starting from the design activities reported in D3.1 and the final list of requirements defined in D2.2, this deliverable identifies and describes a set of artefacts to enable the overall concept of CSI.

Task 3.1 defines interoperable semantic models to describe the nodes available and their capabilities as the first step in enabling a specific use case when adopting the SmartEdge solutions. Furthermore, a shared repository is implemented for storing, retrieving, and querying such descriptions. To enable collaboration among heterogeneous nodes that may join a swarm, a solution is proposed to allow for standardized communication interfaces among them.

However, interoperability of interfaces is not enough since nodes may rely on different information models and data formats that should be harmonized. Task 3.2 addresses these aspects by providing a DataOps toolbox to define mediated data exchanges for semantic interoperability. The flexible execution of data exchanges among nodes is also supported, considering heterogeneous deployment requirements and the potential interplay between nodes on Cloud and Edge. Finally, the performance and scalability of mediated data exchanges are addressed.

Task 3.3., building on the described solutions, provides a dedicated user interface to enable the low-code definition of swarm applications for edge intelligence. Additionally, a set of artefacts is responsible for identifying and orchestrating relevant nodes in order to compose a swarm able to execute the defined applications.

The deliverable describes the first release of the artefacts designed and implemented by WP3 to support the first lab tests and validation phase within WP6, considering the SmartEdge use cases. Deliverable D6.1 will fully report and discuss the validation planning, the results obtained in the first iteration and the fulfilment status for requirements associated with WP3 artefacts.

This document will be updated in month 30 in deliverable D3.3 based on the final implementation of the tools for CSI that will consider the feedback from WP6 activities.

## TABLE OF CONTENTS

1	Introduction.....	1
1.1	Concept of Continuous Semantic Integration.....	1
1.2	Continuous Semantic Integration in SmartEdge .....	3
1.3	Continuous Semantic Integration Artefacts.....	4
1.4	Mapping KPIs and Artefacts .....	6
1.5	Relations to WP4 and WP5 .....	7
1.6	Structure of the Document.....	8
2	Standardized Semantic Interfaces for SmartEdge .....	9
2.1	Final Design.....	9
2.1.1	Final Design of SmartEdge Schema (A3.1) .....	9
2.1.2	Final Design of Recipe model (A3.1) .....	10
2.1.3	Final Design of Middleware with Standardized Semantic Interfaces (A3.2) .....	12
2.1.4	Final Design of Knowledge Graph Repository (A3.3) .....	14
2.1.5	Final Design of Mendix Toolchain (A3.4).....	15
2.1.6	Final Design of Semantic Media Service (A3.11).....	16
2.2	First Implementation .....	17
2.2.1	First Implementation of SmartEdge Schema (A3.1).....	17
2.2.2	First Implementation of Recipe Model (A3.1).....	18
2.2.3	First Implementation of Middleware with Standardized Semantic Interfaces (A3.2) .....	25
2.2.4	First Implementation of Knowledge Graph Repository (A3.3).....	28
2.2.5	First Implementation of Mendix Toolchain (A3.4) .....	29
2.2.6	First Implementation of Semantic Media Service (A3.11) .....	30
3	DataOps Tool for Semantic Management of Things and Embedded AI Apps.....	31
3.1	Final Design.....	32
3.1.1	Final Design of the DataOps Pipeline Components (A3.5) .....	34
3.1.2	Final Design of the DataOps Deployment Templates (A3.6) .....	39
3.1.3	Final Design of Low-code DataOps Configuration (A3.7) .....	43
3.2	First Implementation .....	45
3.2.1	First Implementation of the DataOps Pipeline Components (A3.5).....	45
3.2.2	First Implementation of the DataOps Deployment Templates (A3.6) .....	54
3.2.3	First Implementation of Low-code DataOps Configuration (A3.7).....	58
3.3	DataOps Toolbox Pipelines for SmartEdge.....	62
3.3.1	DataOps Pipeline for harmonised traffic data (UC2).....	62

3.3.2	DataOps Pipeline for OPC-UA support in A3.3 (UC4).....	66
4	Creation and Orchestration of Swarm Intelligence Apps.....	69
4.1	Final Design.....	69
4.1.1	Final Design of Semantic Recipe Integration with Mendix (A3.8) .....	69
4.1.2	Final Design of Recipe-TD Matcher (A3.9) .....	72
4.1.3	Final Design of Mendix Recipe Orchestrator (A3.10) .....	74
4.2	First Implementation .....	75
4.2.1	First Implementation of Semantic Recipe Integration with Mendix (A3.8). 75	
4.2.2	First Implementation of Recipe-TD Matcher (A3.9).....	77
4.2.3	First Implementation of Mendix Recipe Orchestrator (A3.10) .....	78
5	Conclusions.....	84
6	References.....	85
7	ANNEX I – Sample Recipe for UC4 .....	86
8	ANNEX II – Performance and Scalability Evaluation of the Mapping Template Component.....	92
8.1	GTFS Madrid Benchmark .....	92
8.2	Knowledge Graph Construction Challenge .....	94
9	ANNEX III – Evaluation of UC2 DataOps Pipeline on Different Deployment Templates 97	
9.1	Conversion Time and Input Size.....	97
9.2	Memory and CPU Utilisation.....	98
9.2.1	Temurin – GraalVM – GraalVM-Native .....	98
9.2.2	Spring-Temurin – Spring-GraalVM – Spring-GraalVM-Native .....	99
9.2.3	Temurin – Spring-Temurin .....	100
9.2.4	GraalVM – Spring-GraalVM.....	101
9.2.5	Native – Spring-Native .....	102
9	ANNEX III – Sample Recipe for UC4	



## LIST of FIGURES

Figure 1-1: Continuous Semantic Integration for SmartEdge .....	2
Figure 1-2: Continuous Semantic Integration enabled by SmartEdge WP3 artefacts .....	3
Figure 2-1: Semantic models in SmartEdge .....	10
Figure 2-2: Recipe model extended with NLQ.....	11
Figure 2-3: LLM-driven approach to address user's requirements in Recipe development .....	12
Figure 2-4: Standardized Semantic Interfaces in SmartEdge. ....	13
Figure 2-5: Repository for Thing Descriptions and OPC UA Nodesets .....	15
Figure 2-6: Mendix Toolchain .....	15
Figure 2-7: Manufacturing illustration of streaming semantic media service .....	16
Figure 2-8: Factory schematic and corresponding 2D occupancy map .....	17
Figure 2-9: Overview of SmartEdge Schema .....	18
Figure 2-10: Graphical representation of sample Recipe for smart factory application in UC4 .....	19
Figure 2-11 Virtual Scene with Virtual Car .....	27
Figure 2-12: Production Module with its Virtual Counterpart and OPC UA Information Model from UC4 .....	28
Figure 2-13: Thing Description Upload and Retrieval via TDD API.....	29
Figure 2-14: Mendix toolbox for WoT client .....	29
Figure 3-1: DataOps Toolbox related artefacts and their relation .....	32
Figure 3-2: Deployment options for the DataOps Toolbox .....	33
Figure 3-3: Declarative semantic conversion process for interoperability .....	34
Figure 3-4: DataOps Pipeline .....	34
Figure 3-5: Final workflow for generic knowledge conversion enabled by a DataOps pipeline .....	36
Figure 3-6: Example of a Camel route written using the Java domain specific language .....	38
Figure 3-7: Overview of a DataOps pipeline integrating different components.....	39
Figure 3-8: DataOps Deployment Templates .....	42
Figure 3-9: Example source Kamelet that is used to read the status of the swarm and then forward it to a target node .....	44
Figure 3-10: MTL mapping to convert XML data to RDF Turtle .....	47
Figure 3-11: XML input data and corresponding RDF Turtle representation obtained by applying the mapping template .....	47
Figure 3-12: On top, an example of defining multiple readers statically within a mapping. On the bottom, the new possibility of providing multiple readers dynamically from outside the mapping.....	48
Figure 3-13: MTL to RML transformation process .....	49
Figure 3-14: An example RML mapping (above) and the corresponding automatically generated MTL mapping (below). .....	49
Figure 3-15: Example of a ChimeraResourceBean defined using XML .....	51
Figure 3-16: An example SPARQL SELECT and ASK query .....	52
Figure 3-17: Example Chimera route that performs a SPARQL select query and returns the result as JSON .....	52

Figure 3-18: DataOps pipeline defined to demonstrate the deployment templates. ...	56
Figure 3-19: Karavan component palette showing the Chimera graph, mapping-template and rml components .....	59
Figure 3-20: Low-code DataOps Configuration Karavan plugin interface.....	60
Figure 3-21: Example of a ChimeraResourceBean that holds the lifting MTL mapping file defined through Karavan.....	60
Figure 3-22: Example YAML Camel using Chimera components produced by the Visual Studio Code Karavan plugin.....	61
Figure 3-23: Example DataOps pipeline for the semantic conversion of radar data .....	63
Figure 3-24: Example logs monitoring the execution of the DataOPs pipeline .....	64
Figure 3-25: DataOps pipelines enabling support for OPC UA Nodesets in the A3.3 artefact .....	68
Figure 4-1: Mendix Studio Pro: design time App environment.....	70
Figure 4-2: Semantic Recipe Integration. ....	71
Figure 4-3: W3C Thing Description core vocabulary (source: <a href="https://www.w3.org/TR/wot-thing-description/">https://www.w3.org/TR/wot-thing-description/</a> ) .....	72
Figure 4-4: Matchmaking between the capabilities required in the Recipe and the nodes available in the Swarm at design time .....	73
Figure 4-5: Visual representation of the main concepts of the SmartEdge semantic Recipe model, from Artefact 3.1 .....	73
Figure 4-6: Orchestration of Mendix applications in A3.10, based on SmartEdge Recipes. ....	75
Figure 4-7: Snippet of a semantic Recipe for a test Lamp application.....	76
Figure 4-8: JSON-LD snippet of n interaction detail in a sample Recipe. ....	76
Figure 4-9: Mendix flow of a sample application .....	76
Figure 4-10: Example of a Recipe snippet for a healthcare physiotherapy Recipe in Turtle format.....	78
Figure 4-11: Snippet of a TD description of a device including details about its capabilities, in Turtle format. ....	78
Figure 4-12: Mendix flow example, connecting to a TD counter.....	79
Figure 4-13: Reading a property from a TD description from JS code in a Mendix flow. ....	80
Figure 4-14: Writing back to a TD exposed property using JS code inside Mendix. ....	80
Figure 4-15: Accessing TD-retrieved properties from a Mendix-created application page. ....	81
Figure 4-16: Accessing the Node-WoT server, counter example exposed through a TD. ....	82
Figure 4-17: Mendix App frontend. TD accessed through the application during runtime .....	82
Figure 4-18: Runtime invoke action: TD property read through the Mendix runtime ..	83
Figure 7-1: Sample Recipe snippet for smart factory application in UC4 .....	91
Figure 8-1: Evaluation on the GTFS Madrid Benchmark between mapping-template and morph-kgc.....	93
Figure 8-2: Evaluation on the KGCW Challenge for GTFS-scale and GTFS-heterogeneity .....	95
Figure 8-3: Comparison on GTFS-Scale 1 to evaluate the overhead of RML compilation in the mapping-template.....	95

Figure 8-4: Evaluation on the KGCW Challenge for mappings, records, join parameters .....	96
Figure 8-5: Evaluation on the KGCW Challenge for empty values, duplicates and properties parameters.....	96
Figure 9-1: Comparison of conversion time and input size over time for Temurin and GraalVM.....	97
Figure 9-2: Comparison of conversion time and input size over time for Native .....	98
Figure 9-3: CPU Comparison of Temurin, GraalVM and Native .....	99
Figure 9-4: Memory Comparison of Temurin, GraalVM and Native .....	99
Figure 9-5: CPU Comparison of Spring Temurin, Spring GraalVM and Spring Native images.....	100
Figure 9-6: Memory Comparison of Spring Temurin, Spring GraalVM and Spring Native images.....	100
Figure 9-7: CPU Comparison of Temurin and Spring Temurin .....	101
Figure 9-8: Memory Comparison of Temurin and Spring Temurin .....	101
Figure 9-9: CPU Comparison GraalVM and Spring GraalVM .....	102
Figure 9-10: Memory Comparison GraalVM and Spring GraalVM.....	102
Figure 9-11: CPU Comparison Native and Spring Native.....	103
Figure 9-12: Memory comparison Native and Spring Native .....	103

## LIST OF TABLES

Table 1-1: Tools for Continuous Semantic Integration. ....	5
Table 1-2: WP3 Key Performance Indicators for CSI tools and related artefacts. ....	6
Table 3-1: Analysis of PROs and CONs for different deployment alternatives .....	41
Table 3-2: Comparison of deployment templates for the same DataOps pipeline .....	57
Table 3-3: Average/Max/Min metrics for conversion time and input size for each pipeline deployment tested. ....	65

## 1 INTRODUCTION

---

Deliverable 3.2 provides the final design and first implementation of tools for Continuous Semantic Integration (CSI) in the SmartEdge project. In this deliverable we report the status of the work in Work Package 3 (WP3), which aims to provide CSI via three tasks: (i) edge semantics with standardized semantic interfaces for IoT devices; (ii) a DataOps toolbox for continuous semantic integration, and (iii) a declarative and low-code approach for creation and orchestration of swarm apps based on Recipes. To this goal, we design and implement concepts for these three tasks considering the requirements from SmartEdge use cases. The final design is based on its first iteration described in deliverable D3.1 and the final list of requirements reported in D2.2. This deliverable also discusses the first implementation of tools for Continuous Semantic Integration to support the first lab tests and validation phase within WP6. D6.1 provides a validation and report on how the developments described in this deliverable match the SmartEdge requirements defined in D2.2 and the complete set of SmartEdge KPIs.

The following introduction explains the concept of Continuous Semantic Integration (CSI) defined and implemented by WP3 in SmartEdge. We present the different tasks that are required to enable CSI and the associated artefacts that SmartEdge implements to support it. Moreover, we discuss the mapping between KPIs and artefacts and the relations between WP3 and other technical work packages. Finally, we outline the structure of the deliverable.

### 1.1 CONCEPT OF CONTINUOUS SEMANTIC INTEGRATION

The Internet of Things (IoT) together with edge intelligence brings several benefits across various industries and everyday life. These technologies enable the seamless flow of data between devices and systems, leading to improved efficiency and productivity. They can lead to cost savings by optimizing operations. IoT devices generate a vast amount of data. This data can be analysed to gain valuable insights and lead to better decision-making systems. But all these promises come with a hypothesis that the data generated with IoT devices can be easily consumed by intelligent applications. This is not always true, and very often it is a challenge. The reason is that IoT devices have different capabilities, communicate via different protocols, exchange information in different formats, and may change over time. For all these reasons, it is not an easy task to integrate data generated by IoT devices and make them consumable for application developers. Figure 1.1 introduces the concept of Continuous Semantic Integration in the SmartEdge project. It is a building block between IoT devices and added-value apps. Continuous Semantic Integration (CSI) provides access to horizontally and vertically integrated data via standardized communication interfaces. It also provides semantics about data, devices, and applications, and runs on the edge. For example, capabilities of devices are described in a machine-interpretable way with standardized vocabularies. Devices' data is also semantically described and accessible via unified and standardized interfaces. CSI is a prerequisite for the low-code application development in a way that

it facilitates semantic discovery of device skills and provides matchmaking of skills with application requirements. It also integrates data with different formats and semantics, and provides a uniform and standardized access to it. With CSI, the SmartEdge project aims to enable an easy development of low-code applications.

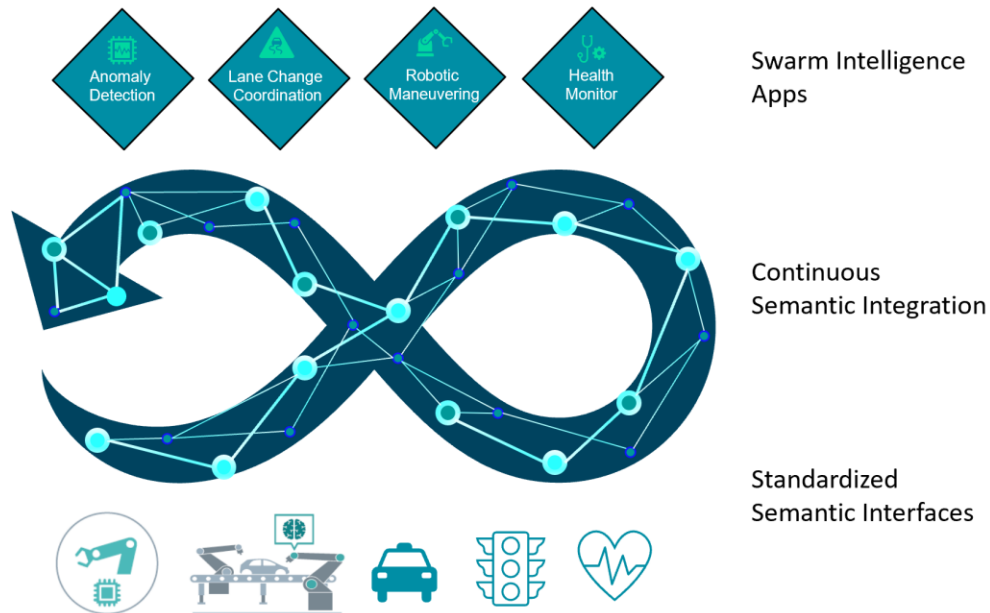


Figure 1-1: Continuous Semantic Integration for SmartEdge

The concept of CSI introduces several innovative contributions to the SmartEdge project. Semantic models in CSI formalize key concepts like Swarm Node, Device, Capability, and Recipe, among others. Through CSI, Recipes are introduced as a practical means to specify, develop, and deploy low-code swarm applications. Built on standardized models and interfaces, CSI promotes interoperability, automates low-code toolchains, and enhances the reusability of Recipe-based applications. CSI also incorporates the DataOps toolbox to enable a declarative specification of pipelines to guarantee semantic interoperability between swarm or device nodes. It supports pipeline deployment across diverse environments and provides a low-code tool for pipeline definition. Additionally, CSI integrates a Knowledge Graph Repository, a specialized semantic repository designed for storing, retrieving, and managing knowledge artefacts. This repository interfaces with two key standards: W3C Web of Things<sup>1</sup> and OPC UA<sup>2</sup>, and stores data in RDF format. This format allows for the integration of multiple semantic vocabularies, supporting a range of SmartEdge use cases. Crucially, it enables an effective application of large language models (LLMs) on this data, simplifying the use of semantic models in low-code application development and enhancing the usability of the SmartEdge low-code toolchain. By addressing complexity and usability, common barriers to the adoption of semantic technologies, CSI tackles these challenges effectively.

The following sections of this document break down the functionalities of CSI into a set of SmartEdge artefacts, each addressing a specific function within CSI. We also cover the design and initial implementation of these artefacts.

<sup>11</sup> <https://www.w3.org/WoT/>

<sup>2</sup> <https://opcfoundation.org/about/opc-technologies/opc-ua/>

1.2 CONTINUOUS SEMANTIC INTEGRATION IN SMARTEDGE

This section provides an overview of the designed integration of WP3 artefacts for enabling Continuous Semantic Integration for a set of nodes that are used to compose a swarm and execute a swarm intelligence application. The diagram in Figure 1-2 summarises the relation among the different artefacts and we discuss their integration considering the different tasks required to adopt the artefacts for a certain use case. Each artefact is described in detail within a dedicated section in the rest of the deliverable.

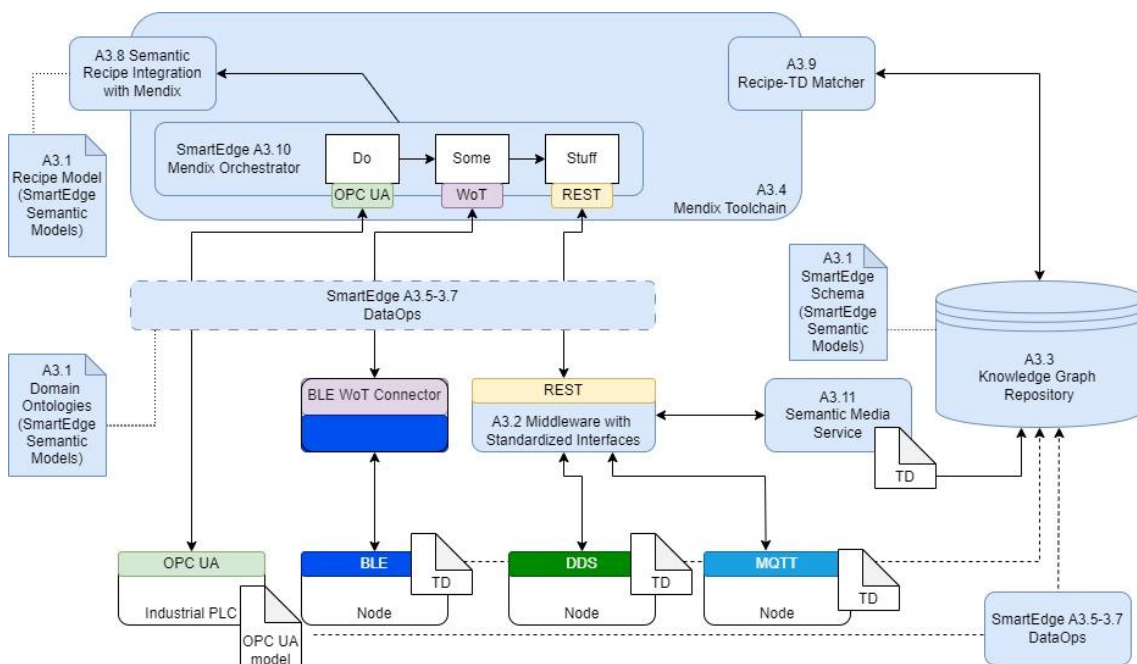


Figure 1-2: Continuous Semantic Integration enabled by SmartEdge WP3 artefacts

**Interoperable description of nodes and their capabilities**

The first task is based on an interoperable description of the nodes (devices) available for a certain use case, i.e., the list of nodes that can be selected to form a swarm and execute a swarm intelligence app. The description of the nodes should be compliant with the SmartEdge Semantic Models (A3.1) and stored within the Knowledge Graph Repository (A3.3). The Knowledge Graph Repository provides support for both W3C Thing Descriptions and OPC UA Nodesets.

**Enable standardized communication interfaces for relevant nodes**

The second task is associated with the need for enabling communications among nodes leveraging different protocols. Standard communication interfaces can be enabled with the support of the artefact Middleware with Standardized Interfaces (A3.2). Moreover, the Semantic Media Service (A3.11) is defined to support the efficient exchanges of data among nodes that do not require the intermediation of the orchestrator.

***Define mediated data exchanges ensuring semantic interoperability***

The DataOps tool provides a set of modular and configurable components (A3.5) that can be used to define a pipeline for mediated data exchanges among nodes in the swarm. Such a pipeline supports a semantic conversion process and guarantees not only the exchange of data among nodes, but also the required schema and data transformations for semantic interoperability, e.g., considering the semantics a specific domain ontology (A3.1). These pipelines may also support the static management of nodes, e.g., the conversion of OPC-UA nodes description for their insertion/retrieval from the Knowledge Graph Repository (A3.3). Moreover, they can support the execution of swarm intelligence applications by supporting the communication between the orchestrator and specific nodes in the swarm.

The definition of pipelines can be simplified by low-code approaches via artefact Low-code DataOps Configuration (A3.7).

***Support execution of mediated data exchanges between nodes on Cloud and Edge***

The execution of Data Ops pipeline should be flexible w.r.t the deployment environment of the different nodes. The artefact DataOps Deployment Templates (A3.6) supports the execution of pipelines in different deployment environments to enable the communication and data management across different types of nodes.

***Define interoperable Recipes for swarm intelligence apps***

In SmartEdge, a Recipe serves as an application template. It formally outlines the application requirements based on the Recipe Model (A3.1). Additionally, it specifies the steps that nodes (devices) must follow to implement a swarm application. These steps, along with the application logic, can be defined using a low-code approach (A3.4). As an application template, a Recipe can be reused to create multiple applications. Therefore, CSI facilitates the discovery and retrieval of existing Recipes within the low-code environment (A3.8). Once a suitable Recipe is identified, then the low code developer can customize it and finish the design of the application.

***Orchestrate interoperable Recipes on a swarm***

Semantic Recipes in SmartEdge, described using the models from A3.1, can be used as the basis for low code developers to build their applications, using artefact 3.8. In order to match the capabilities required by the semantic Recipes, Artefact A3.9 provides the ability to match them with the affordances of available nodes in the swarm. Thanks to the matching features of A3.9, then specific nodes can be bound to the runtime environment, to be later enacted and orchestrated through artefact A3.10.

### 1.3 CONTINUOUS SEMANTIC INTEGRATION ARTEFACTS

Table 1.1 describes the complete list of artefacts representing Continuous Semantic Integration tools developed by SmartEdge. The table summarizes the role of the artefact, where it is described within the document and the current implementation status.

Table 1-1: Tools for Continuous Semantic Integration.

ID	Component	Lead	Section	Description	Implementation Status
A3.1	SmartEdge Semantic Models	SAG	2.1.1, 2.1.2, 2.2.1, 2.2.2	Common data structures to represent the data from diverse machines, applications, and swarms.	Available for the first release. Second release planned.
A3.2	Middleware with Standardized Interfaces	SAG	2.1.3, 2.2.3	Middleware provides standardized data access to the devices' data.	Available for the first release. Second release planned.
A3.3	Knowledge Graph Repository	SAG	2.1.4, 2.2.4, 3.3.2	Specialized semantic repository designed for the storage, retrieval, and management of standardized knowledge artefacts	Available for the first release. Second release planned.
A3.4	Mendix Toolchain	SAG	2.1.5, 2.2.5	A programming environment to create Recipes in a low-code manner.	Available for the first release. Second release planned.
A3.5	DataOps Pipeline Components	CEF	3.1.1, 3.2.1	Set of components to define DataOps pipelines and reusable pipelines defined for SmartEdge use cases.	Available for the first release. Second release planned.
A3.6	DataOps Deployment Templates	CEF	3.1.2, 3.2.2	Templates for DataOps pipeline deployment on Cloud and Edge environments.	Available for the first release. Second release planned.
A3.7	Low-code DataOps Configuration	CEF	3.1.3, 3.2.3	Artefacts and tools to support the low-code definition of DataOps pipelines.	Available for the first release. Second release planned.
A3.8	Semantic Recipe Integration with Mendix	HESSO	4.1.1, 4.2.1	Integration of semantic Recipe directory with Mendix for discovery and retrieval of existing Recipes, related to specific swarm tasks.	Available in the second release, currently being implemented.
A3.9	Recipe-TD Matcher	HESSO	4.1.2, 4.2.2	Implementation of matching of Recipes and thing descriptions according to TD affordances and swarm Recipe specifications	Available in the second release, currently being implemented.
A3.10	Mendix Orchestrator	HESSO	4.1.3, 4.2.3	Orchestrator of Mendix flows following a given Recipe and a given set of swarm nodes previously matched	Available in the second release, currently being implemented.
A3.11	Semantic Media Service	DELL	2.1.5, 2.2.5	Artefact to stream semantic graphs between nodes in the swarm, which will facilitate a shared environmental context.	Available in the second release, currently being implemented.



## 1.4 MAPPING KPIS AND ARTEFACTS

KPIs relevant for WP3 are shown in Table 1-2 and mapped with relevant artefacts designed and implemented to address them. The progress towards KPIs for the first release of the CSI tools is summarized in this table by referencing specific sections of the deliverable.

Table 1-2: WP3 Key Performance Indicators for CSI tools and related artefacts.

KPI number	Description	Related artefacts
<b>K2.1</b>	Semantic integration should be provided for at least 4 brownfield protocols and more than 3 green field devices.	A3.2
<i>Status Update</i>	Semantic integration of communication protocols has been analyzed in Section 3.4, and initially designed in Section 3.5 of D3.1. The final design has been provided as the middleware with Standardized Semantic Interfaces, see Section 2.1.3. Initially, the middleware with Standardized Semantic Interfaces would support four brownfield protocols and more than three greenfield devices. We proceeded with the design and implementation for greenfield protocol support as planned (Section 2.2.3). However, the use case requirement analysis indicated that supporting brownfield protocols was unnecessary. Instead, in use cases UC1 and UC4, it became clear that the SmartEdge middleware should also extend protocol integration within a virtual environment. As a result, our design for the middleware with Standardized Semantic Interfaces now includes support for two additional protocol integrations (OPC UA and MQTT <sup>3</sup> ) for Unity <sup>4</sup> , the preferred virtual environment in SmartEdge.	
<b>K2.2</b>	Message conversion performances increased by at least 80% wrt. the baseline described in [Scrocca21] (140ms conversion time with 50KBytes XML payloads)	A3.5, A3.6
<i>Status Update</i>	Analysis of existing processors for declarative mapping languages to identify relevant components and operations affecting the performance of the semantic conversion process (cf. design of the mapping processor for a DataOps pipeline reported in D3.1). Enhancement of the Mapping Template tool to support the optimized execution of declarative mapping rules for message conversion (both lifting and lowering) in a DataOps pipeline (cf. Section 3.2.1). Performance tests were performed to compare semantic conversion with the Mapping Template tool and existing processors for knowledge graph construction (cf. Section 3.2.1.3). Testing was performed using the DataOps pipeline for traffic data from UC2 (c.f. Section 3.3.1), and the JSON stream payloads (3Kb) were converted to RDF in less than 4 milliseconds on average. Further tests will be executed with bigger payloads for the second release.	

<sup>3</sup> <https://mqtt.org/>

<sup>4</sup> <https://unity.com/>

<b>K2.3</b>	Semantic integration scalability (in terms of maximum concurrent requests and data velocity) increased by at least 50% wrt. the baseline described in [Scrocca21] (100 requests per second with XML payloads of around 50 Kbytes on commodity hardware) on a single converter instance (T3.2)	A3.5, A3.6
<i>Status Update</i>	Analysis of different deployment options for a DataOps pipeline to minimise resource usage of a single instance and enable flexible scalability for increasing demand (cf. Section 3.2.2). Testing was performed using the DataOps pipeline for traffic data from UC2 (c.f. Section 3.3.1), and the JSON stream (10 req/s, 3Kb) was converted without dropping requests. The average conversion time of 4ms should enable processing of 250 req/s. Further tests will be executed with bigger payloads and increased number of concurrent requests for the second release.	
<b>K2.4</b>	Reduced complexity and configuration time (at least 70%) of swarm intelligence Apps through the automatic instantiation and orchestration of template-based specifications.	A3.8, A3.9, A3.10
<i>Status Update</i>	Simplification of the configuration process is part of the facilitated use of semantic Recipes, matching and orchestration, as described in Section 4. These metrics will be evaluated in the context of A3.10 orchestration with the Mendix runtime, expected to be completed for the second release.	

## 1.5 RELATIONS TO WP4 AND WP5

In this section, we discuss the designed interactions between the artefacts for Continuous Semantic Integration (CSI), developed by WP3, and the ones developed within WP4 and WP5. Additional details on the referred artefacts can be found in D4.2 and D5.2

The SmartEdge WP4 focuses on the networking aspects of a swarm and leverages the semantic representations defined by WP3 (A3.1) to exchange interoperable representations of the information on nodes composing a swarm. Additionally, the Task Orchestrator (A5.3.2) uses both the Knowledge Graph Repository (A3.3) and the Distributed Database for Network Information – Address Resolution Table (A4.5) to discover the available swarm nodes together with their semantic description and IP addresses. This information is then provided to the Swarm Coordinator (A4.2) to establish the communication to the Node Managers (A4.3) to request the designated nodes to join the swarm.

To ensure interoperability between the developments in WP3 and WP5, the Mendix toolchain from WP3 will be enhanced to orchestrate nodes within Recipes managed by the SmartEdge runtime engine, developed in WP5. The Mendix toolchain will communicate with the WP5 Orchestrator via Zenoh, sending tasks to the orchestrator, subscribing to retrieve execution results, and passing the data to subsequent Recipe

steps. This integration will enable seamless interaction between SmartEdge nodes, regardless of whether they are controlled by the SmartEdge runtime (WP5) or the Mendix runtime (WP3).

Within WP5, the DataOps toolbox is also used as part of A5.1 to implement the Data Stream Fusion artefact (A5.1.4). Indeed, a set of DataOps pipelines can be customized to process heterogeneous data sources and integrate them according to a shared semantic representation.

## 1.6 STRUCTURE OF THE DOCUMENT

The document has the following sections. Section 2 provides the final design and the first release implementation for Standardized Semantic Interfaces in SmartEdge. This work is primarily the subject of Task 3.1. Section 3 outlines the final design and first release of the DataOps toolbox in SmartEdge, which is in the scope of Task 3.2. Section 4 reports the current contribution in Task 3.3 on a low-code approach for orchestration of swarm edge applications; and finally, Section 5 closes the document, highlighting some of the conclusions found and discussing the next steps in WP3.

## 2 STANDARDIZED SEMANTIC INTERFACES FOR SMARTEDGE

---

Standardized Semantic Interfaces are fundamental in SmartEdge, providing access to integrated data and metadata through standardized communication interfaces. This section discusses the final design and initial implementation of the artefacts dedicated to these interfaces, as part of Task 3.1 of SmartEdge WP3. Specifically, these artefacts include:

- A3.1: SmartEdge Semantic Models
- A3.2: Middleware with Standardized Interfaces
- A3.3: Knowledge Graph Repository
- A3.4: Mendix Toolchain

A3.1 offers common data structures to represent data from various machines, applications, and swarms. This work is based on two widely adopted standards: W3C Web of Things and OPC Unified Architecture (OPC UA). The main tasks, as defined by D2.1 and refined in D2.2, are to enable easy low-code application development, facilitating formal description of nodes (devices), as well as semantic discovery of device skills and matching these skills with application requirements. This functionality is applied uniformly across different use case domains, regardless of the standards used. A3.1 also formalizes key concepts in SmartEdge, such as Swarm Node, Device, Capability, and Recipe.

A3.2 and A3.3 are infrastructure artefacts that enhance the functionalities of other artefacts. A3.2 provides access to data, while A3.3 offers access to metadata (semantics) from any SmartEdge node or device via unified and standardized interfaces, supporting easy low-code application development.

A3.4 is a toolchain that relies on A3.1, A3.2, and A3.3 to facilitate low-code application development. Additionally, this artefact integrates an interface for large language models (LLMs), simplifying the use of semantic models in low-code application development and enhancing the usability of the SmartEdge low-code toolchain.

### 2.1 FINAL DESIGN

#### 2.1.1 Final Design of SmartEdge Schema (A3.1)

This section provides an overview of all SmartEdge semantic models and then briefly mentions the final design of SmartEdge schema. SmartEdge semantic models are the artefacts that provide common data structures to represent the data from diverse machines, applications, and swarms etc. Existing standards such as W3C Web of Things, OPC UA, and existing domain models are reused to create SmartEdge semantic models. Reusing the standards provides harmonized interfaces for diverse machines and use cases and enables interoperability. The project develops several semantic model artefacts such as:

- semantic models for representing data from diverse nodes using Web of Things Thing Description and OPC UA standard. The node semantic models describe the capabilities offered by a node, its static and dynamic attributes required by the applications;

- semantic models for representing SmartEdge applications using Recipe model. The Recipe model describes the application requirements such as capabilities on the nodes required to execute the application, the required static, and dynamic attributes of the nodes;
- semantic models for representing the runtime attributes of a swarm using SmartEdge Schema will be a new semantic model that will be developed in the project. SmartEdge schema describes a swarm in runtime. It describes swarm attributes such as nodes involved in the swarm currently, Recipe that is being executed by the swarm, status of swarm execution etc.

Figure 2-1 gives the overview of semantic models in SmartEdge project. Together, all the semantic models provide common data structures from devices to applications.

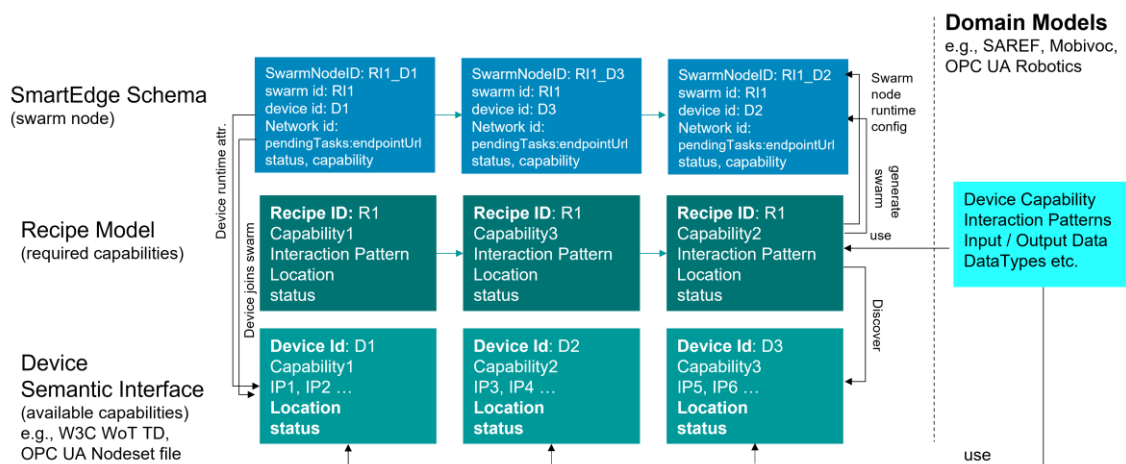


Figure 2-1: Semantic models in SmartEdge

The final design of SmartEdge schema is presented in D3.1 section 3.1. The details about the first implementation of SmartEdge schema are presented in this deliverable in section 17.

### 2.1.2 Final Design of Recipe model (A3.1)

The Recipe model defined in D3.1 is used to define applications in SmartEdge use cases. A Recipe is basically a semantic definition of an application template. It defines the requirements of an application such as the skills / capabilities a node needs in order to execute the applications and interaction between them. This information is needed in a Recipe to discover the right nodes to execute it. The interaction is defined at the higher abstraction in the Recipe model. It just defines which capabilities interact with each other, however the business logic during the interactions is out of scope of Recipe semantic definition. This higher abstraction makes the Recipe model flexible and enables it to be useable in different domains and different use cases. For example, Recipe model can be used to model application in SmartEdge UC1 which is based on W3C WoT standard and UC4 which is based on OPC UA standard. The example Recipes are presented in the first implementation of Recipe model in section 18.

The model defined in D3.1 is extended further to make it usable with AI technologies such as Large Language Models (LLMs). The changes are depicted in Figure 2-2.

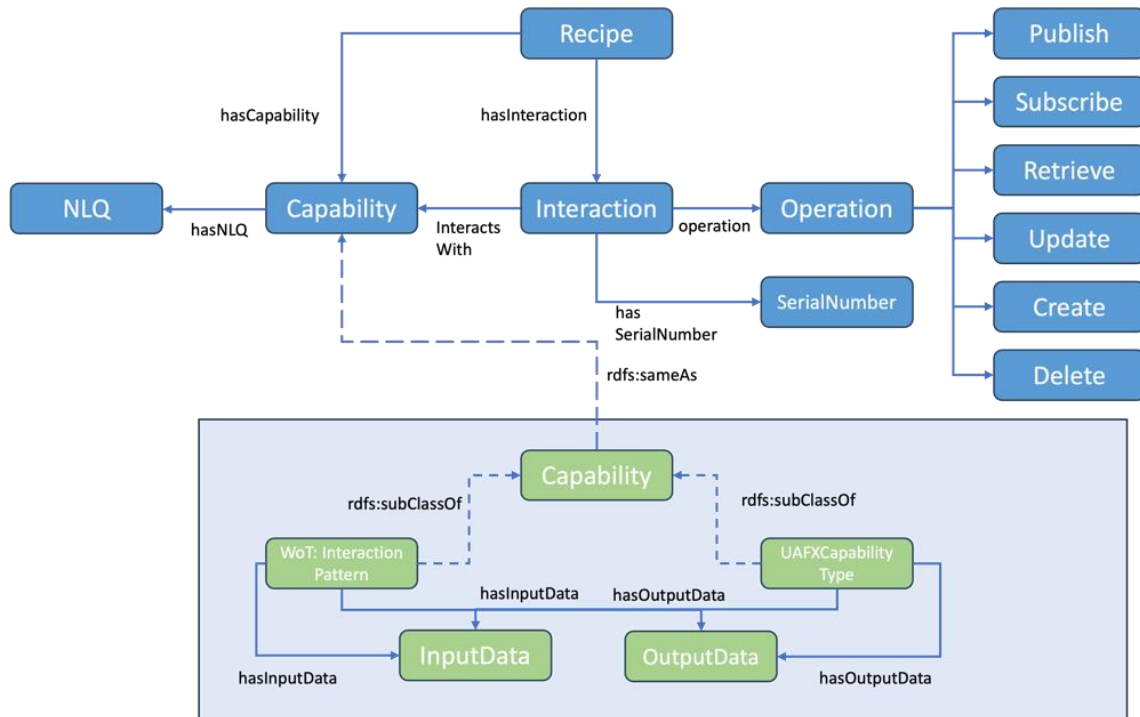


Figure 2-2: Recipe model extended with NLQ

Two new terms are added to the model: Serial Number and NLQ (Natural Language Query).

An application defined using a Recipe can have multiple interactions between different nodes. The Serial Number attribute uniquely identifies an interaction between nodes.

A capability in a Recipe specifies the skills a node should have in order run the application defined by the Recipe. The capability is formally represented in RDF format in a Recipe. However, now we also added a new attribute to a capability called NLQ, which can be used to describe the capability requirements in natural language format that can be easily used by an LLM. Therefore, the requirements in a Recipe can be specified either formally in RDF format or in natural language using the NLQ. The purpose of an NLQ is that an LLM can understand it and search for a matching thing / machine which can fulfill the requirements specified in Recipe. With this approach, we could leverage LLMs to convert the NLQ into SPARQL query to search for a required thing / machine.

Since the skills of a thing / machine can be defined using the XML-based OPC UA standard, we need to convert these skills into the RDF format. This conversion allows us to use SPARQL for discovering skills that meet the required capabilities of a Recipe. To achieve this, we have implemented the work from [Schiekofer19] in the DataOps tool, as detailed in Section 3. Our approach allows us to construct an OPC UA ontology to describe skills and match them with the required capabilities. Given the complexity of the OPC UA ontology, we leverage LLMs to formulate SPARQL queries, which are then used to match skills against capabilities. It makes the application development process

more intuitive and interactive to a user as the user can specify the requirements for the application in natural language.

The use of LLMs has not been originally planned in the description of work in the SmartEdge project. However, meanwhile the use of LLMs and AI, in general, with graph data and semantic models has been increasing significantly. Thus, in the SmartEdge project we started the development of an LLM based approach for Recipe implementation. Currently it is being tested with OPC UA standard. For this purpose, Mendix is extended with an LLM-based application called OPC UA copilot which provides a natural language interface over OPC UA knowledge graphs (A3.3). The OPC UA copilot converts the NLQ into a SPARQL query, executes that query on the underlying OPC UA knowledge graph and returns the results to Mendix to implement the application specified by a Recipe. The LLM driven approach is shown in Figure 2-3.

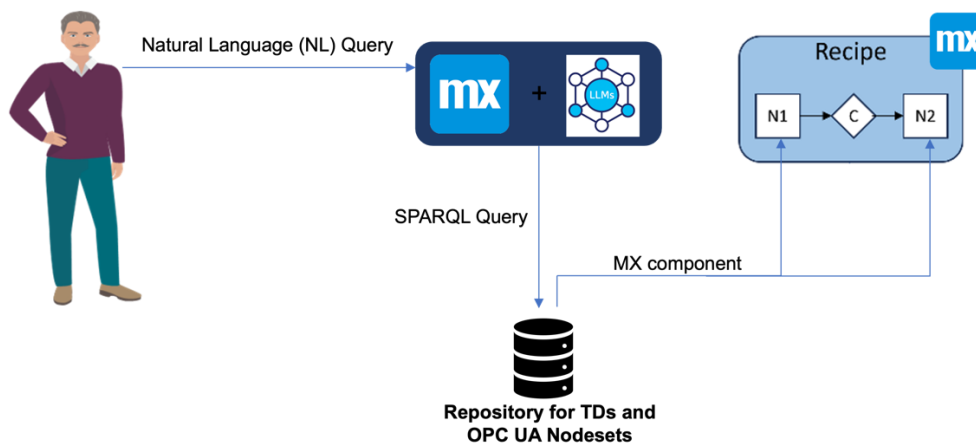


Figure 2-3: LLM-driven approach to address user's requirements in Recipe development

The workflow, which we are implementing for discovery purposes with LLMs in Mendix (MX), is performed in the following steps:

- 1) A User or a Recipe developer expresses his/her requirements in natural language.
- 2) An LLM interface (integrated in MX) produces a SPARQL query. The query captures user's requirements.
- 3) Repository evaluates the query and provides answer to MX.
- 4) An extension of MX will generate a MX component, which connects to a node via OPC UA standardized interface (A3.2).
- 5) Process will repeat for other capabilities in a Recipe.

### 2.1.3 Final Design of Middleware with Standardized Semantic Interfaces (A3.2)

Standardized semantic interfaces provide a common way to access the devices' data from the application level. For the different use cases covered by the project, in deliverable D3.1 we identified a need for communication across diverse protocols, such

as OPC UA, MQTT, DDS<sup>5</sup>, and Bluetooth (BLE). Ensuring interoperability at the protocol level is essential to make use of these interconnected systems.

To overcome the challenges of multi-protocol device communication and to enable interoperability at the protocol layer, we envision using a middleware solution that unifies the messages across different protocols and enables interoperability as shown in Figure 2-4. The messages from different protocols being unified at the middleware layer allow the dataflow vertically and horizontally and, also, enable unified access to the data from the application layer.

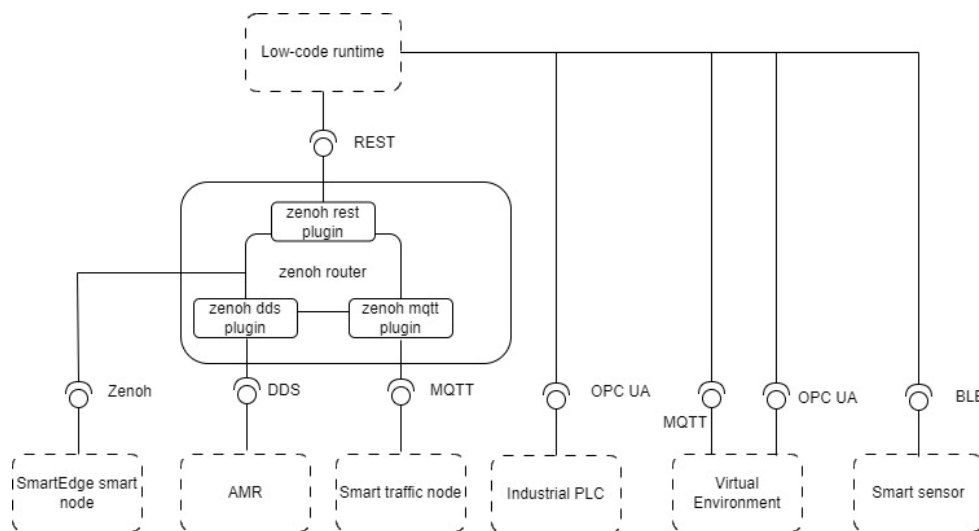


Figure 2-4: Standardized Semantic Interfaces in SmartEdge.

This feature will be delivered in the form of a Docker container, containing Zenoh router and a set of plugins and backend libraries to support use case specific communication protocols. In the first iteration, Zenoh router is extended with MQTT and DDS as southbound interfaces and REST as a northbound interface.

Initially, the project proposal outlined that the middleware with Standardized Semantic Interfaces would support four brownfield protocols and more than three greenfield devices (see KPI K2.1). For greenfield protocol support, we proceeded with the design and implementation as planned (see Section 2.2.3). However, the use case requirement analysis revealed that supporting brownfield protocols was unnecessary. Instead, in use cases UC1 and UC4, it became evident that the SmartEdge middleware should also extend protocol integration within a virtual environment. Consequently, our design for the middleware with Standardized Semantic Interfaces now includes support for two additional protocol integrations (OPC UA and MQTT) for Unity, the preferred virtual environment in SmartEdge (Figure 2-4). Unity is used in SmartEdge in Use Case 1 and Use Case 4.

<sup>5</sup> <https://www.dds-foundation.org/>



In virtual environments, interoperability and communication interfaces are also use case-dependent, much like in real-world scenarios. For instance, in SmartEdge Use-Case 1, interoperability is achieved using Thing Descriptions with protocol bindings like MQTT. In contrast, Use Case 4 utilizes the OPC UA standard.

In Use Case 1 (UC1), a "Thing" in a virtual world exposes its interfaces via a Thing Description (TD) with an MQTT binding, which abstracts whether the thing is virtual or physical. This layer of abstraction enables any component in the SmartEdge ecosystem, including the SmartEdge Recipe model and the Mendix toolchain, to use virtual things in virtual worlds in exactly the same way as physical things with the same Thing Description. The (Remote) Rendering Runtime is responsible for managing the lifecycle—including activation and deactivation—of Thing Descriptions associated with virtual things (see Example in previous subsection).

In Use Case 4 (UC4), the OPC UA standard is employed to describe a "Thing" and facilitate data communication. Unity serves as the Industrial Metaverse environment for virtual commissioning. The low-code runtime interacts with Unity via the OPC UA protocol. Consequently, in SmartEdge, the middleware with standardized semantic interfaces also integrates the low-code environment with the virtual environment using the OPC UA protocol, as illustrated in Figure 2-4.

#### 2.1.4 Final Design of Knowledge Graph Repository (A3.3)

SmartEdge Knowledge Graph Repository is a specialized semantic repository designed for the storage, retrieval, and management of standardized device descriptions. The repository supports W3C Web of Things and OPC UA standard. For example, a device can be described either with a W3C Thing Description (TD) or with an OPC UA Nodeset.

In both cases, the device description can be retrieved via an interface of the Knowledge Graph Repository. Figure 2-5 shows two standardized interfaces of the SmartEdge Knowledge Graph Repository. The repository implements Thing Description Directory (TDD) API, which is a standardized API for TDs<sup>6</sup>. For OPC UA, a standardized API to access data in a knowledge graph does not exist. Still, in both cases we provide a SPARQL RESTful interface. These SPARQL interfaces will be used for discovering devices, which can be used in the matchmaking process when implementing SmartEdge Recipes. Different Triplestore, i.e., databases for RDF graphs, can be configured for A3.3 assuming they are compliant with the SPARQL protocol<sup>7</sup>. SmartEdge Knowledge Graph Repository will be offered as a standalone feature.

It will be also possible to integrate the feature with other features, e.g., the Mendix Toolchain (A3.4) presented in Section 2.2.5. The discussion on how OPC-UA support is enabled via DataOps pipelines is reported in Section 3.3.2, after the discussion of the artefacts and features associated with the DataOps toolbox.

---

<sup>6</sup> <https://www.w3.org/TR/wot-discovery/>

<sup>7</sup> <https://www.w3.org/TR/sparql11-protocol/>

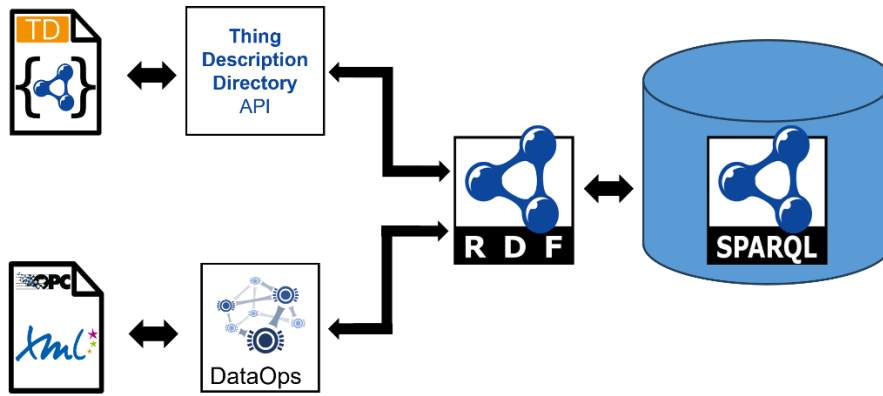


Figure 2-5: Repository for Thing Descriptions and OPC UA Nodesets

2.1.5 Final Design of Mendix Toolchain (A3.4)

The Mendix toolchain comprises Mendix Integrated Development Environment (IDE) and Mendix runtime. The Mendix IDE will be extended to support development of SmartEdge Recipes. The Mendix runtime, is an interpreter that runs Mendix application and provides the frontend to the user. This part will also be extended to facilitate execution of SmartEdge Recipes. Mendix runtime will interact with A3.3 Knowledge Graph Repository over the SPARQL RESTful interface for discovering devices and their capabilities and executing the matchmaking process. Mendix runtime will also interact with the A3.2 Middleware over REST to execute the Recipes on the selected devices.

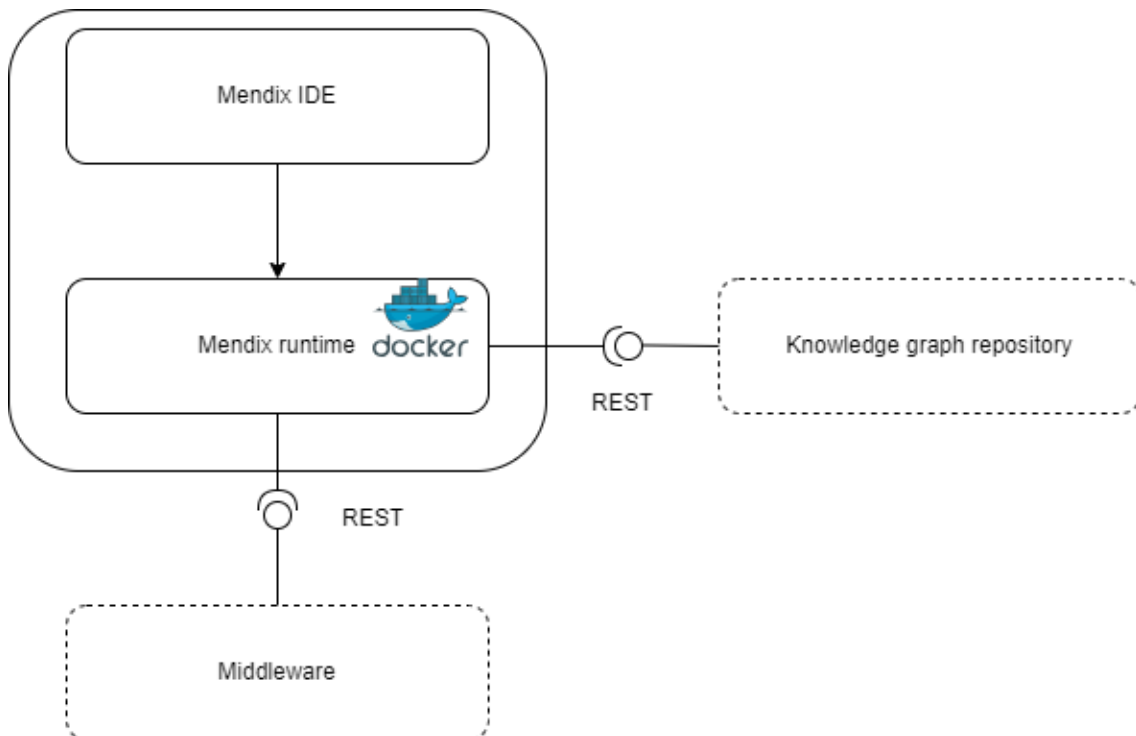


Figure 2-6: Mendix Toolchain

Mendix extensions, allowing interactions with Knowledge Graph Repository and further extensions will be provided in form of Mendix modules<sup>8</sup>. Mendix runtime itself will be delivered in a form of Docker container, as shown in Figure 2-6.

<sup>8</sup> <https://docs.mendix.com/appstore/modules/>

## 2.1.6 Final Design of Semantic Media Service (A3.11)

This artefact will facilitate the sharing of streaming semantic media between smart-nodes in a swarm. Semantic media includes scene understanding graphs, which are generated by artefact A5.1.2.2 described in D5.2. The artefact processes and fuses data from smart-node depth cameras and LiDARs in the swarm to provide an abstract semantic understanding of the objects in the environment from the nodes point of view. The semantic scene graph is then streamed to this artefact, A3.11, as illustrated in Figure 2-7.

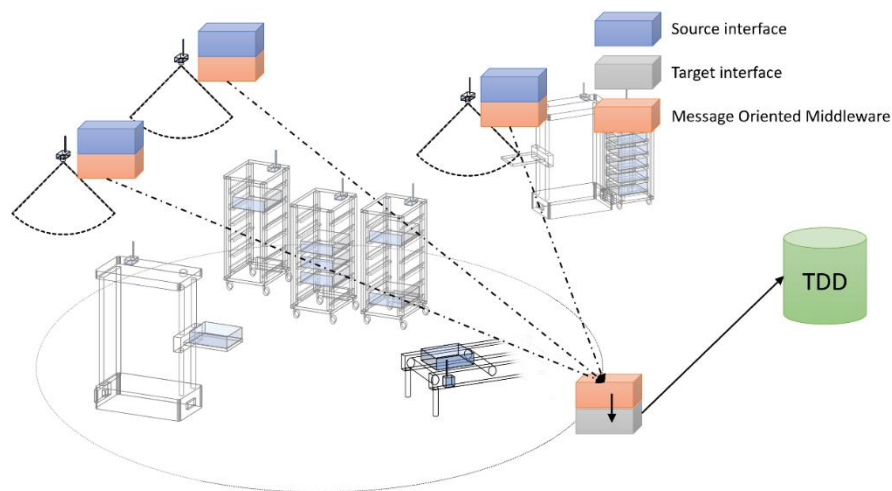


Figure 2-7: Manufacturing illustration of streaming semantic media service

A smart-node sharing semantic scene understanding graphs publishes the graphs as RDF triples on the message topic `/perception/scene_understanding/graph`. The A3.11 artefact subscribes to these messages and consolidates the scene understanding graphs from multiple smart-nodes to build up a 3D model of the environment based on multiple viewpoints and Thing Descriptions from the TDD. The consolidated 3D environment model is also represented as RDF triples in a semantic graph, which can then be streamed to other smart-nodes in the swarm or used to generate other types of semantic media such as a 2D occupancy map, which nodes can use to navigate the environment, as illustrated in Figure 2-8.

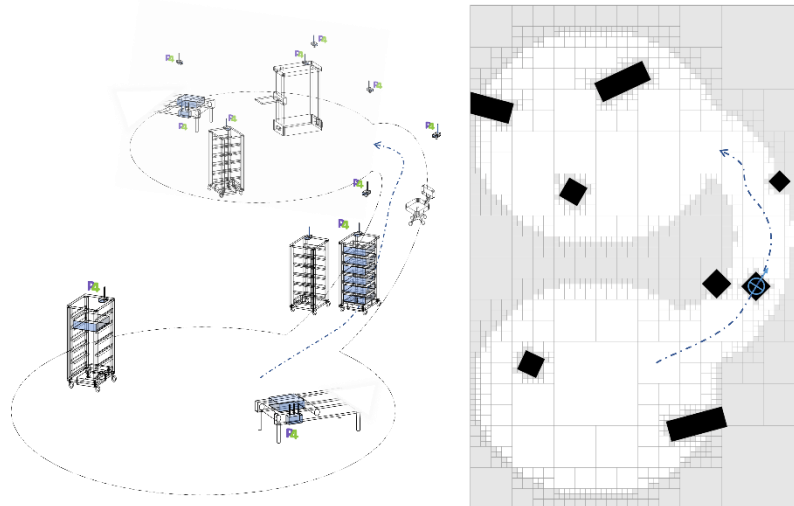


Figure 2-8: Factory schematic and corresponding 2D occupancy map

The semantic media service will allow smart-nodes to share knowledge about their environment and build up a more complete internal model, providing them with perspectives they cannot perceive directly. For example, in the factory scenario, several ceiling-mounted cameras would be able to provide different perspectives on an operational area. Another smart-node in the swarm would be able to subscribe to these semantic graph streams and so perceive the operational area in-the-round, facilitating the construction of a 3D model of the environment by the smart-node.

A smart-node finds the available streaming sources by first querying the TDD via its SPARQL interface and returning all possible streaming smart-nodes in a given region. It then directly subscribes to the source smart-nodes' semantic graph stream. In this way smart-node stream sources can be dynamically bound into the swarm. The artefact also includes source and target components to facilitate the semantic graph stream, which abstract the overlying message-oriented middleware; the only assumption is that the middleware supports the publish and subscribe message paradigm.

## 2.2 FIRST IMPLEMENTATION

### 2.2.1 First Implementation of SmartEdge Schema (A3.1)

SmartEdge schema aims to formally define the important concepts of the SmartEdge architecture which are used in swarm formation and execution. Purpose of the SmartEdge schema is to enable following swarm functions:

- It can be used during design time for the configuration of a swarm;
- It can be used in run time for identifying the nodes with matching skills which can join a swarm;
- It can be used to monitor the execution of a swarm (e.g., entry of a node into swarm, exit of a node and replacing a node in swarm).

The SmartEdge schema defines the concepts that are common to all swarms regardless of the use case applications, such as swarm co-ordinator, its interactions with a swarm orchestrator, industrial knowledge graph to discover nodes with required capabilities

and nodes in the swarm. Detailed explanation about concepts is presented in D3.1. While SmartEdge schema represents the runtime attributes of a swarm, the Recipe model represents an application template. When a Recipe is instantiated then the instance is considered as a swarm, which is dynamic and the nodes in the swarm can be replaced with other suitable nodes in runtime. The relationship between the Recipe and its corresponding swarm is captured in the SmartEdge schema using RecipeID class as represented in Figure 2-9.

The Figure represents the concepts in the SmartEdge schema and the relationships between them. The main concepts in the schema are the SmartEdge node, SmartEdge smart node, swarm co-ordinator and the swarm orchestrator. SmartEdge smart node is a subclass of SmartEdge node where the smart node has the capability to dynamically join or leave the swarm. Each of these nodes has certain attributes and relationships with other nodes which is depicted in Figure 2-9.

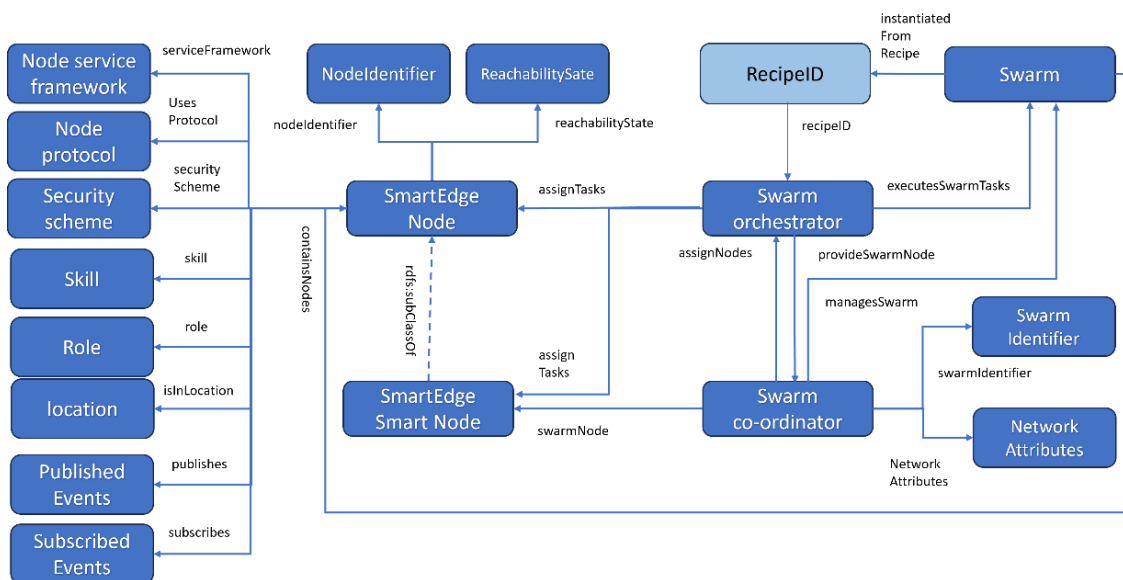


Figure 2-9: Overview of SmartEdge Schema

Each SmartEdge node has the attributes such as: node id, node capabilities, network attributes, location, events it publishes and subscribes, security scheme to connect to the node, its reachability state etc. which are characteristics of a node. A swarm coordinator has attributes such as swarm-id, network attributes etc. as it manages the swarm and connects to the nodes in the network. Swarm orchestrator has a relationship to the Recipe which it runs through the swarm. The first version of SmartEdge schema is implemented in RDF format and it can be found on the SmartEdge repository: <https://gitlab.com/smartedge-project-eu/SMARTEDGE/-/tree/main/WP3/A31>.

SmartEdge use cases could use the schema to describe a swarm in their use cases.

### 2.2.2 First Implementation of Recipe Model (A3.1)

The Recipe model is finalized and it can be found on the SmartEdge GitHub: <https://gitlab.com/smartedge-project-eu/SMARTEDGE/-/tree/main/WP3/A31>. Based on the finalized Recipe model, a first implementation of the discovery with Recipes is

done. The Recipe model is independent of standards (e.g., OPC UA, W3C WoT). The nodes with matching capabilities that are implemented using OPC UA or W3C WoT can be discovered and a Recipe can be implemented with them. Discovery of required nodes is done by generating SPARQL queries from Recipes. The SPARQL queries are pre-defined, and they are different for OPC UA and WoT. That is, we provided a set of SPARQL queries which can discover matching nodes implemented with W3C WoT standard. For Discovery of nodes from descriptions compliant with the OPC UA standard, i.e., OPC UA NodeSets, we use latest AI technologies such as Large Language Models (LLMs). This approach is described briefly in section 2.1.2. Based on the first implementation of Recipe discovery with LLMs in Mendix, here we provide a few example Recipes based on the W3C Web of Things Thing Descriptions and a Recipe for UC4, which is based on the OPC UA standard.

#### 2.2.2.1 Example Recipe for Smart Factory Application based on OPC UA Standard for UC4

Below we provide a simple example Recipe considering nodes described according to the OPC UA standard. The Recipe is related to UC4 in SmartEdge. The sample Recipe does not completely represent any UC4 application, rather it is part of an application described in UC4. The purpose of the Recipe is to assemble a product in a manufacturing unit. The application should load an empty tray into an assembly module of a manufacturing unit where the product gets assembled in multiple steps by inserting 4 different types of blocks onto the tray. After assembly, the product should be unloaded from the assembly module. The corresponding Recipe JSON-LD description for this application is presented in Section 7 (Annex I) of this document.

The discovery of matching capabilities for the above Recipe can be done based on the NLQ in the Recipe using a LLM application that is integrated into Mendix for discovery of OPC UA data points. The above Recipe consists of the workflow depicted in Figure 2-10.

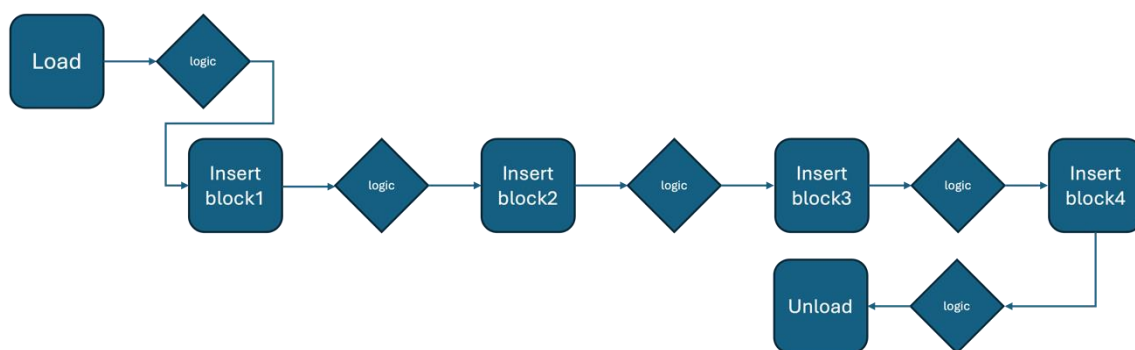


Figure 2-10: Graphical representation of sample Recipe for smart factory application in UC4

#### 2.2.2.2 Example Recipe with W3C Web of Things Thing Descriptions

Below we provide a simple example Recipe. The Recipe is not part of any use case in SmartEdge. We provide here a use case agnostic example which is easily understandable. The purpose of the Recipe is to turn on or turn off a lamp based on the proximity of a person or object to the lamp. The business logic for the application based

on Recipe is not part of the Recipe semantic model. When the Recipe is created in Mendix then business logic can be added to it through Mendix nodes as shown in section 69. The created business logic will be part of the Mendix project. In this example, the Recipe will be instantiated on things implemented with Web of Things standard which have corresponding Thing Descriptions.

```
{
  "@context": [
    {
      "RecipeModel": "http://www.semanticweb.org/SmartEdge/RecipeModel/",
      "saref4bdlg": "https://saref.etsi.org/saref4bdlg/",
      "saref": "https://saref.etsi.org/saref/",
      "iot": "http://iotschema.org/",
      "@id": "http://www.semanticweb.org/SmartEdge/RecipeModel/",
      "@type": [
        "http://www.w3.org/2002/07/owl#Ontology"
      ]
    }
  ],
  "@type": [
    "RecipeModel:Recipe"
  ],
  "title": "Lamp control Recipe",
  "NLQ": "An application to turn off a lamp based on the proximity of a person or an object",
  "RecipeModel:hasCapability": {
    "@type": [
      "iot:LightControl", "iot:MotionControl"
    ]
  },
  "RecipeModel:hasIngredients": [
    {
      "status": {
        "@id": "b4493a89cfd4a062",
        "NLQ": "find a sensor which can detect motion in <room_x>",
        "description": "current status of the lamp",
        "@type": [
          "iot:MotionDetected",
          "RecipeModel:Ingredient"
        ]
      },
      "RecipeModel:hasOutputData": {
        "type": "boolean"
      },
      "RecipeModel:operation": "RecipeModel:Retrieve",
      "iot:capability": [
        { "@type": "iot:MotionControl" }
      ],
      "RecipeModel:interactsWith": [
        {
          "hasSerialNumber": "1",
          "@id": "bcfca6fc0f1c1e8b",
          "RecipeModel:operation": "RecipeModel:Update"
        }
      ]
    }
  ],
  "toggle": {
```

```

"@id":"bcfca6fc0f1c1e8b",
"NLQ": "find a lamp which can be turned on and off in <room_x>",
"description":"Turn the lamp on or off",
"@type":[
  "iot:Toggle",
  "RecipeModel:Interaction"
],
"RecipeModel:hasInputData":{
  "type":"boolean"
},
"iot:capability" :
{"@type" : "iot:LightControl"},
"RecipeModel:operation":"RecipeModel:Update"
}}

```

In order to instantiate the sample Recipe, we should discover the things which have the capabilities specified in the Recipe. In the sample Recipe, the capabilities are MotionControl which can detect the motion in a room (with MotionDetected property) and LightControl capability which can control the lamp in a room (with Toggle action). The discovery can be done by generating the SPARQL queries from the Recipe description manually (or by using an LLM which takes the NLQ in the Recipe as input and automatically generates the SPARQL queries to discover the required things in case of OPC UA).

#### 2.2.2.3 Recipe Discovery

In order to instantiate an application based on a Recipe, we need to discover the nodes which can match the requirements defined in the Recipe. Since the Recipe is an RDF description, SPARQL queries can be used to discover the matching nodes. For this purpose, SPARQL queries should be generated from the Recipe. The queries should be executed on a knowledge graph where the semantic descriptions of nodes are stored. For each standard (e.g., W3C WoT, OPC UA etc.), the queries can be pre-defined. That is, in order to discover matching WoT Thing Descriptions (TD) from a Recipe a pre-defined SPARQL query can be instantiated. The SPARQL queries shown in this section will discover the matching nodes described with W3C WoT TDs which can run the lamp control Recipe described above. The queries discover: (i) a WoT TD with MotionControl capability which has interaction affordance to detect motion, (ii) another WoT TD with light control capability which has interaction affordance to turn on or turn off a lamp. Additionally, the queries search for both nodes located in the same building and the same room.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX td: <https://www.w3.org/2022/wot/td/v1.1/>
PREFIX iot: <http://iotschema.org/>
PREFIX saref: <https://saref.etsi.org/saref/>
PREFIX saref4bdlg: <https://saref.etsi.org/saref4bdlg/>

SELECT ?title ?id ?at ?iat ?op ?href
{
  ?s rdf:type td:Thing .
  ?s rdf:type ?thingType .

```



```

?s td:title ?title .
?s td:id ?id .

?s ?interaction ?interAff .
?interAff rdf:type ?at .
?o1 rdf:type ?iat .
?o1 td:forms ?b .
?b td:op ?op .
?b td:href ?href .

FILTER (?at IN (td:PropertyAffordance, td:ActionAffordance, td:EventAffordance))
FILTER (?thingType = iot:MotionControl) .
FILTER (?iat IN (iot:MotionDetected) ) .

OPTIONAL {?s saref4bdlg:isContainedIn "Room_1" . }
OPTIONAL {?s saref4bdlg:isSpaceOf "Building_1" . }
} LIMIT 100
SELECT ?title ?id ?at ?iat ?op ?href
{
  ?s rdf:type td:Thing .
  ?s rdf:type ?thingType .
  ?s td:title ?title .
  ?s td:id ?id .

  ?s ?interaction ?interAff .
  ?interAff rdf:type ?at .
  ?o1 rdf:type ?iat .
  ?o1 td:forms ?b .
  ?b td:op ?op .
  ?b td:href ?href .

  FILTER (?at IN (td:PropertyAffordance, td:ActionAffordance, td:EventAffordance))
  FILTER (?thingType = iot:LightControl) .
  FILTER (?iat IN (iot:Toggle) ) .

OPTIONAL {?s saref4bdlg:isContainedIn "Room_1" . }
OPTIONAL {?s saref4bdlg:isSpaceOf "Building_1" . }
}
LIMIT 100

```

Below is an example WoT TD that is discovered from the SPARQL queries specified above.

```

{
  "@context": ["https://www.w3.org/2022/wot/td/v1.1",
    {"saref4bdlg": "https://saref.etsi.org/saref4bdlg/",
      "saref": "https://saref.etsi.org/saref/"}],
  "id": "urn:uuid:014139c9-b267-4db5-9c61-cc2d2bfc217d",
  "title": "MyLampThing",
  "@type": ["saref4bdlg:Lamp"],
  "saref4bdlg:isContainedIn": "Room_1",
  "saref4bdlg:isSpaceOf": "Building_1",
  "securityDefinitions": {
    "basic_sc": {
      "scheme": "basic",
      "in": "header"
    }
  }
}

```

```

    }
  },
  "security": "basic_sc",
  "properties": {
    "status": {
      "@type": "saref4bdlg:colorTemperature",
      "type": "string",
      "readOnly": false,
      "writeOnly": false,
      "observable": false,
      "forms": [{
        "op": [
          "readproperty",
          "writeproperty"
        ],
        "href": "https://mylamp.example.com/status",
        "contentType": "application/json"
      }]
    }
  },
  "actions": {
    "toggle": {
      "@type": "saref:Switch",
      "safe": false,
      "idempotent": false,
      "forms": [{
        "op": "invokeaction",
        "href": "https://mylamp.example.com/toggle",
        "contentType": "application/json"
      }]
    }
  },
  "events": {
    "overheating": {
      "@type": "saref4bdlg:Alarm",
      "data": {
        "type": "string",
        "readOnly": false,
        "writeOnly": false
      },
      "forms": [{
        "op": "subscribeevent",
        "href": "https://mylamp.example.com/oh",
        "contentType": "application/json",
        "subprotocol": "longpoll"
      }]
    }
  }
}

```

A SPARQL query template can be extracted from the above queries to discover WoT TDs from a given Recipe. The template can be instantiated based on the capabilities described in a Recipe. Such a sample template is presented in the below snippet.

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX td: <https://www.w3.org/2022/wot/td/v1.1/>

```

```

PREFIX saref4bdlg: <https://saref.etsi.org/saref4bdlg/>
PREFIX saref: <https://saref.etsi.org/saref/>

SELECT ?title ?id ?at ?iat ?op ?href
{
  #Get the title and id of a thing with capability_type
  ?s rdf:type td:Thing .
  ?s rdf:type ?thingType .
  ?s td:title ?title .
  ?s td:id ?id .

  #Get interaction affordances, their data types, their allowed operations and hrefs
  ?s ?interaction ?interAff .
  ?interAff rdf:type ?at .
  ?o1 rdf:type ?iat .
  ?o1 td:forms ?b .
  ?b td:op ?op .
  ?b td:href ?href .

  FILTER (?at IN (td:PropertyAffordance, td:ActionAffordance, td:EventAffordance))
  FILTER (?thingType = <Capability_Type> ) .
  FILTER (?iat IN (<Interaction_1_Semantic_Type>, ..., <Interaction_n_Semantic_Type> )) .

  OPTIONAL {?s saref4bdlg:isContainedIn <Room_no> .}
  OPTIONAL {?s saref4bdlg:isSpaceOf <Building_no> .}
}

```

The query template can be part of the Recipe matchmaker in Mendix (A3.9), which can instantiate the queries based on a given Recipe.

Alternatively, the requirements for capabilities are also specified in textual format as NLQ in the Recipe. This NLQ can be given to an LLM which can generate the SPARQL query, execute it on a given knowledge graph and discover the matching things that can run the Recipe application.

Currently, in SmartEdge, the discovery with LLM approach is being tested for OPC UA standard to discover machines described with OPC UA information models. Section 19 presents a sample OPC UA based Recipe with NLQs. The LLM integrated in Mendix uses the NLQs and discovers the matching nodes by generating SPARQL queries.

#### 2.2.2.4 *Current status and next steps*

Based on these two example Recipes, here we demonstrated that the Recipe model is flexible and domain-independent. It can define diverse kinds of applications in diverse domains. Therefore, it is suitable for the semantic representation of applications in SmartEdge. SmartEdge itself is a proof of concept as its use cases come from diverse domains.

As the next steps we will define a capability model based on W3C WoT TD model and implement it. Moreover, the Recipe model should be aligned with the Mendix flow model (i.e., the Recipe created in Mendix as Mendix flow can be exported in JSON format). The terminology in the Recipe model should be mapped with the Mendix flow terminology and the context should be generated from the aligned Recipe model. This

context can be added to the Recipe Mendix flow, which will transform the Mendix flow into RDF format. Finally, Recipes based on the Recipe model should be developed for each use case. At present Recipe development has started for UC1 and UC4, the examples are provided in this deliverable.

### 2.2.3 First Implementation of Middleware with Standardized Semantic Interfaces (A3.2)

We have been working on testing Zenoh as a technology for a middleware solution that can be used within SmartEdge. We have successfully completed initial testing with Zenoh as a message-oriented middleware. The tests showcased seamless interaction using the DDS and MQTT protocols, demonstrating Zenoh's capability to bridge different communication standards effectively.

Another implementation effort was dedicated to integrating OPC UA client functionality in a virtual environment, e.g., into the Unity platform. Our new implementation enables Unity to connect with an OPC UA server using the OPC UA .NET Standard Stack. This integration includes the ability to read data from nodes, write to nodes, subscribe to variables' updates. This allows for the use of the OPC UA standard within virtual environments. An example, where the OPC UA connector for Unity is used is depicted in Section 2.2.2.1.

Also, a proof-of-concept was developed to integrate MQTT Binding for Thing Descriptions into a virtual scene as a Unity plugin, using a remote rendering artefact. This demonstrates the potential of using MQTT-based communication for dynamic interaction with virtual objects in real-time. An example, where the MQTT connector for Unity is used is depicted in Section 2.2.2.2.

The upcoming tasks focus on further integrating these advancements within the scope of Mendix and Middleware interactions. The next objective is to further test Mendix microflows that can interact with Zenoh. Further efforts will be directed towards integrating virtual scenes with TDDs, including implementing Life Cycle Management for virtual Things. This will enable the use of these virtual objects in Recipes. These steps are crucial for building a robust framework that can handle complex interactions between virtual scenes and real-world systems, paving the way for more dynamic and scalable applications. Additionally, the OPC UA connector for Unity will undergo further tests and, possibly, extensions.

In the following two sub-sections we provide example applications that use MQTT and OPC UA connectors for Unity, respectively.

#### 2.2.3.1 *Example application using MQTT based virtual environment integration*

This subsection shows the first implementation of the MQTT based virtual environment integration using W3C Thing Description with MQTT protocol binding. The JSON-LD example below shows a Thing Description representation of a virtual car from Use Case 1. In this example, the two properties, acceleration and steering, are the most important elements in the Thing Description. They allow an external application, such as ADAC, to control the virtual car in the virtual environment, enabling the following:

- Simulation of complex traffic scenarios in a controlled, repeatable way.
- Time and cost savings, along with increased safety in testing.
- Large-scale testing that would be impractical or unsafe in the real world.

```
{
  "@context": "https://www.w3.org/2022/wot/td/v1.1",
  "title": "MyVirtualCar",
  "id": "urn:uuid:9489991a-7622-45b6-8437-f858b59835d4",
  "securityDefinitions": {
    "nosec_sc": {
      "scheme": "nosec"
    }
  },
  "security": [
    "nosec_sc"
  ],
  "properties": {
    "acceleration": {
      "data": {
        "type": "number",
        "minimum": -1.0,
        "maximum": 1.0
      },
      "forms": [
        {
          "href": "mqtt://192.168.1.187:1883",
          "contentType": "text/plain",
          "op": [
            "readproperty",
            "writeproperty"
          ],
          "mqv:topic": "scene1/things/car1/properties/acceleration"
        }
      ]
    },
    "steering": {
      "data": {
        "type": "number",
        "minimum": -1.0,
        "maximum": 1.0
      },
      "forms": [
        {
          "href": "mqtt://192.168.1.187:1883",
          "contentType": "text/plain",
          "op": [
            "readproperty",
            "writeproperty"
          ],
          "mqv:topic": "scene1/things/car1/properties/steering"
        }
      ]
    }
  }
}
```

Each property in the Thing Description above includes two sections: “data” and “forms”. The “data” section defines the structure and restrictions applied to a property, while the “forms” section defines the binding to the underlying protocol. In the Thing Description example above, the property “steering” accepts only numbers between -1.0 and 1.0, indicating the direction of steering (negative values correspond to left turns, positive values to right turns, and zero to move straight). It also binds the “steering” property to the MQTT protocol via the endpoint provided in “href” and the MQTT topic defined in “mqv:topic”. In this example, any external application can read and change the “steering” property, which maps to MQTT publish or subscribe operations for the topic specified in “mqv:topic”.

The virtual car Thing Description can be used in a virtual scene whereas a virtual scene is a digital replica of a physical environment, such as a factory or traffic situation. In these environments, physical entities like robots, cars, and many others are represented in the virtual scene by their digital entities/assets, or digital twins. Since the SmartEdge ecosystem is represented by a set of interworking artefacts, it is a key requirement to make the digital assets in virtual scenes accessible to other SmartEdge components in the same way as physical assets. W3C Thing Description is a suitable method to enable this functionality. The Rendering Engine exposes all virtual assets in a virtual scene using Thing Description, making them available in a Thing Description repository, where Thing Descriptions of physical assets are also registered. In this way, a SmartEdge application that uses the Thing Descriptions from the repository will not need to distinguish between physical and virtual assets. Figure 2-11 shows an example of a virtual scene for a traffic scenario with virtual cars that mimic the behaviour of physical cars in a real traffic scenario.

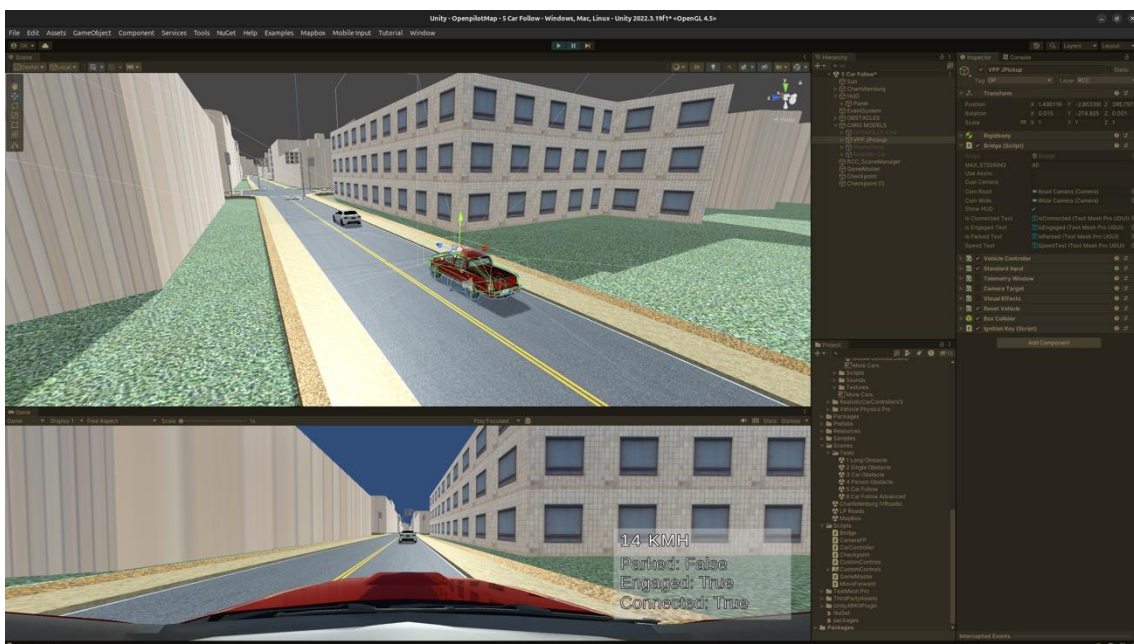


Figure 2-11 Virtual Scene with Virtual Car

### 2.2.3.2 Example application using OPC UA based virtual environment integration

In Use Case 4, Unity<sup>9</sup> is used for the purpose of virtual commissioning in an Industrial Metaverse environment. The Mendix low-code runtime communicates with Unity over OPC UA protocol. Industrial assets, like the assembly module seen in Figure 2-12, have a digital twin in Unity. The real asset is defined using OPC UA's standardized semantics in the form of skills, which are then mapped to its virtual representation. Leveraging these OPC UA skills, the low-code runtime, enhanced with large language models, can swiftly and easily generate Recipes. These Recipes are executed within the virtual environment for virtual commissioning. To ensure the generated Recipe matches the low-code engineer's vision, the virtual environment employs physics simulations to preview the Recipe's execution. This visual confirmation allows the engineer to verify the accuracy and behaviour of the production line.

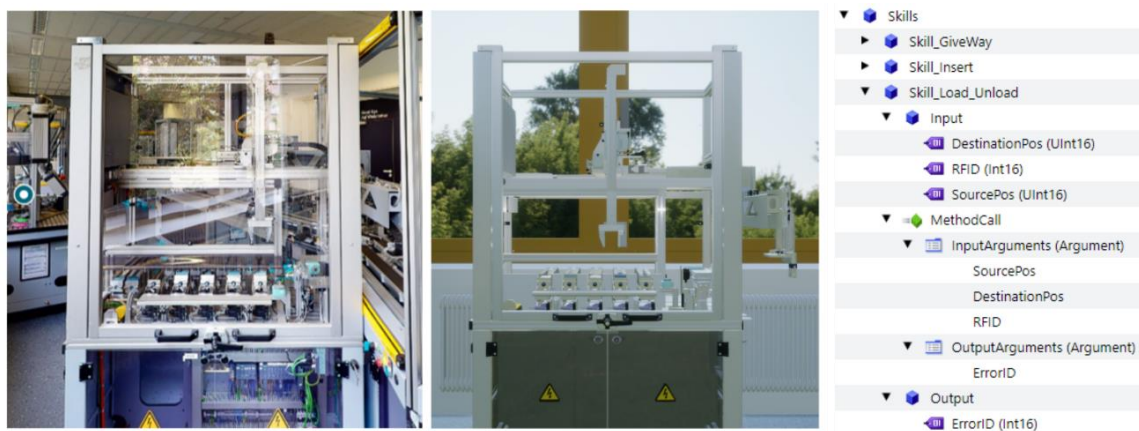


Figure 2-12: Production Module with its Virtual Counterpart and OPC UA Information Model from UC4

### 2.2.4 First Implementation of Knowledge Graph Repository (A3.3)

The Knowledge Graph Repository is the basis for semantic queries and the discovery service in the low-code toolchain. We use the [Domus TDD API](#) from Eclipse Thingweb as our Knowledge Graph Repository. Domus implements a Python and SPARQL-based Thing Description Directory (TDD). It complies with the W3C specifications and implements the Web of Things [Discovery Exploration Mechanisms](#). The API relies on a SPARQL endpoint as a database connection. The supported endpoints are Apache Jena's Fuseki, Ontotext's GraphDB, OpenLink Software's Virtuoso, and AWS Neptune. Fuseki is set as the default endpoint. If one wants to change the endpoint, they can do so using two methods: (i) editing the `config.toml` file with the corresponding `SPARQLENDPOINT_URL` value, or (ii) by using environment variables like so: `export TDD__SPARQLENDPOINT_URL="http://my-new-sparql.endpoint/address"`.

The Figure 2-13 shows how the JSON and RDF files are dealt with in the TDD API.

<sup>9</sup> <https://unity.com/>

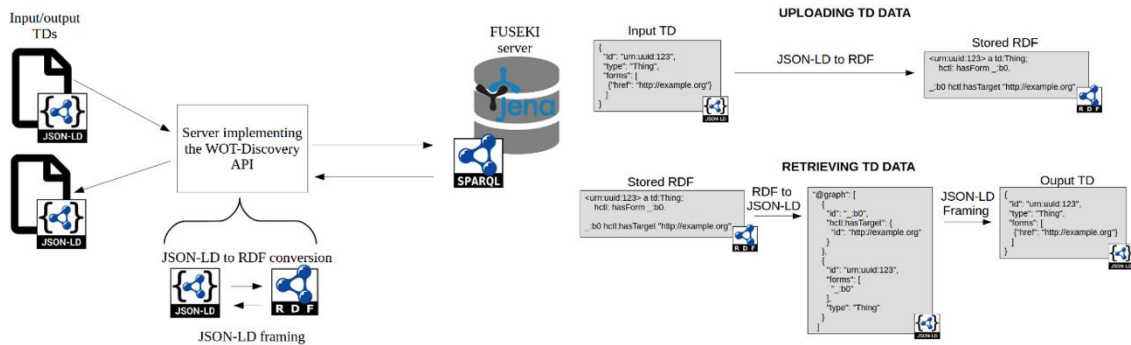


Figure 2-13: Thing Description Upload and Retrieval via TDD API

The Knowledge Graph Repository is released as part of the first SmartEdge release. It is shipped with the docker-compose file and is available on the Docker registry on the project integration environment. To enable storing of OPC UA NodeSets in RDF format, a DataOps pipeline was designed and implemented. The first version of a SPARQL query interface for OPC UA NodeSets is available for the first release and fully documented in Section 3.3.2 (DataOps Pipeline for OPC-UA support).

2.2.5 First Implementation of Mendix Toolchain (A3.4)

Recent progress has been made in extending Mendix capabilities, particularly in supporting various communication protocols and integrating new connectors for advanced use cases. The first version of the WoT client connector has been developed. It allows to read/write a property, subscribe to an event, and/or invoke an action via WoT REST API. The developed functional blocks are shown in Figure 2-14.

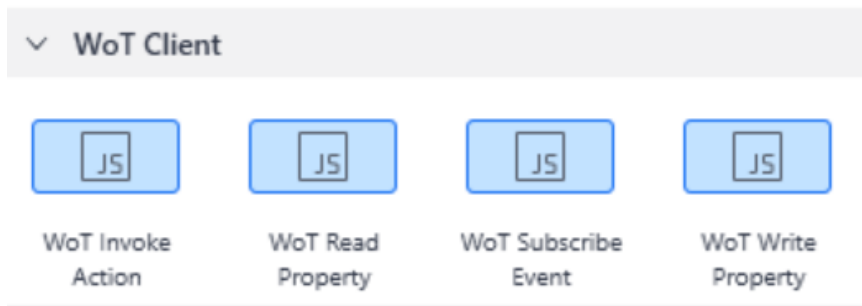


Figure 2-14: Mendix toolbox for WoT client

Besides that, the latest version of Mendix has been successfully extended to include support for BLE communication, specifically tailored for the use case 5b. This connector allows to directly get the data from BLE-devices in the Recipe’s instances, running in Mendix runtime. Furthermore, the OPC UA connector for Mendix has been extended to support the OPC UA method calls. The current version supports only the parameter-less methods and must be extended in the future release to be capable of calling the methods with parameters. With further successful tests of BLE, OPC UA, and REST connectors, Mendix supports various communication protocols and allows to use diverse devices to run the Recipes. Also, the Mendix runtime capable of executing Recipes has been dockerized and provided as an artefact.

The following steps towards the second release of SmartEdge tools, include extending Mendix to support the SmartEdge Recipe model. This will enable the development of



low-code solutions that can leverage predefined Recipes, simplifying the creation and reusability of complex IoT applications.

#### 2.2.6 First Implementation of Semantic Media Service (A3.11)

The artefact is scheduled for the second release of SmartEdge and has been delayed due to resources and other priority work. Initially, the plan was to use projective geometry to calculate the location and pose of the objects in the environment, but this did not prove to be possible, so another technique had to be developed by matching partial images to the object. We have started work on building an image dataset of manufacturing equipment. The dataset will be used for a number of purposes, including training object detectors to classify objects in the environment correctly. The artefact also extensively uses the dataset to model the objects in both CAD and URDF. The models form a digital twin of the object that allows the pose of the object to be discerned from only a partial image.

As next steps, we will complete the modelling of the objects in the environment and start training the object classifiers. Furthermore, we aim to investigate techniques for calculating the location and pose of an object for a partial image. This work is still ongoing but iNeRF looks promising.

### 3 DATAOPS TOOL FOR SEMANTIC MANAGEMENT OF THINGS AND EMBEDDED AI APPS

---

The DataOps toolbox is designed and implemented in SmartEdge to support the continuous integration of Things and Apps, facilitating their deployment from the Cloud to the Edge. This tool aims to provide a solution for enabling data exchanges, harmonisation, and integration in implementing edge intelligence among nodes within a swarm. A special focus is given to the performance and scalability requirements and the need to support different deployment environments.

Considering the SmartEdge requirements elicited in D2.1 and refined in D2.2, the DataOps toolbox has been designed in D3.1 to address two main challenges:

- *Interoperability of static node information*: the description of the node information and its capabilities should be made interoperable and exchanged/retrieved according to common semantics;
- *Interoperability at runtime within a swarm*: a node's runtime information should be made interoperable, or the runtime data exchanges between nodes in the swarm should be mediated to guarantee their interoperability.

To achieve this, the following list of functionalities should be supported by the DataOps toolbox:

- Implementation of mediated data exchanges between an input and target node/component requiring different interaction mechanisms (e.g., from MQTT queue to REST API);
- Conversion of heterogeneous (semi-)structured data from an input format/schema to a target format/schema (e.g., JSON using custom schema to RDF using target ontology);
- Data integration/fusion by leveraging a common semantic representation, i.e., data can be converted to an RDF graph using a shared reference ontology.

Since implementing such functionalities depends on each scenario's specific requirements, a single solution cannot exist. For this reason, the DataOps toolbox is designed as:

- DataOps Pipeline Components (A3.5): a set of composable modules that can be appropriately configured and combined within a pipeline to address heterogeneous integration requirements within a swarm.
- DataOps Deployment Templates (A3.6): reusable templates to provide flexibility in deploying DataOps pipelines both in the Cloud and on the Edge.
- Low-code DataOps Configuration (A3.7): low-code approaches to support developers in the configuration of the pipelines.

As an orthogonal non-functional requirement, we focus on the performance and scalability of the DataOps tool that is evaluated considering the KPI 2.2 and 2.3.

The remainder of this chapter presents the final design and the first implementation of the artefacts A3.5, A3.6 and A3.7.

### 3.1 FINAL DESIGN

This section presents the final design of the DataOps toolbox and highlights the main innovations for each artefact. The diagram in Figure 3-1 represents the three artefacts implemented for the DataOps toolbox and their relation.

We describe the diagram by considering an example scenario related to implementing a mediated data exchange between two nodes within a swarm. The implementation of a proper DataOps pipeline for each data exchange requires the following information:

- Input/output data connector required
- Input and target output data format and schema
- Associated performance/scalability constraints and requirements

Based on these requirements, a set of components should be identified and selected considering the ones made available by A3.5. Such components can be composed and configured to define a DataOps pipeline. The low-code definition of the pipelines is enabled by the adoption of declarative approaches for defining the interactions among the components and the schema and data transformations to be performed.

Moreover, A3.7 enables the possibility of relying on a GUI to configure the pipeline via a drag-and-drop editor that also guides the user in the selection of the parameters for each selected component.

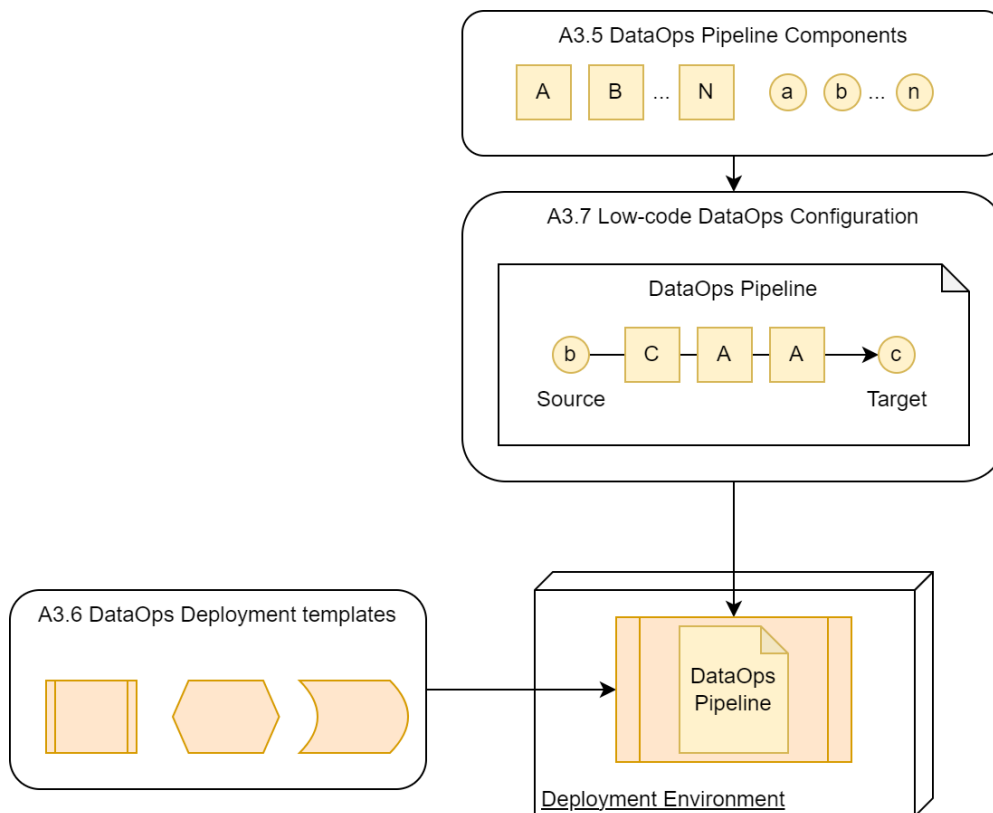


Figure 3-1: DataOps Toolbox related artefacts and their relation

The DataOps Tool should support diverse needs, particularly considering the different strategies for deploying a solution for mediating data formats and semantics. The diagram in Figure 3-2 shows different options that can be adopted for the integrated execution of a DataOps pipeline within the swarm:

- Within a dedicated smart-node<sup>10</sup> (mediation node)
- Embedded in the swarm orchestrator (mediation service)
- Embedded in the middleware/network layer
- Embedded in the source/target smart-node

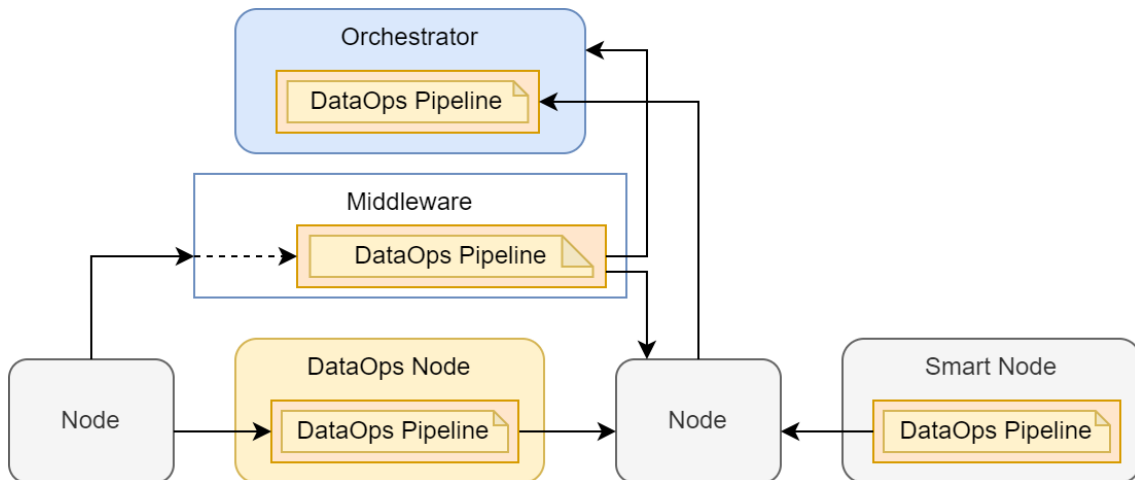


Figure 3-2: Deployment options for the DataOps Toolbox

Each option is associated with different trade-offs and depends on the specificities of the considered scenario. In the first two cases, the orchestrator could possibly enable the deployment of a mediation node or the execution of a mediation service by considering the requirements of the Recipe to be executed and the nodes composing the swarm. The last two cases assume a predefined configuration for either the middleware or specific nodes to enable their cooperation within a Recipe executed by the orchestrator.

A3.6 provides a set of templates to deploy a pipeline in different deployment environments. The right template can be selected considering resource availability and deployment strategies and used to execute the specified pipeline. Generally, the same pipeline can be deployed in different deployment environments without requiring specific modifications.

D3.1 reports the analysis of the state of the art and the main design decisions made for each DataOps artefact. In this section, we briefly summarise the relevant content from D3.1 to ensure that the document is self-contained and describe the final design of A3.5, A3.6 and A3.7.

<sup>10</sup> As in D3.1, we define a smart-node as a node that can be modified to execute SmartEdge components.

## 3.1.1 Final Design of the DataOps Pipeline Components (A3.5)

The DataOps component is designed to provide the necessary building blocks to configure heterogeneous DataOps pipelines for data operations in SmartEdge.

From the state-of-the-art analysis reported in D3.1, the declarative semantic conversion process represented in Figure 3-3 is selected as the approach to enable data interoperability and data integration. Declarative mapping rules are leveraged to configure to/from transformations from a reference conceptual model relying on Semantic Web technologies for syntactic and semantic interoperability. This any-to-one approach reduces the number of mappings to be defined in case of point-to-point integrations and improves scalability when enabling interoperability between numerous data models/standards [Vetere05]. The mapping rules are decoupled from the component responsible for their execution to improve their maintainability and reusability.

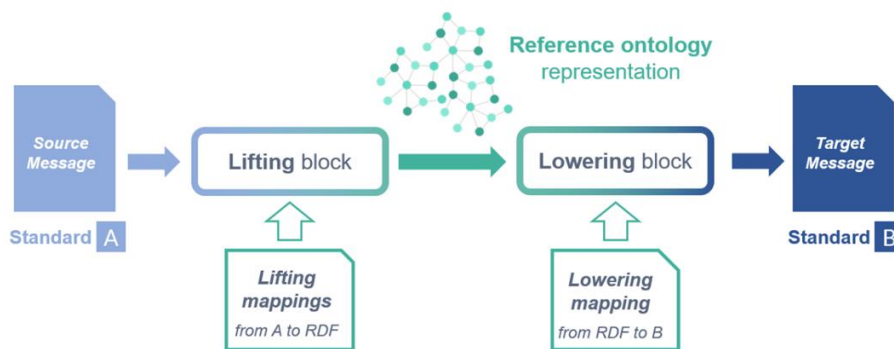


Figure 3-3: Declarative semantic conversion process for interoperability

To implement such an approach, we designed the DataOps pipelines as represented in Figure 3-4. The main types of building blocks are the *Node Data Connector*, which are blocks responsible for enabling data exchanges with different types of interfaces/protocols, and the *Mapping Processor*, which are blocks capable of executing declarative mapping rules for data and schema transformations. Additional blocks may be integrated within a pipeline to perform additional manipulations or to implement *Enterprise Integration Patterns* [Hohpe04].

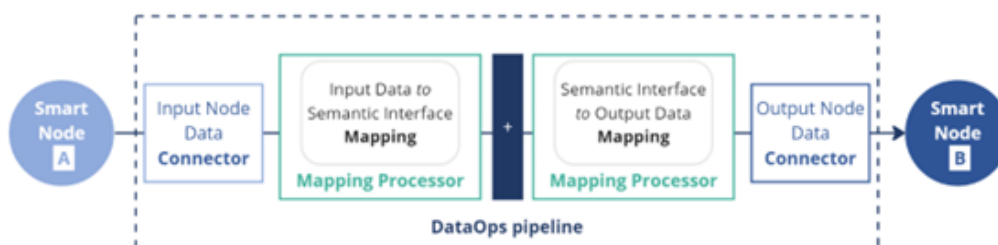


Figure 3-4: DataOps Pipeline

We selected the Apache Camel<sup>11</sup> framework as a solution enabling enterprise integrations through the configuration of building blocks within an executable pipeline.

<sup>11</sup> <https://camel.apache.org/>

Moreover, Apache Camel offers many production-ready components that can be easily integrated within a pipeline as *Node Data Connector* for common protocols and interfaces. Finally, Camel can be easily extended to define custom-defined components to be integrated within a pipeline.

As a reusable component for Apache Camel, the *Chimera* framework<sup>12</sup> introduces operations for constructing, manipulating, and exploiting knowledge graphs within a pipeline. It provides support for operations on an RDF graph (either local or remote), execution of declarative mapping rules adopting the RDF Mapping Language (RML)<sup>13</sup> specification and the execution of template-based mapping rules leveraging the Apache Velocity<sup>14</sup> Template Engine. The Apache Camel components and the one introduced by the *Chimera* framework are defined as A3.5, i.e., the DataOps components for implementing the required pipelines in SmartEdge.

The design and implementation activities for the first release focused on:

- the redesign of the template-based mapping rule component (Mapping Template) to enable generic knowledge conversion via declarative mapping rules while improving performance and scalability;
- a complete refactor of the Chimera framework to increase the solution's overall TRL, improve its reusability and facilitate the configurability and composability of pipelines.

For the second release, we will focus on improving the maturity of the DataOps components with respect to the new functionalities (e.g., improve the integration of the mapping-template within the respective Camel component), and we will address additional requirements emerging from SmartEdge use cases in the piloting activities within WP6 (e.g., in terms of building blocks required within a pipeline). Moreover, considering the evaluation performed for the first release (discussed in Section 3.2.1.3), we will investigate approaches to monitor DataOps pipelines and further improve performance and scalability of the pipeline executions.

#### 3.1.1.1 Mapping Template Component

Starting from analysing mapping languages and mapping processors for declarative RDF Knowledge Graph construction, reported in D3.1, we designed a workflow for generic knowledge conversion [Scrocca2024]. This workflow aims to build on the work done for the declarative materialisation of RDF triples for the definition and execution of declarative mapping rules towards a generic output. Indeed, nodes involved within a swarm are usually not able to directly process an interoperable representation of data in RDF .

The final version of the workflow, represented in Figure 3-5, depicts a general mapping scenario in which data from a source, formatted according to a specified input format

---

<sup>12</sup> <https://github.com/cefriel/chimera>

<sup>13</sup> <https://rml.io/>

<sup>14</sup> <https://velocity.apache.org/>

and model, needs to be converted into a target format and model before being stored in a designated data sink. This mapping scenario may include the integration of extra data sources to produce the output and the application of data and schema transformations throughout the process. The workflow outlines the foundational elements for a generic *declarative mapping language* along with the relevant components necessary for a *mapping process* executing the mappings.

The parsing and extraction process from heterogeneous data sources is generalised considering the concept of *data frame*, i.e. a two-dimensional data structure made of rows and columns. The overall workflow can be summarized as follows:

- the input data sources are accessed according to a specific configuration (e.g., protocol/interaction mechanism);
- the retrieved data are extracted and used to initialize a set of *data frame* structures;
- data transformations or combination operations (e.g., join) can be applied to the *data frame* structures;
- a set of declarative mapping rules is executed to map the content of the *data frame* structures to the target schema;
- the output data are written to data sinks according to specific configurations.

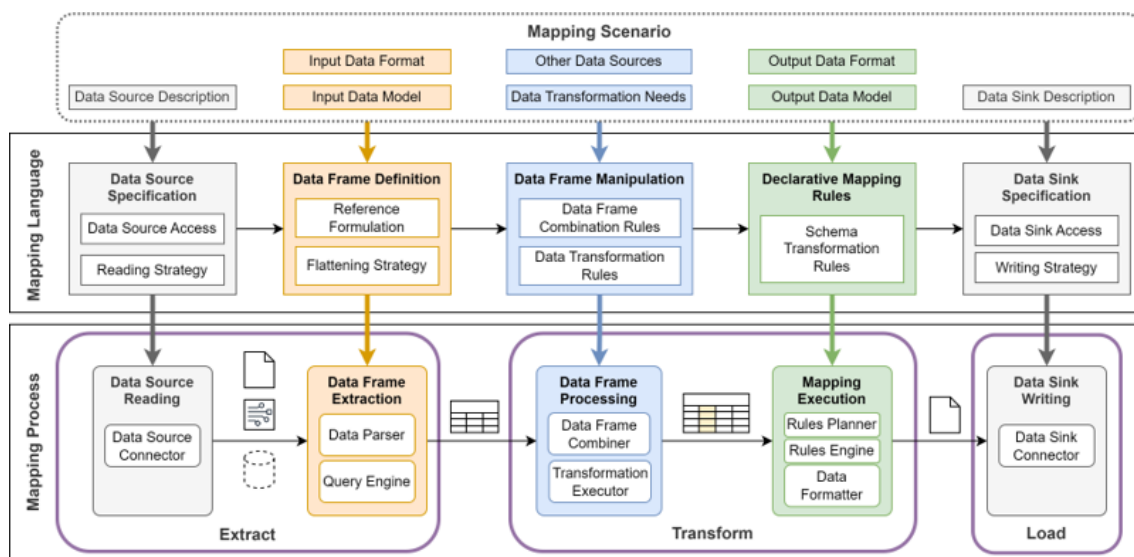


Figure 3-5: Final workflow for generic knowledge conversion enabled by a DataOps pipeline

Based on this workflow, we redesigned and refactored the mapping-template<sup>15</sup> library that supports data and schema transformations by leveraging the Apache Velocity template engine. In particular, we defined a Mapping Template Language<sup>16</sup> (MTL) on top of the Velocity Template Language (VTL) to define mapping rules for generic knowledge conversion according to the defined workflow building blocks. The overall goal is to leverage the competitive performances provided by the adoption of a template engine

<sup>15</sup> <https://github.com/cefriel/mapping-template>

<sup>16</sup> [https://github.com/cefriel/mapping-template/wiki/Mapping-Template-Language-\(MTL\)](https://github.com/cefriel/mapping-template/wiki/Mapping-Template-Language-(MTL))

while providing users with a declarative mechanism to specify mapping rules. The details about the developments performed are reported in Section 3.2.1.

The definition of mapping rules as templates trades some aspects of a fully declarative approach. However, it covers three important requirements emerging in SmartEdge:

- provide flexibility in the generated output since many nodes can not process RDF;
- handle complex transformation scenarios (e.g., requiring functions with side effects);
- facilitate the definition of mapping rules for users unfamiliar with RDF.

Nevertheless, to support users willing to adopt a fully declarative approach and not interested in the mentioned features, we implemented RML compliance for this component, as discussed in Section 3.2.1. This implementation supports the claim that the proposed MTL could generalize the declarative mapping rule specification for KG construction.

As an additional advantage of the new implementation, the decomposition in blocks of mapping rule definitions and executions enables the explicit specification of optimization strategies to improve the performance and scalability of mapping rules considering a target scenario. For example:

- the number of accesses to the input data sources and the information extracted can be optimised by defining the minimum number of data frame required for the mapping rules to be applied;
- the join execution can be optimized by applying appropriate combination rules directly to the data frames.

#### 3.1.1.2 Chimera

To facilitate integration within the Apache Camel ecosystem, Chimera was redesigned to define distinct Camel components, each one compliant with Apache Camel's guidelines<sup>17</sup>. Camel components are collections of related functionalities focused on specific tasks. For instance, the *FileComponent* provides capabilities for file operations like deleting, creating, and copying files. Components expose these functionalities through configurable options known as *endpoint parameters*. Each Endpoint is identified by a URI, which consists of the component's unique identifier followed by its configuration parameters. For example, the URI `file://inputdir/?delete=true` specifies the use of the File component, where *inputdir* is the target directory, and the *delete=true* parameter instructs Camel to delete files in this directory after processing.

Components are then linked sequentially within *Routes*, where each component is executed in the specified order. An example Camel route is shown in Figure 3-6, where data is read from a directory using the *FTP* component and is then sent to the example queue using the *ActiveMQ* component.

---

<sup>17</sup> <https://camel.apache.org/manual/component.html>



```
from("ftp:myserver/folder")
.to("activemq:queue:example");
```

Figure 3-6: Example of a Camel route written using the Java domain specific language

We refactored the Chimera codebase defining three main Camel components:

- The *Chimera graph component* is designed to perform operations on RDF graphs, including reading and serializing RDF data in multiple formats (such as Turtle, RDF/XML, and N-Triples). In Chimera, RDF graphs act as abstractions for diverse RDF data sources, each optimized for specific use cases:
  - *MemoryRDFGraph*, A transient RDF graph stored only in memory.
  - *NativeRDFGraph*, An RDF graph persisted on disk, backed by a specific filesystem.
  - *HTTPRDFGraph*, Enables Chimera to connect to an RDF graph hosted on a remote triplestore.
  - *InferenceRDFGraph*, An RDF graph that incorporates inference capabilities for reasoning tasks.
  - *SPARQLEndpointGraph*, An RDF graph accessible through a SPARQL endpoint.

Over these RDF graphs, the *Chimera graph component* defines a series of operations.

- *GraphGet*, initializes one of the RDF graph types to be used in a Camel route
  - *GraphAdd*, adds RDF triples to an RDF graph
  - *GraphInference*, performs inference to generate new data based on existing graph data
  - *GraphSparqlSelect*, runs a SPARQL select query on an RDF graph
  - *GraphConstruct*, generates triples to be added to an RDF graph via a SPARQL construct query
  - *GraphSparqlAsk*, runs a SPARQL ask query on an RDF graph
  - *GraphShacl*, performs validation of an RDF graph via SHACL<sup>18</sup> shapes
  - *GraphDetach*, severs the connection between the RDF graph and an RDF data source or clears part of, or all triples from a graph
  - *GraphDump*, writes an RDF graph to a file in a specific RDF format
- The *Chimera mapping-template component* is a Camel component wrapper around the mapping-template library<sup>19</sup> discussed in Section 3.1.1.1, making it accessible for data conversion and mapping operations within Camel. In a Camel route, it is used to convert an incoming input according to a set of declarative mapping rules, defined using the MTL, that should be provided as part of the component configuration.

<sup>18</sup> <https://www.w3.org/TR/shacl/>

<sup>19</sup> <https://github.com/cefriel/mapping-template>

- The *Chimera RML component* serves a similar purpose as the mapping-template component but allows the usage of mappings written in RML by wrapping a forked version of the `rmlmapper`<sup>20</sup> library.

As summarized in Figure 3-7, a DataOps pipeline can be configured by integrating and configuring within a Camel route: (i) existing Camel components, (ii) Chimera components, (iii) custom components defined for specific integration scenarios. Additional details on the design of the A3.5 are reported in D3.1. We discuss the main modifications introduced in the implementation work for A3.5 within Section 3.2.1.2, and we discuss examples of pipelines developed for SmartEdge in Section 3.3.

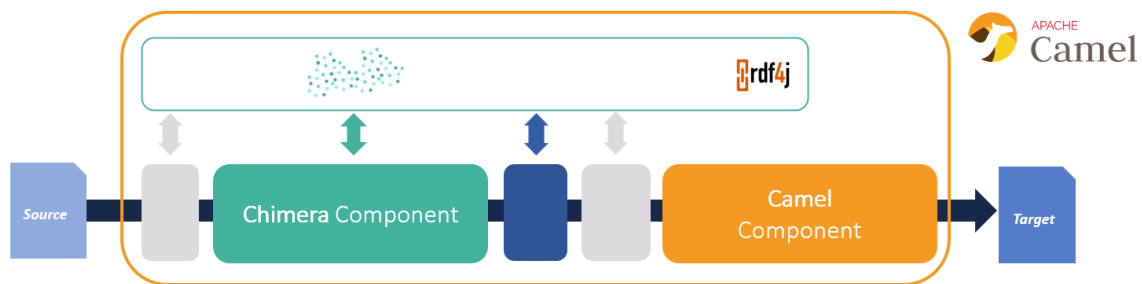


Figure 3-7: Overview of a DataOps pipeline integrating different components

### 3.1.2 Final Design of the DataOps Deployment Templates (A3.6)

Given a DataOps pipeline, a deployment strategy should be selected considering the constraints for its deployment. Different options may be evaluated, depending on the nodes involved in the mediated data exchange, such as the machine hosting a certain node or the possibility of modifying the node's code to be executed.

For this reason, we investigated the alternative options for executing Apache Camel integrations to enable flexibility in deploying DataOps pipelines both on Edge devices and in the Cloud. The implementation of A3.6 focused on the definition of reusable templates to facilitate the deployment of pipelines in different environments.

We identified the following deployment alternatives for a DataOps pipeline:

1. **Library:** A DataOps pipeline can be integrated within a Java Project by importing Apache Camel and Chimera as dependencies and thus integrating the execution of a DataOps pipeline within the already existing source code of a swarm node. In this case, the deployment depends on the parent project integrating the DataOps pipeline.
2. **JAR Files:** JAR files are self-contained executables (Java Archive) that encapsulate all the necessary components for running a DataOps pipeline. This makes them

<sup>20</sup> <https://github.com/cefriel/rmlmapper-cefriel>

highly portable and suitable for a wide range of devices that can run a Java runtime. JAR files can be deployed on various platforms, including desktops, servers, and cloud environments. They offer a high degree of flexibility and can be integrated with different systems and frameworks. Different runtime can be selected to build and package JAR files for executions:

- *Camel Core*: basic runtime for Camel applications. Can be used in applications where lightweight pipelines should be defined without the need for a larger framework supporting many dependencies.
- *Spring for Camel*: Camel's Spring integration allows Camel routes to be configured within a Spring-based application, thus leveraging Spring's dependency injection, lifecycle management, and configuration. Moreover, Camel offers Spring Boot compliant components to facilitate the automatic integration of pipelines within Spring applications.
- *Quarkus*: a Java framework designed to start up quickly applications by implementing specific optimisations and configurations at build time. Quarkus is optimized for low memory usage, making it suitable for resource-constrained environments, serverless environments and microservices architectures. Java libraries should be adapted as Quarkus *extensions*<sup>21</sup> to enable their usage within Quarkus projects.

JAR Files for a DataOps pipeline can be deployed within a node equipped with a JVM or as a dedicated node implementing a mediation service.

3. **Containerization**: To further enhance portability and scalability, JAR files can be packaged as OCI (Open Container Initiative) containers using different Java Virtual Machines (JVMs), such as OpenJDK or Oracle JDK. This allows for efficient deployment and management in containerized environments like Docker and Kubernetes. A DataOps pipeline may be packaged as a standalone container or integrated within the container executing code for a certain node.
4. **Native Executable (GraalVM)**: GraalVM is a high-performance Java runtime that offers ahead-of-time (AOT) compilation. This enables the compilation of DataOps pipelines as native binary executables before deployment, eliminating the need for a JVM at runtime. Native executables generated by GraalVM are typically smaller and have faster startup times than JAR files. This makes them ideal for resource-constrained devices and applications that require quick response times. Native executables can have a smaller memory footprint than JAR files, especially when used on embedded or IoT devices. This can improve performance and reduce resource consumption. GraalVM can generate native binaries for specific platforms, such as Linux, Windows, and macOS. This ensures optimal performance and compatibility for the target environment. Native executables can be generated for the different frameworks discussed above: Camel Core, Spring for Camel, Quarkus. Native executables for a DataOps pipeline within a node or as a dedicated node implementing a mediation service.

---

<sup>21</sup> <https://quarkus.io/guides/writing-extensions>

5. **Kubernetes:** If a Kubernetes environment is available either on a single device or as a deployment environment for multiple nodes composing the swarm, DataOps pipeline(s) can be deployed using different approaches:
- *Sidecar container*<sup>22</sup>: a Docker container running a DataOps pipeline can be executed within a Pod deployed for a node as a sidecar container;
  - *Service*: A Docker container running a DataOps pipeline can be executed as a dedicated Kubernetes Pod and exposed as Kubernetes services. This enables scaling the number of replicas and the automatic load balancing of requests.
  - *Apache Camel K*: a subproject of Apache Camel explicitly designed for running pipelines in Kubernetes-based environments. It can be used to simplify the deployment of DataOps pipelines for use cases involving cloud-native and serverless architectures. One of the core innovations introduced by Camel K is the concept of Kamelets (Kamel route snippets), which are reusable integration templates that provide an abstraction to encapsulate pipelines for specific integration tasks (like accessing data from a certain node in a specific format).
  - *K-Native*: a Kubernetes-based platform to deploy, manage, and scale applications. It can be employed to manage a Camel K deployment of DataOps pipelines in a serverless manner. In particular, K-Native enable automatic scaling of the pipelines<sup>23</sup>, allocating more computing resources to pipelines with increasing workloads and allowing scale-to-zero to pipelines not receiving requests.

Table 3-1 updates the analysis reported in D3.1 on the pros and cons of the different deployment alternatives identified.

Table 3-1: Analysis of PROs and CONs for different deployment alternatives

Deployment	Pros	Cons
<b>Library</b>	<ul style="list-style-type: none"> <li>• Integrated execution (e.g., exchange over the network is not required) can lead to better performance.</li> </ul>	<ul style="list-style-type: none"> <li>• Requires modification to the source code. It is only possible if the codebase is in Java.</li> </ul>
<b>JAR file</b>	<ul style="list-style-type: none"> <li>• Easy to build.</li> <li>• Easy to deploy to a device, everything necessary is contained in the JAR.</li> <li>• [Quarkus] Really low memory footprint and fast start-up.</li> </ul>	<ul style="list-style-type: none"> <li>• Requires the device to run Java.</li> <li>• [Quarkus] Not all the Java libraries support Quarkus.</li> </ul>
<b>Containerization</b>	<ul style="list-style-type: none"> <li>• JVM and needed dependencies are packaged as a single artefact.</li> </ul>	<ul style="list-style-type: none"> <li>• Requires a container runtime to execute it.</li> <li>• Image selected for the container may introduce overhead in resource usage w.r.t. direct execution.</li> </ul>

<sup>22</sup> <https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/>

<sup>23</sup> <https://knative.dev/docs/serving/autoscaling/>

<p><b>Native Executable</b></p>	<ul style="list-style-type: none"> <li>Does not require the device to run Java.</li> <li>Faster start-up and execution times than a JAR file.</li> </ul>	<ul style="list-style-type: none"> <li>Creating a native binary demands more CPU power and RAM compared to building a JAR file.</li> </ul>
<p><b>Kubernetes</b></p>	<ul style="list-style-type: none"> <li>Replication and auto-scaling are supported by Services in Kubernetes.</li> <li>[Kamelets] Allow an even easier re-use of routes inside of a larger integration.</li> <li>[K-Native] Serverless approach enables scale-to-zero to save resources, and scalability via replication for high traffic loads.</li> </ul>	<ul style="list-style-type: none"> <li>It only makes sense in the context of a Kubernetes deployment.</li> <li>[Kamelets] Limitations on the structure of kamelets for reuse.</li> <li>[K-Native] Management of dependencies should be handled to enable execution of pipelines including custom components.</li> </ul>

Figure 3-8 provides an overview of the deployment templates (A3.6) identified for a DataOps pipeline.

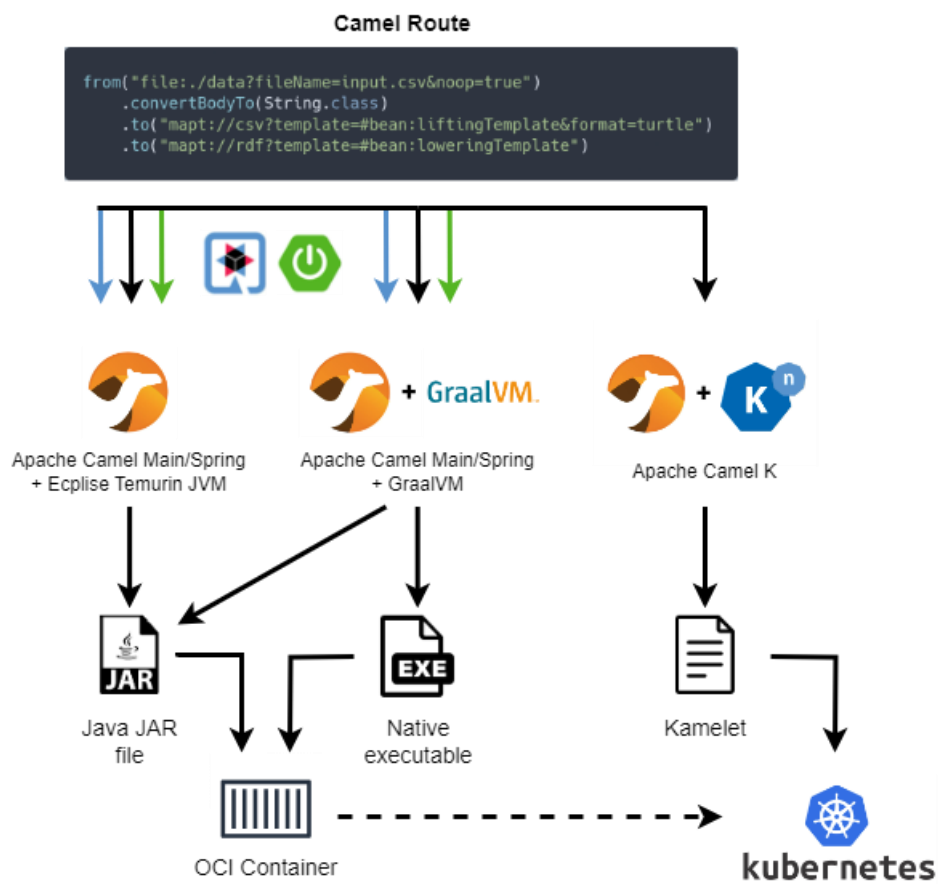


Figure 3-8: DataOps Deployment Templates

The majority of the deployment templates are made available for the first release as described in Section 3.2.2. For the second release, we will further extend the list of

available templates by focusing on Camel-K<sup>24</sup> and Knative<sup>25</sup> for Cloud deployment environments and on Quarkus for Edge environments.

### 3.1.3 Final Design of Low-code DataOps Configuration (A3.7)

The objective of artefact A3.7 is to support easy and low-code implementation of DataOps pipelines for enabling mediated data exchanges between nodes in the swarm. The low-code approach adopted for the definition of DataOps pipelines simplifies application development by emphasizing configuration over manual coding. The overall objective is to enable a declarative configuration of components so that users can reduce the need to implement custom solutions.

For the DataOps tool, the choice of adopting the Apache Camel framework enables the definition of data integration pipelines using the abstraction of *Routes* as a composition of building blocks. This abstraction empowers a no-code approach to data integration, as it exposes all available functionalities of Camel components through well-documented URI parameters, which users can configure when creating a route. This approach also means that modifying the data integration pipeline doesn't necessitate rebuilding the entire software artefact that executes Camel routes; it only requires changes to the file where the route is declared. To enable this configuration over code approach, routes can be defined using several domain-specific languages (DSL), with the most prominent options being XML, Spring XML, and YAML.

A declarative approach also enables the definition of mapping rules, as discussed in 3.1.1, that can then be provided as input to the relevant components within a *Route*.

To streamline the definition of a *Route*, we investigated Apache Camel *Karavan*<sup>26</sup> that provides a graphical user interface as a plugin for Visual Studio Code to configure a *Route* without writing code. This graphical approach significantly eases the process of route definition, as it avoids syntax and logical errors that may happen when manually writing a route in a text file. *Karavan* supports all the components officially included in the Apache Camel Framework, that can be reused within a pipeline defined using the tool. A pipeline configured in *Karavan* can be automatically exported as a Java project for execution.

Additionally, *Karavan* supports predefined and custom Kamelets<sup>27</sup>, which are reusable route templates designed to simplify route construction. Kamelets let users define parameterized routes using the YAML DSL, streamlining the process by hiding unnecessary details. A Kamelet can either be a *Source*, that produces data and can then send it to another component that is passed in as a parameter or as a *Sink*, that receives data from a component passed in as a parameter another fixed component defined in the Kamelet. An example *Source* Kamelet can be seen in Figure 3-9. This example Kamelet demonstrates how this approach can be used within DataOps pipelines to reuse integrations to access or forward data to a certain node. The Kamelet shown in the snippet, sends an HTTP request to check the status of a swarm node at regular intervals

---

<sup>24</sup> <https://camel.apache.org/camel-k/2.2.x/index.html>

<sup>25</sup> <https://knative.dev/docs/>

<sup>26</sup> <https://github.com/apache/camel-karavan>

<sup>27</sup> <https://camel.apache.org/camel-k/2.5.x/kamelets/kamelets.html>

and processes the response. The user reusing this Kamelet will only need to configure the `kamelet:sink` to define the remaining part of the pipelines. If available, another Kamelet may be reused also for the sink.

```

apiVersion: camel.apache.org/v1
kind: Kamelet
metadata:
  name: smart-edge-source
  annotations:
    camel.apache.org/kamelet.support.level: "Preview"
    camel.apache.org/catalog.version: "main-SNAPSHOT"
    camel.apache.org/provider: "Apache Software Foundation"
    camel.apache.org/kamelet.group: "SmartEdge"
  labels:
    camel.apache.org/kamelet.type: "source"
spec:
  definition:
    title: "SmartEdge Source"
    description: |-
      Gets periodically SmartEdge swarm status updates
    type: object
    properties:
      period:
        title: Period
        description: The interval (msec) to wait before getting the next swarm update
        type: integer
        default: 10000
  types:
    out:
      mediaType: text/plain
  dependencies:
    - "camel:kamelet"
    - "camel:timer"
    - "camel:http"
    - "camel:jsonpath"
  template:
    from:
      uri: "timer:smartedge"
      parameters:
        period: "{{period}}"
      steps:
        - to: "https://api.smartedge.io/status"
        - setBody:
            jsonpath: "$.value"
        - to: "kamelet:sink"

```

Figure 3-9: Example source Kamelet that is used to read the status of the swarm and then forward it to a target node

To integrate this tool into the DataOps toolbox and enable the use of Chimera components in a no-code manner, we worked on improving Chimera's compatibility with the Apache Camel framework. This has been done by introducing the concept of *ChimeraResources*, a general approach to handling different resource types in the various Camel DSLs and changes to the parameters defined by the various Chimera components. The technical details of these changes are described in Section 3.2.1.2.2 and Section 3.2.3.

Some usability challenges remain due to the unofficial status of Chimera components. Indeed, a default installation of the Karavan plugin lacks the necessary metadata for enabling the definition and automatic export of a pipeline using DataOps components. Section 3.2.3 discusses how we addressed these aspects for the first release. However, this still requires manual steps from the user, which we seek to eliminate. We will investigate how to improve these aspects in the second release by focusing on the possibility of defining and reusing custom Kamelets in Karavan. Finally, we plan to

develop a set of Kamelets specific to the project's needs and make them available through the Karavan catalog, making their reuse easier.

We also performed an examination of Kaoto<sup>28</sup>, a tool comparable to Camel Karavan and developed by RedHat, for potential adoption but we discarded it for the moment since it cannot support custom Camel components as the ones defined for the DataOps pipelines.

## 3.2 FIRST IMPLEMENTATION

This section discusses the artefacts implemented for the first release of the DataOps Toolbox and how they support their final design. Open-source components adopted for implementing the SmartEdge artefacts are referenced as Git submodules within the public SmartEdge repository on Gitlab<sup>29</sup>.

Specific developments for the SmartEdge use cases (e.g., the pipelines discussed in Section 3.3) are kept within the SmartEdge private repository<sup>30</sup>.

### 3.2.1 First Implementation of the DataOps Pipeline Components (A3.5)

The first implementation of the DataOps Pipeline Components is discussed considering developments for the mapping-template component and the overall Chimera framework. Finally, we report an initial performance and scalability evaluation of the mapping-template against other RML-based processors.

#### 3.2.1.1 Mapping Template Component

Considering the *mapping processors* made available as DataOps components for A3.5, we focused on improvements for the *mapping-template* library and the corresponding component in Chimera (*Mapping Template Component*). We developed this approach as an alternative to the well-known RML mapping processors, taking into account requirements from SmartEdge: (i) address specific mapping rules that are difficult to express with the fully-declarative syntax and targeting a generic output, (ii) facilitate the definition of mapping rules by users that are not familiar with RDF, (iii) address performance and scalability transformations for runtime message conversion and considering resource-constrained devices.

As a result of the workflow for a generic mapping process discussed in the design of A3.5, we reviewed the definition of mapping rules in the *mapping-template*<sup>31</sup> library and we defined a Mapping Template Language<sup>32</sup> (MTL) to provide specification of the intended usage. We also developed examples to compare MTL with RML-based mapping languages and help users adopt the tool<sup>33</sup>. These examples were also used to perform a

---

<sup>28</sup> <https://kaoto.io/>

<sup>29</sup> <https://gitlab.com/smartedge-project-eu/smartedge-public/-/tree/main/dataops>

<sup>30</sup> <https://gitlab.com/smartedge-project-eu/SMARTEDGE>

<sup>31</sup> <https://github.com/cefriel/mapping-template>

<sup>32</sup> [https://github.com/cefriel/mapping-template/wiki/Mapping-Template-Language-\(MTL\)](https://github.com/cefriel/mapping-template/wiki/Mapping-Template-Language-(MTL))

<sup>33</sup> <https://github.com/cefriel/mapping-template/tree/main/examples>



qualitative evaluation of MTL's expressiveness against the requirements for mapping languages for knowledge graph construction [Scrocca24].

Finally, we implemented additional features to enable the tool's use in additional scenarios and we added support for the direct execution of RML mappings via the *mapping-template*.

#### 3.2.1.1.1 Mapping Template Language (MTL)

MTL is the defined language to declaratively specify data and schema transformation for a mediated data exchange within a DataOps pipeline.

The core components enabling the Mapping Template Language (MTL) are *Readers* and *Data Frames*. Readers are format-specific objects used to load input data for mapping, while Data Frames provide a flat, tabular view of that data extracted by format-specific query languages known as *reference formulations*. Each type of data format has a dedicated *Reader* (e.g., CSVReader for CSV files, JsonReader for JSON, etc.). Once input data is loaded by a Reader, it is transformed into a Data Frame using a reference formulation, which extracts data into a tabular structure. This implementation supports the second step of the generic workflow for knowledge conversion, i.e., the *Data Frame Extraction*. For example, hierarchical JSON data is converted into a Data Frame using JsonPath expressions; similarly, XML uses XQuery, RDF uses SPARQL to achieve this tabular format.

Once the data is available in a Data Frame, it can be manipulated (*Data Frame Manipulation* in the workflow) by combining it with other data frames or applying data transformations. For enabling this, we implemented in the *mapping-template* a set of convenience functions accessible via the *\$functions* variable in a MTL template. These include various string operations, such as replacement and hashing, but also join operations between Data Frames. To support a wide range of mapping applications, MTL allows users to extend functionality by adding custom Java functions, which can be loaded and used within mappings.

To enable the *Mapping Execution* step of the workflow, the MTL follows a template approach by allowing the user to express the structure that the output data must follow and how the data from Data Frame(s) should be bound to it. This is shown in the mapping in Figure 3-10 which reads XML data and outputs the data in RDF Turtle format shown in Figure 3-11.

The mapping in Figure 3-10 demonstrates the key features of the MTL language and how it integrates the template language of the Apache Velocity library (VTL). The initial lines define RDF prefixes for the output, which are written exactly as specified, since they are neither directives nor variables. The *#set* directive assigns a value to the *\$query* variable, which contains an XQuery query. In the MTL language, directives are denoted by a leading *#* while variables are denoted by a leading *\$*. Since the input data is in XML format, XQuery serves as the reference formulation, and the *\$reader* is an XMLReader. Together, the Reader and query extract data to create a Data Frame, which is then used

to generate the RDF Turtle output. In the final part of the mapping, a loop iterates over the Data Frame, highlighting MTL's template-based approach. Fixed elements, such as the *rdf:type* literal, are written directly to the output, while expressions like *\$stop.busId* are evaluated based on the current iteration.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix transit: <http://vocab.org/transit/terms/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix ex: <http://trans.example.com/>.

#set( $query = '
  for $stop in /transport/bus/route//stop
  return map {
    "stopId": $stop@id,
    "stopName": $stop/text(),
    "busId": $stop/ancestor::bus/@id
  }')
#set( $data = $reader.getDataframe($query))

#foreach($stop in $data)
ex:$stop.busId rdf:type transit:stop ;
transit:stop "$stop.stopId"^^xsd:int ;
rdfs:label "$stop.stopName" .
#end
```

Figure 3-10: MTL mapping to convert XML data to RDF Turtle

```
<transport>
  <bus id="25">
    <route>
      <stop id="645">International Airport</stop>
      <stop id="651">Conference center</stop>
    </route>
  </bus>
</transport>
```

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix transit: <http://vocab.org/transit/terms/>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix ex: <http://trans.example.com/>.

ex:25 rdf:type transit:stop ;
transit:stop "645"^^xsd:int ;
rdfs:label "International Airport" .
ex:25 rdf:type transit:stop ;
transit:stop "651"^^xsd:int ;
rdfs:label "Conference center" .
```

Figure 3-11: XML input data and corresponding RDF Turtle representation obtained by applying the mapping template

The workflow's *Data Source Reading* and *Data Sink Writing* steps are only partially supported through the MTL for execution via CLI. We chose to decouple these steps to reduce the necessity of incorporating multiple external libraries into the *mapping-template* library. This decision was made with the expectation that the tool could be seamlessly integrated with existing Extract-Transform-Load (ETL) tools, offering various production-ready data connectors right from the start. In this direction, the integration with Chimera guarantees support for the declarative definition of DataOps pipelines leveraging Camel components and MTL to implement the full workflow. Examples are discussed in Section 3.2.3.

### 3.2.1.1.2 Additional features implemented

To support the new MTL specification, we modified the library accordingly. Currently, *Reader* implementations are made available to extract data frames for heterogeneous input data sources: CSV, JSON, XML, RDF, SQL databases (specifically PostgreSQL and MySQL). Moreover, we defined additional functions exposed via MTL to combine and manipulate data frames, e.g., *join* operations.

As an additional feature to facilitate the implementation of complex integration requirements within a DataOps pipeline, users can now refer to multiple input sources via MTL that are then accessed by providing multiple Readers to the *mapping-template* library. This feature eliminates the need to define individual mappings for each data source. It is useful for complex mappings that depend on multiple, potentially dynamic, data sources, e.g., data coming from different nodes within a swarm. While it was previously possible, doing so required the user to know both the type and location of each data source in advance when writing the mapping file, limiting flexibility at design time. This constraint was manageable for static or batch processes, where input data remains constant and is converted only once. However, it hampers template reusability, as users must manually update templates to accommodate new or different data sources. An example of this can be seen in the MTL mapping snippets in Figure 3-12. In the top part of the figure, we can see that the user must specify both the type of data that needs to be loaded, in this case, CSV, and the location of the data file. This means that should the file location change, the mapping should also be changed, limiting reusability as the mapping ideally should be concerned with just the data conversion aspect. In the bottom part of the picture the new functionality is shown, where the Readers used in the mapping are supplied externally and are not defined in the mapping itself.

```
#set ($frequenciesReader = $functions.getCSVReaderFromFile("/data/shared/FREQUENCIES.csv"))
#set ($calendarDatesReader = $functions.getCSVReaderFromFile("/data/shared/CALENDAR.csv"))
```

```
#set ($df1 = $reader1.getDataframe())
#set ($df2 = $reader2.getDataframe())
```

Figure 3-12: On top, an example of defining multiple readers statically within a mapping. On the bottom, the new possibility of providing multiple readers dynamically from outside the mapping.

This new capability is designed to work in the context of a DataOps pipeline by leveraging the Chimera Mapping Template component. In this context, conversions are more dynamic, and often, externally supplied data inputs need to be adapted. By contrast, the conversion remains a one-time batch process when using the *mapping-template* component as a standalone application.

#### 3.2.1.1.3 Support for RML mappings

Building on the work done on the generic workflow for knowledge conversion and on the previously introduced enhancements, we implemented the capability to execute RML mappings in the mapping-template library automatically.

This functionality enables users to leverage existing RML mappings by translating them into MTL syntax, providing the capability to:

- leverage the *mapping-template* as an RML mapping processor, or

- adapt the generated MTL file for finer control over the output or to introduce optimisations.

Notably, the translation from RML to MTL is implemented via MTL, showcasing the alignment of the *mapping-template* solution to the approach for declarative knowledge graph construction adopted by RML. Despite being a not trivial effort, implementing this feature required much less work with respect to the definition of an RML mapping processor from scratch.

The MTL mapping generated from this process produces an equivalent output to the one generated by original RML mapping. The template parses the data from the data sources specified in the RML mappings and applies the same mapping rules.

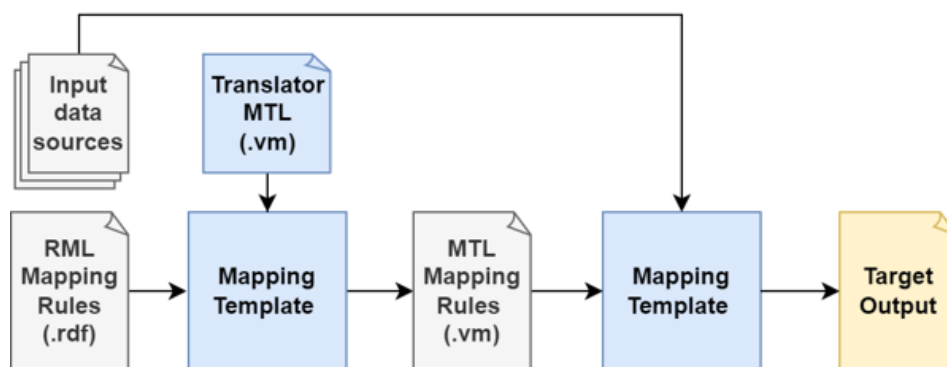


Figure 3-13: MTL to RML transformation process

```

@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rml: <http://w3id.org/rml/> .

<http://example.com/base/TriplesMap> a
  rml:TriplesMap;
  rml:logicalSource [ a rml:LogicalSource;
                    rml:referenceFormulation rml:CSV;
                    rml:source [ a rml:RelativePathSource;
                                rml:root rml:MappingDirectory;
                                rml:path "student.csv"
                              ]
                  ];
  rml:predicateObjectMap [
    rml:objectMap [
      rml:reference "Name"
    ];
    rml:predicate foaf:name
  ];
  rml:subjectMap [
    rml:template "http://example.com/{Name}"
  ] .
  
```

```

#set($foo = $functions.setBaseIRI("http://example.com/base/"))
#set($reader_zbffbifagdd = $functions.getCSVReaderFromFile("/data/shared/student.csv"))
#set($foo = $reader_zbffbifagdd.setHashVariable(true))
#set($foo = $reader_zbffbifagdd.setOnlyDistinct(true))
#set($columns = "Name")
#set($dataframe_zbffbifagdd = $reader_zbffbifagdd.getDataframe($columns))
#foreach($i in $dataframe_zbffbifagdd)
  #set($refs = [{i.cecadjf}])
  #if($functions.checkStrings($refs))
    <$functions.resolveIRI("http://example.com/{i.cecadjf}")> <http://xmlns.com/foaf/0.1/name> "{i.cecadjf}" .
  #end
#end
  
```

Figure 3-14: An example RML mapping (above) and the corresponding automatically generated MTL mapping (below).

This process is shown in Figure 3-13, with a comparison of the original RML and the resulting MTL shown in Figure 3-14. As can be seen, the automatically generated MTL mapping is meant to be used by the library and not the user. As such it is not human readable, and the automatically generated variable names are randomly generated and assigned to avoid naming collisions. The user is, however, free to manually edit the resulting MTL mapping and introduce possible optimizations which rely on external knowledge not present in the original RML mapping.

The mapping template is currently compliant with the *rml-core* specification (<https://w3id.org/rml/portal>) and the execution against all the *rml-core* test cases is reported and documented online<sup>34</sup>. The RML mapping can be passed with a specific option for usage via CLI and a test case is made available to exemplify the usage as a library.

For the second release, we will investigate the possibility of improving the compiler from RML rules to MTL by evaluating the support for additional RML modules<sup>35</sup> (e.g., RML-CC and RML-star) and the automatic definition of DataOps pipelines for accessing data sources and targets defined via RML-IO.

### 3.2.1.2 Chimera

The Chimera framework<sup>36</sup> has seen numerous improvements to support the first release of DataOps pipeline components (A3.5). In particular, aiming at increasing the Technology Readiness Level (TRL) and enhancing its compatibility with Apache Camel.

#### 3.2.1.2.1 Component operations and parameters

Apache Camel components use a configuration-over-code approach, allowing users to set parameters that define the component's behaviour within a Camel route. Although this approach minimizes the need for custom code, it can be confusing for users because not all parameters are compatible, and some combinations can lead to invalid configuration states.

To address this, we refactored Chimera as a set of Camel components associated with a set of specific operations that a user should explicitly configure for execution within a DataOps pipeline. This explicit configuration makes the component's functionality clearer for users. Additionally, on the backend, we redesigned the code to eliminate invalid configuration states<sup>37</sup> entirely, following principles inspired by functional programming and ML-style languages. We achieved this by using Java Records and Sealed Interfaces, features introduced in Java 14 and 17 respectively, to enforce these constraints in a robust and expressive way.

As an example, the *GraphGet* operation is used to create an *RDFGraph* and this *RDFGraph* can be of different types, described in Section 3.1.1.2 depending on the

---

<sup>34</sup> <https://github.com/cefruel/mapping-template-eval/tree/main/kgc-challenge-2024/track1>

<sup>35</sup> <https://w3id.org/rml/portal>

<sup>36</sup> *Chimera v4.1.1*, <https://github.com/cefruel/chimera>

<sup>37</sup> <https://fsharpforfunandprofit.com/posts/designing-with-types-making-illegal-states-unrepresentable/>

provided component configuration. Certain configuration parameters, such as the *ServerUrl* option which can be provided to create an *HTTPRDFGraph* do not make sense when used in conjunction with the *PathDataDir* which must be set to create a *NativeRDFGraph*. Apache Camel does not stop the user from providing both options, which would lead to an undefined configuration state. By controlling the validity states, we can detect the issue and inform the user that the configuration provided is invalid.

#### 3.2.1.2.2 External Resource Access

To further simplify user configuration of DataOps pipelines, we introduced *ChimeraResourceBeans*. A resource represents any data source required for an operation. For example, in the *Graph Add* operation, RDF triples are added from a file to an RDF graph. Here, the RDF file containing the triples is the resource. However, a resource can also be a remote file that may require authorized access.

To handle this variety of resources flexibly and intuitively, *ChimeraResourceBeans* were designed to define key details such as an access URL, serialization format, and, optionally, an authentication method. The access URL could be a local file path (e.g., `file://someFile`) or a remote address (e.g., `https://someRemoteResource`), which indicates both the type of resource and the access mechanism. This approach aligns with Apache Camel's use of URIs and URLs, providing a consistent experience.

As an example, Figure 3-15 shows a *ChimeraResourceBean* that is used to access a file in the RDF Turtle format that is stored locally.

```
<bean id="triples" class="com.cefriel.util.ChimeraResourceBean">
  <property name="url" value="file:///home/inbox/my-source.ttl"/>
  <property name="serializationFormat" value="turtle"></property>
</bean>
```

Figure 3-15: Example of a *ChimeraResourceBean* defined using XML

*ChimeraResourceBeans* have been integrated into all three Chimera components and are now the main way to access resources. An example of usage of a *ChimeraResourceBean* can be seen in Figure 3-17, where it is used to provide a SPARQL query to the graph component SPARQL select operation.

#### 3.2.1.2.3 Additional Graph Component features

The Chimera Graph Component was enhanced to include two new operations, SPARQL SELECT and ASK queries, needed to implement DataOps pipelines that dynamically access RDF repositories such as the SmartEdge Knowledge Graph Repository (A3.3). SPARQL SELECT queries retrieve specific data from an RDF graph by allowing users to define the exact pattern of triples they need, effectively pulling detailed information from datasets. SPARQL ASK queries, on the other hand, simply check whether a particular pattern exists in the graph, returning a boolean result to confirm its presence or absence. Examples for these types of queries are shown in Figure 3-16, on top a simple select query that pattern matches all the subject, predicates and objects and

returns all triples while on the bottom an ask query that returns 'true' if there exist a *?person* which is of type *Author* that *hasWritten* a specific *Book1*.

```
SELECT ?s ?p ?o
WHERE { ?s ?p ?o}

ASK {
  ?person rdf:type ex:Author .
  ?person ex:hasWritten ex:Book1 .
}
```

Figure 3-16: An example SPARQL SELECT and ASK query

The SPARQL select operation can be configured to retrieve the query results in different formats, these being JSON, XML, CSV and TSV. If no output format is chosen, then the result is kept in memory in for further processing. The result of SPARQL ask queries is always a Java Boolean value, either true or false. An example route showing the SPARQL select operation is shown in Figure 3-17, where an in-memory RDF graph is obtained, triples are added to it and then a SPARQL select query is performed and the result returned as JSON. As explained in the previous paragraphs, all resources, RDF triples and the SPARQL select query are passed in as *ChimeraResourceBeans*.

```
from("graph://get")
.to("graph://add?chimeraResource=#bean:triples")
.toD("graph://select?chimeraResource=#bean:selectQuery&dumpFormat=json")
...
```

Figure 3-17: Example Chimera route that performs a SPARQL select query and returns the result as JSON

Additionally, the Chimera graph component has been enhanced to support handling multiple RDF named graphs simultaneously. This upgrade significantly expands Chimera's data handling capabilities by enabling the retrieval of results across multiple graphs in a single query. This feature was implemented to address the common practice of storing different types of information in separate RDF named graphs, e.g., in A3.3, the storage of nodes description as separate graphs. SPARQL queries now by default consider all the named graphs provided by the user to be part of the same RDF default graph. This behaviour is not defined by the RDF and SPARQL specifications and is something that is left to the specific triplestore implementation<sup>38</sup>. Users are still able to query specific graphs by either specifying them in the SPARQL query or by specifying which named graphs should be considered when creating the *RDFGraph* through the *GraphGet* operation.

Finally, in handling these cases and using multiple *RDFGraphs* in the same pipeline we fixed an inconsistency in behaviour in the underlying *HTTPRepository*, depending on the

<sup>38</sup> <https://blog.metaphacts.com/the-default-graph-demystified>

remote Triplestore implementation, and *MemoryRepository*, implemented by the RDF4J library. These objects had an undocumented difference in their behaviour when pattern matching values specified in the SELECT clause of a SPARQL query does not find a match in the graph. In the first case, the unmatched values will be returned with a null value, indicating that no match was found. In the second case, those unmatched values will be completely skipped. This led to a difference in results when performing the same query on the same graph content but on different repository types. Because of this, non-matching values are excluded, meaning that the *HTTPRepository* behaviour is applied also to the other *Repository* types, leading to consistent results.

#### 3.2.1.2.4 Additional Mapping Template Component features

The Chimera Mapping Template component has been enhanced in a similar way to the Graph component, with improvements focused on simplifying user configuration and streamlining resource usage via *ChimeraResourceBeans*. Additionally, users now have the capability to integrate custom Java functions for data transformations into their mappings. Previously available through the mapping-template library, this functionality is now accessible directly within the Chimera Mapping Template component using *ChimeraResourceBeans*. This feature is particularly useful when custom functions are required, such as geolocation functions for coordinate system conversions that are not included in the standard Java library.

#### 3.2.1.2.5 Overall improvements

For all Chimera components, additional unit tests have also been implemented to enhance the robustness of the Chimera graph, mapping-template, and RML components. These tests ensure consistent functionality and help prevent regressions during development, providing greater stability and reliability in these components.

Chimera was upgraded to version 4.4.1 of Apache Camel, the latest available at the time, to leverage recent improvements in the Camel framework. The Chimera tutorial<sup>39</sup> was also updated to showcase example pipelines that utilize these components, streamlining user onboarding.

Finally, following a structured semantic versioning process, we developed the required mechanism to make each release available on Maven Central, simplifying reuse and integration in other projects. Thanks to the availability of components on Maven, their improved integration with the Camel ecosystem, the extensive documentation, and the tutorial, it is now much easier for users to re-use the DataOps pipeline components. Of course, each component can also be used independently and applied to heterogeneous data integration needs that differ from those investigated within the SmartEdge project.

---

<sup>39</sup> <https://github.com/cefriel/chimera-tutorial>



As a measurement of the effectiveness of the new developments, the Chimera repository almost doubled the number of *GitHub stars*<sup>40</sup> from external users interested in the project from 14 to 27 since the beginning of the SmartEdge project.

### 3.2.1.3 Mapping Template Performance and Scalability Evaluation

To evaluate the performance and scalability of the Mapping Template component, we performed an evaluation considering state-of-the-art benchmarks from the Knowledge Graph Construction (KGC) Community Group<sup>41</sup>. The usage of well-known benchmark allowed us to compare our tool with existing mapping processors implementing a declarative approach for data and schema transformations to RDF. We report here the primary outcomes of the evaluation performed. Complete details of the experiment performed, and related visualizations can be found in Annex II (Section 8).

The main result is that the designed and implemented approach for generic knowledge conversion maintains performance levels comparable to leading mapping processors for RDF graph construction tasks and can outperform them in specific scenarios. This advantage is mainly attributed to the efficient operation of the template engine and the possibility offered by MTL of introducing custom optimizations in mapping rules. On the one side, this aspect guarantees very good performance in specific scenarios like the ones addressed in SmartEdge, i.e., service mediation with small messages to be converted quickly. Conversely, the template engine is a sort of black box that prevents a more granular memory consumption optimization.

These tests could not be used to assess the KPIs 2.2 (execution time) and 2.3 (concurrency of requests) defined in SmartEdge for mediated data exchanges (service mediation use case). Indeed, benchmarks from the KGC community do not properly address these scenarios. For this reason, we performed additional tests, reported in Section 3.3.1, that address the KPIs by defining a DataOps pipeline developed specifically for a SmartEdge use case. We will work to enhance the performed evaluation for the second release and investigate the definition of a structured benchmark for dynamic data that considers the metrics assessed in SmartEdge for KPI 2.2 and 2.3.

### 3.2.2 First Implementation of the DataOps Deployment Templates (A3.6)

For this first release, we defined a first set of deployment templates from the ones identified for a DataOps pipeline. Then, we implemented demonstrator Java projects executing a DataOps pipeline that could be used to exemplify and test the different deployment templates. The DataOps deployment templates and the demonstrator pipelines are made available and documented online to make them easier to reuse<sup>42</sup>.

---

<sup>40</sup> <https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars>

<sup>41</sup> <https://www.w3.org/community/kg-construct/>

<sup>42</sup> <https://github.com/cefriel/chimera-deployment-templates>

A set of deployment template is defined to build the pipeline as a JAR and package it into a lightweight OCI container<sup>43</sup> by leveraging a multi-stage build<sup>44</sup>. A multi-stage build leverages an appropriate base image with all the required dependencies to execute the build and then copies the generated JAR within a base image containing the minimum set of dependencies to execute it. This ensures the optimisation of the final OCI container. We make two deployment templates of this type available considering two open-source Java Virtual Machines (JVMs): Temurin JVM and the alternative GraalVM Community JVM. The same approach defined in the Dockerfile for the multi-stage build can be used to run the JAR locally without containerization, assuming all the necessary dependencies are installed.

The obtained container can be executed on a container runtime and a container orchestrator like Kubernetes. We provide as part of the deployment templates the required Kubernetes manifests to execute the container as a Service<sup>45</sup> with potentially multiple replicas.

Additionally, we implemented a deployment template to run a DataOps pipeline as a native executable, thus not requiring a JVM installation on the host.

### 3.2.2.1 Deployment Templates Description

The deployment templates are exemplified by considering two distinct Camel applications that were developed for demonstration purposes: *minimal-chimera-app*<sup>46</sup> and *minimal-chimera-spring-app*<sup>47</sup>. Each application initializes the Camel context and executes a basic DataOps pipeline defined using the Camel Java DSL<sup>48</sup>. The primary distinction between the two applications lies in their underlying framework compatibility. The first is a standalone Camel application, operating independently of any external dependency injection framework. The second application, however, has been engineered to integrate with the Spring framework seamlessly. We did this test to assess the potential drawbacks from a deployment perspective of adopting an overarching framework such as Spring to execute the DataOps pipeline.

The considered pipeline, shown in Figure 3-18, is responsible for orchestrating a mediated data exchange from its source to its destination while applying a series of data and schema transformations. The steps performed are:

- Read a local file containing raw JSON data
- Execute a lifting process using the Mapping Template component to transform the data read to RDF format
- Execute a lowering process using the Mapping Template component to transform the RDF to a harmonized JSON file

Both the lifting and the lowering operations are configured via appropriate *ChimeraResourceBeans* referencing the mapping rules to be executed.

---

<sup>43</sup> <https://opencontainers.org/>

<sup>44</sup> <https://docs.docker.com/build/building/multi-stage/>

<sup>45</sup> <https://kubernetes.io/docs/concepts/services-networking/service/>

<sup>46</sup> <https://github.com/cefriel/chimera-deployment-templates/tree/main/minimal-chimera-app>

<sup>47</sup> <https://github.com/cefriel/chimera-deployment-templates/tree/main/minimal-chimera-spring-app>

<sup>48</sup> [MyRouteBuilder.java](#)

```

from("file:./data?fileName=input.csv&noop=true")
  .log("Reading file: ${file:name}")
  .convertBodyTo(String.class)
  .log("File content: ${body}")
  .log("Lifting...")
  .to("mapt://csv?template=#bean:liftingTemplate&format=turtle")
  .log("After lifting: ${body}")
  .log("Lowering...")
  .to("mapt://rdf?template=#bean:loweringTemplate")
  .log("After lowering: ${body}");

```

Figure 3-18: DataOps pipeline defined to demonstrate the deployment templates.

The second application contains the same configuration as the first one but takes advantage of Spring's powerful dependency injection mechanism to handle Camel context and Chimera pipeline dependencies.

Three different deployments templates have been defined for both the core and the spring version of the minimal chimera application, which differ in terms of the docker image used and the type of build:

- Temurin<sup>49</sup>: Minimal chimera app (Core and Spring) built with JVM and running on Temurin-17 docker image
- GraalVm<sup>50</sup>: Minimal chimera app (Core and Spring) built with JVM and running on GraalVm-17 docker image (community edition)
- Native<sup>51</sup>: Minimal chimera app (Core and Spring) app native-built with GraalVm JDK 17 and running on Alpine docker image

The deployment templates repository is organized into three primary folders: *Temurin*, *GraalVM*, and *GraalVM-Native*. Each folder contains two subfolders: *example* and *example-spring*. These subfolders provide Dockerfiles and Docker Compose configurations for building and running Java applications using the respective runtime environments. To build and to run the images from the source code, it is possible to navigate to example folder of the specific case and use the following docker commands:

```

docker-compose build
docker-compose up

```

For example, to run the *Temurin* image of the *core* version, it is possible to navigate to the folder *chimera-deployment-templates/Temurin/example* and execute the two commands listed above. This folder contains the *docker-compose*, which uses a *Dockerfile* located in the same folder to execute a multi-stage build of the application *minimal-chimera-app*.

<sup>49</sup> <https://github.com/cefriel/chimera-deployment-templates/tree/main/Temurin>

<sup>50</sup> <https://github.com/cefriel/chimera-deployment-templates/tree/main/GraalVM>

<sup>51</sup> <https://github.com/cefriel/chimera-deployment-templates/tree/main/GraalVM-Native>

We provide additional details for building Native images since it involves a slightly different process. For the Native case, the first stage uses a specific base image *cefriel/native-builder:v17*. This base image is just a "wrapper" that combines the required dependencies for building and running a native application built with GraalVM. It is based on the *ghcr.io/graalvm/native-image-community:17* image, which is required for running a native executable and an `apache-maven-3.9.6` installation for executing the native build. We made available via DockerHub the *cefriel/native-builder:v17* image<sup>52</sup> to simplify the build process using the provided Dockerfile. The Dockerfile also documents the operations for a user executing the same building process on a hosting machine without Docker.

The native build is executed with a specific maven command:

```
mvn -Pnative -Dagent=false -DskipTests package
```

The advantage of this approach is that the resulting OCI container, generated using the multi-stage build, can leverage a minimal Docker image since in this case we do not need a JVM. An Alpine Linux's latest version is used as base image for running the native executable obtained as output of the first step of the build.

### 3.2.2.2 Deployment Templates Comparison

This evaluation compares the performance and resource utilization of the deployment approaches enabled by the templates and the DataOps pipeline discussed in the previous section. The comparison focuses on the following characteristics and metrics:

- *Template*: Identifier of the DataOps Deployment Template
- *Framework*: Framework used for the project
- *JVM*: Java Virtual Machine (JVM) used for building and execution of the pipeline
- *Base docker image*: Docker image used as base for executing the pipeline
- *Executable dimension (MB)*: size of the executable artefact obtained as output of the build process
- *Image Size (MB)*: size of the container generated for execution
- *Startup Time (ms)*: time to startup the Camel Context as measured by Camel
- *CPU %*: average CPU percentage to execute the pipeline
- *Memory MB*: average memory consumption to execute the pipeline

Table 3-2 reports the metrics for each deployment template considered.

Table 3-2: Comparison of deployment templates for the same DataOps pipeline

Template	Framework	JVM	Base docker image	Executable Dimension (MB)	Image Size (MB)	Start Up Time (ms)	CPU %	Memory MB
Temurin	Maven	Build+Execution Temurin	eclipse-temurin:17	67(Jar)	528	168	0.15/1.5	128
GraalVM	Maven	Build+Execution GraalVM	ghcr.io/graalvm/jdk-community:17	67(Jar)	821	145	0.15/3	107
Native	Maven	Build GraalVM + Execution Without JVM	alpine:latest	109(Binary)	224	17	0.03/0.9	30

<sup>52</sup> <https://hub.docker.com/repository/docker/cefriel/native-builder>

Spring Temurin	Maven Spring	+	Build+Execution Temurin	eclipse-temurin:17	140(Jar)	603	30	0.16/1.8	308
Spring GraalVM	Maven Spring	+	Build+Execution GraalVM	ghcr.io/graalvm/jdk-community:17	140(Jar)	896	21	0.17/2.7	233
Spring Native	Maven Spring	+	Build GraalVM + Execution Without JVM	alpine:latest	145(Binary)	295	1	0.03/0.9	56

The results demonstrate the benefits of GraalVM native images in terms of improved resource efficiency and faster application execution. GraalVM images utilize slightly less resource than the one with Temuring JVM but have a higher image size.

Projects using Spring generate a JAR of higher dimension and demonstrate optimal startup time, higher memory consumption and similar CPU usage.

It should be noted that the Community Edition of GraalVM is less efficient than the enterprise edition<sup>53</sup>, therefore, the latter's usage is recommended if the user has a license.

To further evaluate the different deployment templates, we performed additional testing in the context of the performance and scalability evaluation on the demonstrator DataOps pipeline defined for SmartEdge Use Case 2. These results are discussed in detail in Section 3.3.1.

### 3.2.3 First Implementation of Low-code DataOps Configuration (A3.7)

To support the low-code configuration of DataOps pipelines (A3.7), we made various adjustments to Chimera to simplify the configuration of the DataOps components, and we enabled the usage of Camel Karavan for pipeline configuration via a graphical interface.

#### 3.2.3.1 Simplify DataOps Pipeline Configuration

For the first release, we focused on improving the integration of the Chimera components with the Apache Camel ecosystem and bringing the improvements made to the various components to the Apache Karavan tool. These main adjustments to facilitate the configurability of the DataOps components within a pipeline consist of (i) refactoring and better documenting the configurable parameters of each Chimera component (as discussed in Section 3.2.1.2.1), (ii) changing how external resources (e.g., declarative mapping rules) are configured to be accessed by the Chimera components (as discussed in Section 3.2.1.2.2).

In this section, we provide additional details on specific changes enabled to facilitate configurability via DSL and enable the integration with the low-code *Karavan* tool.

Currently, it is not possible to define a list of *Beans* in the Camel YAML DSL, which is the DSL also used by the *Karavan* plugin to configure a *Route*. For this reason, we refactored the Chimera components in order to support the configuration of each operation by providing one *ChimeraResourceBean* for each parameter. For example, it is possible to

<sup>53</sup> [https://www.oracle.com/a/ocom/docs/graalvm\\_enterprise\\_community\\_comparison\\_2021.pdf](https://www.oracle.com/a/ocom/docs/graalvm_enterprise_community_comparison_2021.pdf)

apply multiple *GraphAdd* operations (i.e., the operation adding RDF triples to the graph managed within the pipeline) in sequence by referencing different data sources.

The second change involves adding a parameter to specify the operation to be executed. Previously, the operation could only be defined as a URI path parameter. However, by default, the VS Code Karavan plugin could not parse the URI of custom components correctly. This issue is effectively resolved by introducing a standard parameter for specifying the operation. For example, a route step that previously had to be written as *from("graph://get?...")* can now also be expressed as *from("graph://?operation=get")*. This provides a backward compatible and reliable workaround, allowing the Karavan tool to properly interpret and handle Chimera components and their configuration.

### 3.2.3.2 Karavan Integration

The refactoring of Chimera components has improved their alignment with the behaviour and structure of standard Apache Camel components. As a result, we could enable the integration with the Karavan tool so that the user can select Chimera components from the component palette available in Karavan while defining a DataOps pipeline. This facilitates the integration with other components of the pipeline and guides the configuration of the required parameters for each component.

To install and use the Chimera components with the rest of the Camel framework in the VS Code Karavan plugin a few installation steps should be followed. The configuration files for the Chimera components can be downloaded from the Chimera GitHub repository<sup>54</sup>, which also contains the detailed installation steps alongside examples.

Once the Chimera components are installed, they become available for immediate use. At this stage, users can start creating a new Camel project with the required dependencies to integrate Chimera components. After setting up the project, the Karavan tool can be used to visually design and configure a *Route*. Single components can be picked from the Karavan component palette, as shown in Figure 3-19 and assembled into complex routes.

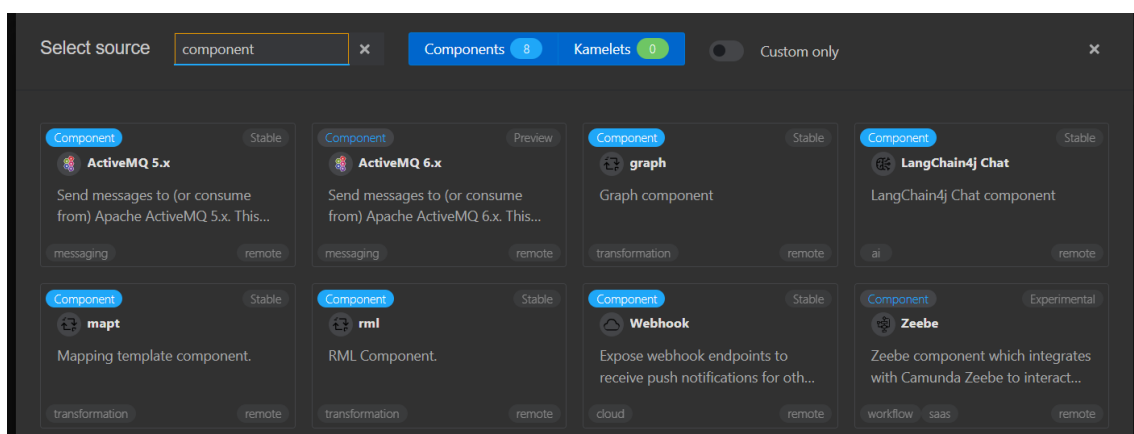


Figure 3-19: Karavan component palette showing the Chimera graph, mapping-template and rml components

<sup>54</sup> <https://github.com/cefriel/chimera/tree/master/karavan>

Once a route has been built, the components can be configured according to their needed functionality. For example, given the route shown on the left side of Figure 3-20, by clicking on a component the configuration menu shown on the right of Figure 3-20 is opened. This menu allows users to configure the component's parameters according to their needs. Depending on the metadata supplied by the authors of the specific Camel component, some configuration options may be already configured, while certain parameters might offer a limited set of predefined values for user selection.

An example of how the tool can guide the user in the configuration, is provided by the *chimeraResource* parameter for the Chimera components. Karavan can determine that the parameter expects a *Bean*, and as such, allows the user the option to choose the *Beans* that have been declared through the Karavan plugin, using the dedicated *Bean* declaration functionality shown in Figure 3-21.

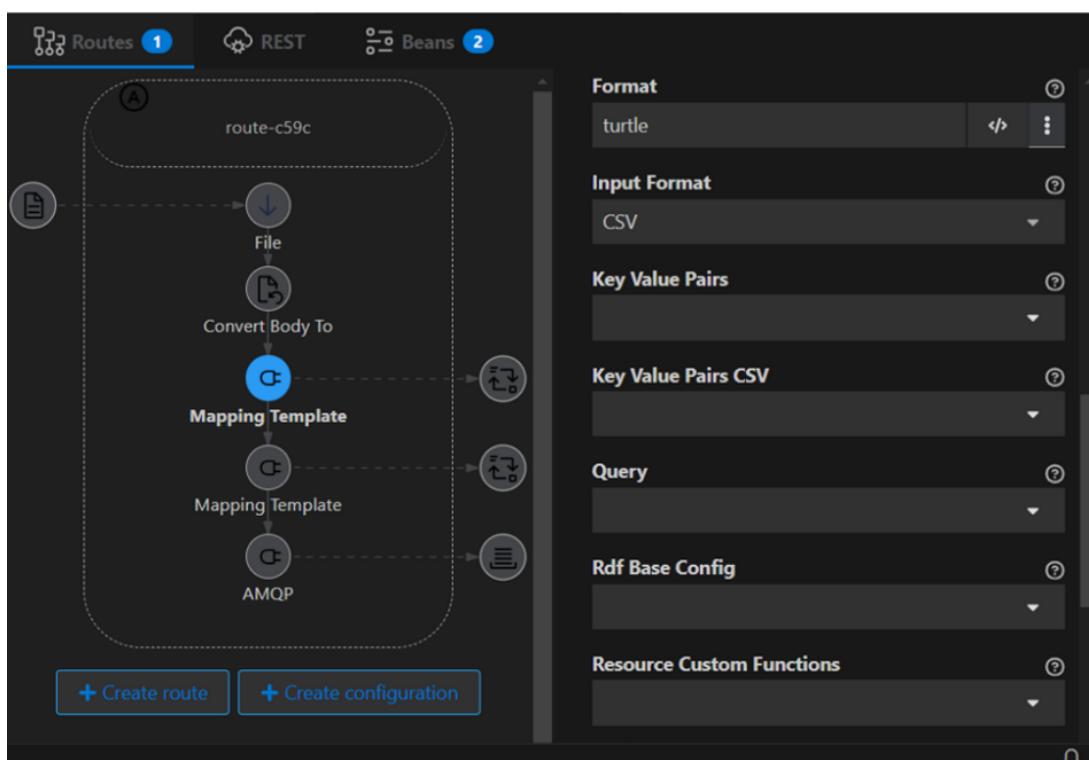


Figure 3-20: Low-code DataOps Configuration Karavan plugin interface

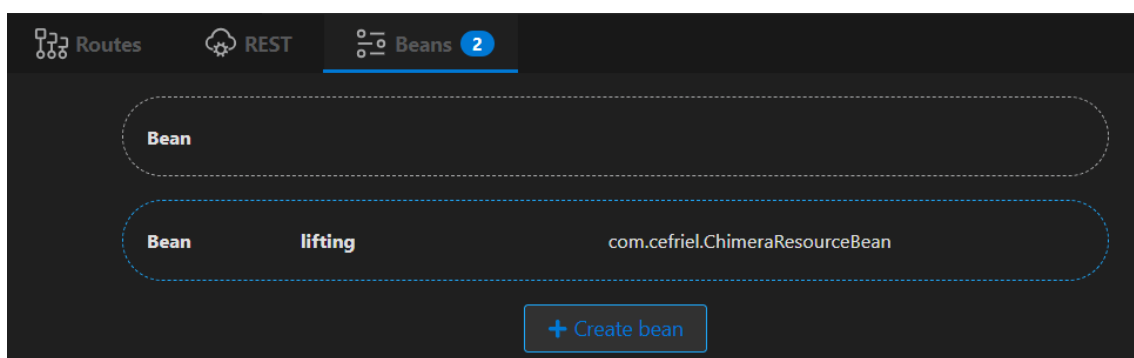


Figure 3-21: Example of a *ChimeraResourceBean* that holds the *lifting* MTL mapping file defined through Karavan

Through this graphical approach to building routes, Karavan creates a corresponding DataOps pipeline defined using the YAML DSL, shown in Figure 3-22.

```

- route:
  id: route-c59c
  nodePrefixId: route-e88
  from:
    id: from-41c6
    uri: file
    parameters:
      directoryName: ./data
      fileName: input.csv
      noop: true
  steps:
    - convertBodyTo:
      id: convertBodyTo-69b0
      type: String
    - to:
      id: to-5f43
      uri: mapt
      parameters:
        inputFormat: CSV
        template: "#bean:lifting"
        format: turtle
    - to:
      id: to-4517
      uri: mapt
      parameters:
        inputFormat: rdf
        template: "#bean:lowering"
    - to:
      id: to-d5ae
      uri: amqp
      parameters:
        destinationName: myQueue
- beans:
  - name: lifting
    type: com.cefriel.util.ChimeraResourceBean
    properties:
      serializationFormat: vtl
      url: file://./data/lift.vm
  - name: lowering
    type: com.cefriel.util.ChimeraResourceBean
    properties:
      serializationFormat: vtl
      url: file://./data/lower.vm

```

Figure 3-22: Example YAML Camel using Chimera components produced by the Visual Studio Code Karavan plugin

For this route, a file's content is read, a lifting mapping is applied to convert the data into RDF format, followed by a lowering mapping that transforms the RDF data into another specified format. The transformed data is then routed to an AMQP exchange for further processing or distribution. At the bottom of Figure 3-22 the *ChimeraResourceBeans* used in the route are defined, these being the MTL lifting mapping file and the MTL lowering mapping file. Both resources are defined by specifying the location of the mapping file and its serialization format and by associating a name to these resources. The Mapping Template component which performs both the lifting and lowering operations refers to these resources with these names using the *#bean:lifting* and *#bean:lowering* syntax.

Considering the YAML file generated by *Karavan*, another advantage is that once the route has been defined, the user is no longer bound to use the user interface. Instead,



changes can be made directly to the YAML file. This is especially useful when routes are part of a version control system like Git.

Once the route is completed, Karavan can export it to a dedicated Java project. The project is initialized with all the components present in the route, and Karavan specifies all the needed dependencies. At the moment, Karavan does not directly support the automatic addition of dependencies for custom components, and users have to explicitly add dependencies for Chimera components. Nevertheless, once a project has the required dependencies, the user can modify the pipeline through the Karavan interface, export it, and simply replace the YAML file within the Java project.

A similar problem presented itself when testing the maturity of Karavan in defining and using Kamelets relying on Chimera components. Support for specifying external dependencies has been found to be lacking in Karavan, but we plan to further investigate this option for the next release by considering the latest Karavan developments<sup>55</sup>. We also plan to investigate a tighter integration with Karavan (e.g., for direct deployment of the pipelines) now that Chimera has been upgraded to the latest Camel version, which was one of the completed activities for the first release of the DataOps components (A3.5).

For the second release of A3.7, we will focus on facilitating the automatic export and execution of pipelines from Karavan, e.g., by providing dedicated deployment templates as part of A3.6. Furthermore, we plan to explore the use of Kamelets, which can be catalogued within the Karavan plugin. These Kamelets offer the potential to allow users to easily incorporate commonly used route snippets into their projects with minimal configuration. By leveraging Kamelets, we hope to simplify further the process of designing a DataOps pipeline for the users.

### 3.3 DATAOPS TOOLBOX PIPELINES FOR SMARTEDGE

This section discusses examples of DataOps pipelines explicitly developed to address the requirements of SmartEdge use cases. These pipelines exemplify the usage of the artefacts implemented for the DataOps toolbox and offer additional insights.

In Section 3.3.1, we consider a demonstrator DataOps pipeline defined for SmartEdge Use Case 2 on traffic data to evaluate the performance and scalability of the overall DataOps solution. In particular, we focused on evaluation against KPIs 2.2 and 2.3 and comparing performances for different deployment templates.

In Section 3.3.2, we discuss how we implemented support for OPC-UA nodes for Use Case 4 in artefact A3.3 via a dedicated set of DataOps pipelines.

#### 3.3.1 DataOps Pipeline for harmonised traffic data (UC2)

The DataOps pipeline defined for the SmartEdge use case 2 exemplifies the usage of the DataOps toolbox to implement a semantic conversion process to a stream of traffic data.

---

<sup>55</sup> <https://camel.apache.org/blog/2024/03/camel-karavan-4.4.0/>

The pipeline ingests real-time data from the city of Helsinki radars, which is transmitted via WebSockets in JSON format. This data includes information on the number and types of vehicles detected, such as whether they are cars, trucks, or other vehicle types. Once collected, the data is converted into RDF to facilitate semantic interoperability and structured analysis. Two ontologies are employed for this purpose: the ASAM OpenXOntology<sup>56</sup>, which models road and vehicle-related data, and the SOSA<sup>57</sup> (Sensor, Observation, Sample, and Actuator) ontology, which is used to describe sensor-generated data. Together, these ontologies provide a standardized and meaningful representation of traffic and sensor data, enabling more effective data integration. The DataOps pipeline is illustrated in Figure 3-23. Declarative mapping rules are defined using the MTL and executed via the Mapping Template component.

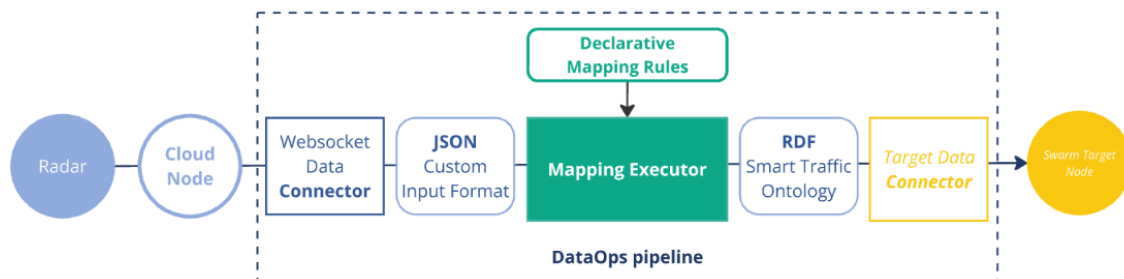


Figure 3-23: Example DataOps pipeline for the semantic conversion of radar data

The pipeline considered for evaluation involves different DataOps pipeline components (A3.5) to fetch data from the WebSocket, transform it using a predefined mapping template, and collect performance metrics. The pipeline saves the transformed sample to a file, however, by introducing an appropriate *Node Connector*, the same pipeline can be leveraged to forward the data to a generic swarm target node.

A modified version of this pipeline, performing data integration among different data sources, and details on the mapping rules defined are discussed in Deliverable D5.1 as part of the work done for the Data Stream Fusion artefact (A5.1.4).

To test the different DataOps deployment templates (A3.6), we defined a *Core* and *Spring* project for the same pipeline as done in Section 3.2.2 for the demonstrator pipeline. For each DataOps deployment template, the provided files were customized and executed to obtain the following set of images:

- Temurin Camel Core
- Temurin Camel Spring
- GraalVM Camel Core
- GraalVM Camel Spring
- Native-GraalVM Camel Core
- Native-GraalVM Camel Spring

The images were then uploaded to the Docker registry of the WP6 integration environment and executed to collect performance and scalability metrics over time.

<sup>56</sup> <https://www.asam.net/standards/asam-openxontology/>

<sup>57</sup> <https://www.w3.org/TR/vocab-ssn/>

The following metrics are collected and logged during the execution of the pipeline, as shown in Figure 3-24:

- CM (Count Messages): Message counter
- CPT (Current Processing Time): Time required to harmonize the sample
- APT (Average Processing Time): Average processing time related to the number of messages harmonized
- MXPT (Maximum Processing Time): Maximum processing time detected for harmonizing a sample
- MNPT (Minimum Processing Time): Minimum processing time detected for harmonizing a sample
- CSS (Current Sample Size): Size of the JSON sample received from the WebSocket URL
- ASS (Average Sample Size): Average size of the Json samples received from the WebSocket URL
- MXSS (Maximum Sample Size): Maximum size of the Json samples received from the WebSocket URL
- MNSS (Minimum Sample Size): Minimum size of the Json samples received from the WebSocket URL

```
INFO CM=[1] - CPT=[11ms] - APT=[11.00ms] - MXPT=[11ms] - MNPT=[11ms] CSS=[2699byte] - ASS=[2699.00byte] -
MXSS=[2699byte] - MNSS=[2696byte]
INFO CM=[2] - CPT=[9ms] - APT=[10.00ms] - MXPT=[11ms] - MNPT=[9ms] CSS=[2707byte] - ASS=[2703.00byte] -
MXSS=[2707byte] - MNSS=[2696byte]
INFO CM=[3] - CPT=[9ms] - APT=[9.67ms] - MXPT=[11ms] - MNPT=[9ms] CSS=[2707byte] - ASS=[2704.33byte] -
MXSS=[2707byte] - MNSS=[2696byte]
INFO CM=[4] - CPT=[7ms] - APT=[9.00ms] - MXPT=[11ms] - MNPT=[7ms] CSS=[2708byte] - ASS=[2705.25byte] -
MXSS=[2708byte] - MNSS=[2696byte]
INFO CM=[5] - CPT=[13ms] - APT=[9.80ms] - MXPT=[13ms] - MNPT=[7ms] CSS=[2707byte] - ASS=[2705.60byte] -
MXSS=[2708byte] - MNSS=[2696byte]
....
```

Figure 3-24: Example logs monitoring the execution of the DataOps pipeline

Additionally, we record the memory and CPU usage of the container being executed every five seconds.

The performance test is automated using a script<sup>58</sup> that can be configured in terms of:

- Number of test replicas executed for the same test case
- Duration of the test
- Optional parameter to specify a time interval between one test and the other

The testing script is primarily structured around a *Docker Compose* template. This template is a blueprint for running various Docker images obtained from the different DataOps deployment templates. The script dynamically replaces the placeholders with the relevant test-specific values in the Docker Compose template. Subsequently, the modified template is used to start the Docker image. Performance metrics are collected as discussed above.

<sup>58</sup> [https://github.com/cefriel/chimera-deployment-templates/blob/main/evaluation/run\\_tests.sh](https://github.com/cefriel/chimera-deployment-templates/blob/main/evaluation/run_tests.sh)

For the following evaluation, we considered 7000 JSON samples collected from the WebSocket radar “*lidar.otaniemi.2.json*” in around 12 minutes with a frequency of 10 messages per second. To better highlight the differences between the various test cases, the first 15 samples were removed for every case to slightly mitigate the impact of these initial spikes. In the following, we discuss the main results obtained and compare the performances between the different test cases.

Table 3-3 reports the average, maximum and minimum value for conversion time and input size for each pipeline tested. Annex III (Section 9) also reports a visualization comparing the trends of sample size and corresponding conversion time over all the samples.

From the values, it can be easily noted how images running the pipeline using *Camel Core* and the Temurin JVM recorded a better conversion time on average (3.25ms). In contrast, the same *Camel Core* version with GraalVM kept the conversion time lower than 15ms, while Temurin reached spikes of 44ms.

The Native versions obtained higher conversion times, but still around 10ms on average. The Spring version of each image performed slightly worse on average in terms of conversion time.

Table 3-3: Average/Max/Min metrics for conversion time and input size for each pipeline deployment tested.

	Avg. Conversion time (ms)	Max. Conversion time (ms)	Min. Conversion time (ms)	Avg. Input size (B)	Max. Input size (B)	Min. Input size (B)
Temurin	3.25	44	1	2771	5322	708
GraalVM	4.93	15	<1	2416	5061	413
Native	9.99	78	1	2218	5937	126
Temurin Spring	7.08	26	1	2398	5335	418
GraalVM Spring	6.97	19	1	2724	5327	417
Native Spring	10.92	87	1	3826	7380	998

In terms of CPU and memory utilisation, we summarise the main insights for each deployment template:

- *Camel Core*: the results from the Camel core tests demonstrated nearly identical performance between the Temurin and GraalVM images. This suggests that, for the considered DataOps pipeline, the choice of runtime environment does not significantly impact CPU and memory utilization.
- *Camel Spring*: when testing the Spring version, both Temurin and GraalVM images exhibited similar performance levels. However, a slight performance advantage in terms of CPU utilization was observed for the Spring applications running on the GraalVM image. This marginal improvement indicates that GraalVM might offer a modest performance boost for Spring-based workloads.
- *Camel Native (Core and Spring)*: Native images instead demonstrated a significant performance improvement in terms of both CPU and memory utilization compared to the standard (Temurin and GraalVM) images. This is

attributed to the ahead-of-time compilation that native images undergo, resulting in smaller, faster-starting applications. When comparing native Camel core and Spring applications, the Spring version consistently outperformed the core version in terms of both CPU and memory usage. This suggests that the Spring framework, when compiled into a native image, offers additional benefits in terms of resources utilization.

Annex II (Section 7) presents detailed visualizations of memory and CPU utilization for each test case and analyses them by comparing the different deployments tested.

In summary, Native images offer lower resource utilization but register higher conversion times. On the contrary, images with Camel Core run through a JVM obtain the best conversion time performance while having higher CPU and memory utilization. Considering the KPI 2.2 and 2.3, the JSON stream (10 req/s, 3Kb) was converted without dropping requests with an average conversion time lower than 4ms. Regarding the baselines [Scrocca21] of 100ms for 50KB XML and 100 concurrent requests/s, the average conversion time registered (<4ms) shows a potentially huge improvement and should enable processing of 250 req/s. However, we will have to perform additional tests for the second release, considering bigger payloads and higher concurrency of requests. Moreover, the test showcased how the input data source generated messages with unpredictably varying frequency and input size.

For these reasons, to obtain better comparable results for each image for the second release, we plan to (i) record the data from the original input data source, (ii) keep messages with a bigger payload, or manually edit them to reach at least 50kB, and (iii) reproduce the stream with messages sent at regular intervals. This would also enable the possibility of executing tests by varying the interval between requests, i.e., reaching the 100 req/s (interval 10ms) of the considered baseline.

### 3.3.2 DataOps Pipeline for OPC-UA support in A3.3 (UC4)

The support for OPC-UA for the Knowledge Graph Repository (A3.3) is developed using the DataOps toolbox and designed to define pipelines that perform a data harmonization process using the Chimera components. Such pipelines enable the description in the repository of swarm nodes for Use Case 4 that are compliant with the OPC UA standard<sup>59</sup>. Moreover, it enables the searching of OPC UA nodes according to specific capabilities.

More specifically, DataOps pipelines are used to insert and retrieve data from the Triplestore used by the Knowledge Graph Repository by carrying out two harmonization processes: a lifting operation that transforms data from an OPC UA NodeSet in XML format to RDF format, and a lowering operation that queries data in RDF format and returns an OPC UA NodeSet in XML. The first release of the pipeline is able to transform and retrieve an OPC UA NodeSet in its entirety. Moreover, it could be used to perform any query on the integrated RDF graph. For the second release, we will explore the

---

<sup>59</sup> <https://camel.apache.org/>

possibility of generating a custom OPC UA Node Set based on the result of a custom query by the user, i.e., containing only the nodes matching the provided query.

The implemented DataOps pipeline leverages a remote Triplestore for storage and querying. For the first release, we implemented and tested the solution with two Triplestore with open license: RDF4J Server<sup>60</sup> and GraphDB Free<sup>61</sup>.

The application is available through a Docker image built using the DataOps deployment templates. Two docker-compose files are defined depending on the Triplestore to be used. The Docker image can be easily configured via environment variables to provide the correct endpoints for connecting to an already existing Triplestore. The images are uploaded in the WP6 Docker Registry and executed on Kubernetes in the integration environment via a dedicated set of manifest files.

The pipelines leverage Jetty Camel component<sup>62</sup> for creating a standalone rest service which provides the following API endpoints:

1. POST: <Server-Url>:<port>/api/v1/graph
2. POST: <Server-Url>:<port>/api/v1/sparql
3. GET: <Server-Url>:<port>/api/v1/graph?named\_graph\_id={Named graph id}
4. GET: <Server-Url>:<port>/api/v1/graph/names

The POST method on the */api/v1/graph* endpoint takes as body an OPC UA NodeSet in XML. It executes a lifting transformation to produce RDF triples according to the OPC UA ontology. RDF content is saved in the repository in a dedicated named graph. The identifier of the named graph is extracted from the request body, considering the OPC UA Model associated with the NodeSet. Currently, we expect each OPC UA NodeSet to define nodes for a single OPC UA Model.

The GET method on the */api/v1/graph* endpoint returns the content of the named graph, specified with the parameter *named\_graph\_id*. The RDF is retrieved from the repository and converted via a DataOps pipeline to a corresponding OPC UA XML NodeSet.

The GET method on the */api/v1/graph/names* endpoint returns the list of all the named graphs which are saved on the triple store.

The POST method on the */api/v1/sparql* endpoint takes as body a SPARQL query as text and executes it on the triple store returning the result set. The result can be requested according to different data formats (e.g., CSV, JSON). This enables querying the repository for finding OPC-UA nodes according to specific capabilities.

---

<sup>60</sup> <https://rdf4j.org/documentation/tools/server-workbench/>

<sup>61</sup> <https://graphdb.ontotext.com/>

<sup>62</sup> <https://camel.apache.org/components/4.8.x/jetty-component.html>

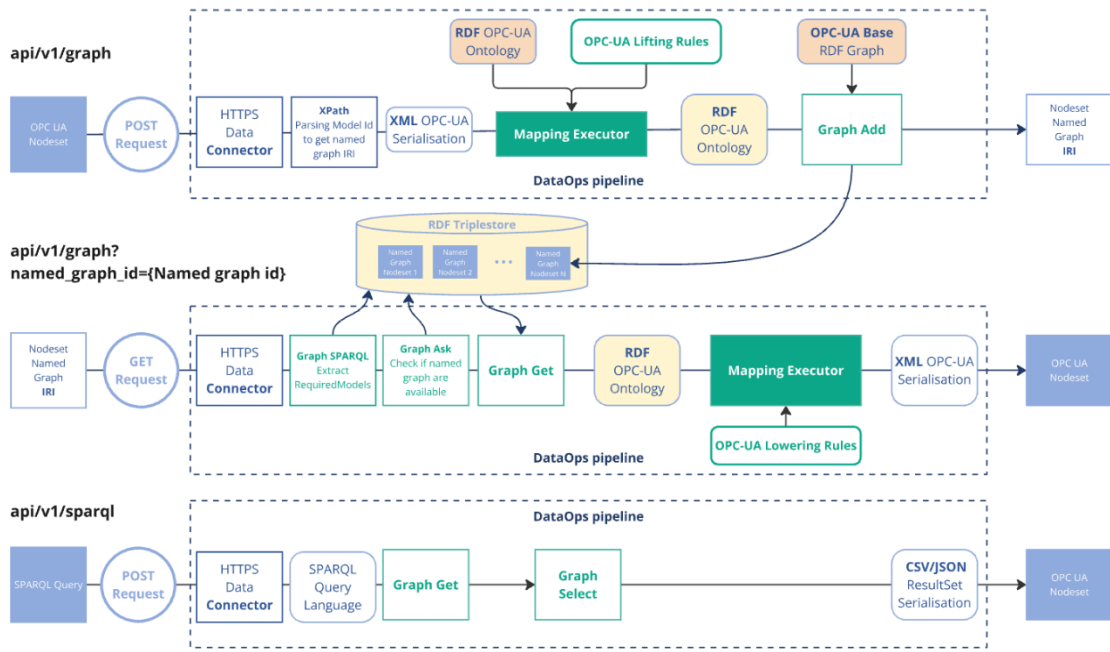


Figure 3-25: DataOps pipelines enabling support for OPC UA Nodesets in the A3.3 artefact

Figure 3-25 describes each pipeline and the DataOps components (in green) used to implement them. A dedicated set of mappings defined using MTL and custom functions (*OPCUALiftingUtils* and *OPCUALoweringUtils*) are executed using the Mapping Template component to perform the lifting and lowering operations.

Notably, the defined pipelines can initialize a dedicated repository on the Triplestore if it is not already available. Furthermore, we implemented a basic authentication mechanism that authenticates all the requests received on the REST endpoints.

For the second release, we plan to improve the management of NodeSets relying on the same set of companion specifications and to implement a strategy for dealing with different versions of the same OPC UA NodeSet in the RDF Graph.

## 4 CREATION AND ORCHESTRATION OF SWARM INTELLIGENCE APPS

---

In this section, we discuss the final design and first implementation of the artefacts dedicated to the creation and orchestration of swarm intelligence Apps, as part of Task 3.3 of SmartEdge WP3. More specifically, these are artefacts:

- A3.8: Semantic Recipe Integration with Mendix
- A3.9: Recipe-TD Matcher
- A3.10: Mendix Recipe Orchestrator

### 4.1 FINAL DESIGN

This section presents the final design of each artefact.

#### 4.1.1 Final Design of Semantic Recipe Integration with Mendix (A3.8)

One of the key advantages of the SmartEdge ecosystem lies in the ability to create domain-specific Apps using Low-Code development tools. The idea behind this concept is to facilitate and accelerate development time in Edge-enabled systems, unburdening application engineers from device configuration and other complex specific settings. As explained in the previous section, in the SmartEdge approach we propose the creation of Recipes that encapsulate the structural key characteristics of an application of a given domain.

These Recipes work as template or stereotype that indicates which are the main steps or operations that the swarm App has to fulfil, along with the goals, capabilities needed from the nodes, interactions among them, as well as input and output data. As seen in artefact A3.1, these Recipes are specified as knowledge graphs based on semantic models, with the ability to reuse existing vocabularies from domain-specific areas. These Recipes are then the basis for the SmartEdge approach for low-code swarm App development, as they constitute a declarative representation of what the app should do and how it should be structured.

With these considerations at hand, the Mendix platform provides an interesting starting point for enabling the implementation of the Recipes specified in SmartEdge. Mendix is a low-code platform with both design-time and runtime environments that facilitates the process of developing applications in different domains, including IoT components. Mendix Studio Pro (currently in version 10.x) is the design-time component of Mendix, which provides “a visual model-driven IDE with customizable themes, drag-and-drop functionality, reusable components, and full-stack capabilities”<sup>63</sup>. This is shown in Figure 4-1. The orchestration in SmartEdge is configured at design time using this tool, which permits organizing the different data sources (e.g., coming from edge nodes), establishing a flow of tasks and computations that need to be performed, and the nodes that are involved.

---

<sup>63</sup> <https://www.mendix.com/platform/ide/>



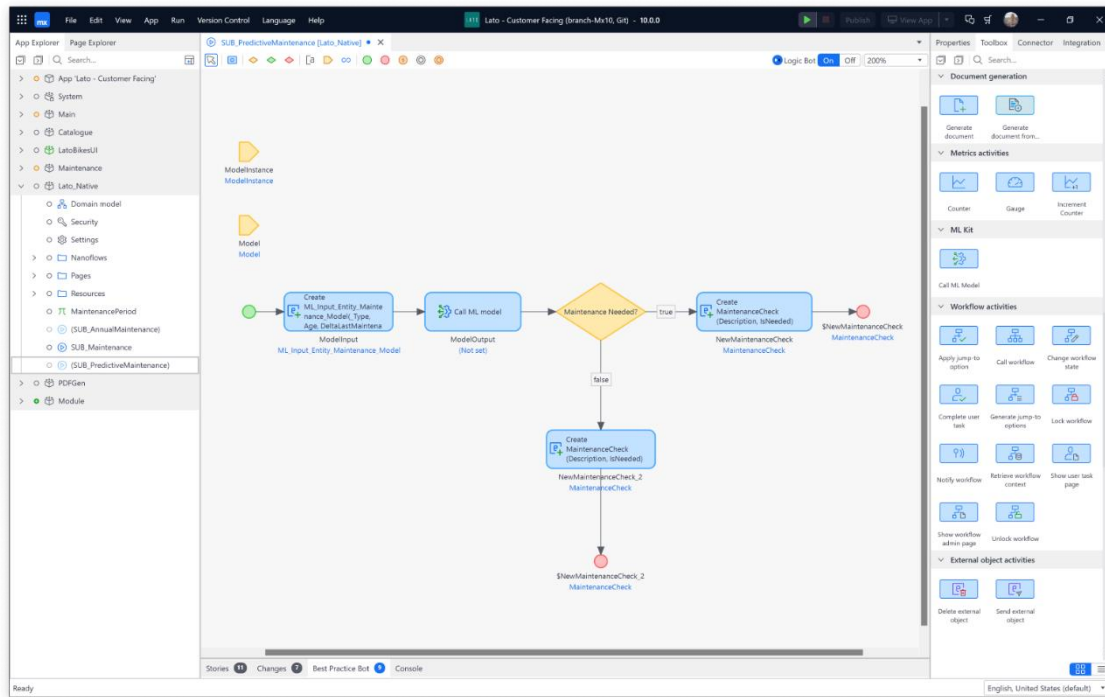


Figure 4-1: Mendix Studio Pro: design time App environment.

However, Mendix Studio Pro lacks the ability to employ **ontologies** to represent nodes in the system, capabilities, steps, or flows. It also lacks the option of including external semantic vocabularies, e.g., from existing standards, in order to represent input, outputs, or metadata information. Although Mendix Studio Pro counts with a number of plugins in the Mendix marketplace, it does not have built-in components able to connect with Triple stores or Knowledge Graphs, in order to connect with the Recipes proposed in SmartEdge.

The SmartEdge **Artefact A3.8** precisely addresses this gap, and consists of an integration component that enables the reuse of semantic Recipes within the Mendix development environment, i.e., Mendix Studio Pro.

With the integration of the semantic Recipe Knowledge Graph (a triple store containing the semantic recipes) within Mendix, it will be possible to discover and retrieve existing Recipes, related to specific swarm tasks. For example, a Recipe created for monitoring temperature measurements using a swarm of sensors could be made available in the Knowledge Graph. This Recipe would use RDF to describe what are the goals of the task, and the capabilities required from participating sensors in the swarm (e.g., measure temperature values, with a given frequency, etc.).

The architectural view of Artefact A3.8 is depicted in Figure 4-2. The Semantic Recipe integration for Mendix has access to Recipes stored as RDF knowledge graphs in a triple store database. Through SPARQL queries, it is possible to query and filter suitable Recipes that can be imported into Mendix, providing an initial set of steps in a micro- or nano-flow. Once the Recipe is loaded, the low-code developer can make all necessary modifications to complete the application (based on the Recipe) and customize it as needed.

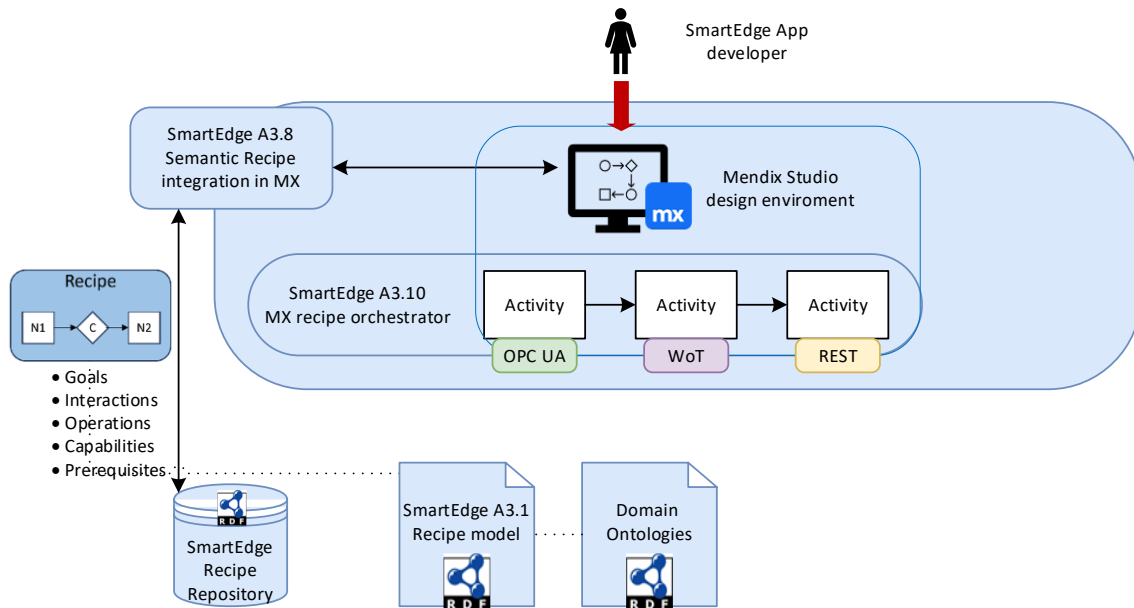


Figure 4-2: Semantic Recipe Integration.

The Mendix semantic Recipe integration functionalities can be summarized as follows:

- Recipe search: list available Recipes and search/filter according to different criteria: name, domain, capabilities, input/output, interactions.
- Recipe selection: from the Recipes available in the repository, select one and import it into the Mendix design-time environment, where further modifications can be made.
- Recipe export: from an existing Mendix flow, export a semantic Recipe and store it in the repository, from where it can later be retrieved.

**Assumptions:** This artefact depends on other SmartEdge artefacts, and makes certain assumptions as we detail next:

- This component requires the SmartEdge schema (ontology) from A3.1, which provides the blueprint for specification of nodes in the swarm. The SmartEdge schema defines the concepts of coordinator, orchestrator and other nodes that will participate in the Recipe.
- The artefact also requires the SmartEdge Recipe model, which is directly used to model the operations and capabilities in the Recipe, then included in the Mendix flow.
- The artefact makes use of the SmartEdge Knowledge Graph repository (A3.3) where the semantic Recipes are stored and queried.
- This artefact assumes the usage of Mendix as low-code development tool, although it could be in the future adapted to other similar tools based on flow-shaped declarative application development environments.

## 4.1.2 Final Design of Recipe-TD Matcher (A3.9)

Having the Recipes integrated into the Mendix Studio Pro environment, it is necessary to link the capabilities specified in the Recipe, with actual nodes available in the actual deployment. Nodes in the system are specified using the WoT TD specification<sup>64</sup>. In some scenarios nodes can also connect through OPC UA connectors as well.

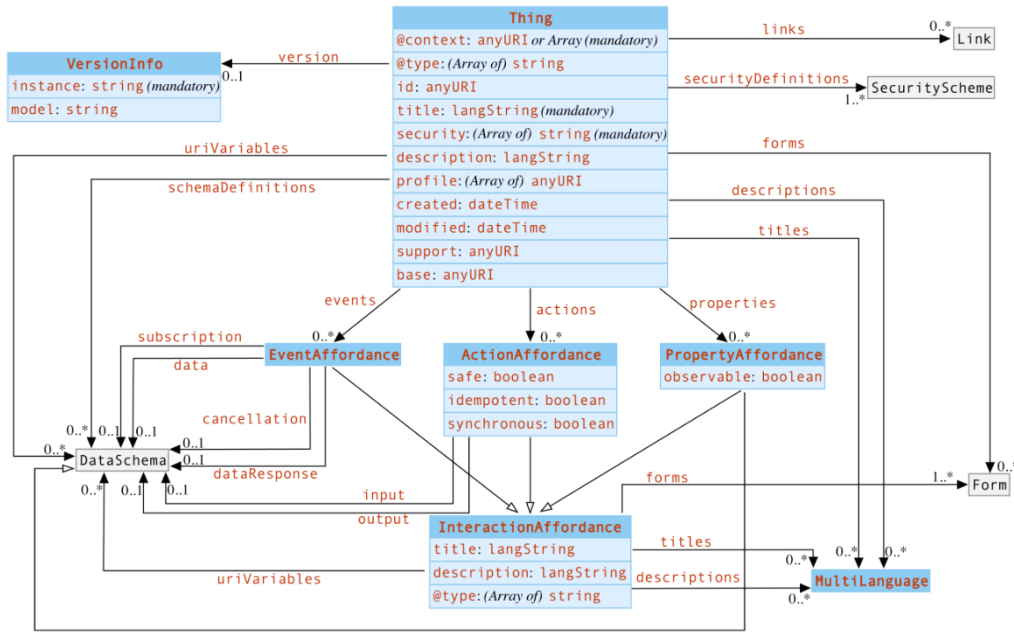


Figure 4-3: W3C Thing Description core vocabulary (source: <https://www.w3.org/TR/wot-thing-description/>)

As we can see in Figure 4-3, the TD core vocabulary provides the essential information needed to characterize a Thing, e.g., a device or sensor that maps to a node in the swarm. An essential information contained in the TD is the specification of what capabilities it has, which are needed by the Recipe. Capabilities in TD are specified through the concept of “affordances”, e.g., action, event or property affordances.

The goal of **Artefact 3.9** is to match the requirements and specifications described in the semantic Recipes, to the capabilities indicated in the WoT TD representations of the swarm nodes.

Taking a specific Recipe, the matching tool will look for the swarm devices available in the TD directory and identify those that satisfy the necessary conditions of the Recipe. Using the results of the matching, the application flow can be completed at design time, including the swarm nodes that have been suggested by the matcher. Affordances in the TDs will provide the necessary abstractions to represent the device capabilities, as well as the technical means to access them (endpoints, interfaces). As it can be seen in Figure 4-5 the SmartEdge Recipe model directly links capabilities to affordances at the conceptual level.

<sup>64</sup> <https://www.w3.org/TR/wot-thing-description/>

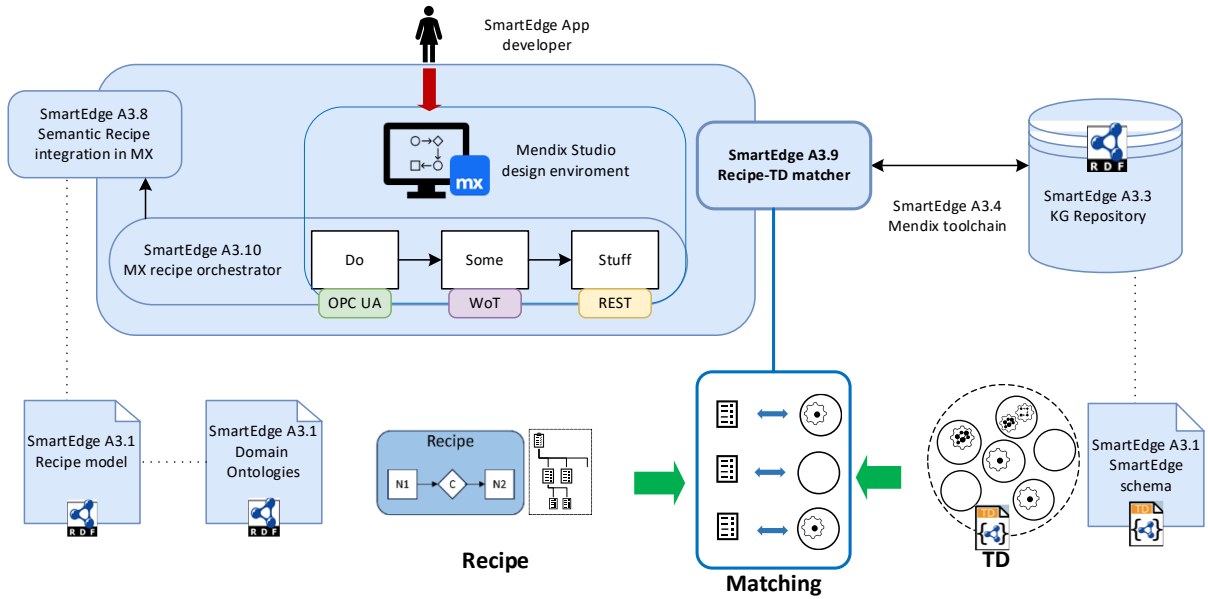


Figure 4-4: Matchmaking between the capabilities required in the Recipe and the nodes available in the Swarm at design time

Assumptions: This artefact depends on other SmartEdge artefacts, and makes certain assumptions as we detail next:

- This artefact requires the existence of semantic Recipes as described in the artefact A3.1 and used in artefact A3.8.
- TDs for existing swarm devices, and ideally the KGs and TD directory to host them, respectively.
- Nevertheless, for a first version the matcher may also work as a standalone version only connected to Recipe/TD endpoints.

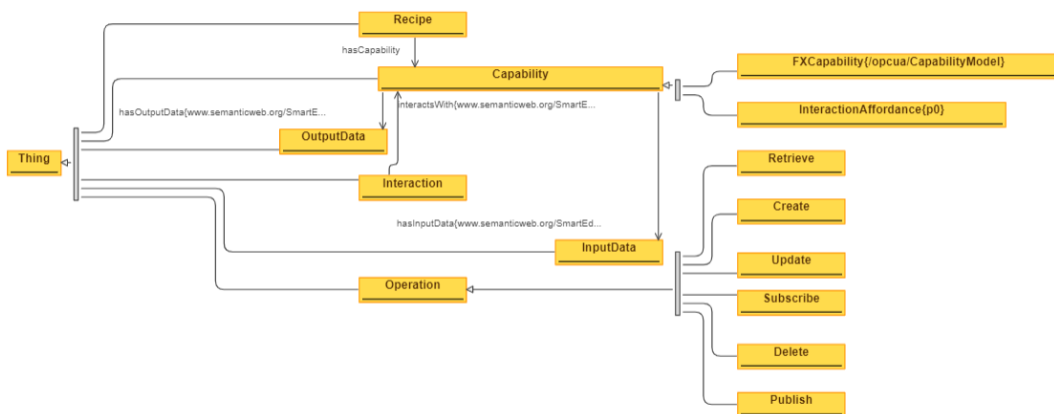


Figure 4-5: Visual representation of the main concepts of the SmartEdge semantic Recipe model, from Artefact 3.1

#### 4.1.3 Final Design of Mendix Recipe Orchestrator (A3.10)

The goal of this artefact is to enact the Recipes created with Mendix following a given Recipe and after the matching with existing devices has been performed.

This will include the coordination of the operations to be executed by different nodes in the swarm. Therefore, the orchestrator will also include the instantiation of the semantic description of the tasks, goals, sub-tasks, and skills established in the Recipe. The coordinator will then need to find and discover which nodes comply with these requirements. In cases where the orchestrator and coordinator roles are implemented by the same component (e.g., Mendix runtime), these two roles can be merged in only one entity. In certain cases, the orchestrator may not find the necessary resources to achieve the Recipe, and it could either fail or latently wait until the necessary resource can be scheduled.

This component assumes the usage of Mendix to perform the orchestration tasks. Using an existing Recipe, the low code developer will adapt it in order to customize the application, based on the Recipe. Then with the help of the previous artefact the matching of the Recipe and the nodes in the swarm will be performed. Once the bindings with the necessary devices have been established, the Mendix flow will be ready to be instantiated by the runtime. Different steps in the flow may require the interaction with edge and IoT devices. Using the Mendix tool chain artefact, different connectors will be made available. These connectors will allow interfacing devices through REST APIs, Bluetooth, etc. For instance, connection to Bluetooth devices can be specified through TD semantic affordances.

This artefact takes the following assumptions:

- Application flows are built based on a semantic Recipe using the SmartEdge Recipe model.
- Nodes in the swarm are matched against the Recipes as specified in the matcher artefact.
- The orchestration runtime is provided by Mendix.
- Interactions with IoT and edge devices are provided by connectors embedded in the Mendix flow.

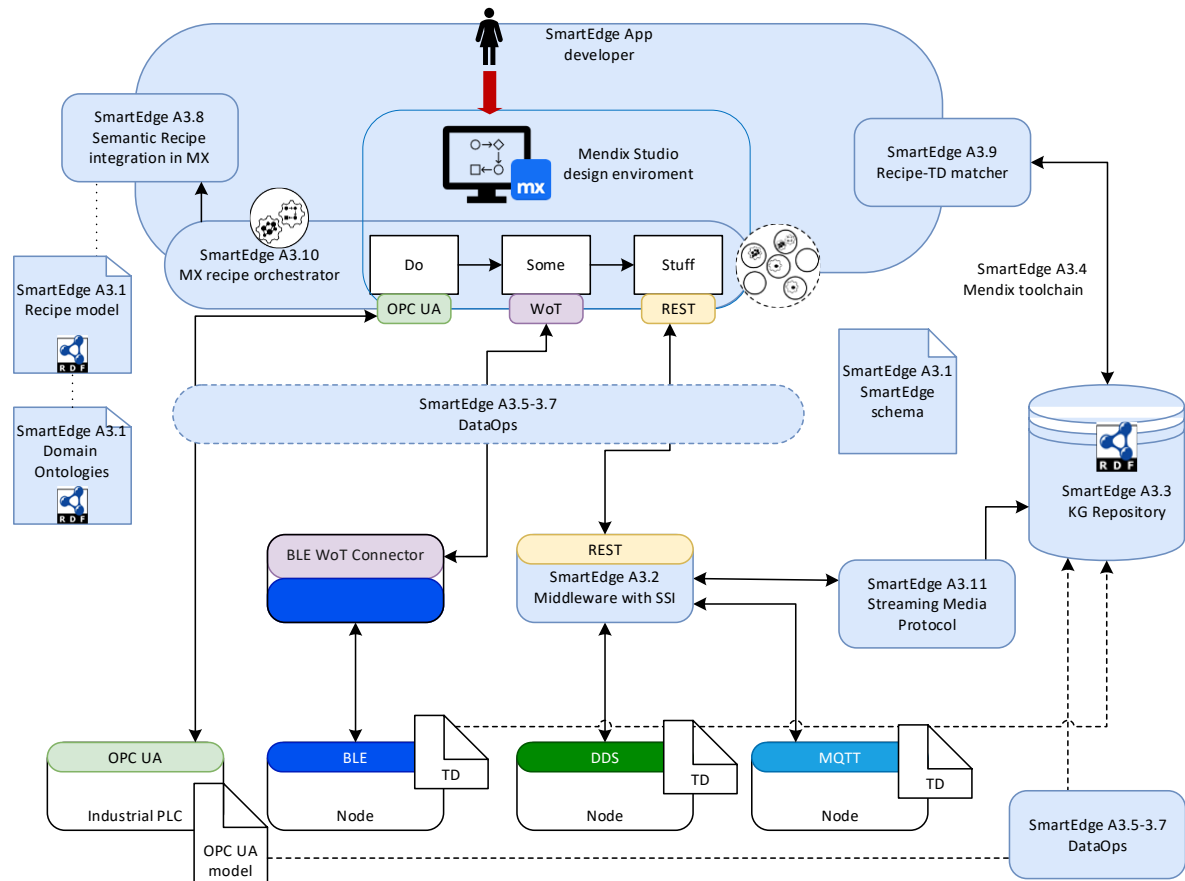


Figure 4-6: Orchestration of Mendix applications in A3.10, based on SmartEdge Recipes.

## 4.2 FIRST IMPLEMENTATION

### 4.2.1 First Implementation of Semantic Recipe Integration with Mendix (A3.8)

This component is targeted for the SmartEdge solution 2 release. For milestone M1, the integration is at an initial state. The semantic Recipe model is still being developed as part of WP3, including the Knowledge Graph (KG) that will host the Recipes. In parallel, as part of Task T3.3, we have studied Mendix's flows in order to determine how the integration implementation will be incorporated. Java extensions in Mendix have been tested, including usage of REST service calls, which could be employed to access the KG.

Examples of Recipes used so far include simple lamp control-based examples, as well as Recipes based on the SmartEdge use cases. For instance, the following snippet of a semantic Recipe is designed to describe an application flow for a simple Lamp activation system. Notice that the Recipe reuses the semantic artefact A3.1, but can also reference other external ontologies and vocabularies. For example, in the snippet below we refer to the SAREF model<sup>65</sup>.

<sup>65</sup> <https://saref.etsi.org/core/v3.1.1/>

```

{
  "@type": [
    "RecipeModel:Recipe"
  ],
  "title": "Lamp control Recipe",
  "RecipeModel:hasCapability": {
    "@type": [
      "saref4bdlg:Lamp"
    ]
  }
}
...
}

```

Figure 4-7: Snippet of a semantic Recipe for a test Lamp application.

Within the Recipe, different elements can be specified. For instance, interactions may include operations to be executed at the device level. In the example below, an operation of data retrieval is specified, which obtains the status of the Lamp. The type of interaction, as well as operation details, input and output data, are specified using a semantic representation, as in the following JSON-LD snippet.

```

"RecipeModel:hasInteraction": [
  {
    "status": {
      "description": "current status of the lamp",
      "@type": [
        "saref4bdlg:colorTemperature",
        "RecipeModel:Interaction"
      ]
    },
    "RecipeModel:hasOutputData": {
      "type": "string"
    },
    "RecipeModel:operation": "RecipeModel:Retrieve"
  }
]

```

Figure 4-8: JSON-LD snippet of an interaction detail in a sample Recipe.

Different approaches are currently explored to integrate these Recipes into the Mendix design-time environment, potentially importing nanoflows into Mendix, or adding flow components that are able to communicate with the TD repository (artifact A3.3) and SmartEdge Recipe Knowledge Graph. For example, the Recipe used the example above reflects the workflow presented in the Mendix flow in the figure below.

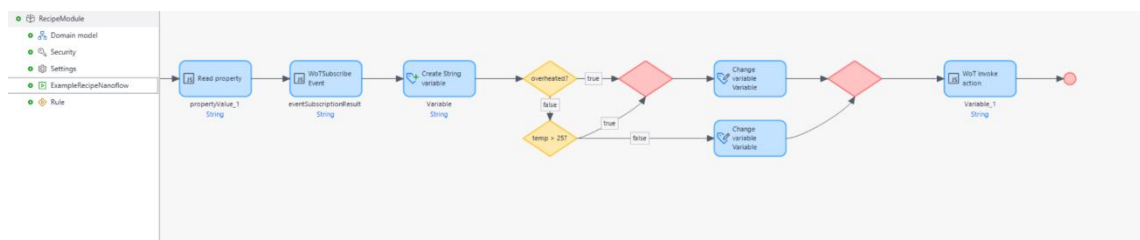


Figure 4-9: Mendix flow of a sample application

## Next steps:

- Different options are being tested regarding the importing of semantic Recipes into Mendix flows. Mendix supports a JSON format for representing the flows, and Chimera can be used as a means to translate from one model to the other, although it remains to implement the full automation of the loading of Recipes. For the time being this is an option to export Mendix flows, although there is not yet a corresponding import option at the moment.
- Testing of sample Recipes, in particular from the SmartEdge use cases will be key to demonstrate the efficacy of the artefact in managing Recipes and integrating them with the Mendix Studio Pro tool.

## 4.2.2 First Implementation of Recipe-TD Matcher (A3.9)

The matching component is targeted for the SmartEdge solution 2 release. As part of Task T3.3, we have provided sample TD RDF descriptions using JSON-LD and started using local deployments of the Node-WoT<sup>66</sup> servient for hosting TDs. The inclusion of Node-WoT within Mendix is currently under testing. It has so far been used by using a browser bundle running within Mendix.

As an example for the matching process, the Recipe snippet below represents part of the description of an exercise in a smart healthcare solution for digital rehabilitation. The Recipe snippet includes information, among other things, about the required devices needed, specified in terms of the capabilities that they should provide. In the example below (Figure 4-10) this refers to the capacity of providing rotation data for the limbs of the patient.

```
ex:exerciseRecipe1
  rdf:type          se:Exercise ;
  schema:identifier "1" ;
  schema:additionalType PhysicalTherapy ;
  schema:name       "Movement control tests-1" ;
  schema:description "Active cervical flexion and extension" ;
  schema:video      <https://youtu.be/uKjSvHty1Uo> ;
  schema:duration   "120s" ;
  ex:repetition     "3" ;
  ex:requiredDevice ex:headSensor,ex:shoulderSensor;
  ex:requiredMeasurements fe:hasConnectionFunction, fe:hasRotationFunction ;
  ex:procedureType   ex:Noninvasive ;
  schema:howPerformed [
    schema:text "The patient flexes the cervical spine so that the chin moves
  towards the sternum. The patient then extends the cervical spine into extension as
  far as possible and finally returns to the upright position."
  ] ;
```

<sup>66</sup> <https://github.com/eclipse-thingweb/node-wot>



```

    schema:description      "Allow head movements, do not allow shoulder
movements" ;
    ex:successMeasurements fe:hasSuccessEulerMeasurements,
fe:hasSuccessQuaternionMeasurements ;
    ex:alert                "positiveCount_anyShoulderMovements, wrongHeadAngle" .

```

Figure 4-10: Example of a Recipe snippet for a healthcare physiotherapy Recipe in Turtle format.

The Recipe-TD matcher then needs to perform SPARQL queries to identify which TDs contain the capabilities that are needed in order to fulfill the goals of the Recipe. In the example below (Figure 4-11), a wearable device includes in its TD descriptor information including the capabilities of the sensor. Given that it implements one type of rotation monitoring function (e.g., Euler rotation), it can be one of the devices potentially matched for the Recipe provided above. Further details could also be included in the mapping, including sensor accuracy, trust, frequency, etc.

```

se:thingy52
  rdf:type          se:Device ;
  se:role           se:Sensor ;
  se:title          "smartEdgeSensor" ;
  se:description    "Detects and responds to some type of
input from the physical environment—e.g., head movements" ;
  se:location       "head or shoulder" ;
  se:properties     fe:hasCapability, fe:identifier_service ;
  se:actions        fe:hasConnectionFunction,
fe:hasColorFunction ;
  se:events         fe:hasApplicationFunction ;
  se:goals          fe:hasRotationFunction ;
  se:knowledge      "compensationInfo" .

```

Figure 4-11: Snippet of a TD description of a device including details about its capabilities, in Turtle format.

Next steps:

- Refine the SPARQL queries to perform the matching between Recipes and TDS.
- Provide advanced matching parameters that may include detailed capability details, e.g., data quality, frequency, etc.
- Given that both Recipes and TDs can be externalized in independent Knowledge Graph stores, the artefact A3.9 will also be made available as an independent library.

#### 4.2.3 First Implementation of Mendix Recipe Orchestrator (A3.10)

This component is targeted for the SmartEdge solution 2 release. For the moment, the Mendix runtime has been tested with initial versions of UC4 and UC5b. This was important, in order to test device data retrieval from Mendix Apps, (e.g., using Nordic Thingy52 devices through BTE), and in general to test the application flow. For the next steps, we will continue with more complete flows from the use cases that will make use of Mendix, so that we capture more complete flows. These will be used to provide more complete Recipes, which will be matched with device capabilities as explained in the previous artefact.

One of the developments in this artefact includes the inclusion of microflow actions in Mendix that allow interacting with Thing Description servers. For instance, in the example below (Figure 4-12), an Invoke Action is called to perform an action exposed through an affordance in a TD. In the example it is simply incrementing a counter through HTTP verbs, but following the TD standard it can be used to interface any IoT device able to interpret those affordance invocations.

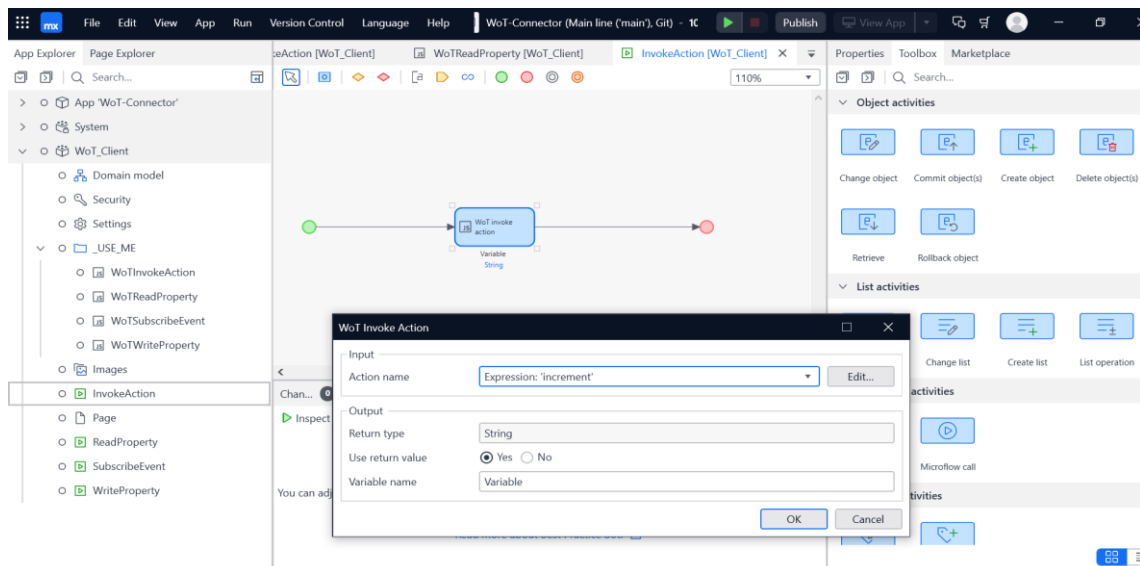


Figure 4-12: Mendix flow example, connecting to a TD counter.

In the figure below (Figure 4-13) we can see how a property affordance can be called, e.g., to read a certain property (in the simple example the value of the counter), in JavaScript code. Custom code can be added by the developer at this level in case of needing further functionalities.

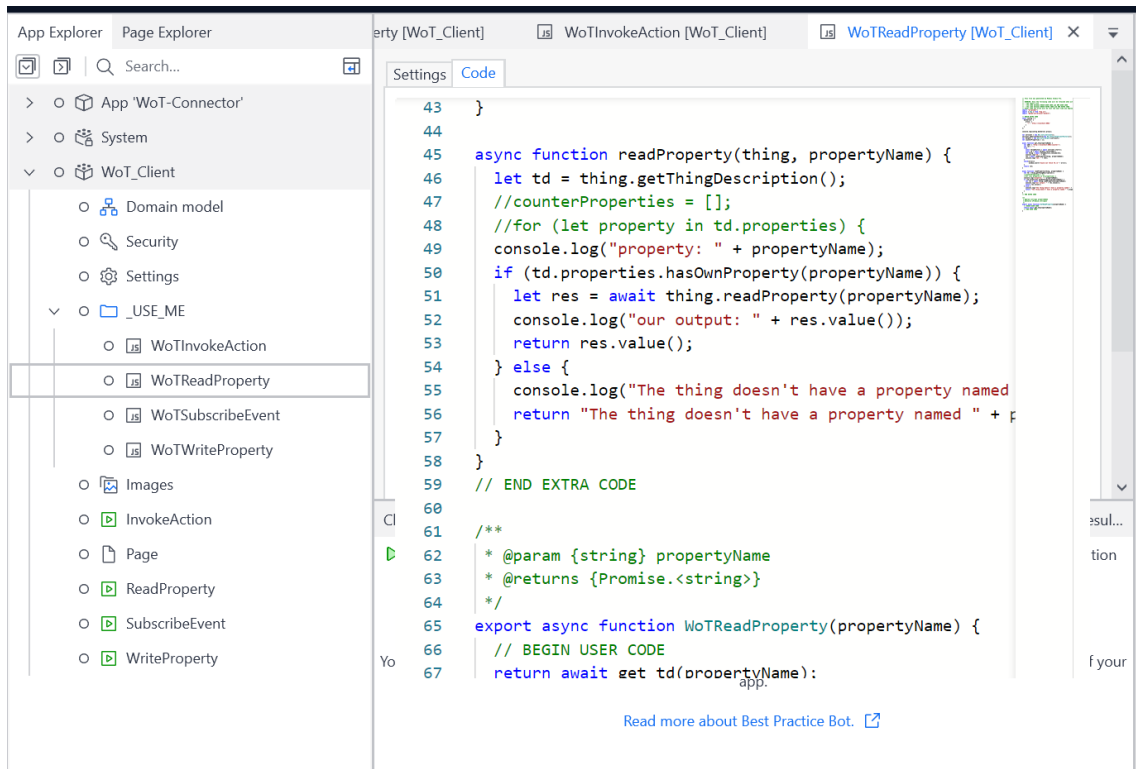


Figure 4-13: Reading a property from a TD description from JS code in a Mendix flow.

Similarly, through the property affordance of the TD, it is possible to modify the value of a given property, as seen in the example below (Figure 4-14). The same function can be reused to set any IoT device property, e.g., a state, parameter, or any other kind of value that is exposed through the TD interface.

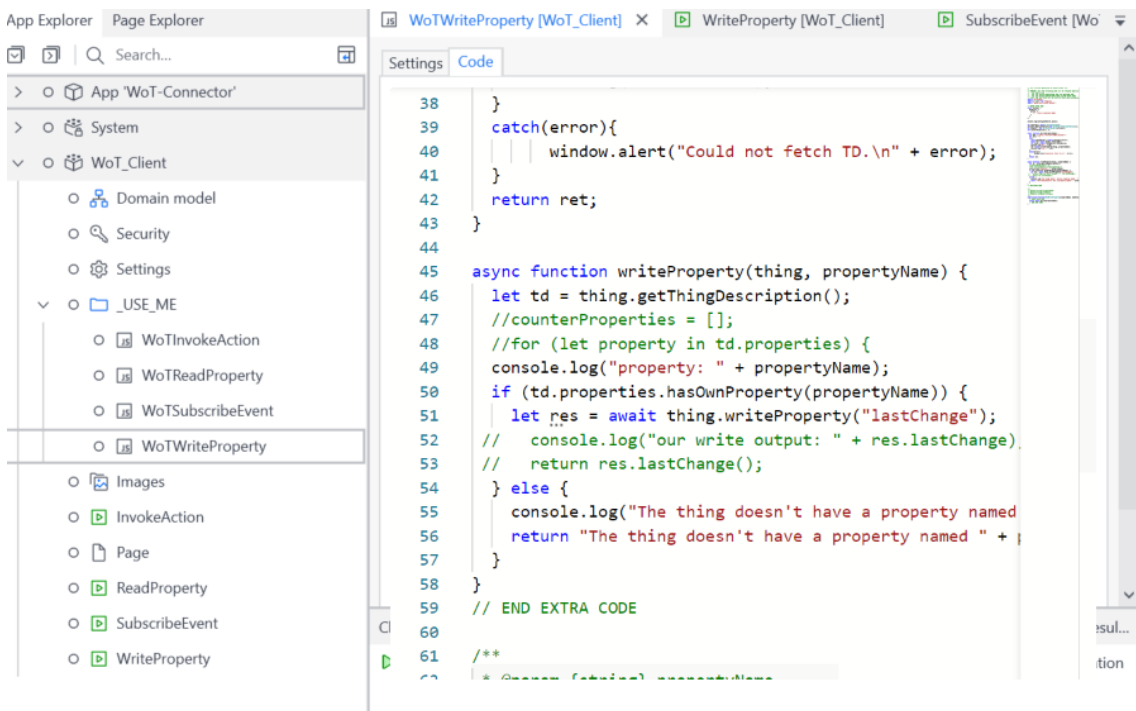


Figure 4-14: Writing back to a TD exposed property using JS code inside Mendix.

Once these interactions through the TD interface are established, then the low-code developer can complete the application, e.g., a front-end webpage designed to display the values of the counter. In the screenshot below (Figure 4-15) we can see how this is done for the simple counter example on a basic front end web page designed in Mendix.

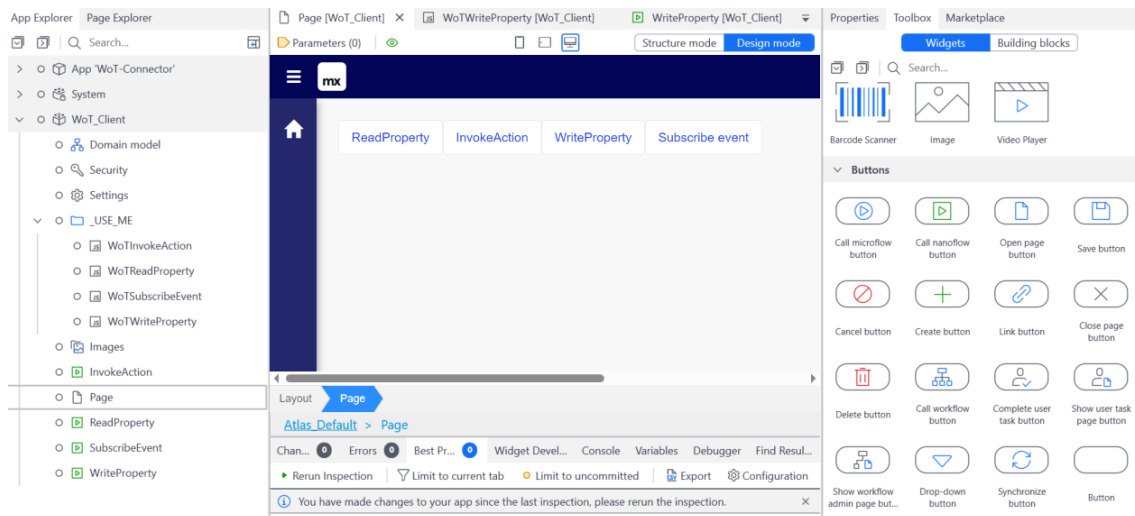
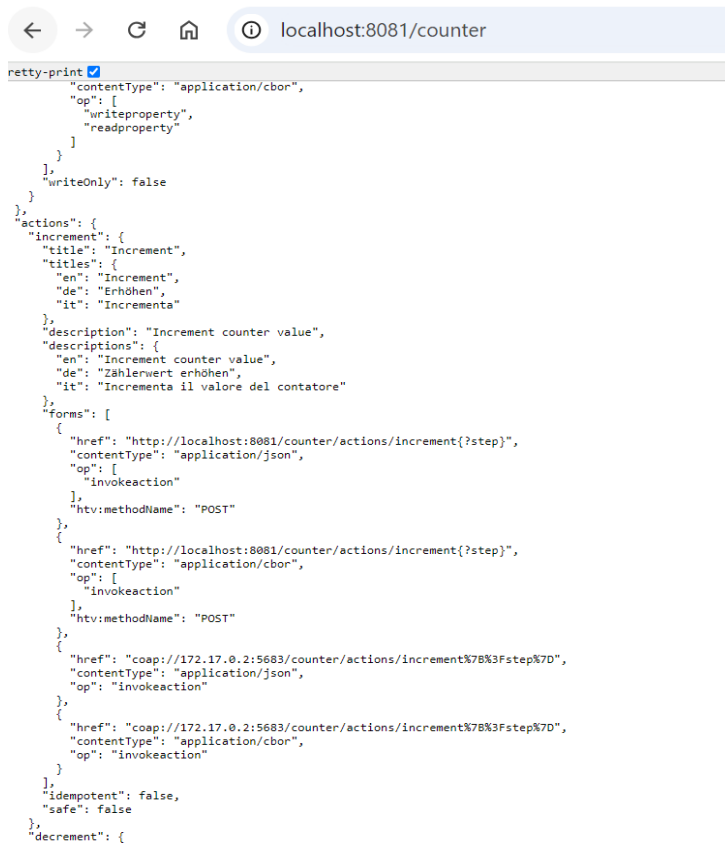


Figure 4-15: Accessing TD-retrieved properties from a Mendix-created application page.

At runtime, we have tested these TD invocation activities in a flow orchestrated by the Mendix runtime. To do so, first the Node-WoT server (an Eclipse implementation of a WoT server through TDs) is started independently through a Docker container (Figure 4-16).

```
C:\Users\banani.anuraj\node-wot>docker run -it --init -p 8081:8080/tcp -p 5683:5683
/udp -v "%cd%/examples:/srv/examples --rm wot-servient /srv/examples/scripts/count
er.js -p http.baseUri=http://localhost:8081
Produced Counter
Counter ready
```



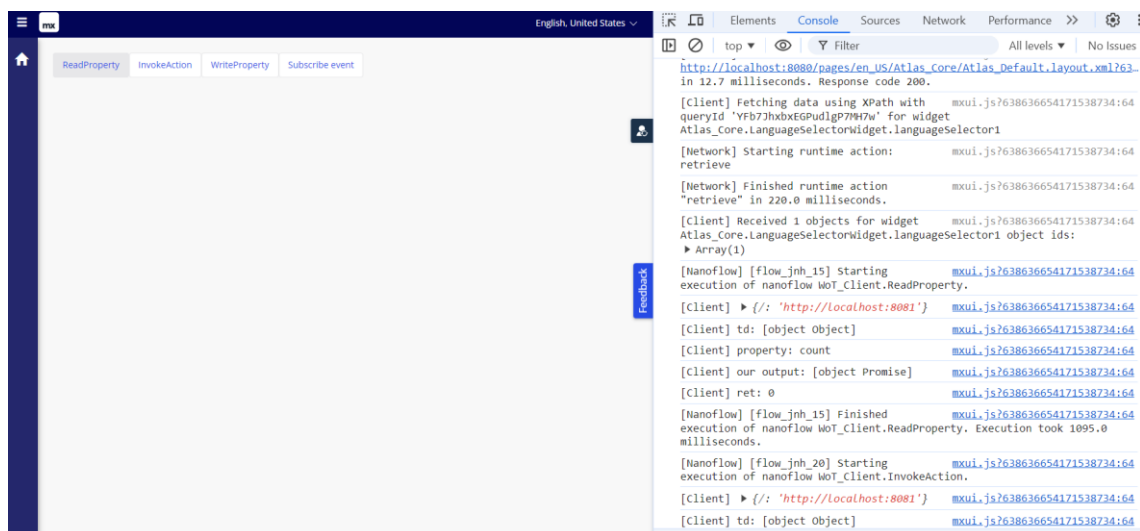
```

retty-print
{
  "contentType": "application/cbor",
  "op": [
    "writeproperty",
    "readproperty"
  ]
}
},
"writeOnly": false
},
"actions": {
  "increment": {
    "title": "Increment",
    "titles": {
      "en": "Increment",
      "de": "Erhöhen",
      "it": "Incrementa"
    },
    "description": "Increment counter value",
    "descriptions": {
      "en": "Increment counter value",
      "de": "Zählerwert erhöhen",
      "it": "Incrementa il valore del contatore"
    },
    "forms": [
      {
        "href": "http://localhost:8081/counter/actions/increment{?step}",
        "contentType": "application/json",
        "op": [
          "invokeaction"
        ],
        "htv:methodName": "POST"
      },
      {
        "href": "http://localhost:8081/counter/actions/increment{?step}",
        "contentType": "application/cbor",
        "op": [
          "invokeaction"
        ],
        "htv:methodName": "POST"
      },
      {
        "href": "coap://172.17.0.2:5683/counter/actions/increment%7B%3Fstep%7D",
        "contentType": "application/json",
        "op": "invokeaction"
      },
      {
        "href": "coap://172.17.0.2:5683/counter/actions/increment%7B%3Fstep%7D",
        "contentType": "application/cbor",
        "op": "invokeaction"
      }
    ]
  },
  "idempotent": false,
  "safe": false
},
"decrement": {

```

Figure 4-16: Accessing the Node-WoT server, counter example exposed through a TD.

And then the Mendix project is run, and the application can read, write and display the values exposed by the TD, as it can be seen in the console output and the application frontend (Figure 4-17).



```

http://localhost:8080/pages/en_US/Atlas_Core/Atlas_Default.layout.xml?63-
in 12.7 milliseconds. Response code 200.
[Client] Fetching data using XPath with queryId 'YFb7jhxbxEGPudlgP7#47w' for widget Atlas_Core.LanguageSelectorWidget.LanguageSelector1
[Network] Starting runtime action: retrieve
[Network] Finished runtime action "retrieve" in 220.0 milliseconds.
[Client] Received 1 objects for widget Atlas_Core.LanguageSelectorWidget.LanguageSelector1 object ids:
  ▶ Array(1)
[NaNoflow] [flow_jnh_15] Starting execution of nanoflow Wot_Client.ReadProperty.
[Client] ▶ [/: 'http://localhost:8081']
[Client] td: [object Object]
[Client] property: count
[Client] our output: [object Promise]
[Client] ret: 0
[NaNoflow] [flow_jnh_15] Finished execution of nanoflow Wot_Client.ReadProperty. Execution took 1095.0 milliseconds.
[NaNoflow] [flow_jnh_20] Starting execution of nanoflow Wot_Client.InvokeAction.
[Client] ▶ [/: 'http://localhost:8081']
[Client] td: [object Object]

```

Figure 4-17: Mendix App frontend. TD accessed through the application during runtime

Runtime Invoke Action (Increment) and read property again through the TD interface (Figure 4-18):

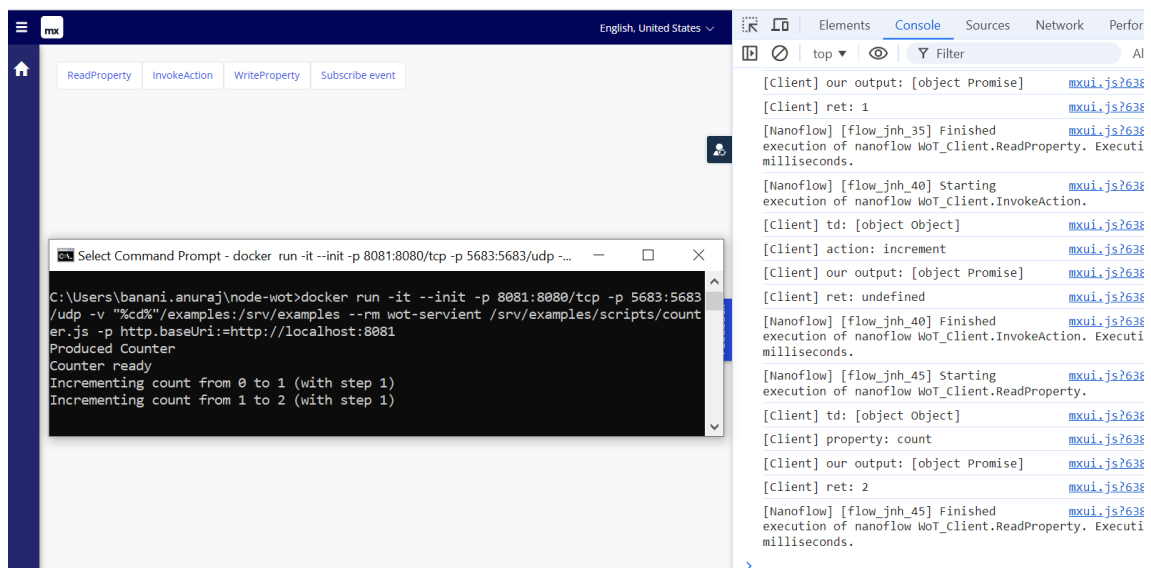


Figure 4-18: Runtime invoke action: TD property read through the Mendix runtime

Next steps:

- Consolidate the implementation and integration of TD invocation actions in Mendix
- Integrate the TD-Recipe matching tool into the Mendix orchestration workflow.
- Test the orchestration and execution of instantiated Recipes in collaboration with use case owners.

## 5 CONCLUSIONS

---

This document described the first release of artefacts implemented in SmartEdge to enable the concept of Continuous Semantic Integration (CSI). This concept is broken down into (i) Standardized Semantic Interfaces (Section 3); (ii) the DataOps toolbox for semantic management of things and embedded AI apps (Section 4); (iii) Creation and orchestration of Swarm Intelligence apps (Section 5). This deliverable described the final design of the CSI tools by revising and extending deliverable D3.1 and considering the final list of requirements from D2.2. Section 1, provided an overview of the tasks required for CSI and described how the 11 artefacts identified for WP3 are expected to be integrated to enable CSI for a SmartEdge use case. Section 2 also reported the expected integration with WP4 and WP5 and the current status for each artefact.

This deliverable provided the following contributions considering the SmartEdge Obj.2 “Middleware and tools for continuous semantic integration”:

- *standardized semantic interface*: first implementation of interoperable semantic models for the description of nodes (statically and at runtime) and applications (i.e., Recipes), repository for the store and retrieval of interoperable descriptions, middleware solutions for standardized interfaces among nodes;
- *continuous conversion process based on declarative mappings and scalable from edge to cloud*: first implementation of reusable and modular component for the declarative definition of conversion pipelines, templates for scalable deployments of pipeline on Edge and Cloud devices, low-code approach for pipeline definition;
- *declarative approach for the creation and orchestration of apps based on swarm intelligence*: final design and initial developments to support the definition of semantic Recipes, perform matchmaking of nodes for Recipes, and orchestrate Recipes across nodes.

The second release will focus on extending and improving artefacts released for the first two objectives by implementing feedback from the first validation phase within WP6. Furthermore, the artefacts planned for release 2 will be made available.

The status of KPIs for Work Package 3 (WP3) is presented in this deliverable (see Table 2.4), considering the progress made on the first implementation of the tools. A full report against SmartEdge requirements and KPIs will be provided in D6.1 considering the first release of the integrated SmartEdge solution.

The successor of this deliverable, i.e., D3.3 will describe the final implementation of tools for Continuous Semantic Integration based on the collected feedback.

## 6 REFERENCES

---

[Arenas21] J. Arenas-Guerrero, M. Scrocca, A. Iglesias-Molina, J. Toledo, L. Pozo-Gilo, D. Doña, Ó. Corcho and D. Chaves-Fraga, “Knowledge Graph Construction with R2RML and RML: An ETL System-based Overview”, Proceedings of the 2nd International Workshop on Knowledge Graph Construction co-located with 18th Extended Semantic Web Conference (ESWC 2021), 2021, CEUR Workshop Proceedings, <http://ceur-ws.org/Vol-2873/paper11.pdf>

[Arenas22] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M.S. Pérez and O. Corcho, “Morph-KGC: Scalable knowledge graph materialization with mapping partitions”, Semantic Web (2022). doi:10.3233/SW-223135

[Hohpe04] Hohpe, Gregor, and Bobby Woolf. 2004. “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions”. Addison-Wesley Professional.

[Schiekofer19] Schiekofer, Rainer, et al., "A Formal Mapping between OPC UA and the Semantic Web.", IEEE 17th International Conference on Industrial Informatics (INDIN), 2019, doi: 10.1109/INDIN41052.2019.8972102.

[Scrocca21] Scrocca M., Carenini A., et al., “Semantic Conversion of Transport Data Adopting Declarative Mappings: An Evaluation of Performance and Scalability”, Sem4Tra 2021 @SEMANTICS, CEUR-WS.org, <http://ceur-ws.org/Vol-2939/paper2.pdf>

[Scrocca24] Scrocca Mario, Alessio Carenini, Marco Grassi, Marco Comerio, and Irene Celino, "Not Everybody Speaks RDF: Knowledge Conversion between Different Data Representations.", Fifth International Workshop on Knowledge Graph Construction@ ESWC2024, 2024, <https://ceur-ws.org/Vol-3718/paper3.pdf>

[Vetere05] Vetere, G., and M. Lenzerini. 2005. “Models for Semantic Interoperability in Service-Oriented Architectures.” IBM Systems Journal 44 (4): 887–903. <https://doi.org/10.1147/sj.444.0887>

[Vleeschauwer24] E.d. Vleeschauwer, P. Maria, B.D. Meester and P. Colpaert, “RML-view-to-CSV: A Proof-of-Concept Implementation for RML Logical Views”, Proceedings of the 5th International Workshop on Knowledge Graph Construction, CEUR Workshop Proceedings, Vol. 3718, 2024, <https://ceur-ws.org/Vol-3718/paper2.pdf>



## 7 ANNEX I – SAMPLE RECIPE FOR UC4

This annex presents a sample Recipe for UC4 which is described in Section 2.2.2.1.

```
{
"@context":[
  {
    "RecipeModel":"http://www.semanticweb.org/SmartEdge/RecipeModel/",
    "saref4bldg": "https://saref.etsi.org/saref4bldg/",
    "saref": "https://saref.etsi.org/saref/",
    "iot": "http://iotschema.org/",
    "@id":"http://www.semanticweb.org/SmartEdge/RecipeModel/",
    "@type":[
      "http://www.w3.org/2002/07/owl#Ontology"
    ]
  }
],
"@type":[
  "Recipe"
],
"title":"Metaverse product assembly Recipe",
"NLQ":"An application to simulate the assembly of a product in assembly station in metaverse",
"hasCapability":{
  "@type":[
    "SmA:kill_Insert" , "SmA:Skill_Load_Unload"
  ]
},
"hasIngredients":[
  {
    "load":{
      "@id":"b4493a89cfd4a062",
      "NLQ": "find a skill to load the plate into an assembly module",
      "description":"load plate to a module",
      "@type":[
        "SmA:kill_Load_Unload",
        "Ingredient"
      ],
      "hasInputData":{
        "type":{
          "argument1": {
            "name": "sourcePos",
            "type": "number"
          },
          "argument2": {
            "name": "DestinationPos",
            "type": "number"
          },
          "argument3": {
            "name": "RFID",
            "type": "number"
          }
        }
      },
      "hasOutputData":{
        "type":{
          "argument1": {
            "name": "ErrorID",
```

```

        "type": "number"
      }
    }
  },
  "operation": "Update",
  "interactsWith": [
    {
      "hasSerialNumber": "1",
      "@id": "ccfca6fc0f1c1e9c",
      "operation": "Update"
    }
  ]
},
{
  "insert": {
    "@id": "ccfca6fc0f1c1e9c",
    "NLQ": "find a skill to insert a block into the plate",
    "description": "insert a block into the plate",
    "@type": [
      "SmA:Skill_Insert",
      "Ingredient"
    ],
    "hasInputData": {
      "type": {
        "argument1": {
          "name": "Position",
          "type": "number"
        },
        "argument2": {
          "name": "BuildingBlockTypeID",
          "type": "number"
        },
        "argument3": {
          "name": "Orientation",
          "type": "number"
        },
        "argument4": {
          "name": "RFID",
          "type": "number"
        },
        "argument5": {
          "name": "CurrentConfiguration_BuildingBlockTypeid",
          "type": "number"
        },
        "argument6": {
          "name": "CurrentConfiguration_Orientation",
          "type": "number"
        }
      }
    },
    "hasOutputData": {
      "type": {
        "argument1": {
          "name": "ErrorID",
          "type": "number"
        }
      }
    }
  }
}

```

```

    }
  },
  "RecipeModel:operation":"RecipeModel:Update",
  "interactsWith":[
    {
      "hasSerialNumber": "2",
      "@id": "dcfca6fc0f1c1e9d",
      "operation":"Update"
    }
  ]
}
},
{
  "insert":{
    "@id":"dcfca6fc0f1c1e9d",
    "NLQ": "find a skill to insert a block into the plate",
    "description":"insert a block into the plate",
    "@type":[
      "SmA:Skill_Insert",
      "Ingredient"
    ],
  },
  "hasInputData":{
    "type":{
      "argument1": {
        "name": "Position",
        "type": "number"
      },
      "argument2": {
        "name": "BuildingBlockTypeID",
        "type": "number"
      },
      "argument3": {
        "name": "Orientation",
        "type": "number"
      },
      "argument4": {
        "name": "RFID",
        "type": "number"
      },
      "argument5": {
        "name": "CurrentConfiguration_BuildingBlockTypeId",
        "type": "number"
      },
      "argument6": {
        "name": "CurrentConfiguration_Orientation",
        "type": "number"
      }
    }
  },
  "hasOutputData":{
    "type":{
      "argument1": {
        "name": "ErrorID",
        "type": "number"
      }
    }
  }
},

```

```

"RecipeModel:operation":"RecipeModel:Update",
"interactsWith":[
  {
    "hasSerialNumber": "3",
    "@id": "ecfca6fc0f1c1e0e",
    "operation":"Update"
  }
]
},
{
  "insert":{
    "@id":"ecfca6fc0f1c1e0e",
    "NLQ": "find a skill to insert a block into the plate",
    "description":"insert a block into the plate",
    "@type":[
      "SmA:Skill_Insert",
      "Ingredient"
    ],
    "hasInputData":{
      "type":{
        "argument1": {
          "name": "Position",
          "type": "number"
        },
        "argument2": {
          "name": "BuildingBlockTypeID",
          "type": "number"
        },
        "argument3": {
          "name": "Orientation",
          "type": "number"
        },
        "argument4": {
          "name": "RFID",
          "type": "number"
        },
        "argument5": {
          "name": "CurrentConfiguration_BuildingBlockTypeId",
          "type": "number"
        },
        "argument6": {
          "name": "CurrentConfiguration_Orientation",
          "type": "number"
        }
      }
    },
    "hasOutputData":{
      "type":{
        "argument1": {
          "name": "ErrorID",
          "type": "number"
        }
      }
    }
  },
  "RecipeModel:operation":"RecipeModel:Update",
  "interactsWith":[

```

```

    {
      "hasSerialNumber": "4",
      "@id": "fcfca6fc0f1c1e1f",
      "operation": "Update"
    }
  ]
}
},
{
  "insert": {
    "@id": "fcfca6fc0f1c1e1f",
    "NLQ": "find a skill to insert a block into the plate",
    "description": "insert a block into the plate",
    "@type": [
      "SmA:Skill_Insert",
      "Ingredient"
    ],
    "hasInputData": {
      "type": {
        "argument1": {
          "name": "Position",
          "type": "number"
        },
        "argument2": {
          "name": "BuildingBlockTypeID",
          "type": "number"
        },
        "argument3": {
          "name": "Orientation",
          "type": "number"
        },
        "argument4": {
          "name": "RFID",
          "type": "number"
        },
        "argument5": {
          "name": "CurrentConfiguration_BuildingBlockTypeId",
          "type": "number"
        },
        "argument6": {
          "name": "CurrentConfiguration_Orientation",
          "type": "number"
        }
      }
    },
    "hasOutputData": {
      "type": {
        "argument1": {
          "name": "ErrorID",
          "type": "number"
        }
      }
    },
    "RecipeModel:operation": "RecipeModel:Update",
    "interactsWith": [
      {
        "hasSerialNumber": "5",

```

```

    "@id": "g4493a89cfd4a063",
    "operation": "Update"
  }
]
},
{
  "Unload": {
    "@id": "g4493a89cfd4a063",
    "NLQ": "find a skill to unload the plate from an assembly module",
    "description": "load plate to a module",
    "@type": [
      "SmA:kill_Load_Unload",
      "Ingredient"
    ],
    "hasInputData": {
      "type": {
        "argument1": {
          "name": "sourcePos",
          "type": "number"
        },
        "argument2": {
          "name": "DestinationPos",
          "type": "number"
        },
        "argument3": {
          "name": "RFID",
          "type": "number"
        }
      }
    },
    "hasOutputData": {
      "type": {
        "argument1": {
          "name": "ErrorID",
          "type": "number"
        }
      }
    },
    "operation": "Update"
  }
}
}}

```

Figure 7-1: Sample Recipe snippet for smart factory application in UC4

## 8 ANNEX II – PERFORMANCE AND SCALABILITY EVALUATION OF THE MAPPING TEMPLATE COMPONENT

---

This annex provides a detailed description of the quantitative assessment of the performance and scalability of the *mapping-template* component reported in Section 3.2.1.3.

All the diagrams presented in this section report the average metrics over multiple test repetitions on a logarithmic scale ( $\log_{10}$ ).

### 8.1 GTFS MADRID BENCHMARK

For an initial assessment of the performance and scalability of the *mapping-template* tool, we utilized the GTFS-Madrid-Bench<sup>67</sup> following the methodology and the RML mappings established in the evaluation by Arenas et al. [Arenas21]. The benchmark includes a variety of (R2)RML mappings and a generator for producing input data sources in different formats and sizes. We focused on three data formats (CSV, XML, and JSON) and tested three scaling factors (1, 10, and 100), comparing the *mapping-template* tool to the *morph-kgc* v2.3.1<sup>68</sup> RML processors. The configuration details and the raw data results are made available online<sup>69</sup>.

*Morph-kgc* was chosen for evaluation due to its state-of-the-art performance and scalability with respect to other RML mapping processors [Arenas22]. We executed the *morph-kgc* processor in both parallel (*morph-kgc-p*) and sequential (*morph-kgc*) modes. We generated a set of templates using MTL that followed the same mapping rules defined for the RML mappings, but that could be executed directly via the *mapping-template* tool. To evaluate the impact of join conditions on the *mapping-template*, we defined two types of mapping templates: (i) the first one performs join operations between data frames (*mapping-template*) (ii) the second one leverages the generation of corresponding IRIs to obtain the same output without performing join operations (*mapping-template-nj*).

For the evaluation, we measured execution time (with a timeout of 24 hours) and the maximum memory usage (with each processor running inside a Docker container limited to 64GB memory). The experiments were conducted on a virtual machine equipped with 12 Intel(R) Xeon(R) E-2136 CPUs running at 3.30GHz, along with 128 GB RAM and SSD

---

<sup>67</sup> <https://github.com/oeg-upm/gtfs-bench>

<sup>68</sup> <https://github.com/morph-kgc/morph-kgc/releases/tag/2.3.1>

<sup>69</sup> <https://github.com/cefriel/mapping-template-eval/tree/main/engines-compare>

storage. Figure 8-1 illustrates the metrics recorded for each configuration, with each test repeated three times.

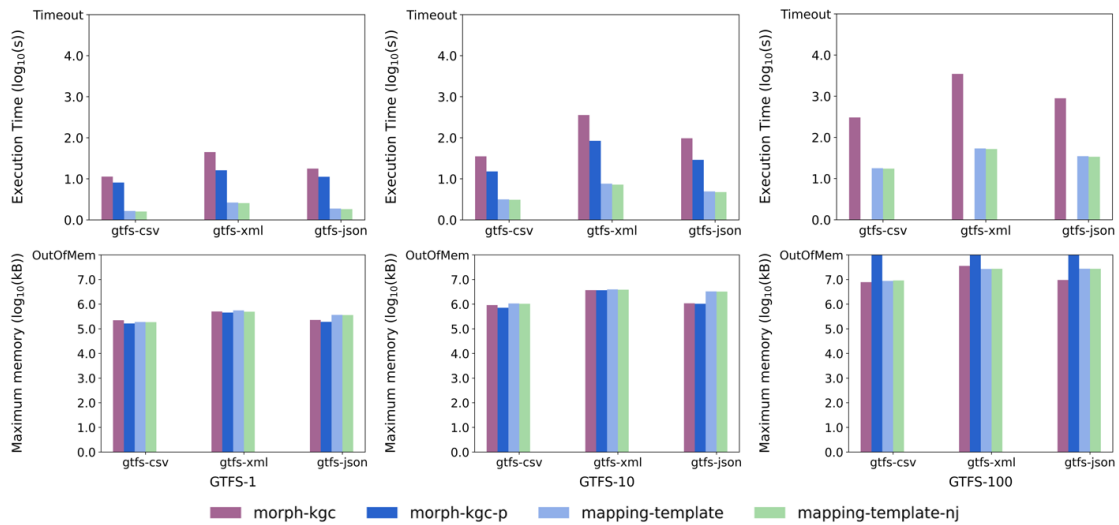


Figure 8-1: Evaluation on the GTFS Madrid Benchmark between mapping-template and morph-kgc

The findings indicate that the *mapping-template* tool executes the task with reduced execution time while demonstrating comparable memory consumption across all three data formats. In contrast, the *morph-kgc* with parallel processing encountered memory issues when handling input data at scale 100. Notably, while adding join conditions typically impacts the performance of processors using RML, the metrics for the *mapping-template* tool showed minimal variation during the evaluation.

This outcome can be attributed to the tool's advantage from the effective and optimized execution of templates offered by the Velocity Engine. However, as highlighted by the test cases presented in the following paragraphs, the execution time comes at the cost of higher memory consumption, and this may be problematic when increasing the input size or in the presence of constraints on memory resources available.

Additionally, it's worth noting that the MTL enables users to refine the mapping rules to fit specific mapping scenarios, e.g., minimizing the number of data frames extracted from the input sources. While fully declarative mapping languages aim at introducing these optimisations without explicit modifications to the mapping rules, not all the optimisations may be automatically inferred by an engine by only relying on the mapping rules, i.e., without knowing the actual data on which the set of mapping rules is applied. In these contexts, the flexibility enabled by MTL can lead to great advantages in terms of performance. Currently, an extension of RML is being investigated to allow users to specify data access methods for improved performance declaratively [Vleeschauer24].

Lastly, it's important to mention that although the inputs and mappings used in the evaluation did not produce duplicate triples, the execution time for *morph-kgc* could still be affected due to its inherent implementation that ensures the elimination of duplicate triples before output serialization.



## 8.2 KNOWLEDGE GRAPH CONSTRUCTION CHALLENGE

To further examine and compare the performance of the *mapping-template* tool against other mapping processors, we participated in track 2 of the Knowledge Graph Construction Challenge 2024<sup>70</sup>. This track focused on performance comparison by requiring each tool to convert input data sources to RDF according to specific RML mapping rules. The first part of the challenge utilized the GTF5-Madrid-Bench to evaluate the tools' behaviour with varying scales of the same data sources (1, 10, 100, 1000) and incorporated diverse combinations of data source types (tabular, files, nested, mixed). The second phase focused on various parameters that could influence the mapping process, establishing different test cases by altering the number of data records, properties, duplicate values, empty values, mapping rules (*PredicateObjectMaps*), as well as join operations. Although different types of joins were tested, we do not report those results here as no significant differences were observed. The organizers provided a tool for reproducible execution of the challenge, along with metric collection and the resulting outcomes. Each participant received the same virtual machine with the following specifications: 4 vCPUs, 16 GB RAM, 130 GB HD, running Ubuntu OS. The complete testing specifications and the set of raw results from each tool are available on Zenodo<sup>71</sup>.

We took part in the challenge before the introduction of direct support for RML mappings in the *mapping-template* tool. Consequently, we manually created an MTL template for a limited set of test cases to carry out the same knowledge graph construction task. We did not explore test cases that varied the number of joins and mapping rules due to the need to manually adapt numerous mapping files. Despite the penalty for executing a limited set of test cases, the *mapping-template* tool secured third place overall in the challenge. Notably, the *mapping-template* tool excelled in execution time and CPU usage, though it did not perform as well in terms of memory consumption.

To achieve a comprehensive performance comparison, we supplemented the challenge results by executing the same test cases with the updated version of the *mapping-template*, providing the RML mapping files directly as input. The evaluation configuration and raw results are available online<sup>72</sup>. In the following sections, we will report and discuss the challenge results alongside the ones recorded afterwards executing the *mapping-template* with RML mappings (referred to as *mapping-template-rml* in the figures). Each test was conducted five times, and we present the results for the other three mapping engines involved in both parts of track 2: FlexRML<sup>73</sup>, RPT/Sansa<sup>74</sup>, and RML-Streamer<sup>75</sup> with RML-view-to-CSV [Vleeschauwer24].

---

<sup>70</sup> <https://kg-construct.github.io/workshop/2024/challenge.html>

<sup>71</sup> <https://zenodo.org/records/11577087>

<sup>72</sup> <https://github.com/cefriel/mapping-template-eval/tree/main/kgc-challenge-2024>

<sup>73</sup> <https://github.com/wintechis/flex-rml>

<sup>74</sup> <https://github.com/Scaseco/R2-RML-Toolkit>

<sup>75</sup> <https://github.com/RMLio/RMLStreamer>

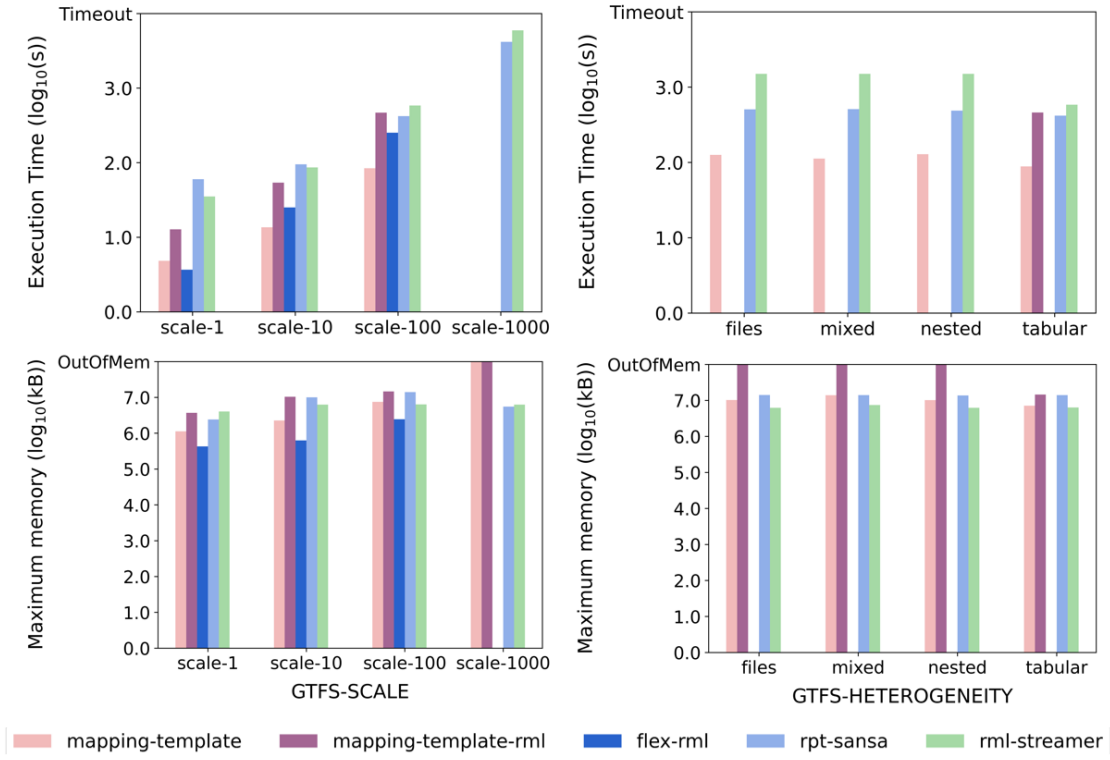


Figure 8-2 shows the results for the first part related to the GTFSS-Madrid-Bench. The performance of *mapping-template-rml* is not directly comparable to that of the *mapping-template* when executing MTL mappings. This disparity arises from the necessity to incorporate additional checks for accurate output generation in the generic case of a translation from RML to MTL. In this case, several optimisations are not applied, as done instead during the manual definition of templates.

In the heterogeneity tests, *mapping-template-rml* failed to run three tests due to the use of a JSON file for the Shapes file, which led to out-of-memory errors due to the challenge of optimizing multiple JSONPath accesses to the input file.

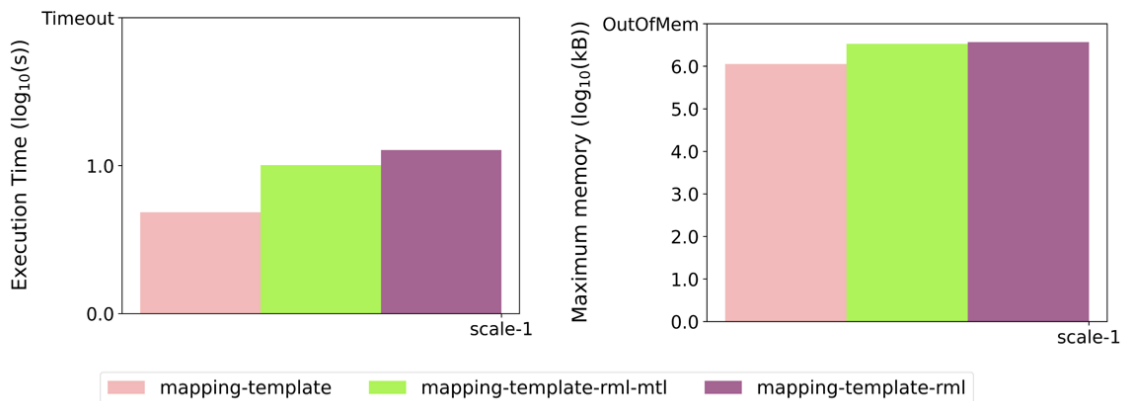


Figure 8-3 illustrate the GTFSS-Madrid-Bench results for scale-1, incorporating metrics from a case (*mapping-template-rm-mtl*) in which an MTL template generated from RML

is executed directly with the tool. This case eliminates the translation overhead that is usually introduced when providing RML mappings directly to the *mapping-template* tool. This diagram indicates that performance differences cannot be solely attributed to this overhead, even with small input files.

Figure 8-4 and Figure 8-5 detail the results for all other test cases influenced by different parameters affecting knowledge graph construction. The previously mentioned trends are observable across these test cases. Overall, it is evident that the *mapping-template* delivers good execution times, although it struggles with memory optimization. Furthermore, the tool performs better with smaller input sizes. Results from executing manually defined MTL templates directly highlight the benefits of tailoring mapping templates to specific mapping scenarios.

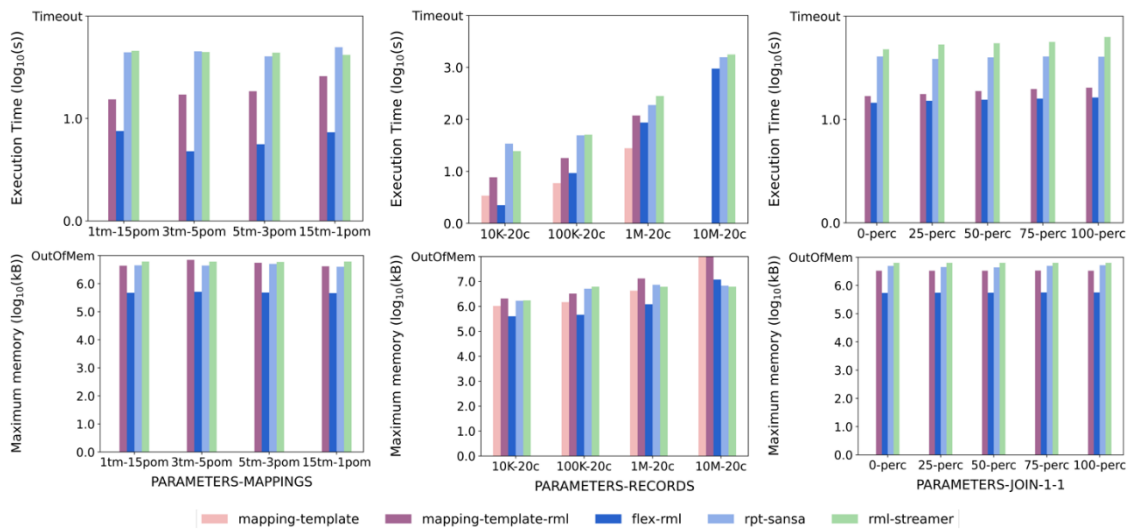


Figure 8-4: Evaluation on the KGCW Challenge for mappings, records, join parameters

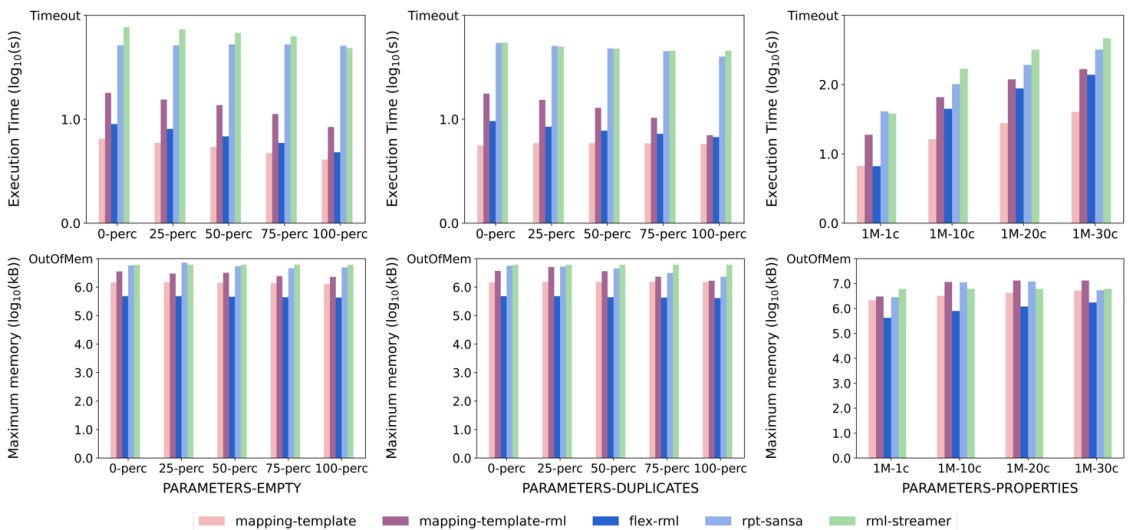


Figure 8-5: Evaluation on the KGCW Challenge for empty values, duplicates and properties parameters

## 9 ANNEX III – EVALUATION OF UC2 DATAOPS PIPELINE ON DIFFERENT DEPLOYMENT TEMPLATES

This annex provides visualisations to compare the behaviour of different deployments of the DataOps pipeline discussed in the evaluation in Section 3.3.1.

### 9.1 CONVERSION TIME AND INPUT SIZE

For the evaluation of conversion time and input size, we visualize the metrics trend over time in the reported graphs. The metrics are normalized between 0 and 1 to show the overall trends.

It can be noticed that the input size varies continuously and there is no specific trend. Nevertheless, the average value is, in all cases, a medium value between maximum and minimum.

On the conversion time, Temurin recorded values with lower variance but greater spikes not correlated with the input size. GraalVM recorded more stable values on average.

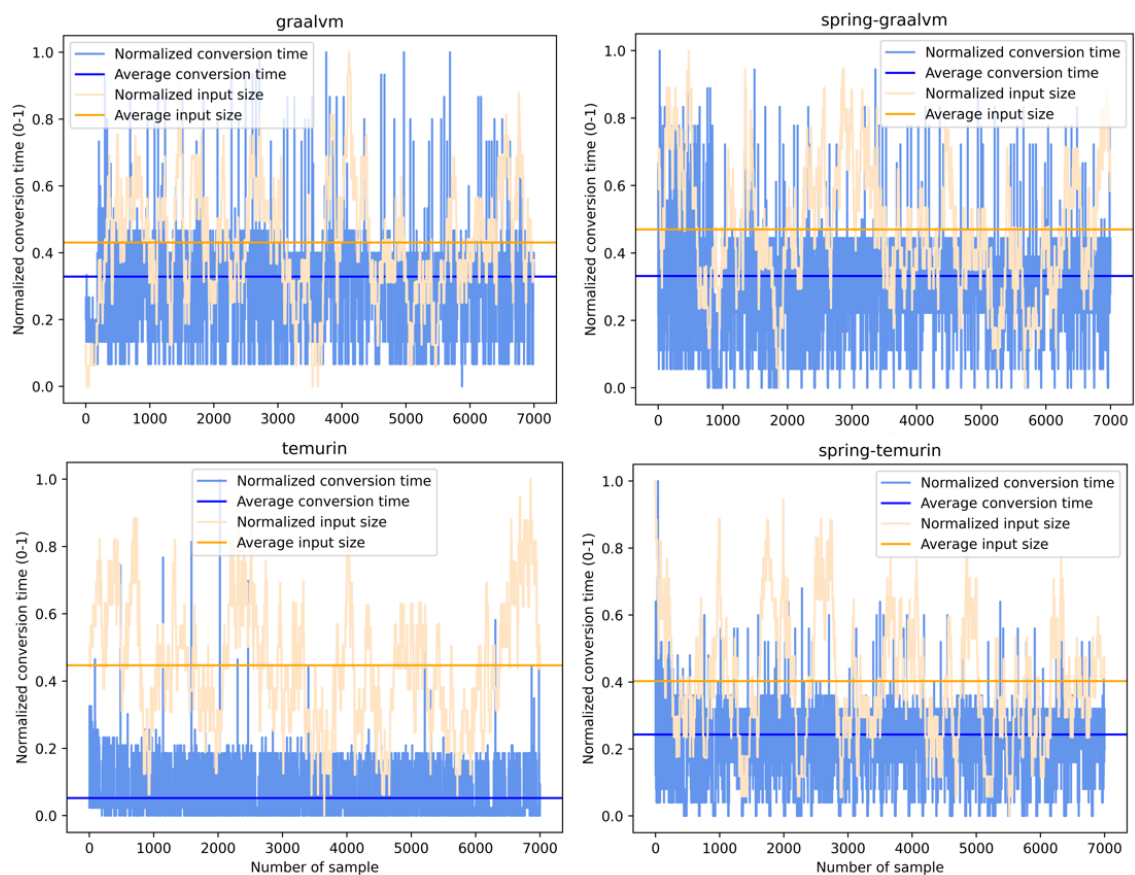


Figure 9-1: Comparison of conversion time and input size over time for Temurin and GraalVM

Native images recorded higher variance in the conversion time despite keeping the average much lower than the maximum values registered.

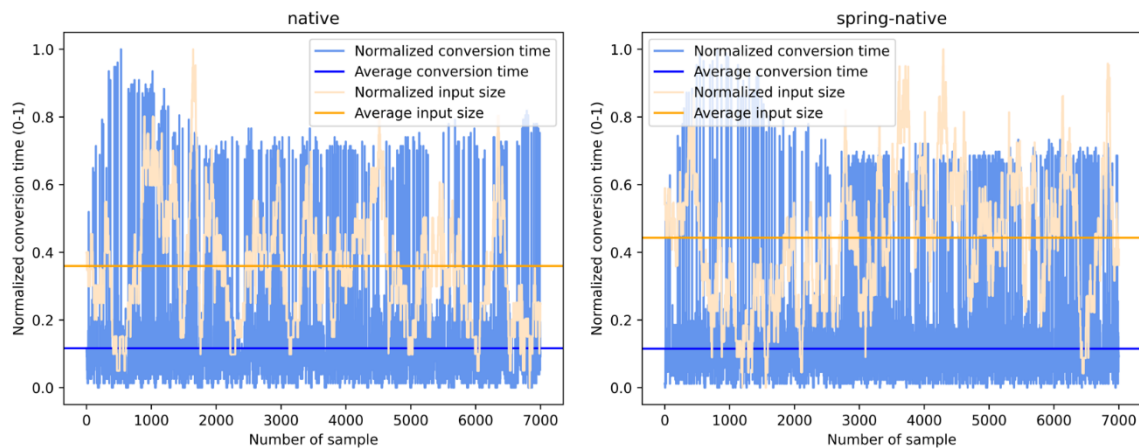


Figure 9-2: Comparison of conversion time and input size over time for Native

## 9.2 MEMORY AND CPU UTILISATION

These visualizations comprehensively compare memory and CPU consumption across the different Docker images.

The following comparisons of CPU/Memory utilization are presented:

- Temurin – GraalVM – GraalVM-Native
- Spring-Temurin – Spring-GraalVM – Spring-GraalVM-Native
- Temurin – Spring-Temurin
- GraalVM – Spring-GraalVM
- Native – Spring-Native

### 9.2.1 Temurin – GraalVM – GraalVM-Native

The test results indicate that for Docker images using the core version, CPU performance for Temurin and GraalVM is substantially similar and slightly better compared to the Native version. Additionally, all of them exhibit an initial spike. Regarding memory, Temurin and GraalVM images also have the same performance. However, there is a noticeable improvement in the Native version for the memory usage with respect to the other images.

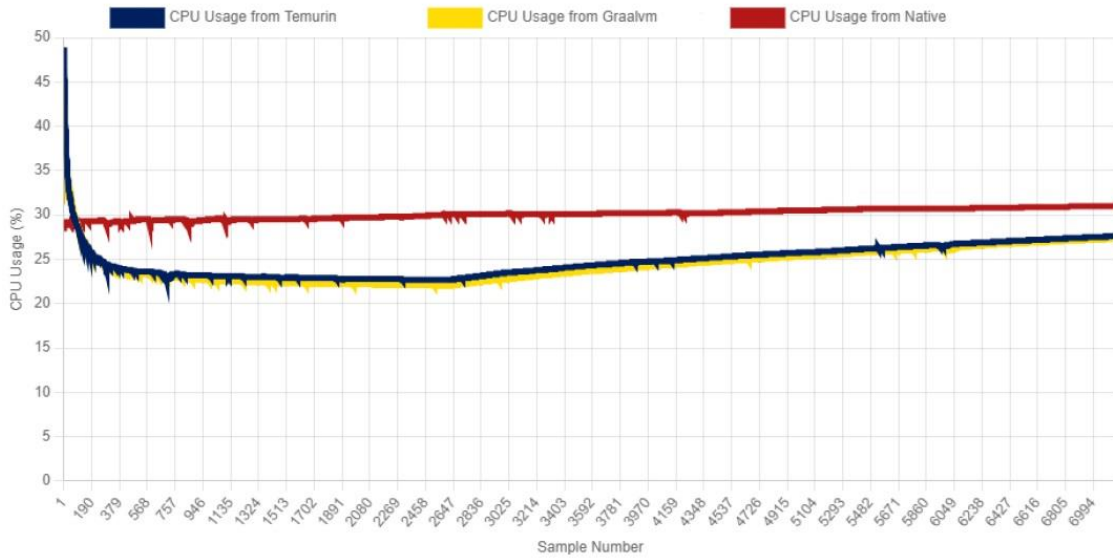


Figure 9-3: CPU Comparison of Temurin, GraalVM and Native

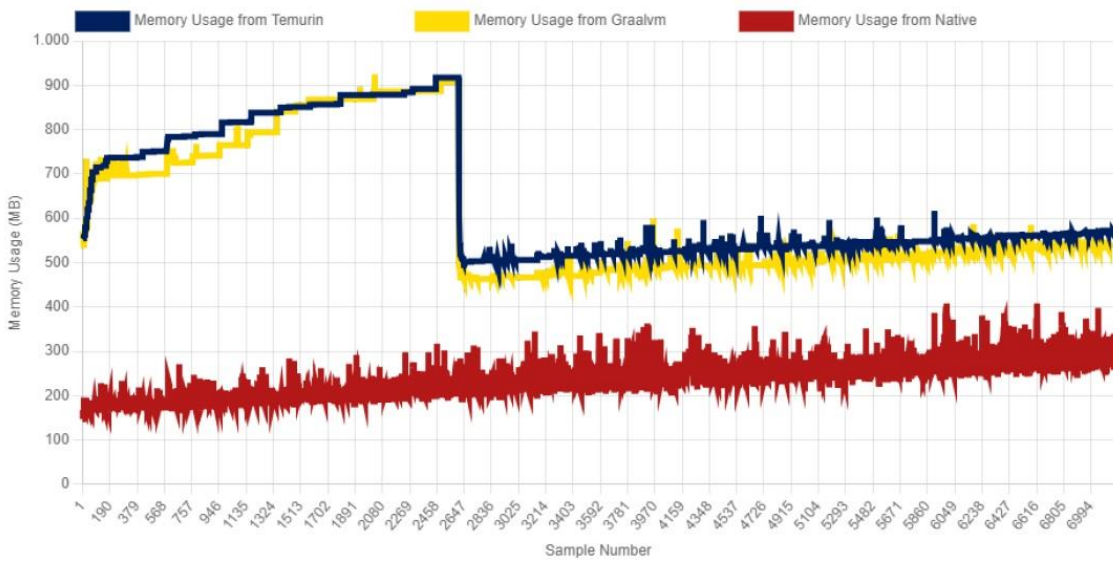


Figure 9-4: Memory Comparison of Temurin, GraalVM and Native

### 9.2.2 Spring-Temurin – Spring-GraalVM – Spring-GraalVM-Native

As shown in the graphs below, the CPU performance of the three tested Docker Spring images is substantially similar, and all exhibit an initial spike that is much more pronounced for the core version than the native version. In terms of relative Memory comparison, however, the Spring GraalVM version shows slightly better performance compared to the core version. It is, however, very evident that the native version exhibits a much lower memory usage compared to the other two images.

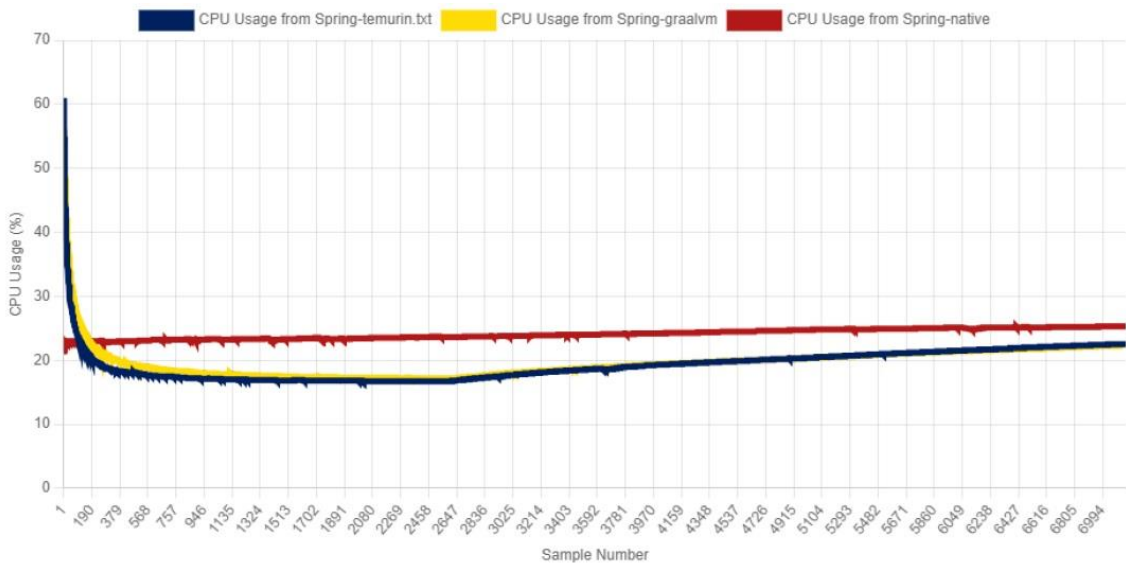


Figure 9-5: CPU Comparison of Spring Temurin, Spring GraalVM and Spring Native images

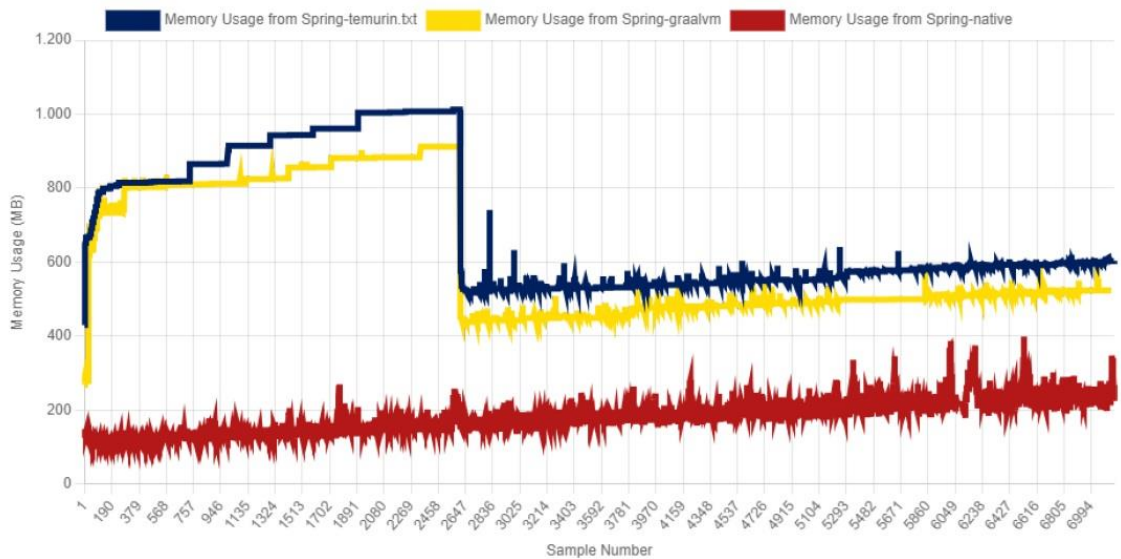


Figure 9-6: Memory Comparison of Spring Temurin, Spring GraalVM and Spring Native images

### 9.2.3 Temurin – Spring-Temurin

The results of this comparison demonstrate that no significant differences were found between the core and spring versions of the Temurin image. The core version exhibits slightly better performance in terms of memory usage, while the spring version shows slightly better behaviour in terms of CPU usage, although it exhibits a much more pronounced initial spike compared to the core version.

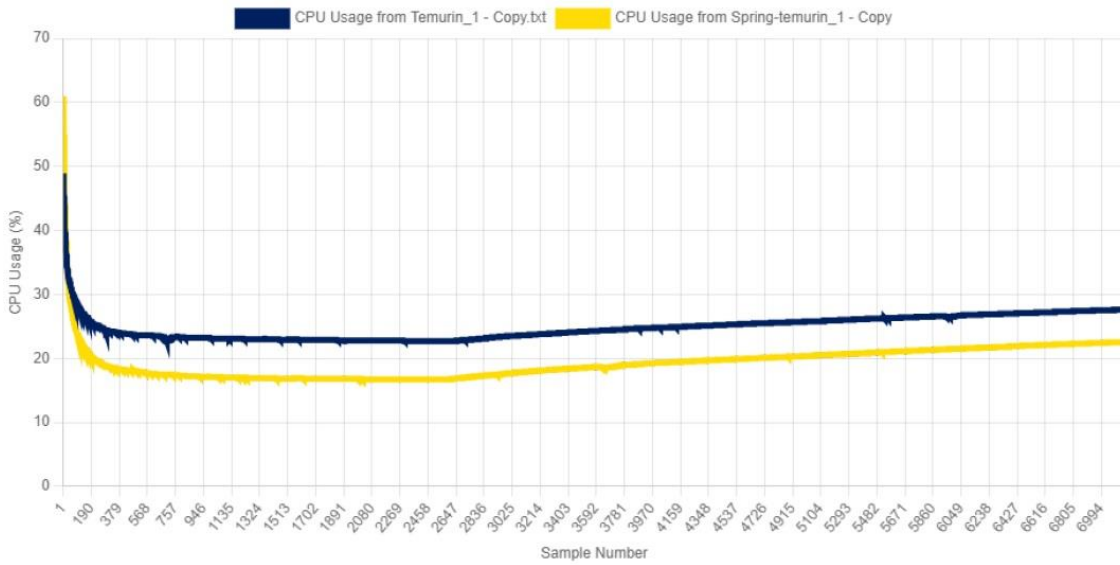


Figure 9-7: CPU Comparison of Temurin and Spring Temurin

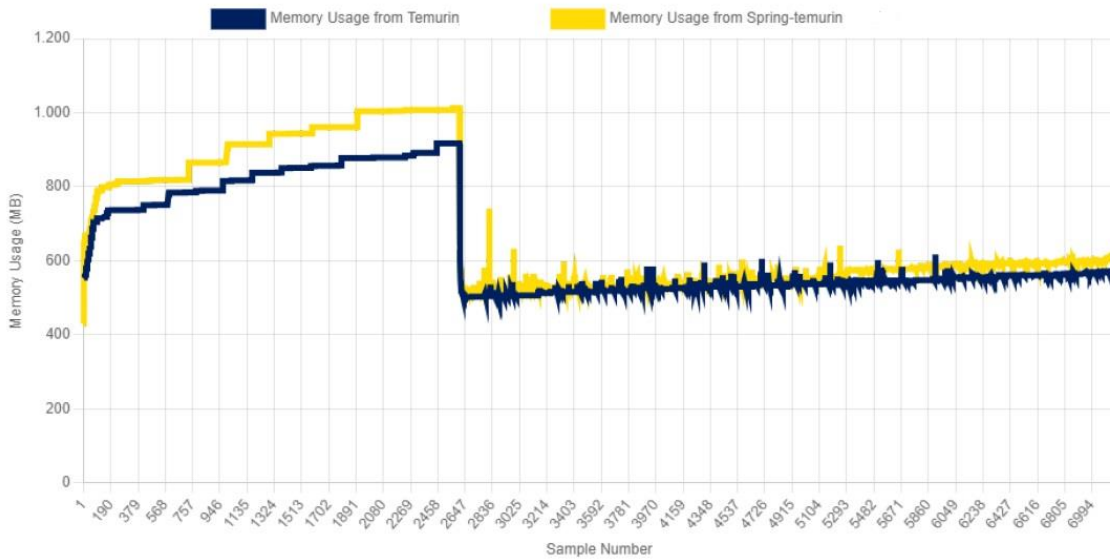


Figure 9-8: Memory Comparison of Temurin and Spring Temurin

9.2.4 GraalVM – Spring-GraalVM

In this test, as with the previous one, no substantial differences are highlighted between the two tested Docker images. However, also in this case, a slightly better performance in terms of memory is noted for the core version and a slightly better behaviour in terms of CPU for the Spring version, which also in this case shows a more pronounced initial spike compared to the core version.



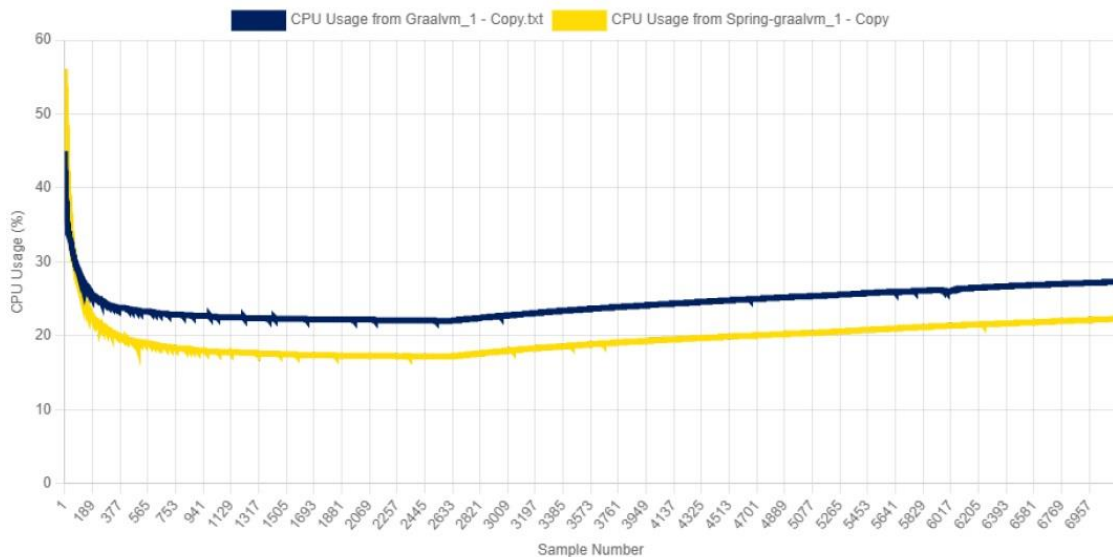


Figure 9-9: CPU Comparison GraalVM and Spring GraalVM

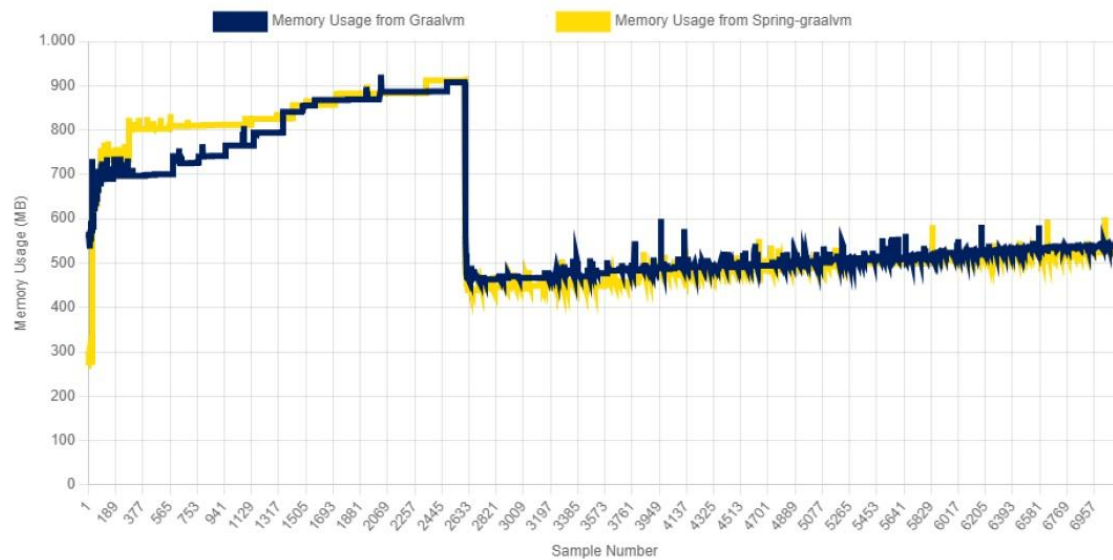


Figure 9-10: Memory Comparison GraalVM and Spring GraalVM

### 9.2.5 Native – Spring-Native

Regarding the comparison of native versions, as can be easily seen in the graphs below, the native spring image shows better performance in terms of both memory and CPU compared to the native core version. However, as with all other cases, even for the native image, the spring version experiences a much higher initial CPU spike compared to the native core version.

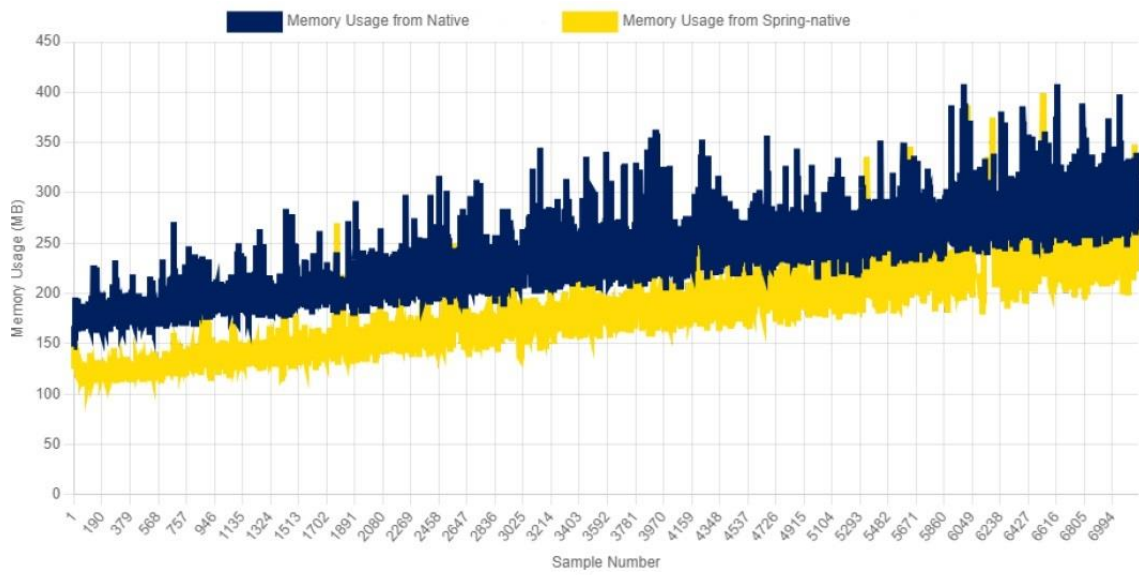


Figure 9-12: Memory comparison Native and Spring Native