SmartEdge

Deliverable D5.2

First implementation of low-code programming tools for edge intelligence

Lead Editor Trung Kien Tran (BOSCH)

Contributors M. Bagheri (CONV), L. Bassbouss (FhG), D Bowden (DELL), P. Cudre-Mauroux (FRIB), K. Dorofeev (SAG), D. Anicic (SAG), A. Ganbarov (TUB), M. Grassi (CEF), X. Guo (TUB), I. Kosonen (AALTO), A. Le-Tuan (TUB), M. Nguyen-Duc (TUB), G. Michelangelo, F. Cugini (CNIT), M. Milich (BOSCH), A. Paul (FhG), L. Bassbouss (FhG), A. Zoubarev (FhG), S. Paul (TUB), E. Petrova (IMC), D. Raggett(W3C), M. Scrocca (CEF), D. Tran(BOSCH), TK. Tran, L. Halilaj (BOSCH), J. Yuan (TUB), N. Zilberman (UOXF)

Version 4.1

Date 19.12.2024

Distribution PU

Semantic Low-code Programing Tools for Edge Intelligence

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

DISCLAIMER

REVISION HISTORY

Revision	*Date*	*Responsible*	*Comment*
0.1	*01/04/2024*	*BOSCH*	*Layout and Structure*
0.2	*01/05/2024*	*BOSCH*	*Initial content*
0.3	*01/07/2024*	*BOSCH*	*Updated layout and structure*
0.4	*01/09/2024*	*BOSCH*	*50% of content*
0.5	*01/10/2024*	*BOSCH*	*75% of content*
0.6	*01/11/2024*	*BOSCH*	*95% of content*
1.0	*07/11/2023*	*BOSCH*	*All content and assigned reviewers*
2.0	*21/11/2024*	*BOSCH*	*Revised version*
3.0	*30/11/2024*	*BOSCH*	*Revised version with new section numbers*
4.0	*09/12/2024*	*BOSCH*	*Submitted for Quality Review*
4.1	*19/12/2024*	*BOSCH*	*Revised version after quality check*

GLOSSARY

Acronym	*Description*
ABR	*Adaptive Bitrate Streaming*
AMR	*Autonomous Mobile Robot*
API	*Application Program Interface*
ARM	*Acorn Reduced Instruction Set Machine*
CAD	*Computer Aided Design*
CAN bus	*Controller Area Network bus*
CaS	*Compare and Swap*
CCTV	*Closed Circuit Television*
CDN	*Content Delivery Network*
CNN	*Convolutional Neural Network*
COCO	*Common Object in Context*
CPU	*Central Processing Unit*
CQELS	*Continuous Query Evaluation over Linked Streams*
CXL	*Compute Express Link*
DASH	*Dynamic Adaptive Bitrate over HTTP*
DCAT	*Data Catalog vocabulary*
DDS	*Data Distribution Services*
DETR	*Detection Transformer*
DKG	*Dynamic Knowledge Graph*
DMA	*Direct Memory Access*
DPU	*Data Processing Units*
D-RDMA	*Declarative Remote Direct Memory Access*

DSL	*Domain-specific Language*
FaA	*Fetch and Add*
FAIR	*Findability, Accessibility, Interoperability, and Reusability of digital asset*
FPGA	*Field Programmable Gate Arrays*
FRCNN	*Fast Region-based Convolutional Neural Network*
GDS	Global Data Space
GeoSPARQL	*Resource Description Frame geospatial query language*
GPM	*Graph Pattern Mining*
GPS	*Global Positioning System*
GPU	*Graphical Processing Unit*
HLS	*Hypertext Transfer Protocol Live Streaming*
HPC	*High-performance Computing*
HTTP	*Hypertext Transfer Protocol*
IDM	*Identity Management*
IMU	*Inertial Measurement Unit*
IoT	*Internet of Things*
IoU	*Intersection over Union*
IRI	*Internationalized Resource Identifier*
ISD	*Integrated Surface Dataset*
iWARP	*Internet Wide Area Remote Direct Memory Access Protocol*
JPEG	*Joint Photographic Experts Group image format*
JSON	*JavaScript Object Notation*
JSON-LD	*JavaScript Object Notation Linked Data*
JVM	*Java Virtual Machine*
LAN	*Local Area Network*
LiDAR	*Light Detection and Ranging*
LLM	*Large Language Model*
MAC	*Media Access Control*
MAT	*Match-action Tables*
MAU	*Match-action Unit*
MEMS	*Micro-Electro-Mechanical Systems*
ML	*Machine Learning*
MOM	*Message-Oriented Middleware*
NARF	*Normal Aligned Radial Feature*
NAS	*Neural Architecture Search*
NAV	*ROS Navigation Stack*
NCDC	*National Climatic Data Center*
NCR	*Non-contiguous Regions*
NIC	*Network Interface Card*
OMG	*Object Management Group*
ONOS	*Open Network Open System*
OBU	*On-board Unit (V2X wireless communication hardware inside a connected vehicle)*
P2P	*Peer-2-peer*

P4	*Programming Protocol-independent Packet Processors*
PNG	*Portable Network Graphics*
QET	*Query Execution Time*
QoS	*Quality of Service*
QP	*Queue Pairs related to Remote Direct Memory Access*
QR	*Quick Response an evolution of bar codes*
RAM	*Random Access Memory*
RCNN	*Region-based Convolutional Neural Network*
RTSP	*Real-Time Streaming Protocol*
RDF	*Resource Description Frame*
RDMA	*Remote Direct Memory Access*
REST	*Representational State Transfer*
RGB	*Red, Green, Blue referring to color images*
RGBD	*Red, Green, Blue, Depth referring to color images with a depth channel*
RML	*Resource Description Frame Mapping Language*
RoCE	*Remote Direct Memory Access over Converged Ethernet*
ROS	*Robot Operating System*
RPC	*Remote Procedure Call*
RSU	*Road-side Unit*
SGE	*Scatter-gather Element*
SHACL	*Shapes Constraint Language*
SLAM	*Simultaneous Localization and Mapping*
SPARQL	*Resource Description Frame query language*
TCP	*Transmission Control Protocol*
TD	*Thing Description part of the WoF*
TDD	*Thing Description Directory part of the WoF*
TTL	*Time-to-live*
UDP	*User Datagram Protocol*
URDF	*Unified Robot Description Format*
URI	*Uniform Resource Identifiers*
USB	*Universal Serial Bus*
UUID	*Universally Unique Identifier*
V2X	*Vehicle to everything*
WebRTC	*Web Real-Time Communication*
WoF	*Web of Things*
WR	*Work Request*
YOLO	*You Only Look Once a common object detector*

EXECUTIVE SUMMARY

Deliverable D5.2 reports the first implementation along with the revised design of low-code programming tools for Edge Intelligence in the SmartEdge project folowing the desing reported in D5.1. The document focuses the first implementation and the next design of four parts of the toolchain corresponding to four tasks of WP5: 1) Semantic-driven Multi-modal sensor fusion for edge devices; 2) Swarm Elasticity via Cloud-Edge Interplay; 3) Adaptive Coordination and Optimization; 4) Cross-layer toolchain for Device-Edge-Cloud Continuum.

WP5 aims to provide an integrated toolchain to lower the effort in building Edge Intelligence. Based on semantic descriptions of sensing and computing capabilities as well as data queries, the toolchain will decouple the application logic to underlying complicated software, hardware and networking elements. On the other hand, the semantic descriptions and specifications are the key enabler to integrate the elements into execution pipelines at run time without a prior knowledge of them. Hence, the semantic data model is the unified data presentation as the integration point for all components designed in D5.1. Firstly, the sensor fusion of multimodal data of T5.1 will use RDF as the intermediate data representation among the operations that provide the unified input/output data presentations to operators that can be processed and integrated in T5.2, T5.3 and T5.4. Secondly, the declarative programming approach for low-code programming across the layers (e.g, network, RDMA, sensor fusion, orchestration, optimization and runtime) can provide different domain-specific languages (DSLs) that be seamlessly integrated via RDF data model and graph query patterns. In particular, T5.2 allows T5.1 to offload their sensor fusion operations that can be expressed a graph query patterns and similarly T5.3 also can orchestrate the federated processing workloads represented in SPARQL-like query languages. Eventually, the SmartEdge runtime pushes one step further in using RDF data as dynamic knowledge graphs (DKGs) that unifies traditional knowledge graphs and semantic streams. DKGs help to integrate sensory data from T5.1 with operational and environment data, e.g. network telemetries, hardware configuration, training data, host environments, into queryable form with graph query language like SPARQL so that SmartEdge nodes can programmatically access it in a unified via in a distributed fashion.

Table of Contents

Table of Figures

1 INTRODUCTION

D5.2 reports the first implementation of the design of the low-programming tools for edge intelligence (D5.1). In the first implementation of such tool, we report implemented artifacts grouped by tasks as following: i) Artifacts related to semantic-based multimodal sensor fusion for edge devices (Section 2 for T5.1), ii) Artifacts related to Swarm elasticity via edge-cloud interplay (Section 3 for T5.2); iii) Artifacts related to Adaptive Coordinator and Optimization (Section 4 for T5.3); iv) Artifacts related to Cross-layer toolchain for device-edge-cloud continuum (Section 5 for T5.4). The design of such tools employs the same approach for low-code programming, declarative programming (cf Section 1.1 of D5.1). The artifacts of such tools along with the ones produced in WP3 and WP4 will be integrated and deployed as a runtime for a SmartEdge Swarm node as illustrated in following Figure 1-1.

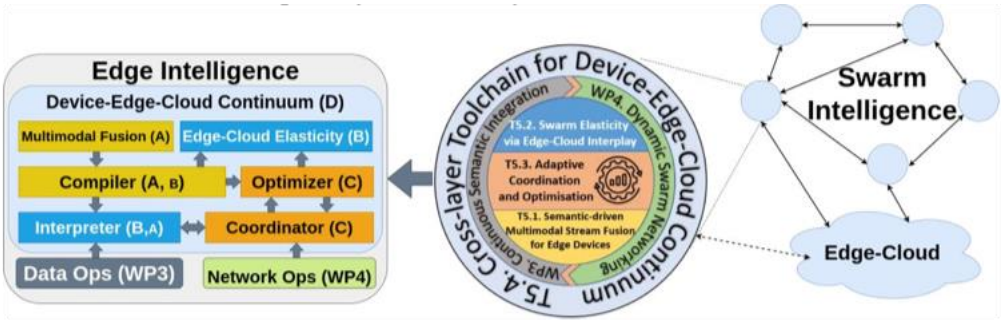


Figure 1-1. From SmartEdge low-code toolchain to SmartEdge runtime (from D5.1)

Recall from D5.1 that the hallmark of low-code programming of SmartEdge lies in its declarative programming model, an approach that introduces delarative query languages or DSLs so that domain expert could specify the application logics according to domain-spefic workflows without having to write imperative code in C++/Java/Python for most underlying software components. At its core, D5.2 realizes this model by using Dynamic Knowledge Graphs (DKG) to interlink domain-expert knowledge with input data and necessary components. DKG is based on data schema provided in WP3 and extended to UCs with the integration with knowledge base to represent runtime context and networking configuration. DKG serves as the foundation for constructing runtime instances and workflows in artifacts of T5.3 and T5.4 that edge devices can execute or interpret.

The domain experts in SmartEdge are UC owners, traffic-, car-, or robot-engineers who will declare the 'what' a swarm of edge nodes should do via Domain-Specific Languages (DSLs) that embody semantics familiar to their application domains, e.g. traffic or factory floor, called Semantic DSLs. Such DSLs are constructed from standardized data models and ontologies, e.g. RDF, SPARQL, SHACL and Recipes (cf. D3.1). By reducing the complexity of traditional programming constructs, e.g. C++ or python, Semantic DSLs empower domain experts to directly contribute to the development process, thus mitigating the necessity for in-depth programming expertise in Data Ops, Network Ops and AIOps.

Furthermore, SmartEdge low-code tools are not solely focused on the high-level application design; they also strive to enhance the productivity of developers working at the lower layers of the technology stack of Edge Intelligence, e.g. such as networking programming, Remote Direct Memory Access (RDMA), and C++/python for ROS. By automating the time-consuming and

monotonous tasks traditionally performed manually, low-code toolchain aims to reduce the potential for error and the overall burden on developers.

1.1 RELATION OF WP5 TO OTHER WPS

WP5 provides tools and solutions to perform *smart* swarm operations supported by semantic-driven multimodal stream fusion components for edge devices, elastic edge-cloud Interplay and smart adaptive coordination and optimization mechanisms for device-edge-cloud continuum. WP5 depends to artifacts from WP4 for establishing network connection in declarative fashion via P4 and employ the semantic-based data integration artifacts of WP3 to transform the data from heterogenous sources, protocols and formats into standardized RDF formats driven by SmartEdge Schema (A3.1).

The data transformation modules provide in artifacts A3.5 and A3.6 will be used to feed data in RDF into SmartEdge Runtime A5.4.1. The artifact A5.4.1 also extends SmartEdge Schema of A3.1 to capture runtime context and inputs/outputs of multimodal stream data sources of artifact A5.1.4. Similarly, the network telemetry streams provided by WP4 are also annotated by vocabularies extended from SmartEdge Schema. Receipt models in A3.2 are the integration points for wiring application specifications to network configuration of WP4 with execution workflow deployed in SmartEdge runtime of WP5.

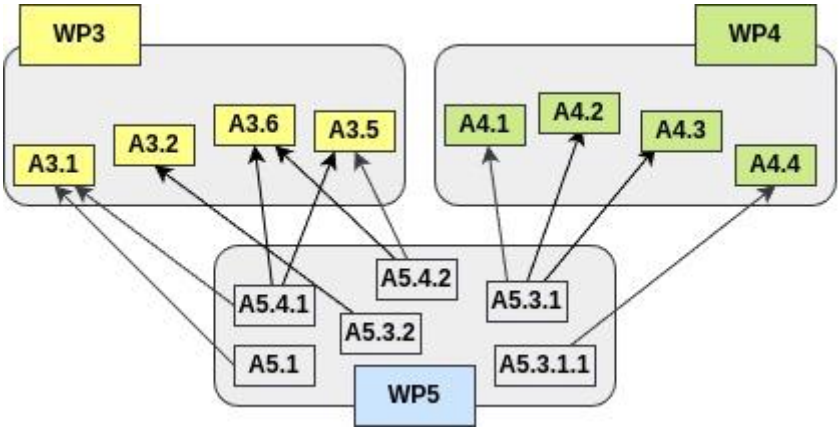


Figure 1-2. Dependencies of Artifacts of WP3, WP4, and WP5

Next, WP4 enables WP5 smart networking capabilities by providing a secure and reliable *networking* solution which includes automatic discovery and dynamic swarm formation operating at the network level.

The Swarm Orchestrator (A5.3.1) developed within WP5 and the Network Swarm Coordinator (A4.2) developed within WP4 are two key artifacts that, following the requirements defined by WP2 (see D2.2), guarantee that the smart swarm operations are effectively supported by adequate configurations at the network level. For example, with reference to Figure 1-3, the orchestrator by WP5 is in charge of considering the available set of skills of the available swarm nodes and to form a swarm recipe. Then, the swarm coordinator by WP4 receives recipe requirements in terms of the number of swarm nodes needed and their requested capabilities. The coordinator then, operating at the network layer, performs a look up in the Address Resolution Table and finds matching nodes that are currently available to join a swarm. The network layer by WP4 then makes sure the connectivity is established properly, guaranteeing

secure networking and isolation. Additional details are reported in D4.2, which also includes specific examples of interaction between WP4 and WP5 components as well as the detailed list of WP2 requirements, particularly in the context of Swarm Management and Communication, clarifying which ones are addressed by WP4 and WP5.

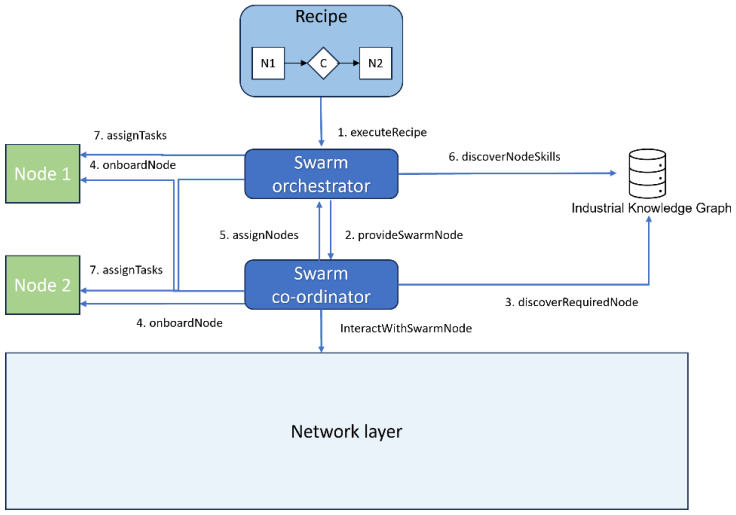


Figure 1-3. Interaction between Recipe (WP3), swarm orchestrator (WP5) and Network Swarm Coordinator (WP4) for the formation of the swarm [D3.1]

1.2 ARTIFACTS FOR LOW-CODE PROGRAMMING TOOLS FOR EDGE INTELLIGENCE

ID	Component	Lead	Section	Description	Status
A5.1	Low Code Programming	BOSCH, CEF	2	Multimodal semantic Stream Fusion	Currently being implemented
A5.1.2.1	Low Code Programming	BOSCH	2.2.1	Vision Scene Understanding for Traffic	First release, continue to release 1.5 and 2.0
A5.1.1.1	Low Code Programming	FhG	2.2.3	Media Stream Processing	Currently being implemented
A5.1.2.2,	Manufacturing scene understanding pipeline	DELL	2.2.2	Manufacturing scene understanding artifact,	Currently being implemented
A5.2.1	Hardware-Accelerated Data Processing	FRIB	3.2.2	Low-level component taking advantage of hardware to accelerate data processing	Currently being implemented
A5.2.2	Runtime Optimizer	FRIB	3.2.3	Hardware-accelerated	Currently being implemented

				component optimizing complex workloads	
A5.3.1	Swarm Adaptive Coordinator	TUB	4.2.1	To form a swarm and coordinate its nodes.	First release, continue to release 1.5 and 2.0
A5.3.2	Swarm Dynamic Orchestrator	TUB	4.2.2	To orchestrate the tasks among swarm members.	First release, continue to release 1.5 and 2.0
A5.3.3	Swarm Optimizer	TUB	4.2.3	To optimize the performance and resource consumption of a swarm.	Currently being implemented
A5.4.1	SmartEdge Runtime	TUB	5.2.1	Runtime to execute low-code tool chain	First release, continue to release 1.5 and 2.0
A5.4.3	SmartEdge plugins	TUB	5.2.2	Plugins as sub artifacts (A5.4.3.x) to extend capabilities of Runtime A5.4.1	First release with first versions of A5.4.3.1-4. Other plugins and next version to be released in 2.0
A5.4.4	Low-code IDE	TUB	5.3.3	An IDE help to lower the effort to write code and interact with the SmartEdge tool chains via GUIs and interactive workflow	Four sub-artifacts (A5.4.4.1, A5.4.4.2,A5.4.4.3, A5.4.4.4) are included in the first release, the completed/advanced versions will be included in release 2.0

Table 1-1. The list of artifacts and their status for WP5

1.3 MAPPING KPIS TO ARTIFACTS

The following table presents the overview of the relation between artifacts and KPIs. Further details are presented in D6.1.

ID	Descriptions	Related Artifacts
K4.1	Ability to free developers from specifying capabilities of hardware and sensors at the design phase of stream fusion pipelines with end-to-end latency guarantee (e.g., 20-75% lower latency to baselines [DTH+21, PET21]	A5.1.1, A5.4.4.3 A5.1.2.1, A5.1.2.2

K4.2	Ability to elastically scale 200-500% better than the state of the art, e.g., [Cudre-Mauroux13, Duc21, Schneider22].	A5.2.1 A5.2.2
K4.3	Can dynamically optimize resource to be 50%-150% better in terms of computing resources and bandwidths;	Swarm Optimizer (A5.3.3), Adaptive Coordinator (A5.3.1), Dynamic Orchestrator (A5.3.2)
K4.4	Lower the effort in building swarm intelligence with the target of reducing the coding effort in comparison to imperative programming paradigms by 80-90%, e.g., Python or C++, with SMARTEDGE low-code tool chain.	SmartEdge Runtime (A5.4.1) SmartEdge Plugins (A5.4.3): - P4 plugin (A5.4.3.1), - In-Network ML (A5.4.3.2) - Security (A5.4.3.3) - Mendix PlugIn (A5.4.3.4) - Chunk&Rule (A5.4.3.5) Low-code IDE (A5.4.4): - Model Builder (A5.4.4.1) - Metric Report & Visualization (A5.4.4.2) - Remote Rendering (A5.4.4.3) - Ego-vehicle Visualization (A5.4.4.4) - Swarm Visualization (A5.4.4.5)
K4.5	Support AI operations and coordination on a large number of heterogeneous IoT devices (20-50 types of 200-1000 devices) and smart systems (5-10 application domains) to achieve a higher resilience in terms of being able to integrate new sensors and participant nodes at runtime without interrupting the current application logic	Integrated systems that pack SmartEdge Runtime (A5.4.1) in performance related several artifacts, e.g Federator (A5.3.2), Coordinator, Network (A5.3.1) and elastic scale artifacts (A5.2.1, A5.3.3)

Table 1-3. Mapping KPIs to Artifacts

2 SEMANTIC-DRIVEN MULTIMODAL STREAM FUSION FOR EDGE DEVICES

2.1 MAIN COMPONENTS AND FUNCTIONALITIES

In the task T5.1, we create components to integrate multimodal stream data and make it usable for various applications. We will briefly revisit the design of T5.1 for multimodal semantic stream fusion within SmartEdge. Figure 2-1 provides an overview of the initial implementation of this pipeline. As detailed in Deliverable D5.1, this pipeline processes media data, such as camera frames, alongside sensor data, producing a semantic description of these inputs for a range of downstream tasks across different applications.

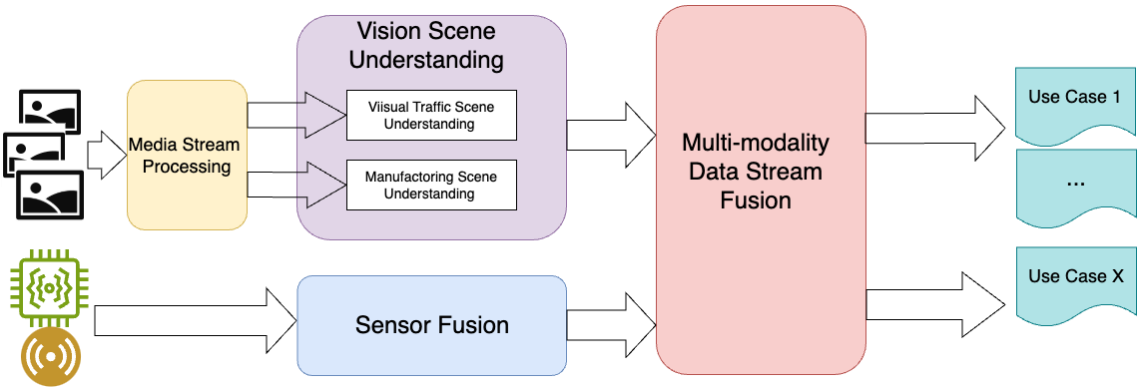


Figure 2-1. Overview of the semantic multimodal stream fusion pipeline.

The Media Stream Processing Component (Section 2.2.3) includes the software necessary to format media streams from devices like 2D cameras and LiDAR cameras on cars or robots for use in data fusion and other applications. The **Sensor Fusion Component (Section 2.2.2)** aggregates sensor data to calculate traffic indicators and perform prediction tasks. The **Vision Scene Understanding (Section 2.2.1)** aims to identify objects, their relationships, and environmental details in a scene. It includes sub-components for traffic scene understanding and manufacturing scene understanding. Finally, the **Data Stream Fusion (Section 2.2.4)** integrates all information, adding semantic descriptions using a unified ontology/schema and semantic web standards like RDF(S) and provides the results/information to the use cases.

2.2 COMPONENTS IMPLEMENTATIONS

2.2.1 Vision Scene Understanding

This component consists of two sub-components Visual Traffic Scene Understanding and Manufacturing Scene Understanding as described in the next sub-sections.

2.2.1.1 Visual Traffic Scene Understanding

2.2.1.1.1 Main Functionalities

In this section, we detail the first implementation of the module “Visual Scene Understanding” in the architecture outlined in Figure 2-1. Overview of the semantic multimodal stream fusion pipeline., which is also documented in the deliverable D5.1.

This module takes video frame from camera as input and provides the corresponding *scene graph* as output. Each element in a video frame is represented as a node in the scene graph and a scene graph is a directed graph constituted by a set of triples of the form (Subject, Predicate/Relation, Object), which describes how the "object node" is related to the "subject node". With that design and functionalities, there are two main components in the module as shown in Figure 2-2, the Object & Motion Detection and Visual Relationship Estimation components.

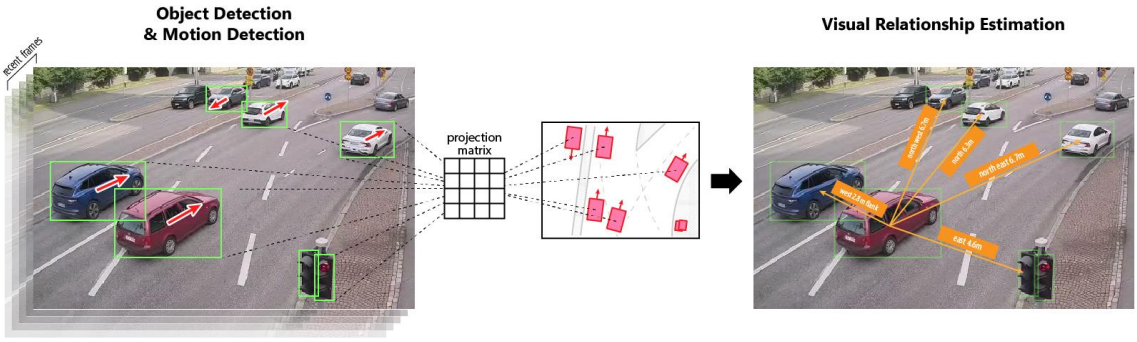


Figure 2-2. The overview of scene graph generation.
The pipeline takes an image as an input and generates a visually grounded scene graph.

Given a video frame, the goal of the system is to generate a directed graph that strictly reflects the semantic relationship between objects within the scene. Using an off-the-shelf object detector, the system first extracts explicit features for objects, specifically their visual features, bounding boxes, and labels. Then, the motion detector detects vehicle movement, providing additional data to enhance the accuracy of subsequent stages. Finally, relation classifiers are used to predict the relationships between each pair of objects, and the scene graph is generated.

2.2.1.1.2 Component Implementations

Object Detection

In this work, we employ YOLOv9 [Wang24], which is an improved architecture and training technique that enhances detection performance, especially in real-time applications. It features more efficient network layers, optimized anchor boxes, and refined loss functions, which together contribute to superior accuracy in detecting and localizing objects in images. The balance of computational efficiency and high precision makes it particularly suitable for deployment on edge devices and in scenarios requiring rapid processing and decision-making. Our objective is to create a comprehensive scene graph by detecting all potential objects within the image, grouping them into pairs, and utilizing the features of their combined area as the fundamental representation for relation estimation.

From the objects that were detected by the object detector, we construct the explicit features for the relationship of each pair of objects in three aspects: visual, spatial, and semantic. Visual features are the CNN features of the two objects. Spatial features are the bounding boxes and coordinates of the two objects which encode their spatial layouts.

Visual Relation Prediction

Many existing approaches for visual relation prediction are slow and resource-intensive, making them impractical for use on edge devices. To address this, we propose a traditional method that

relies solely on arithmetic operations, which are computationally efficient. Our approach begins with estimating the projection matrix from the camera input. Using this matrix, we project the bounding box of all objects into a new space as if they are viewed from above. This top-down projection simplifies the spatial relationships and allows us to calculate angles and distances between objects in the scene more accurately. By reducing computational complexity, our method offers a viable solution for real-time applications on resource-constrained devices.

Motion Detection. For each frame, along with object detection, we also track the motion of each object by mapping it across two consecutive frames, as visualized in Figure 2-3. We maintain a buffer of the N most recent frames and calculate the motion vector for each pair of consecutive frames. These motion vectors are then averaged to obtain the final motion vector for each object. We match objects between frames based on similarity of feature vectors of bounding boxes. It could happen that vehicles change from frame to frame but we can mitigate that by shortening the time between frames. This method ensures a more accurate and stable estimation of object motion by smoothing out short-term fluctuations and providing a clearer understanding of movement patterns over time.

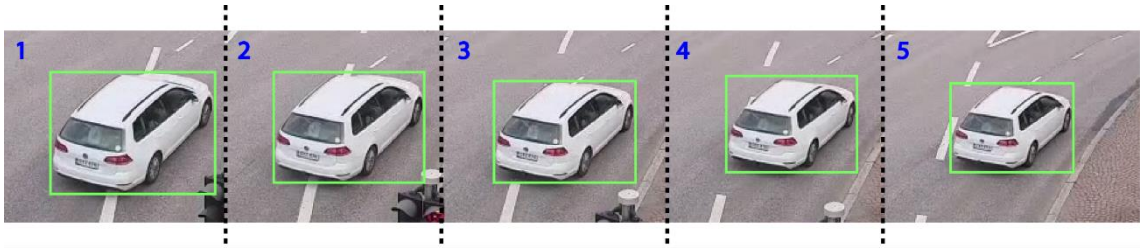


Figure 2-3. Motion detection using bounding boxes matching between frames.

Visual Relationship Estimation. Consider a pair of objects in a scene: we define eight possible directions based on the angle between the two objects. Additionally, we categorize the distances between these objects into five distinct types of ranges. To further classify their interaction, we determine the motion type based on the moving direction of the two objects. If the angle between them is between 0 and 20 degrees, the motion is classified as flank. Conversely, if the angle falls between 160 and 200 degrees, it is classified as approach. This systematic approach allows for precise characterization of spatial and motion relationships between objects in the scene. Figure 2-4 visualizes our definition of angles and distances, as well as the relative position between two objects.

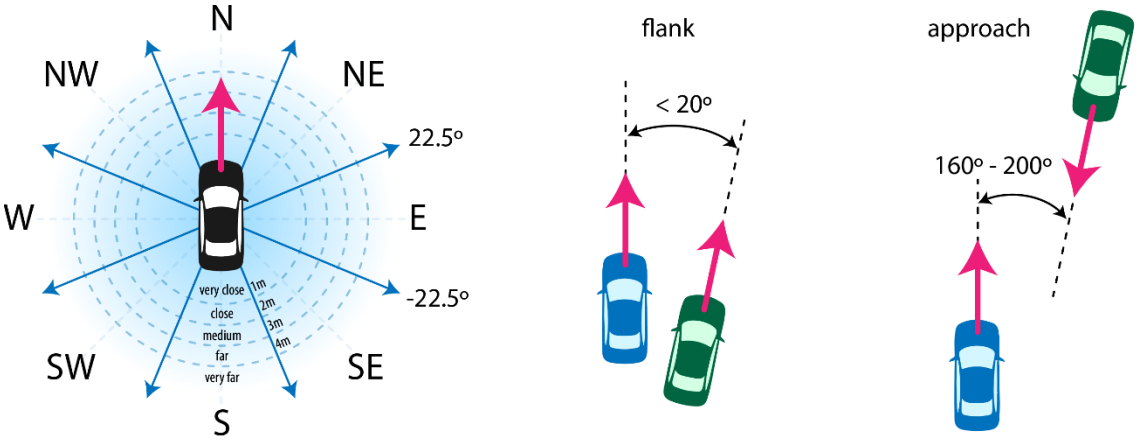


Figure 2-4. Predefinition of object angles, distances and relative positions between two objects.

2.2.1.1.3 Component Usage

The Scene Understanding in Traffic module plays a crucial role by analyzing traffic scenes and producing a detailed scene graph represented as a set of triplets. Each triplet captures key relationships between objects in the traffic environment. This scene graph is then passed to other components to perform various tasks, including ADAS test case generation and virtual environment creation.

Usage in Use Case 1: The output is a scene graph comprising triplets that represent the relationships between traffic objects (e.g., cars, pedestrians, traffic lights, etc.). Other components can process these triplets to automatically generate test cases for Advanced Driver Assistance Systems (ADAS). The test cases are derived from real-world traffic scenarios by interpreting the relationships defined in the scene graph, such as (Car A, is approaching, Pedestrian B). From this triplet, this component could create a test case to assess how ADAS reacts to a pedestrian crossing the street in front of a moving car.

Dependency: The quality and complexity of the test cases generated are directly influenced by the richness of the scene graph produced by the module Scene Understanding. If the scene graph captures subtle traffic dynamics, the ADAS test cases will reflect real-world complexities more accurately.

How to use the component

1. Input Specifications
   The component accepts the following types of input:
   - Image Files: Single image files representing a snapshot of the traffic scene.
   - Video Files: A video file containing multiple frames of a traffic scene.
   - Stream Data: Continuous real-time image or video stream from a camera or sensor.

   Supported Formats:
   - Images: .jpg, .png, .bmp
   - Videos: .mp4, .avi
   - Stream: Real-Time Streaming Protocol (RTSP)

2. Running the component

Option 1: Using Single Image or Video File

i. Prepare the input file: Make sure that the input file is saved in one of the supported formats and is accessible to the component.

ii. Run the component with the following command:
```
python start.py --input_file <path_to_image_or_video>
```
Example:
```
python start.py --input_file traffic_scene_01.jpg
```
This will process the image traffic_scene_01.jpg and generate the corresponding scene graph in the form of triplets.

Option 2: Using Stream Data

i. Connect to a stream: Ensure the stream source (e.g., a camera feed or network stream) is active and accessible.

ii. Run the component with the stream URL or feed:
```
python start.py --stream_url <stream_url>
```
Example:
```
python start.py --stream_url rtsp://192.168.1.100:554/stream
```

This will continuously process frames from the stream and generate triplets for each detected relationship in the traffic scene.

3. Output Specifications
The output will be a scene graph represented as a list of triplets in the format:

```
<object_1, spatial_relationship, object_2>
```

Each triplet describes the relationship between two objects (vehicles, pedestrians, etc.) in the traffic scene, along with their relative positions and distances.

Example of Output:

```
<car_2, north_west 2m flank, car_1>
<car_3, east 4.5m, car_2>
...
```

By following these instructions, users can process traffic scenes and generate scene graphs to support further tasks like ADAS test case generation and virtual environment creation.

2.2.1.1.4 Experiment and Demonstration

Experimental setup w/wo Baselines

Object Detection. We use a checkpoint YOLOv9c pretrained on Microsoft COCO dataset specifically for the object detection task. The model is fast and lightweight while maintaining sufficient accuracy, striking a balance between speed and precision.

Motion Detection. We retain the N most recent video frames for motion detection, with N adjustable based on practical use cases. By default, N is set to 5. Currently, we heuristically estimate the projection matrix for each camera. Since our cameras are fixed, this estimation requires minimal effort. Moving forward, we plan to collect camera intrinsic parameters for more precise calculations.

```
class RelationPredictor:...

def lookup_object(interested_object, list_of_objects, image, tolerance=5):...

def compare_crops(crop1, crop2, tolerance=1.1):...

def get_center(box):...

def get_angle(vector1, vector2, normalize=True):...

# distance_scale: how many pixels in the image to be considered 1 meter?
def predict_single_pair(box1, box2, motion1=None, motion2=None, distance_scale=50, distance_limit=8, number_mode=False):...

def simplify_relations(relations):...

def transform_box(box, homo_matrix):...

def transform_point(x0, y0, homo_matrix):...
```

Figure 2-5. A fragment of the source code from the current implementation

Experimental results

Our initial results are shown in Figure 2-6, where we focus on two cars for easier visualization. We can detect most objects and their relationships, describing each relationship by the angle and distance between objects. The system runs at an average frame rate of 5.8 fps, with the main delay caused by the object detector. Additional optimization techniques are essential to reduce latency and enhance the overall responsiveness of the system.

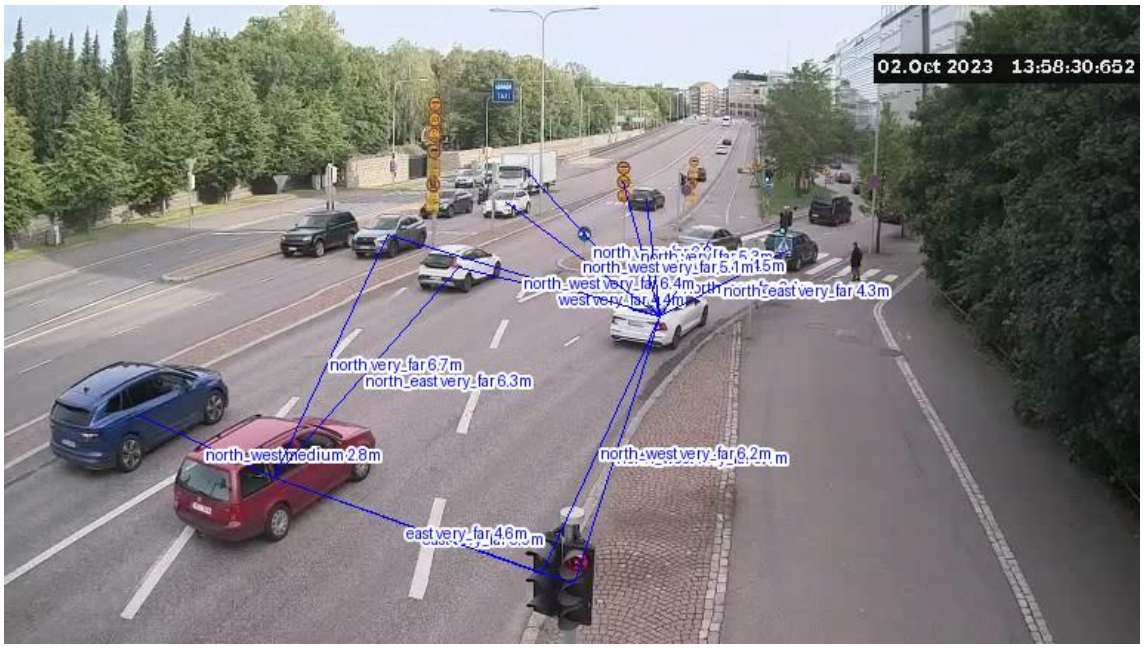


Figure 2-6. Visualization of initial results. For clarity, scene graphs are shown only for two cars within an 8-meter radius. Here for each relation, we show the angle and distance between the two objects.

2.2.1.2 Manufacturing Scene Understanding

The manufacturing scene understanding artifact, A5.1.2.2, along with artifact A3.11, is part of a solution that aims to enable Autonomous Mobile Robots (AMRs) and other smart devices to identify objects in the manufacturing setting, comprehend surrounding objects in their operational area, and handle dynamic partially observable non-deterministic environments more effectively. The solution will be validated and demonstrated primarily in use case 3 – mobile robots in smart factories. By exchanging their perception with other intelligent robots and edge devices, they establish a shared understanding of their environment and collaborate to achieve common goals. Sharing scene information enables them to model aspects they cannot directly perceive through their own sensors, leading to enhanced efficiency and adaptability in the manufacturing process.

With real-time scene understanding, AMRs can navigate, avoiding moving obstacles, and react appropriately to different scenarios. For instance, they can notify staff about an unexpected pallet and seek an alternate route, or stop and issue an alert if the obstacle is a person. While most automated manufacturing environments rely on fixed positions for efficiency, this limits flexibility. AMRs with scene understanding can adapt to variations, like product racks in different locations, or people inadvertently moving items. Even with limited intelligence, this adaptability allows them to handle some of the changes in their environment, improving overall smart factory operation.

The manufacturing scene understanding artifact is similar to the traffic scene understanding artifact, A2.1.2.1, in that their primary input sensor is an RGB camera generating video streams and their output is a streaming scene understanding graph. However, there are significant differences in the way they process the images and draw specific inferences. This is due in part to the differences in the environments and the way objects move and interact within them. Traffic scenes are typically:

- outdoors
- varying lighting conditions, e.g., day/night
- varying atmospheric conditions, e.g., rain
- objects tend to follow predictable motion paths, e.g., cars do not move sideways
- objects tend to maintain their distance
- relatively limited number of object classifications
- object occlusions do occur but are typically transient, e.g., a car may temporarily be hidden behind a lorry
- the ground surface is uneven

Manufacturing scenes are typically:

- indoors
- constant lighting conditions
- limited atmospheric conditions, e.g., occasionally smoke or steam
- objects can move at unpredictable speeds and directions, e.g., Mecanum wheels enable an autonomous mobile robot (AMR) to crab sideways, as well as forward and backwards
- objects often come very close to each other or appear to merge
- relatively large and diverse number of objects, e.g., many different types of robot
- object occlusions occur often and may be long lived, e.g., a mobile rack may remain hidden behind a large piece of equipment for some time

- the floor surface is usually flat

These differences mean that different assumptions can be made about the environment and objects within it and require different approaches to address scene understanding.

2.2.1.2.1 Main Functionalities

The artifact consists of several parts:

- Combined RGB camera and depth sensor
- Location and calibration squares
- Media stream processing
- One or more sensor processing pipelines
- Semantic scene graph fusion

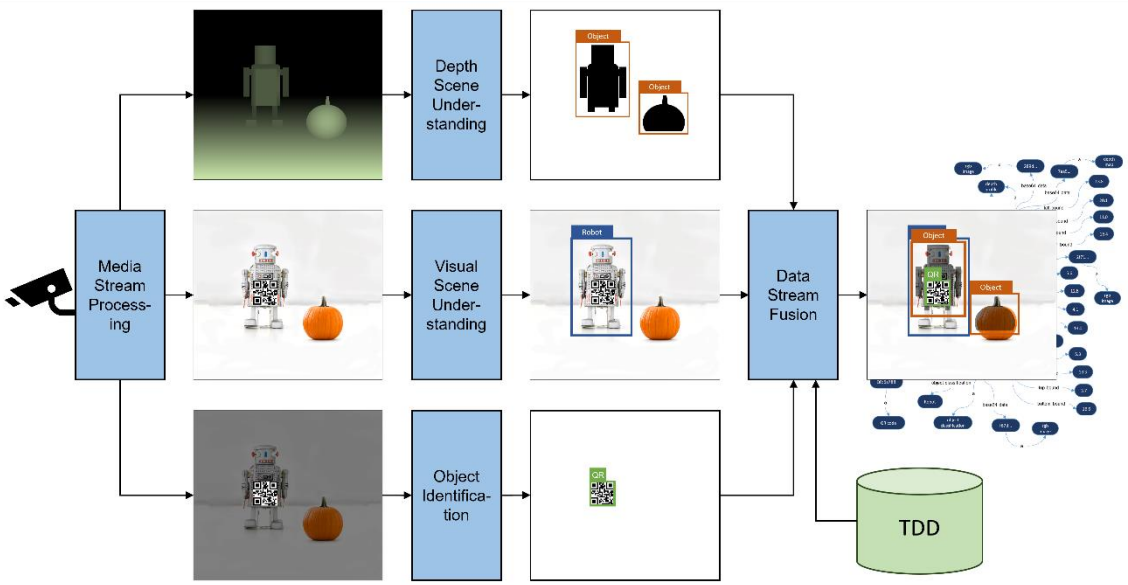


Figure 2-7. Manufacturing scene graph schematic.

Figure 2-7 illustrates the concept of the scene understanding pipelines.

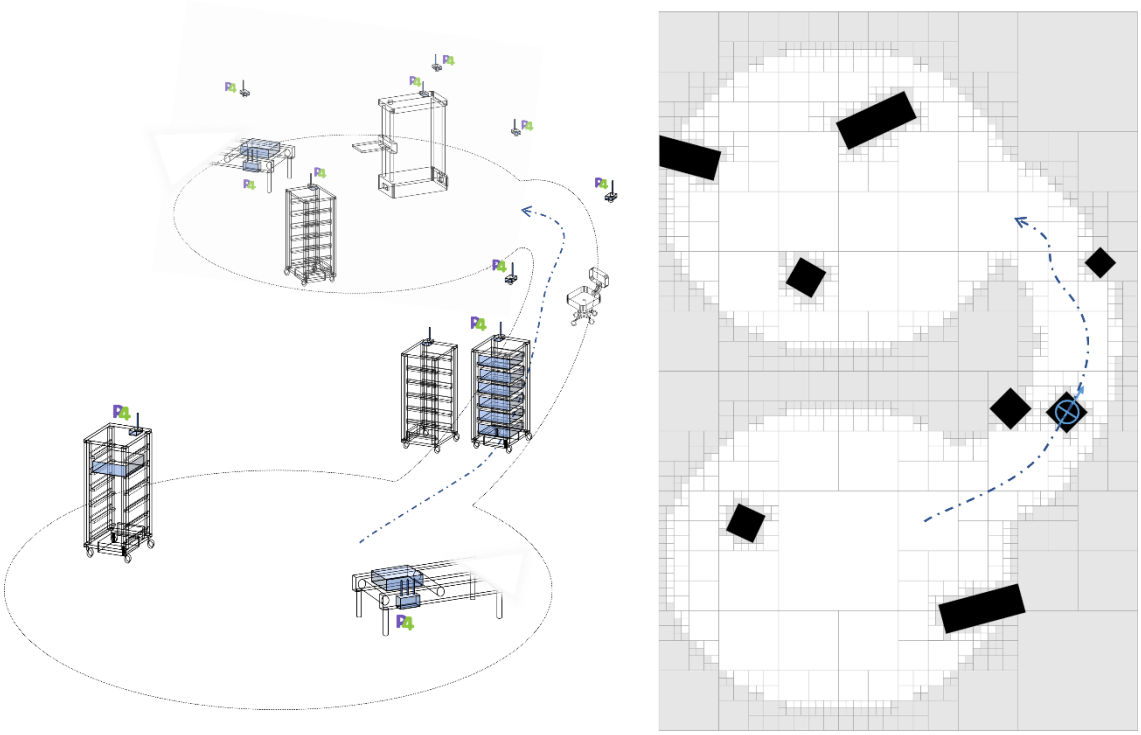


Figure 2-8. Illustration of rack moving between operational areas (left) and a 2D occupancy map (right)

Figure 2-8 illustrates how the streaming scene understanding graph can be used in the smart factory use case 3. The cameras are directly connected to IoT gateways, which host the software that streams the scene understanding graph built from the camera image and depth sensor feeds. The camera and IoT gateway can be considered a single entity and implemented as a SmartEdge smart-node that can be part of a swarm. Scene understanding graphs are streamed from multiple static cameras (smart-nodes) overlooking the operational areas. Each operational area will be overlooked by several cameras providing scene understanding graphs from different viewpoints, which helps the identification of objects that may be occluded from other views. All the nodes in the swarm are linked over a wireless network implemented using special wireless access points that support SmartEdge swarms, and are being developed as part of task T4.1, artifacts A4.1-6. As the nodes are part of the same swarm, they can exchange application information over the Zenoh Message-Oriented Middleware (MOM) implemented by artifact A3.2. The camera smart-node publishes the scene understanding graph on a specific topic, which can be subscribed to by other nodes in the swarm; the graph consists of RDF triples. The smart-node information, topic path, and camera field of view are all stored in the Thing Description Directory (TDD), which is implemented by artifact A3.3. By querying the TDD any smart-node can identify the static cameras that overlook operational areas they are interested in and subscribe to the topic feed. In the smart factory use case a smart-node, with sufficient computational capacity, hosts an application the takes multiple scene understanding feeds and fuses them into a single semantic 3D model of the operational area. The semantic 3D model has many possible uses, one is to generate a 2D occupancy map as illustrated on the right in Figure 2-8, which gives a plan view of the operational area and the objects within it. The 2D occupancy map is also published on a topic, and can be used by any smart-node to navigate the operational area, even if it is not fully observable by its own onboard sensors. The navigation is performed by NAV2, which is a standard ROS2 package for mobile robots. The above functionality is implemented in artifact A3.11.

2.2.1.2.2 Component Implementations

Combined RGB camera and depth sensor

Figure 2-9. MEMS LiDAR cameras (left) and stereoscopic depth cameras (right)

The static cameras are positioned at fixed locations with a high vantage point, typically, they are mounted on the ceiling or high up on structural frameworks. In addition to the static cameras, it is also possible to use cameras mounted on AMRs, but this is more computationally intensive as the camera must recalibrate its ego position each time the AMR moves. So, for initial experiments only static cameras are used. The cameras are a combination of RGB camera and depth sensor, which has the advantage that they have a common frame of reference. Two types of camera technology are being evaluated, the Intel L515 MEMS LiDAR camera and Intel D455 stereoscopic depth camera, as illustrated in Figure 2-9. Both types of sensor provide RGB images and depth maps (range to a target) in a single sensor. The LiDAR uses a scanning laser to detect range, whereas the stereoscopic camera uses stereo photogrammetry to triangulate key features in the image. Both cameras have approximately the same range of about 6-9m. Obtaining RGB and depth information from the same sensor ensures that the two data streams have the same frame rate and field of view, although they do have different resolutions. The cameras are connected to a Dell Technologies 5200 IoT gateway via a USB cable. The gateway hosts the scene understanding pipelines, as well as other software components of the artifact.

Location and calibration squares

Figure 2-10. Rack with QU identification code and floor mounted QR code and calibration squares

Object detection and classification in manufacturing can be problematic, especially when identifying specific items like individual AMRs. To simplify this, QR codes will be attached to certain pieces of equipment, as illustrated in Figure 2-10. These unique codes, linked to object properties in the Thing Description Directory (TDD), enable easy identification when captured by cameras, enriching the scene's semantic understanding. Furthermore, QR codes and calibration squares at known locations will aid camera calibration and object positioning. A static

camera will always see at least one of these floor or wall mounted codes. Like other objects, these codes are also stored as "things" in the TDD. This technique enhances object recognition by identifying individual objects and provides a common frame of reference for objects across the operational areas.

As part of an initial calibration technique, the static QR codes and calibration squares on the floor and walls are used to calculate the projection matrix for the camera's field of view. As the QR codes and calibration squares are modelled as Things in the TDD, their location and orientation can be retrieved. It is therefore possible to calculate the transformation matrix that map the camera's image plane onto the absolute coordinates of the factory floor. This allows all objects to be placed in the same coordinate system.

Media Stream processing

The RGB depth sensor feed is split into different parts by the media stream processing unit and supplied as input to each of the processing pipelines. Which sensor information feed is input to which pipeline depends on the type of processing being performed, e.g., for the foreground object detection only the depth information is required. In order for the output scene graphs from each pipeline to be semantically fused, it is critical that all pipelines process the same frame at the same time and that they share the same field of view. The resolution of the RGB and depth information is different, but this is normalised before passing the sensor feeds to the processing pipelines.

One or more sensor processing pipelines

Currently, three sensor pipelines are planned: foreground object detection, object classification, and identifying mark detection. The output of each processing pipeline is merged in the data stream fusion unit. As all objects in the scene share the same frame of reference, their enclosing bounding boxes can be correlated using Intersection over Union (IoU) [Rezatofighi19]. In the future additional sensor pipelines may be added either to support new processing techniques or new types of sensor feeds. For example, thermal cameras may be used to pick up heat signatures, which is useful for detecting humans as most of the manufacturing plant is colder; or using Large Language Models (LLMs) to describe the objects in a scene by directly producing a scene understanding graph. Whilst LLMs can be prone to hallucinations, scene graphs from other processing pipelines could be used to substantiate their predictions.

The foreground object detection pipeline uses depth information from the camera to separate foreground objects from the background scene. There are various techniques for performing foreground object detection, but in the manufacturing setting we can take advantage of the fact that the floor is level and uniformly flat. As part of a calibration step, a baseline floor value is calculated for each cell in the depth map. There are existing techniques to detect plains in a depth map, even if it contains static foreground objects. The plain representing the floor can then be extrapolated for each cell in the depth map and is stored as a constant for future processing. Detecting foreground objects in the scene is a simple arithmetic operation of subtracting the extrapolated floor depth from the current depth for each cell in the current frame. This technique will detect static objects, such as support columns, as well as moving objects, such as the mobile racks. The static objects are not regarded as part of the background, even if they are fixed to the floor. Instead, they are modelled as discrete objects in the 3D environment with their own unique Thing Description in the TDD and help to provide fixed landmarks at known locations. The output of the pipeline is a scene understanding graph of the detected foreground objects in the sensor's view at a specific point in time. Each object is given

a unique identifier, and its position is enclosed by a bounding box, which is defined by its location in the depth map.

The object classification pipeline uses YOLO9 [Wang24] and a fine-tuned COCO dataset, to detect objects in the scene. YOLO is a powerful object detection model capable of detecting multiple objects at different distances. The COCO dataset is commonly paired with YOLO, and can classify 91 objects, from cars and busses to laptops and mobile phones. Unfortunately, there are almost no datasets for pieces of common manufacturing equipment, and they are typically proprietary and not publicly available. We therefore intend to start from the COCO dataset, as it contains some items we need to identify, and add additional training data for the missing objects. The training data will be a mixture of real and synthetic images. Ideally, we will use the Model Builder (A5.4.4.1) being developed by TUB to fine tune the YOLO/COCO model to recognize the manufacturing equipment. Below is a list of manufacturing equipment that will be added to the COCO dataset:

Autonomous mobile robot	Smart mobile rack	Conveyer belt
Person	Computer server	Table
Chair	Caster chair	Box
Laptop	Monitor	Cup
QR codes	Calibration square	Floor
Robot	Tray	Tray trolley
Pillar	Disk drive	Pallet

Computer servers are the products being moved around the factory. Trays hold products when they are being moved on conveyer belts. The products are lifted off the trays when they are put into the mobile racks, and the trays are stacked onto the tray trolley. Pillars are support structures for the roof, and are static objects with identifying marks, which are useful landmarks when navigating the operational areas. If an object can be detected, but cannot be classified with sufficient certainty, it is given the classification “Unknown”. The images are also used to generate CAD models of the various pieces of equipment, which are then converted into URDF models that can be rendered in Gazebo. The Gazebo renderings are captured and used as synthetic images for fine tuning the YOLO/COCO object detection model. The CAD and URDF models are also used in 2.5D Visualization in artifact A5.4.4.5 and 3D environment construction in artifact A3.11.

The previous pipelines are effective at detecting objects and classifying them, but they are often unable to discriminate two different instances of the same object, an occurrence that is common with factory equipment. To improve the detection of individual instances of objects we will use a processing pipeline to recognize identifying mark detection. In our implementation we will use QR codes attached to certain pieces of factory equipment such as AMRs and mobile racks. The processing pipeline will take an RGB image feed as input, use a common QR code detector such as pyzbar or QReader, and output bounding boxes with the decoded QR codes.

Semantic scene graph fusion

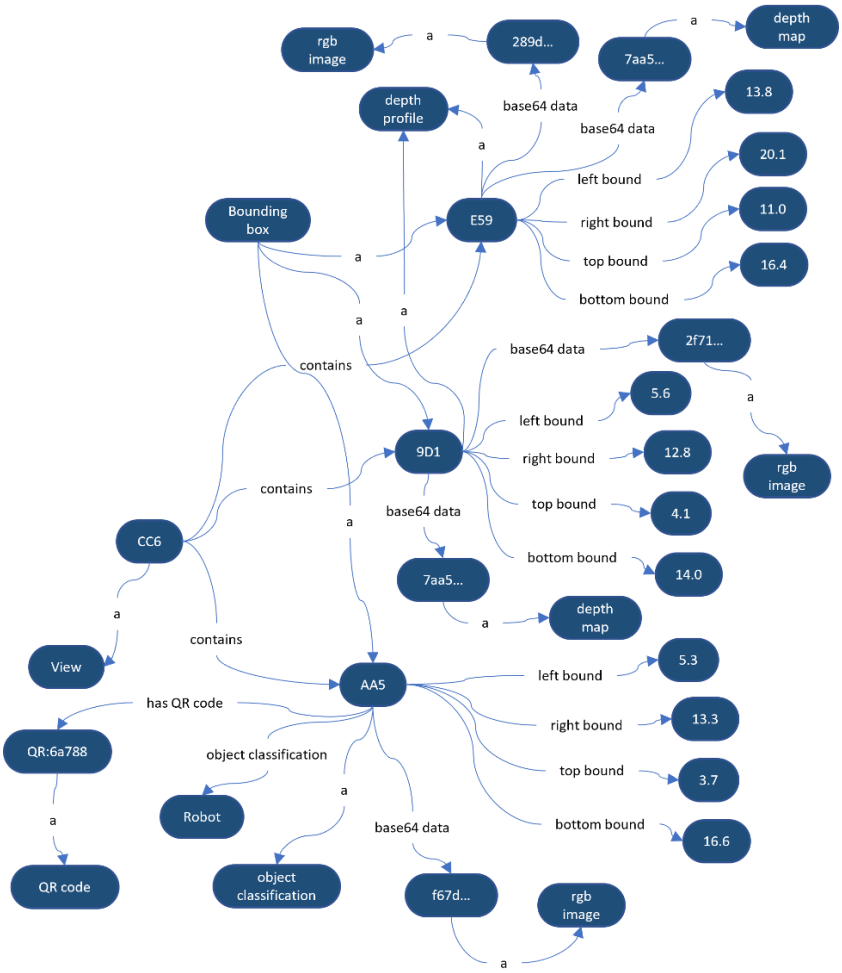


Figure 2-11. Semantic scene graph

It is important that scene understanding graphs maintain the continuity of object IDs between successive frames. Objects that are temporarily occluded by other objects should, for a period of time, be maintained in the scene understanding graph for the current frame as a priori knowledge of an object. Ideally, when the object reappears, they should be reassociated with their previous ID. The semantic scene graph fusion component does this by using DeepSORT [Wojke17] to predict the position of scene objects in future frames. The DeepSORT concept will be modified to generate a predicted scene graph for the next future frame, G_{t+1}. Once the new frame is captured by the sensor at t+1, each sensor processing pipeline is applied to the scene frame G_{t+1} to correct the predicted positions of the objects based on the confidence score from each pipeline and the weight given to each pipeline.

The graph can also be enhanced with Thing properties from the Thing Description Directory (TDD). In SmartEdge, Things are defined as swarm nodes, independent agents, or objects within the use case environment, with their properties, actions, and events stored in the TDD. For instance, a robot's weight and its ability to move are examples of such properties. These TDD properties enrich semantic graphs, offering additional context for detected objects. Additionally, sub-images from bounding boxes could be linked to the semantic stream and transmitted alongside it. These sub-images, which use less bandwidth, could be valuable for subsequent processing, as discussed in [Bowden22].

2.2.2 Integrate Multi-modal Sensor Data Sources

2.2.2.1 Main Functionalities

This task corresponds to *Artifact A5.1.3* and it is performed in collaboration with other partners. UC2 aims at smart traffic management of road junctions and option zones, thus requires real-time data from various sensing sources to construct a digital representation of the traffic scene. Because of the weaknesses of each sensor type, it is important to employ sensor fusion to improve the quality of the scene understanding. Advantages and disadvantages of each sensor type are considered when **comparing** and **combining** data of these sensing sources captured from the same traffic scene.

As an example, let us consider combining radar and camera sources to detect moving road-user objects. Radars are good in detection movement and can work much better than cameras in bad weather conditions like rain and snow. However, it has weakness in detecting smaller objects, namely pedestrians and bicycles, while monitoring these road users is important for smart traffic control. The doppler-based radar is not good in separating objects that are moving slowly or are in stand-still (like vehicles in a queue). Radar can also miss some vehicles or generate ghost vehicles which do not exist. Camera on the other hand has the advantage of detecting all moving object types including pedestrians and bicycles, while it may suffer from image clarity in bad weather conditions.

To sum this up, we need to achieve a more reliable and holistic understanding of the traffic scene by combining data from various sensing sources such as traffic radars, cameras, loop detectors, and signal states (traffic lights). Next, we derive **traffic indicators** from the above scene understanding and give them as inputs to the traffic signal control nodes. These traffic indicators involve momentum, pressure and option-zone vehicles. The momentum is the number of vehicles and their speed approaching a green signal aiming to extend the green time. The pressure is the number of queuing vehicles behind conflicting red traffic signals aiming to stop the ongoing green to get green themselves. The number of vehicles at the option-zone is monitored when the green signal is about to end. The less vehicles at the option-zone, the safer green termination.

2.2.2.2 Component Implementations

The technical implementation of this task comprises the following components: a) Sensing Pipelines, b) Sensor Data Fusion (Artifact A5.1.3), and c) Nodes communication with NATS-MQTT interface.

a) Sensing Pipelines:

The sensing pipeline is responsible for processing data from various sensing sources including radars, cameras, loop detectors, and traffic signals.

Radar sensor processing pipeline:

Radar object detection and its swarm networking interface has been implemented, deployed and successfully tested in the field. The pipeline uses proprietary traffic radars with built-in object detection capability together with customized Jetson Nano edge units that we have

deployed and connected to the radars. These radars are installed at junctions in the pilot area in Helsinki, where one radar faces each road-segment. Each edge device receives the detected objects from the radar serial port in binary format in real-time, converts them into JSON string, and publishes them to specific topics per junction. Data Fusion nodes subscribe to the topics and use this data

Camera sensor processing pipeline:

Similar to radar installations, we have installed cameras (each connected to one of our edge devices) in junctions so that one camera faces each road-segment. For this pipeline, the video-based object detection will be performed by other SmartEdge artifacts like A5.1.2.1 (Vision Scene Understanding for Traffic). This is being implemented using deep learning and GPU acceleration to detect and classify moving objects in real-time.

As a prerequisite for camera detection, the geometries of the relevant sections on the road together with option zone boundaries are specified as an input to the camera object detection pipeline. In addition, we have implemented video streaming to share the real-time camera video frames with the nodes that perform image-based object detection. This video stream sharing feature is implemented using RTSP and via MediaMTX media server and is currently under test. Later, video sharing can be performed using the A5.1.1 (Media Stream Processing) artifact.

Loop detector processing pipeline:

This sub-component has been implemented and is being tested in the field. Vehicle detectors such as inductive loops are directly connected to controller devices at each junction. The detections they provide can be used as part of the sensor fusion. The edge unit connected to each controller parses the raw data from loop detectors and converts them into occupancy statuses as well as vehicle counts, serialized as JSON, before publishing to the network to be used by Data Fusion nodes in real-time (with specific MQTT/NATS subjects).

Traffic signal data processing:

Signals i.e. traffic lights installed at each junction, are all connected to a controller. The statuses of these signals (e.g. red, green, amber) can be fetched from the controllers via the edge unit and relayed to the backend services in a similar manner to the loop detector data described above.

b) Sensor Data Fusion (Artifact A5.1.3):

Object detection streams from the above sensing sources are given as **input** to the data fusion pipeline. It is important to note that each individual road-user object might be detected by multiple sensing sources at the same time, therefore the fusion algorithm should consider this phenomenon. The **format** and the **nature** of the data can also be different depending on the sensing source. As seen in Figure 2-12, data fusion employs detectors, radars, and signal data. AI processed camera data provides similar object lists as radar. This data is not yet available (the grey box)

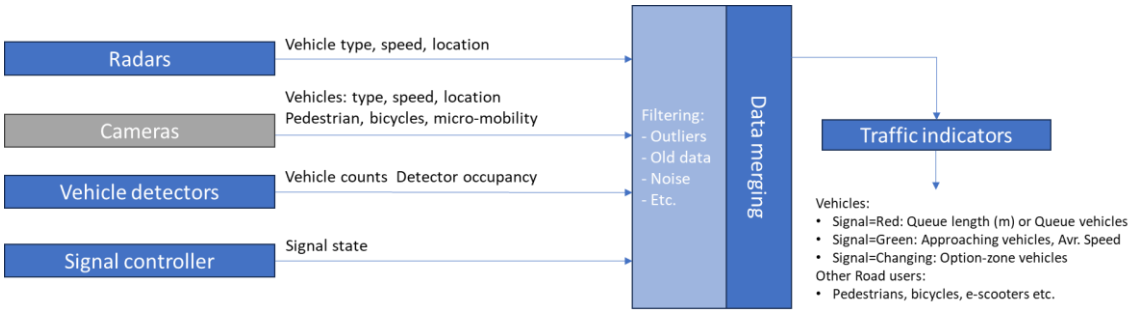


Figure 2-12. Tentative approach for the sensor fusion

The vehicle detectors (usually inductive loops) send binary occupancy data, which indicates if the given location of the loop sensor is partly or fully occupied by a vehicle. The detector data depends on the configuration, location and type of detectors used in each intersection. Generally speaking, an upstream detector (e.g. at 40m from the stop line) can be used to count the incoming vehicles. Downstream detectors located exactly at the stop line can be used to count vehicles leaving the intersection. This approach allows rough estimation of the number of vehicles between the detectors, but it is prone to cumulative error. However, loop detectors may provide only limited details about individual vehicles.

The radar on the other hand, provides detailed object lists with object type, location and speed. However, there are certain limitations related to noise, range, speed and occlusion. For sensor fusion, loop detector data can be used to filter the radar data for anomalies such as “ghost vehicles”, “random spawns”, and “twin detections”. It should be noted that since a detector covers only certain locations, the radar data can be filtered only at that location.

We plan to extend the sensor fusion capabilities with the camera stream. Once the AI-based object detection from camera data becomes available, a more advanced solution will be implemented. From the camera stream we will be getting a list of road users with attributes such as type, position, lane, direction and speed. Camera is also the only data source for pedestrians, cyclists and micro-mobility.

The state of traffic light signal is an input that indicates if the traffic stream is moving or not. When the traffic signal is **red** and vehicles are not moving, the radar data is not reliable, and we should rely more on loop detectors and on camera data. If the camera can detect an object list from a standstill queue, that will be used. Alternatively, the queue length could be detected directly from the camera data and computed into vehicle counts. When the traffic signal is **green** and vehicles are moving, then the radar data becomes more reliable. In addition, the camera data is used to filter out anomalies. When the traffic signal is **changing** (from green to yellow) then we aim to obtain the number of vehicles at the option-zone.

After the filtering is done, all data is merged into a database. The latest messages from various sources are used, so at the moment the timestamps are not used for synchronization. Traffic state estimation process sends queries to this database and estimates the traffic state by using queuing theory and traffic flow theory. Dynamic knowledge graphs (DKG) could also be used if available. The traffic state for the signal controller is provided by traffic indicators as indicated before in Figure 2-12.

Figure 2-13 and Figure 2-14 illustrate a **demo** of the initial implementation of data fusion (artifact A5.1.3). The shown terminals present **one of the data fusion nodes** responsible for one specific junction ("fi.helsinki.270") of the pilot corridor in Helsinki. As explained before, the data fusion node subscribes to the subjects (topics) relevant to the desired junction's local swarm, where various sensing nodes (e.g. loop detectors, radars, signal status) are constantly publishing the data of the moving objects as well as the traffic light statuses. Figure 2-13. Loop data of several lanes received by a Data Fusion node (at junction fi.helsinki.270). presents the loop detectors pipeline of this junction, where each row represents the cumulative number of vehicles that have passed through each lane's upstream and downstream detectors. Figure 2-14. Output of loop-radar-signal fusion for one road lane, performed by a Data Fusion node (at junction fi.helsinki.270). shows the result of fusing the signal, loop, and radar data for one of the lanes (no. 3). Queue lengths, density, and flow are derived from the **loop detector** data while also considering the **traffic signal** status (whether light is red or green). The queue length is determined by the difference between downstream and upstream cumulative vehicle counts. Density refers to the number of vehicles in a specific area, expressed in vehicles per kilometer, while flow measures the number of vehicles passing a fixed point within a given time frame. Density and flow are essential indicators that help determine whether a road is congested. Next in this figure, you can see the resulting filtered list of queued and approaching vehicles as detected by the **radar**. To leverage sensor fusion, any inaccuracies in the radar object detection are being filtered by referring to the loop data.

Figure 2-13. Loop data of several lanes received by a Data Fusion node (at junction fi.helsinki.270).

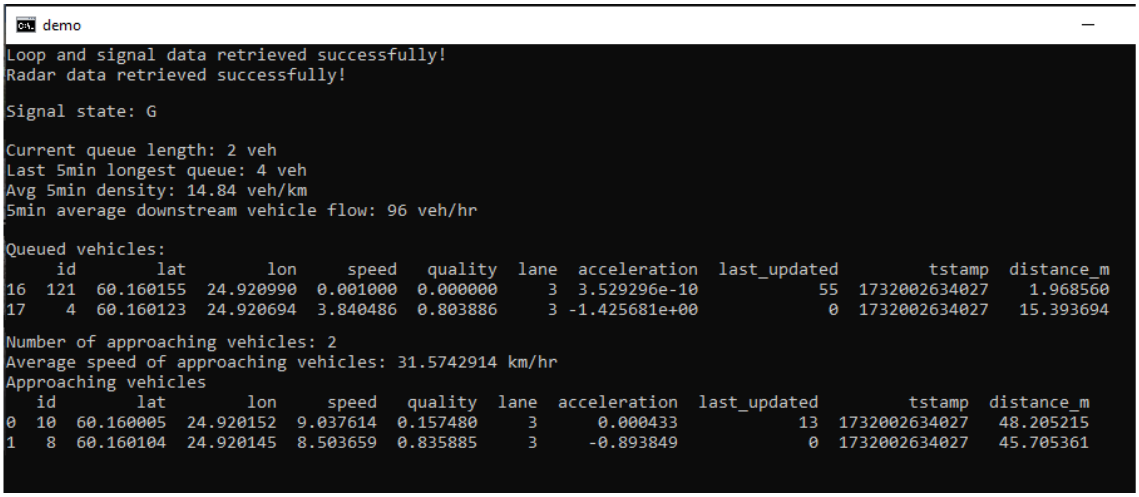


Figure 2-14. Output of loop-radar-signal fusion for one road lane, performed by a Data Fusion node (at junction fi.helsinki.270).

c) Nodes Communication with NATS-MQTT interface:

The role of this component is to enable communication between various sensing nodes and the data fusion nodes as well as possibly with other artifacts. NATS-message broker acts as a middleware component in Conveqs sensor data collection system. In SmartEdge, however, a more widely used MQTT-protocol is preferred. Because of this, we need a mechanism for relaying NATS messages between Conveqs platform and components implemented in SmartEdge. While NATS documentation states that it should be capable of handling MQTT messages, this functionality is relatively new and was not tested by us. As part of A5.1.3, we have now implemented and tested this approach in the field on a NATS server, as illustrated in Figure 2-15. NATS MQTT interface: Integration of heterogeneous publish/subscribe messaging . Results show that having the NATS server configured properly, the *connection*, *publish* and *subscribe* work both ways successfully between **nodes** no matter whether they use MQTT or NATS.

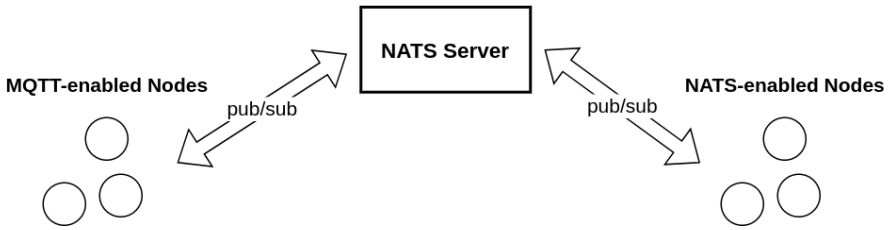


Figure 2-15. NATS MQTT interface: Integration of heterogeneous publish/subscribe messaging technologies.

The following steps should be followed to enable a NATS server to fully support MQTT.

Customized NATS config file ("nats.conf") to support MQTT:

```
# Enable JetStream; Required for mqtt support (standalone):
jetstream {
}

# Enable MQTT support
mqtt {
    # Specifying the port is enough.
    # Server assumes "0.0.0.0" IP by default, therefore listens to "0.0.0.0:1883"
    port: 1883
}
```

Run the NATS Server:

a) Native NATS

Assuming your custom nats config file is in directory "./config/", run nats-server by passing it as an argument.

```
nats-server -c ./config/nats-server.conf
```

b) Containerized (from a NATS image)

Revise the nats section of your **docker-compose.yml** file as follows. "volume" copies your custom nats.conf file to the container internal filesystem.

```
  nats_server:
    image: nats:latest

    command: "-c /etc/nats/nats.conf" # CORRECT
    # command: "-c nats-server.conf" # WRONG PATH

    ports:
      - "4222:4222" # NATS
      - "1883:1883" # MQTT
      # - "8222:8222"
      # - "6222:6222"

    volumes:
      - ./config/nats.conf:/etc/nats/nats.conf

    # network_mode: "host" # DO NOT USE UNLESS STRICTLY NEEDED
```

Now run it as a container:

```
sudo docker compose up
```

MQTT to NATS topic conversion between heterogeneous nodes:
It should be noted that MQTT-enabled nodes should pass "/" in topic names, while NATS-enabled nodes pass "." for the same topic names. The server does the conversion between the two formats automatically. Examples shown below.
MQTT: "radar/*/*/objects_port/json"
NATS: "radar.*.*.objects_port.json"

2.2.2.3 Integration

Integration with UC2:

The following figure shows how A5.1.3 and other functionalities are integrated within the context of smart traffic use-case (UC2). The overall data flow is depicted in the following figure. The processing is performed in different phases starting with fusion of real-time object detection from various sensing sources (dynamic data), where artifact A5.1.3 is responsible for this step. The resulting fused data is then passed to the components that compute various traffic indicators required for realization of traffic management in UC2. As shown in Figure 2-16, these

components also require static data such as the road-network semantic and geometrical description.

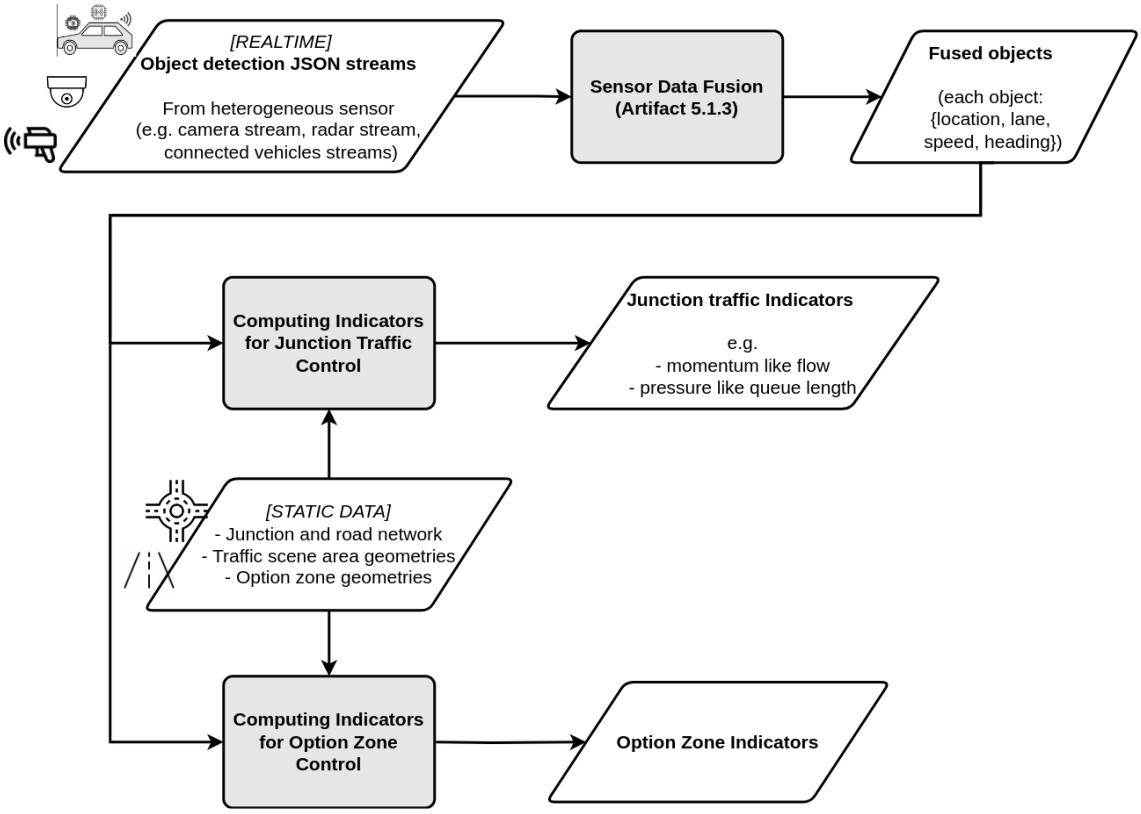


Figure 2-16. Data flow of sensor fusion and traffic indicators for smart traffic management (Grey rectangle boxes indicate a process while other boxes indicate input/output data)

Low-code Recipe-based Implementation:

SmartEdge also provides low-code artifacts that facilitate recipe-based use-cases development as well as quick customization and configuration of the use-case logic. Thus, we have started using these features for smart traffic, aiming at eventually implementing several functionalities using semantic low-code approach. Here we explain an example of recipe development for realization of camera-based object detection. As shown in Figure 2-17 below, the application developer draws the areas subject to traffic management and presents them as polygons (array of points) to be used as *static* input to the recipe for camera object detection.

Figure 2-17. Example of recipe configuration. Traffic areas subject to traffic management, given as static input (as polygons) to the camera object detection recipe.

As examples show in Table 2-1 below, recipe requirements are defined in the form of required input, primitive (required operations) and the expected output. Camera object detection requires receiving real-time video frames as *dynamic* input, and through several steps, should be able to generate real-time object detection output as detailed in the table. The output should contain attributes such as object classification (passenger car, truck, tram, bicycle, pedestrian, etc.), speed and road-lane. Furthermore, given that the traffic scene can be described as a dynamic knowledge graph (DKG) in RDF format, it becomes possible to use graph queries with SmartEdge's low-code approach to automatically generate traffic indicators. The queries written in SPARQL can be passed to the low-code artifacts to extract aggregations and some traffic indicators from the DKG.

Pipeline	Operation	Input	Primitive	Output
01	01	Camera (Image Frame)	• Object Tracking (Yolo tracking)	List of Objects [{ "Object id": int "Box": [float, float, float, float] "Type" : "car/truck …" },]
	02	Json	RDFizer	RDF Graph

	03	List of Objects (from primitive 01)
		[{
		"Object id": "1"
		"Box": "top, right, bottom, left "
		"Type" : "car/truck …"
		},
		]
			Graph Query:
			To query the objects inside the Option zone
				List of Objects with detail: lane id, option zone id,
				[
				{
				"Object_id": int
				"Lane_id": int
				"Speed": int
				}
				]

Table 2-1. Defining recipe requirements and the data flow

2.2.3 Media Stream Processing

As introduced in D5.1, the Media Stream Processing Pipeline includes the software components required to make the media streams coming from capturing devices like 2D Cameras and LiDAR Cameras from cars or robots or even from virtual environments of the remote rendering pipeline to be available in the appropriate format for further components like Data Fusion or use case specific requirements. **Video codecs**, **video bit rates** and **transmission protocols** are key aspects in media streaming. These factors decide whether a media stream with a specific configuration can be created or consumed by a specific provider (media stream source) or consumer (media stream target). In the scope of SmartEdge, media refers to video and image data. For an overview of these concepts please refer to D5.1.

2.2.3.1 Main Functionalities

The Media Stream Processing pipeline consists of a **Video Encoder** that receives raw video streams from capturing camera devices. An optional **Video Packager** which is required if Adaptive Bitrate Streaming (ABR) is used. **Streaming Clients** which are responsible for preparing the video streams for transmission over the network. A **Streaming Server** as the component that receives the media streams from source devices and makes the streams available to consumer devices. And the **Video Decoder**/**Image Extractor** which are responsible for decoding image frames from a video stream and provide them in an appropriate format to the next component in the pipeline such as displaying on a monitor or editing the image or video. For a more detailed overview of the components refer to D5.1. The Media Stream Processing Pipeline is provided in Figure 2-18.

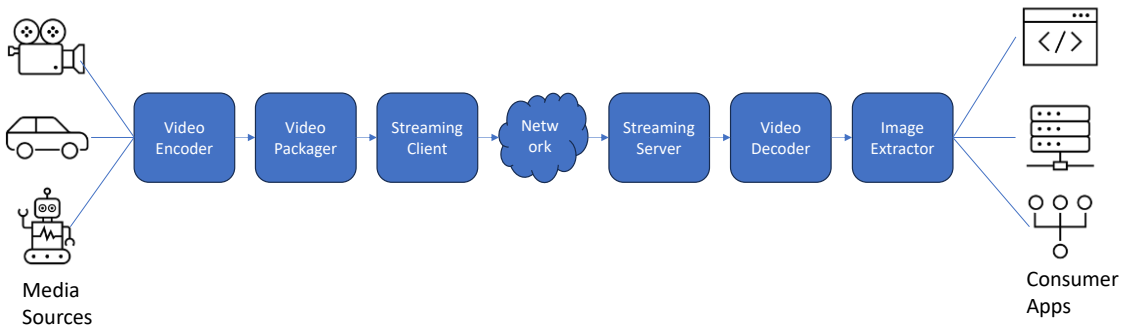


Figure 2-18. Media Stream Processing Pipeline

For hardware devices, the video encoders, packagers and streaming clients are usually predefined and can be configured to a certain degree (e.g. street cameras in UC2). In the case of software that acts as a streaming source, these components can be selected depending on the intended use and runtime environment (e.g. UC1 Virtualization Environment).

2.2.3.2 Component Implementations

As Streaming Server for the SmartEdge Media Stream Processing Pipeline, the open source server Simple Realtime Server (SRS (Simple Realtime Server) | SRS (ossrs.io)) is used as basis for SmartEdge extensions. SRS provides a set of commonly used protocols and allows to translate between them.

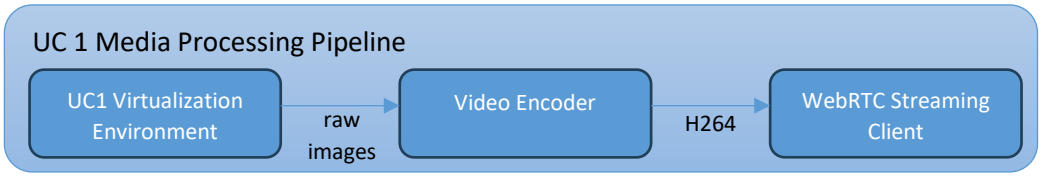


Figure 2-19. UC1 media streaming pipeline

Figure 2-19 shows an example for Use Case 1 where the virtualization environment is one of the sources of a video stream that is encoded as H264 video which is currently provided through a WebRTC Streaming Client to consumers. As mentioned, the first three components of the media streaming pipeline depend on the used hardware and software of a specific use case. For the next phase its planned to extend the client to also support more generic protocols like RTSP so that the video stream can be consumed also by non-Web-based consumers.

For use case 1 the video that is created by the remote rendering artifact must be processed to act as input for the components that are needed to implement the use case. Namely, the ADAS (adaptive driver assistance system) and the scene understanding in traffic artifact. Both need images as input rather than a complete video stream. To provide this functionality as a generic feature the Streaming Server must be extended with additional playout mechanisms.

Figure 2-20 visualizes the extension for the generic media streaming pipeline and its interactions using use case 1 as an example. Orange boxes belong to use case specific implementations. Blue ones are provided by WP5 low code tool chains.

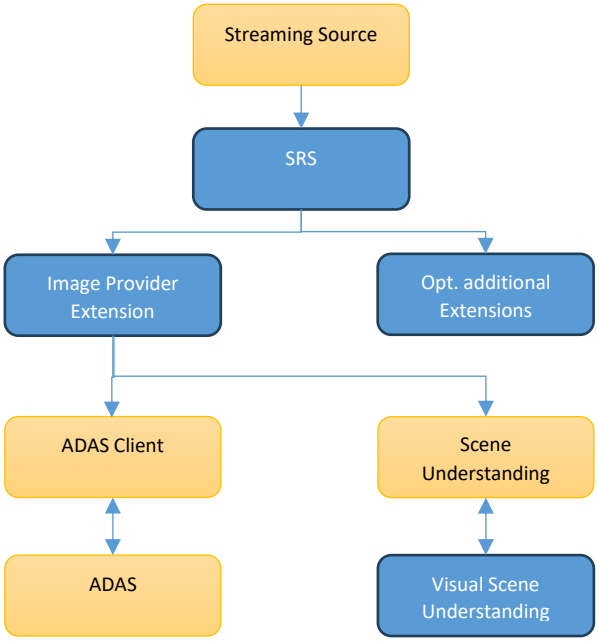


Figure 2-20. Streaming Server Extension and its interactions

The 'opt. additional extensions' box is a placeholder for further development and optional integration with other use cases. In use case 3, for example, a specific colour space of the incoming video relieves the calculation from the use case itself into the media stream processing pipeline. This pre-processing can be integrated directly into the media processing pipeline, which in turn allows the pre-processed video input to be delivered to multiple targets without the targets having to take care of the required video pre-processing themselves.

2.2.3.3 Experiment and Demonstration

The Remote Rendering Artifact can be connected with the Media Stream Processing pipeline. With this, multiple clients, currently as web applications with a WebRTC client, can be connected to virtual cameras without bothering the Remote Rendering Artifact itself with too many video streaming clients. Also the ADAS system which is used by use case 1 can receive images so that it can in turn sent back vehicle control commands. Connecting the scene understanding component is not yet done but targeted for the first months of 2025.

2.2.4 Semantic Data Stream Fusion and Declarative Mapping Rules

We attempt to adhere to FAIR principles as much as possible. We use URIs to identify resources, including images and metadata. The generated data and metadata can be queried using standardized languages like SPARQL and its streaming extension, discussed in the next section. To ensure interoperability, we use as many ontological concepts from Wikidata as possible, each with a unique identifier, and align them with the SmartEdge ontology developed in WP3. Figure 2-21 shows the concept in Wikidata and part of our internal ontology.

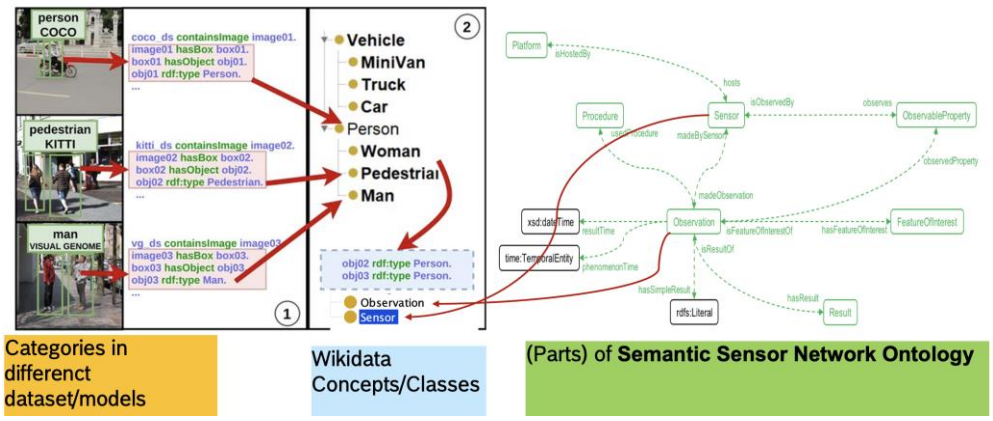


Figure 2-21. The concept in Wikidata and a part of the internal ontology

The first release of the DataOps tool, developed by WP3 and described in deliverable D3.2, provides a solution to configure pipelines for mediated data exchange across different data representations. In the context of artifact A5.1, such a solution can be leveraged as described in deliverable D5.1. In this deliverable, we report the developments performed to define DataOps pipeline that could support the implementation of the Data Stream Fusion artifact (A5.1.4) by leveraging Semantic Web technologies.

2.2.4.1.1 Main Functionalities

The following functionalities are designed and developed via dedicated DataOps pipelines:

- declarative transformation of the output generated by the Vision Scene Understanding artifact (A5.1.2.*) to a common RDF output according to the target ontology;
- enrichment of the graph extracted from multimodal stream fusion with additional information (e.g., contextual information) extracted from static data sources (e.g., datasets).

For the first release, a set of DataOps pipeline is defined to support such functionalities and implemented as a demonstrator considering data sources for scene understanding from the SmartEdge use case 2. We plan to finalise the integration with other artifacts and further refine such pipelines as part of the second SmartEdge release.

2.2.4.1.2 Component implementations

The DataOps pipelines depicted in Figure 2-22 shows the composition and configuration of DataOps components (A3.5) from WP3 to implement the semantic data stream fusion functionality.

In the above part of the diagram, we show how the Vision Scene Understanding component generates from the input data sources (e.g., Sensor, Camera and LIDAR) a stream in a custom JSON format of the detected objects. At runtime, this stream is consumed using an appropriate data connector, e.g., a WebSocket connector[1]. The stream is processed by a *Mapping Executor* component that leverages declarative mapping rules to convert the content of the stream from the JSON representation to an interoperable serialization in RDF according to a target ontology.

[1] https://camel.apache.org/components/4.8.x/vertx-websocket-component.html

The advantage of using the DataOps tool for this transformation refers to the declarative nature of the mapping rules defined that avoids hard-coded solutions, ensuring better maintainability. Since the target data format is RDF, the mapping rules can be defined using both the RML Mapper component and the Mapping Template Component provided by the DataOps tool[2]. Both components support the RDF Mapping Language (RML)[3] to specify the mapping rules as documented in D3.2.

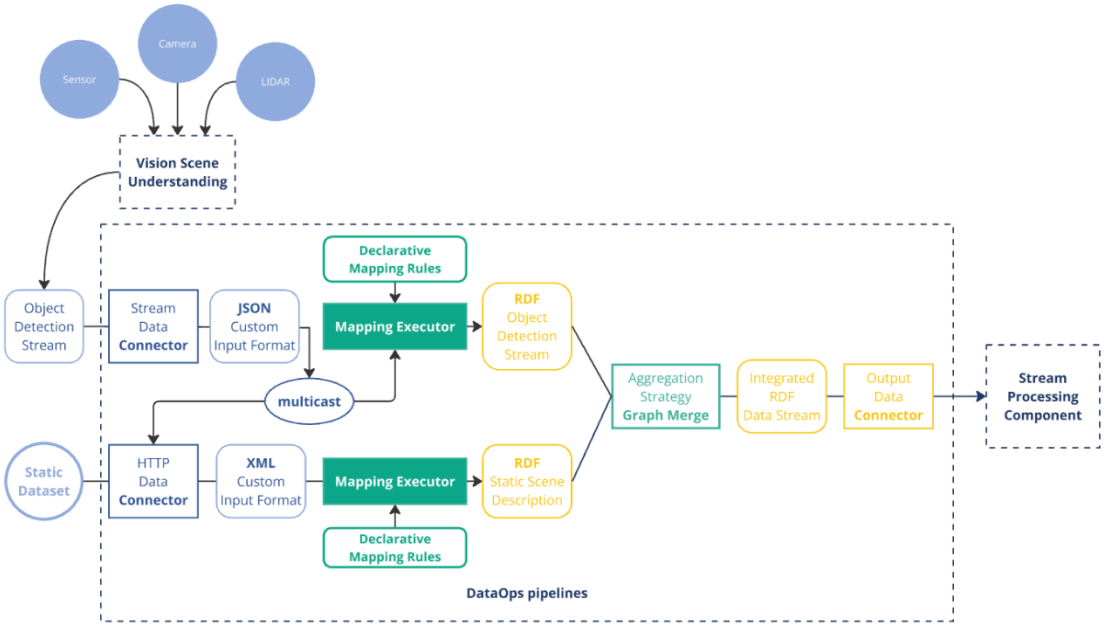


Figure 2-22: Overview of the DataOps pipelines for Semantic Data Stream fusion.

As a result of the transformation, an RDF stream containing an interoperable representation of the detected objects is generated. Such stream is constantly enriched by leveraging the *Multicast* Enterprise Integration Pattern[4] enabling the parallel processing of static information retrieved by additional data sources. As an example, if the scene considered is a street, we can enrich the data with information on its topology. Alternatively, if the scene considered is an industrial shop floor we may enrich the data with information about the robots currently operating in that specific area. The diagram in Figure 2-22 represents an advanced use case in which such information changes often or unpredictably and should be retrieved from an external data source whenever needed. Simpler scenarios may support a once-in-a-while transformation to RDF of the static information that can be directly accessed without requiring an on-the-fly transformation to RDF. In the considered case, an HTTP connector[5] is used to access an external data source represented in a custom XML data format. The data are then processed by a *Mapping Executor* that applies the mapping rules to generate an RDF representation according to the same target ontology applied for the stream.

[2] https://github.com/cefriel/chimera
[3] https://w3id.org/rml
[4] https://camel.apache.org/components/4.8.x/eips/multicast-eip.html
[5] https://camel.apache.org/components/4.8.x/http-component.html

Finally, the graph-based nature of RDF is leveraged to merge the two graphs generated into one. In the resulting graph, common node identifiers are automatically merged and shared semantics are guaranteed by the underlying ontology. The resulting graph is then exposed by an appropriate connector for further processing. For example, a stream processing or stream reasoning engine can be leveraged to perform continuous querying operations for detecting specifying conditions in the processed scene.

2.2.4.2 Experiment and Demonstration

For the first release, we implemented the designed DataOps pipelines in the context of SmartEdge use case 2 as shown in Figure 2-23. Such pipelines demonstrate a process for converting and fusing heterogeneous data sources according to a shared semantic representation.

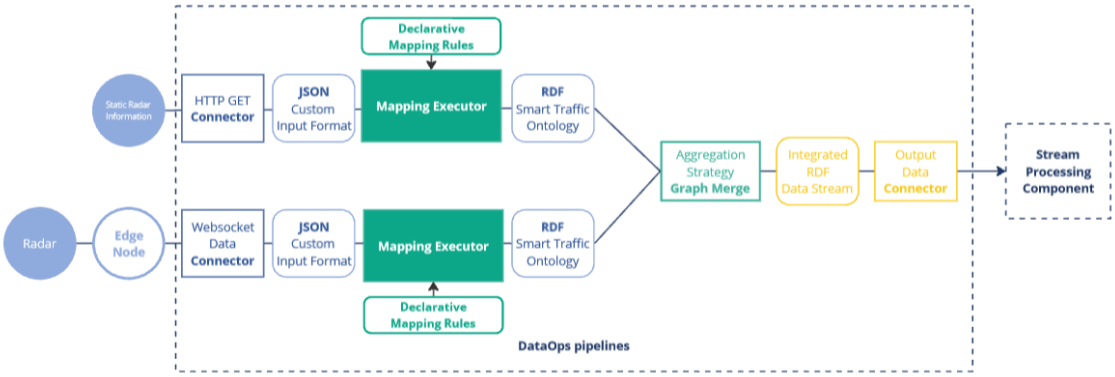


Figure 2-23: Demonstrator DataOps pipeline for the Helsinki use case. Static data and real-time data are converted to an RDF representation and then merged for further possible processing

The pipeline ingests real-time data from radar systems in Helsinki, transmitted via WebSockets in JSON format. Such data are represented using a custom data format and an example is reported in Figure 2-24. This data includes details on the count and types of vehicles detected, such as cars, trucks, and other classifications.

These data are complemented by processing static data on the specific sensor that is responsible for a measurement in the live data. This second data source is obtained from invoking a REST API. Both data sources are transformed into RDF according to a shared "RDF Smart Traffic Ontology" to enable semantic interoperability and data fusion.

For the first release, a preliminary "RDF Smart Traffic Ontology" is identified according to the best practice of ontology reuse. We identified and reused two existing ontologies: the ASAM OpenXOntology[6], which provides a model for road and vehicle-related data, and the SOSA[7] (Sensor, Observation, Sample, and Actuator) ontology, which is designed for describing sensor-generated data. These ontologies offer a standardized, meaningful representation of traffic and sensor data, supporting more effective data integration. Additionally, as discussed in the introduction of this sub-section, we referred relevant entities from Wikidata. Mapping rules are

[6] https://www.asam.net/standards/asam-openxontology/
[7] https://www.w3.org/TR/vocab-ssn/

specified through the Mapping Template Language (MTL) and executed using the Mapping Template component.

As an example, let's examine one specific radar and one specific measurement from it, shown in Figure 2-24. The semantic model identified for the target RDF output is illustrated in Figure 2-25. The static information from the Helsinki radar is described as a "*Sensor*" according to the SOSA ontology. This categorization is enhanced by referring to specific Wikidata entities: Q47528 identifying a *Radar* in Wikidata, and Q167676 for a *Sensor*. The radar's name and geographical position are also included by reusing the Basic Geo (WGS84 lat/long) Vocabulary[8].


```
{
  "source": "radar.269.1.objects_port.json",
  "status": "OK",
  "tstamp": 1706539570737,
  "nobjects": 7,
  "objects": [
    {
      "id": 59,
      "lat": 60.16221173661866,
      "lon": 24.922739519454993,
      "speed": 10.964308,
      "bearing": -147.36752,
      "quality": 0.10896696,
      "length": 5.4,
      "class": 4,
      "lane": 1,
      "acceleration": -0.00020736878,
      "last_updated": 15
    },
    ...]
}
```


Figure 2-24: Example measurement from one Helsinki radar. The position, length, speed and bearing of a vehicle are measured. The class of the vehicle is also identified, in this case the '4' classification corresponds to a car.

In the SOSA ontology, a key class is "Observation," representing a measurement taken by a specific Sensor. Each Observation is linked to the radar, allowing us to identify real-time observations associated with that sensor. Each Observation also relates to an entity being measured, representing specific properties of that entity. In this context, we use the ASAM OpenXOntology to model vehicle characteristics such as length, speed, and angle.

A detected entity uniquely identified here as *"se-data:59"* is declared as a SOSA *featureOfInterest*, meaning that it is being observed by the radar. Additionally, since the Helsinki radars can identify vehicle types, this entity is defined as a *Car* class from the ASAM ontology and Q1420 from Wikidata (*Motor Car*). Finally, the measured values for the car such as length, speed, and angle are categorized by their respective measurement types in the ASAM ontology and are defined as SOSA *Results*, meaning they are the resulting values from specific *Observations* made by the radar.

[8] https://www.w3.org/2003/01/geo/

```
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix asam: <http://ontology.asam.net/ontologies/Domain#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix se-data: <http://uc2.smartedge.com/ontologies/smartedge/data#> .
@prefix wd: <http://www.wikidata.org/entity#>
@prefix sosa: <http://www.w3.org/ns/sosa#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

se-data:Sensor-radar.269.1.objects_port.json a sosa:Sensor, wd:Q167676, wd:Q47528;
    rdfs:label "jatkasaari_269_1";
    geo:lat "60.16208267211914"^^xsd:float;
    geo:long "24.923004150390625"^^xsd:float .

se-data:59 a sosa:FeatureOfInterest, asam:Car, wd:Q1420;
    sosa:isFeatureOfInterestOf se-data:Observation-59-radar.269.1.objects_port.json-1706539570737-speed,
                               se-data:Observation-59-radar.269.1.objects_port.json-1706539570737-angle,
                               se-data:Observation-59-radar.269.1.objects_port.json-1706539570737-length;
    asam:hasLength se-data:LengthQuantity-59;
    asam:hasSpeed se-data:SpeedQuantity-59;
    asam:hasAngle se-data:AngleQuantity-59;
    geo:lat "60.16221173661866"^^xsd:float;
    geo:long "24.922739519454993"^^xsd:float .

se-data:Observation-59-radar.269.1.objects_port.json-1706539570737-speed a sosa:Observation;
    sosa:madeBySensor se-data:Sensor-radar.269.1.objects_port.json;
    sosa:observedProperty se-data:ObservableProperty-speed .

se-data:Observation-59-radar.269.1.objects_port.json-1706539570737-length a sosa:Observation;
    sosa:madeBySensor se-data:Sensor-radar.269.1.objects_port.json;
    sosa:observedProperty se-data:ObservableProperty-length .

se-data:Observation-59-radar.269.1.objects_port.json-1706539570737-angle a sosa:Observation;
    sosa:madeBySensor se-data:Sensor-radar.269.1.objects_port.json;
    sosa:observedProperty se-data:ObservableProperty-angle .

se-data:LengthQuantity-59 a asam:LengthQuantity, sosa:Result;
    asam:hasValue "5.4"^^xsd:float;
    sosa:resultTime "1706539570737"^^xsd:dateTime .

se-data:SpeedQuantity-59 a asam:SpeedQuantity, sosa:Result;
    asam:hasValue "10.964308"^^xsd:float;
    sosa:resultTime "1706539570737"^^xsd:dateTime .

se-data:AngleQuantity-59 a asam:AngleQuantity, sosa:Result;
    asam:hasValue "-147.36752"^^xsd:float;
    sosa:resultTime "1706539570737"^^xsd:dateTime .
```

Figure 2-25: An example of a complete mapping process: transforming JSON data from a single radar and one of its observations into the final RDF representation.

The conversion of both JSON data sources to RDF is defined using an MTL mapping and executed with the Chimera Mapping Template component. Figure 2-26 shows a snippet of this MTL mapping, which demonstrates how real-time observations, such as those in the JSON file in Figure 2-24, are converted to the output format shown in Figure 2-25. Firstly, the relevant data is extracted from the JSON file into a Data Frame. Then, following the mapping-template

approach, the data is written to the RDF Turtle format. The approach and functioning of the Mapping Template Component are explained in more detail in the deliverable D3.2.

```
...

#set ($observations = $reader.getDataframe('{
          "iterator": "$.objects[*]",
          "paths": { "id": "id",
                     "latitude": "lat",
                     "longitude": "lon",
                     "speed": "speed",
                     "bearing": "bearing",
                     "quality": "quality",
                     "length": "length",
                     "type": "class",
                     "lane": "lane",
                     "acceleration": "acceleration",
                     "last_updated": "last_updated" }}'))

#foreach ($o in $observations)
se-data:Observation-$o.id-$sensor.source-$sensor.timestamp-speed rdf:type sosa:Observation ;
  sosa:madeBySensor $sensorSubject ;
  sosa:observedProperty se-data:ObservableProperty-speed .
se-data:Observation-$o.id-$sensor.source-$sensor.timestamp-length rdf:type sosa:Observation ;
  sosa:madeBySensor $sensorSubject ;
  sosa:observedProperty se-data:ObservableProperty-length .
se-data:Observation-$o.id-$sensor.source-$sensor.timestamp-angle rdf:type sosa:Observation ;
  sosa:madeBySensor $sensorSubject ;
  sosa:observedProperty se-data:ObservableProperty-angle .

se-data:LengthQuantity-$o.id rdf:type asam:LengthQuantity ;
  rdf:type sosa:Result ;
  asam:hasValue "$o.length"^^xsd:float ;
  sosa:resultTime "$sensor.timestamp"^^xsd:dateTime .

se-data:SpeedQuantity-$o.id rdf:type asam:SpeedQuantity ;
  rdf:type sosa:Result ;
  asam:hasValue "$o.speed"^^xsd:float ;
  sosa:resultTime "$sensor.timestamp"^^xsd:dateTime .

se-data:AngleQuantity-$o.id rdf:type asam:AngleQuantity ;
  rdf:type sosa:Result ;
  asam:hasValue "$o.bearing"^^xsd:float ;
  sosa:resultTime "$sensor.timestamp"^^xsd:dateTime .
...
```

Figure 2-26. Snippet of the MTL mapping performing the conversion for the DataOps pipeline shown in Figure 2-23. In the portion shown, observations from the input json file are converted to RDF Turtle.

The interoperable and fused output of such pipeline can be exploited by configuring its forwarding to an RDF stream processing component (e.g., to enable a continuous querying task). Alternatively, an additional transformation to a target data format (e.g., JSON) can be defined within the DataOps pipeline with specific mapping rules and the output can be forwarded to another node within the swarm.

For the next release, we plan to improve the implemented pipeline, refine the semantic model considering its integration with downstream components, enhance the integration with other artifacts and evaluate its applicability for other SmartEdge use cases.

3 SWARM ELASTICITY VIA EDGE-CLOUD INTERPLAY

3.1 MAIN COMPONENTS AND FUNCTIONALITIES

As previously described in D5.1, Task T5.2 ("Swarm elasticity via edge-cloud interplay") focuses on offloading parts of the computations running on the SmartEdge platform from edge nodes to potentially more powerful nodes (e.g., specialized SmartEdge nodes benefitting from some acceleration capabilities, or powerful nodes running in the Cloud) or better-connected nodes (e.g., central nodes, routers or switches). Specifically, the task develops a series of mechanisms to elastically use resources and offload specific stateful subqueries and AI operations from edge nodes to further nodes. To do so, this task is based on three technical pillars: i) a dedicated declarative data exchange language, zero-copy networking protocol and interconnect stack to exchange data with the best possible performance and lowest latency between nodes; ii) a set of accelerated operators to dynamically offload portions of the SmartEdge workload to specific accelerators running on further nodes; and iii) a runtime to optimize and streamline some of the more complex offloaded operations.

The rest of the section dedicated to this task is structured as follows:
Sections 3.1.1, **3.1.2**, and **3.1.3** briefly introduce each of the three technical pillars central to this task; **Section 3.1.4** explains the low-code, declarative approach we took for all technical pillars.

Sections 3.2.1, **3.2.2**, and **3.2.3** describe the first implementation of each pillar, including details on their integration with use-cases.

Sections 3.3, finally, describes preliminary results for some of the most advanced components introduced in this section.

3.1.1 Declarative Data Exchange

The first technical pillar of Task T5.2, Declarative Data Exchange, comprises a set of new protocols and interconnect standards to exchange data as efficiently as possible between two SmartEdge nodes. To do so, we leverage a series of new technologies to bring latency as low as possible, while freeing CPU cycles by utilizing hardware acceleration as often as possible.

The first technology we utilize towards that goal is Remote Direct Memory Access (RDMA). RDMA is a network technology that enables fast data transfer between computers by allowing one machine to directly read from or write to another machine's memory, bypassing the operating system and the CPU. This bypassing significantly reduces data transfer latency and CPU usage, making RDMA ideal for high-performance computing or environments that require quick, efficient data sharing, like SmartEdge. RDMA achieves this by offloading data transfer tasks to the network adapter, which directly handles memory access, resulting in both higher throughput and reduced CPU workload. However, RDMA poses many problems in practice. In data-intensive applications it often results in fragmented and smaller data transfers that are detrimental to the overall performance [Ryser22]. To remedy this, the first part of T5.2. leverages dedicated Declarative-RDMA (D-RDMA) protocols to speed up data transfer in SmartEdge; Instead of transmitting individual data fragments through RDMA (which wastes CPU cycles and memory bandwidth), the application specifies in a declarative, low-code language

what data should be transmitted, letting the networking card optimize the transfer by issuing larger DMAs.

Two further next-generation technologies are used to streamline the transfers: CXL and advanced Flash controllers. Compute eXpress Link (CXL) is a very new, high-speed interconnect standard designed to optimize data transfer and resource sharing between CPUs and further devices (like GPUs, memory modules, or accelerators, e.g., for data-intensive operations or AI). CXL provides low-latency, high-bandwidth connections that enhance data flow efficiency. Its three sub-protocols (CXL.io for I/O operations, CXL.cache for shared cache, and CXL.mem for memory pooling) work together to allow flexible memory sharing and pooling, enabling devices to access and use memory directly and coherently across different servers. CXL enables more scalable, flexible, and resource-efficient architectures in environments like SmartEdge where efficient, low-latency data exchange is crucial.

Finally, we designed and implemented a framework to optimize memory operations on Flash channel controllers. Our new framework, called BABOL, exposes an asynchronous programming model in which Flash operations are written in software and enqueue instructions that are later executed by programmable hardware. This allows to develop advanced, optimized operations more easily than in traditional synchronous, hardware-only controllers, hence pushing the performance of Declarative Data Exchange even further.

Our Declarative Data Exchange framework is further described in Section 3.2.1 (in terms of implementation details and integration) and in Section 3.3.1 (preliminary results).

3.1.2 Accelerated Operators

Once the necessary data has been transmitted to a remote node or accelerator in the SmartEdge swarm (or on the cloud) thanks to the technological stack introduced in the previous section, one needs an efficient operator to carry out the task on the new node. Many different operators could be designed and implemented in our context, for tackling tasks as diverse as data-intensive operations, signal processing, or AI. Given the peculiarities of SmartEdge, we decided to focus on three very different cases.

First, we designed and implemented an operator to handle a specific AI task central to some of the SmartEdge use-cases: that of face blurring images (e.g., for privacy concerns). More specifically, we designed and implemented a lightweight operator taking as input either individual pictures or videos, and automatically identifying and blurring faces in the content leveraging a dedicated Data Processing Unit (DPU, i.e., a programmable computer processor that tightly integrates a general-purpose CPU with network interface hardware).

Second, as efficient data gathering from multiple edge nodes is one of the key data-intensive tasks in SmartEdge, we decided to work on a component to do so. This second operator works by offloading the construction of an integrated data structure (i.e., a *view*) gathering and consolidating information from several smart devices in an incremental manner.

Finally, as the SmartEdge architecture (D2.2) specifies, all SmartEdge nodes are supposed to have some degree of P4 capabilities. P4 (Programming Protocol-Independent Packet Processors)

is a high-level programming language designed to define and control how packets are processed on networking devices or hardware accelerators, enabling more flexible, efficient, and application-specific data processing and transfer functions. Taking advantage of this fact, we designed a series of operators for accelerating data processing using P4 accelerators. More specifically, we focused on accelerating generic graph operations and SPARQL queries, which are both very common in SmartEdge.

Those various operators are further described in Section 3.2.2 (in terms of implementation details and integration) and in Section 3.3.2 (preliminary results).

3.1.3 Runtime Optimizer

The last component of Task T5.2 is a component that runs on the SmartEdge Orchestrator (or SmartEdge nodes having advanced P4 hardware acceleration) and optimizes complex data-intensive tasks (implemented T5.3 Section 4). As with all other components designed in the context of this task, the optimizer uses high-level, low-code declarative constructs as input, more specifically a declarative program describing the complex operations to offload. The optimizer then represents this program into a Directed Acyclic Graph of operations, that is then optimized.

The optimization process itself leverages a system catalog containing set of rules to rewrite and optimize the offloading operation into an efficient physical pipeline that can be run on one or several P4-accelerated nodes (in a way similar to the optimization of complex programs in data-intensive systems). The optimization process spans different layers: i) physical optimizations to pick the most efficient plans to run the complex offloading, ii) logical optimizations based on rules to move various operators and obtain more efficient offloading plans, and iii) hardware-specific optimizations, for example for fusing operators that can be jointly run in a common P4 pipeline.

The optimizer is further described in Section 3.2.3 (in terms of implementation details and integration) and in Section 3.3.3 (preliminary results).

3.1.4 Low-Code, Declarative Programming

In the end, the underlying vision behind task T5.2. is to combine recent advances in networking and hardware platforms with high-level, declarative, and low-code constructs. We call this vision *heterogenous, low-code computing,* which extends the general low-code approach mentioned in Section 1 above. The idea is to leverage a subset of very recent technologies and heterogeneous hardware platforms to run some of the SmartEdge workloads as efficiently as possible.

For our data exchange scheme, this declarative approach is visible using RDMA verbs (verbs are the API functions that define how applications interact with RDMA hardware) and the declaration of non-contiguous data regions to transfer. Since D-RDMA is declarative by definition, its operations can be directly serialized in a recipe and invoked in a declarative manner by a SmartEdge node.

For our offloaded operators, we adopt a fully declarative approach for our components running on P4. Our offloading graph operations and offloading SPARQL components are fully declarative as they only require the input of a specific graph pattern, or a specific SPARQL query, respectively. Finally, our runtime optimizer works in a typically declarative way also, by taking complex operations in the shape of trees of operators and optimizing them into more efficient pipelines using our various optimization strategies.

3.2 COMPONENTS IMPLEMENTATIONS

We describe below the first implementation of the various technologies we just introduced above, covering Declarative Data Exchange (Section 3.2.1), Offloaded Operators (Section 3.2.2), and finally our Runtime Optimizer (Section 3.2.3).

3.2.1 Declarative Data Exchange Implementation

We describe below our first implementation of our Declarative Data Exchange stack, starting with D-RDMA (Section 3.2.1.1) and CXL extensions (Section 3.2.1.2), before covering our new Flash channel controller (Section 3.2.1.3). We explain our declarative, low-code approach in Section 3.2.1.4 before commenting on the integration with SmartEdge use-cases in Section 3.2.1.5.

3.2.1.1 D-RDMA

We leverage our previous extension of RDMA, called D-RDMA (for Declarative-RDMA) for initiating and orchestrating data transfers between high-speed SmartEdge nodes. D-RDMA was already introduced in Deliverable D5.1, and we only describe the basics of our technology for completeness below. The rest of Section 3.2.1. subsequently describes the brand-new features that we implemented in the past year, namely CXL integration and flash channel optimization.

D-RDMA optimizes RDMA schedules of data-intensive operations using a declarative approach. When using RDMA canonically without optimization, systems often generate lots of small DMA operations for copying relevant data only (as we originally showed in [Ryser22]). Towards that goal, D-RDMA allows to define non-continuous regions to transfer. We call such regions Non-Contiguous Regions (NCRs). The most important NCRs are called *Strided* Regions. RDMA's work requests can be seen as a rudimentary language that can only describe contiguous regions. To make it more expressive, Strided Regions are a construct to capture whole regions that present data and gaps in regular patterns. A Strided Region is defined using a base pointer, a period made of one or more elements, the width of the elements, and a stride. The stride is described by a frequency, e.g., 1 every 2 elements, and an optional start position, if different than the base address. Strided Regions are expressive enough to handle most data transfers in SmartEdge and other data-intensive projects. Figure 3-1 below gives a reminder of a simple example of a Strided Region.

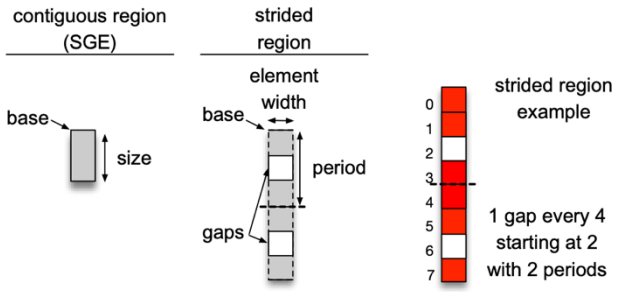


Figure 3-1. Contiguous Regions are insufficient to capture data patterns. Non-Contiguous Regions, such as Strided Regions, can be used to describe data and gaps in a compact way, declarative, and high-level manner and to optimize RDMA.

A D-RDMA request containing NCRs is handled by a specific runtime on a node. Figure 3-2 (a) gives an overview of the workflow from a system's point of view. First, the application sets up the connections (queues) to the remote hosts as it would in an RDMA scenario. It can then use the RDMA verbs API to send transmission instructions to the network card. Certain verbs would take Non-Contiguous Regions (NCRs) to describe the requests. Upon receiving an NCR-based request, the runtime in the node forwards it to an optimizer, which determines the fastest DMA schedule to bring the data from the host. The runtime then executes this DMA schedule, and, as data arrives, it assembles the payloads contained in the NCRs before forming and sending out the packets.

Internally, the runtime comprises five components, shown in Figure 3-2 (b). Two of these components are like those we would encounter in a regular RDMA setting: the DMA Engine is responsible for transferring data from the host's memory into the network card's; and the Packetizer envelopes payload data with headers and trailers for the network protocol the card is handling. The third component, the Segmented Memory, is also present in regular settings but it is implemented slightly differently in D-RDMA: it considers smaller, independent memory buffers, which are used by the DMA engine to write data in a stripped way. The remaining two components, the Optimizer and the (Payload) Assembler, are extensions required to process D-RDMA and are respectively responsible for deciding whether to transmit data chunks individually or to regroup them, and to dynamically assemble the payload of each packet to be transmitted.

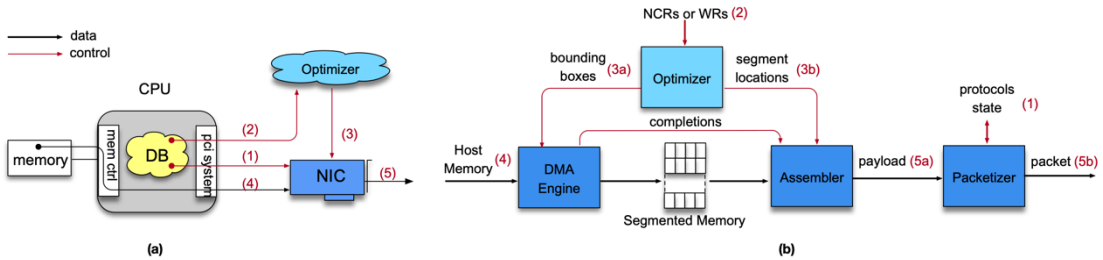


Figure 3-2. The implementation of D-RDMA from a system's perspective (a) and from a NIC's perspective (b). The application sets up a connection as usual (1). It uses declarative, Non-Contiguous Regions instead of SGEs to post work to the card (2). The card determines a DMA schedule upon receiving the NCR list (3,3a,3b). The card issues the DMAs (4). The card uses the row window for that request to find and packetize the data (5,5a,5b).

As a result, D-RDMA can describe data transfers declaratively, and can operate the transfers much closer to line speed, even for fragmented data and data-intensive scenarios such as those presented in SmartEdge.

3.2.1.2 CXL Extensions

The second piece of technology that we leverage to accelerate the offloading operation in the context of this task is CXL. Compute Express Link (CXL) is a very new and open standard for high-speed interconnect technology designed to enhance data transfer between a central processing unit (CPU) and hardware accelerators (GPUs, FPGAs), memory, and storage devices. CXL is built on top of the PCIe physical layer but offers significant advantages in terms of reduced latency, memory coherence, and scalability, making it especially beneficial for modern computing workloads such as data-intensive and IoT tasks.

CXL enables direct, high-speed communication between the CPU and other components, bypassing traditional bottlenecks that occur in systems with multiple independent buses. By leveraging PCIe's established physical layer and adding protocol enhancements, CXL allows for faster data transfers with reduced latency, which is critical in our context for real-time data processing and high-throughput communication between accelerators and memory.

One of the most important features of CXL is memory coherence. In traditional systems, when accelerators access data from the CPU's memory, they often have their own memory space, requiring expensive and time-consuming data copies between the CPU and the accelerator. With CXL, memory can be shared between the CPU and other devices in a coherent manner, meaning both the CPU and the accelerator can access and modify the same data without needing redundant copies. This improves both performance and efficiency by reducing the amount of memory replication and synchronization overhead.

CXL also enables memory pooling, which allows multiple devices to access a shared pool of memory resources. For some SmartEdge nodes, this means reducing the need for dedicated memory on each device, resulting in higher memory utilization and reduced costs. Devices can dynamically request and release memory as needed, improving scalability and resource efficiency. While very recent, CXL is an open standard, meaning that it will be supported by a wide range of hardware vendors in the future, creating a more unified and interoperable ecosystem by standardizing the interconnects between CPUs, memory, and accelerators.

We designed an extension of CXL called *CXL kernels* [Lee24] to automate part of the offloading operations in SmartEdge. The idea is to leverage advanced hardware operation to streamline read/writes at the application layer. Nodes will be able to expose a Database Kernel (DBK) such that read/writes against some specific memory range would trigger data-intensive computations that the kernel would perform directly inside the device. The nodes will use coherence traffic to monitor requests, prepare ahead of time, and ultimately answer data requests more efficiently. We believe that CXL Kernels can support new generations of heterogeneous data platforms (such as SmartEdge) with unprecedented efficiency, performance, and functionality.

CXL devices come in three main types, each designed to optimize different aspects of data handling and computing efficiency:

- Type 1: Accelerator Devices. These devices, like GPUs, FPGAs, and AI/ML accelerators, leverage CXL primarily for high-speed data and cache sharing between the CPU and the accelerator. Two specific protocols, CXL.cache and CXL.io, enable them to access and modify data in CPU memory without duplicating it, reducing latency and increasing performance in compute-intensive tasks.

- Type 2: Accelerators with Local Memory. Type 2 devices are similar to Type 1 accelerators but include their own dedicated memory in addition to accessing CPU memory. This type uses all three CXL protocols (CXL.io, CXL.cache, and CXL.mem), allowing them to cache data, use local memory, and share CPU memory. This flexibility enables complex applications to use both local and shared memory resources efficiently.

- Type 3: Memory Expanders. Type 3 devices are memory expansion modules that add extra memory capacity to the system without any processing capabilities. They use only the CXL.mem and CXL.io protocols, allowing the CPU to treat this memory as an extension of its own, ideal for memory pooling and expanding memory in data-intensive environments like SmartEdge.

Figure 3-3 below depicts CXL operations ensuring data coherence between two CPUs, and between a CPU and a memory expander (a *Type 3 device*).

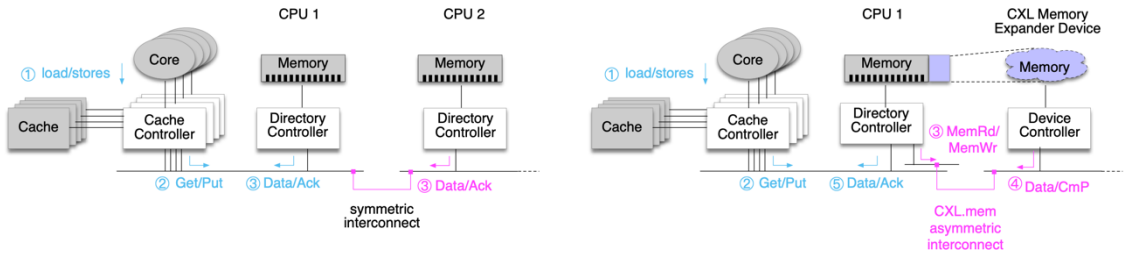


Figure 3-3. (Left) CXL ensuring data coherence across two CPUs: To access or modify the contents of a memory address, a core brings a copy of it to its cache (1). This can be triggered by issuing a load or a store instruction. Upon receiving the instruction, the Cache Controller issues a request to either get a copy or put (write) it's copy of the modified content from/back to memory (2). The Directory Controller receives this message and executes the required memory access, either sending a copy of the read data to the Cache Controller or acknowledging that the modified data was written (3). The Cache Controller can then signal to the core that the instruction is complete. Note that if the address required were held by a remote Directory Controller, the Cache Controller would have targeted it instead (3). (Right) CXL data coherence with a memory expander device: The Cache Controller asks or sends a cache line as before but is unaware of who is backing that address. Upon noticing that the request is for the expanded memory area, the Directory Controller issues the proper command to the Device Controller (3), which in turn interacts with the local memory (4) and responds. It is the Directory Controller that sends the cache line or the acknowledgment back as if the line accessed was local (5).

We propose a component that supports simple and efficient access to expanded memory through memory reads and writes, and rich semantics associated to operations on a given memory range. Our component amounts to a Type 2 device. It exposes memory regions to the server through cxl.mem, and it caches memory from the host through cxl.cache, as depicted in

Figure 3-3. We provide a simple API to map physical addresses from the device onto process memory. Once this mapping is done, applications access these memory ranges “as usual.”

Internally, however, the device offers a powerful indirection mechanism. It associates a range of addresses with what we call a *kernel*. A kernel is a function that provides well-defined semantics for reads and writes. A kernel that implements memory expansion semantics will simply redirect reads/writes to a selected memory type. We will provide such a kernel with the device that can opt between DRAM or NAND-Flash as backing memory. As Figure 3-4 (left) shows, the device can still present itself to the system as an NVMe device and offer a traditional data path. Nothing prevents a legacy application from using it that way. As explained above, a Type 2 device can contribute memory to the system and cache data from the system locally. To do so, it implements a Cache Controller that interacts with the system’s Directory Controller via cxl.cache. Interestingly, Type 2 devices release control of their memory range to the Directory Controller on the host), which forces the device to notify the Directory Controller if it wishes to cache its own memory. In our device, a kernel has the option to request shared cached lines on any portion of the address ranges it exposes. The benefit of this arrangement is subtle but powerful. If a core on the host wishes to access a cached memory address in exclusive mode—e.g., it wishes to write a new entry in an exposed area—the device can be notified of this intent through a cache invalidation message. Figure 3-4 (right) illustrates this case. This early notification of an intent to write gives the device much more time to prepare for the write than it would have if it learned about the write as it was requested.

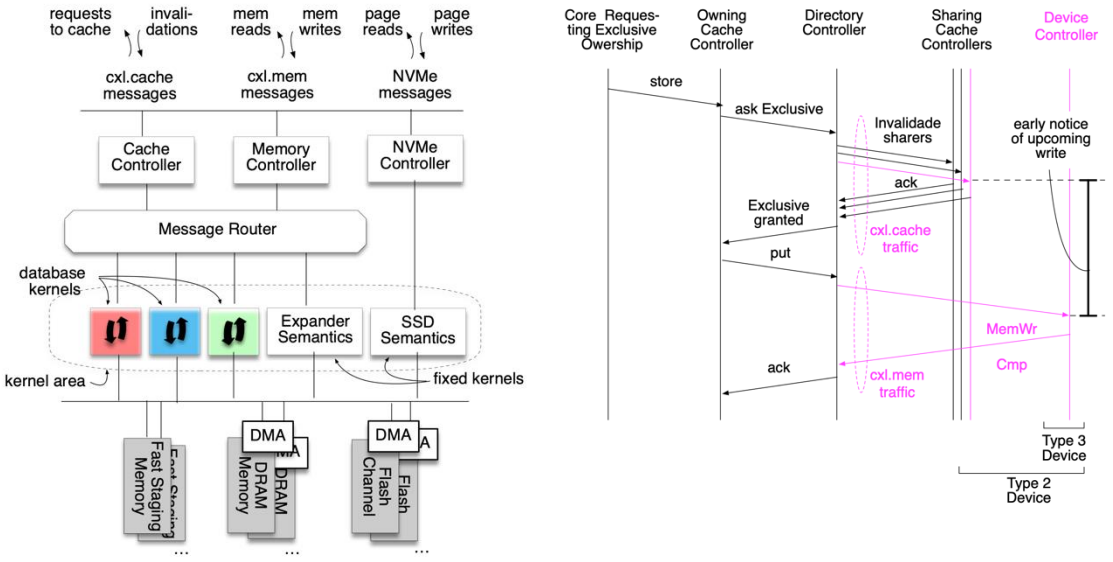


Figure 3-4.(Left) Our device can be accessed as a conventional SSD or through CXL. In the latter case, the messages to a given memory address range will be directed to its assigned kernel. The kernel can choose which kind of storage type to use and how. (Right) As a CXL Type 2 device, our device learns about the early intent to write. The reason is that, to give a core exclusive access to a memory address, the Directory Controller must invalidate all accesses given before. The invalidation is an early signal to the Type 2 device that it should prepare to hear a write request for that address in the short future, giving it ample time to prepare.

3.2.1.3 Flash Channel Controller

The third and last piece of technology we leverage to optimize the data offload is a new software-defined Flash controller that can be used to optimize complex data transfer

operations. NAND Flash Storage Controllers are a crucial component of data-intensive systems. They provide an abstraction of Flash packages to the SSD firmware by translating high-level operations, such as a Page Program or a Block Erase, into low-level signals. In theory, the Open NAND Flash Interface (ONFI) specification standardizes this interface. The standard specifies the number and voltage of pins a compliant Flash package must have and outlines how different data transfer speeds and modes (synchronous and asynchronous) can be achieved via these pins. From an electrical interoperability perspective, the standard can be considered successful. However, the standard lacks important optimizations since it abstracts away relevant internal Flash Array details.

Our new controller, introduced in [Park24], exposes an asynchronous programming model in which Flash operations, written in software enqueue instructions, are later executed by optimized programmable hardware. This asynchronous programming model allows architects to develop advanced, optimized operations more easily than in traditional synchronous, hardware-only controllers.

Our architecture is depicted in Figures 3-5 (left), and is built on two key principles:

1. Separation of scheduling and execution: in our new architecture, a description of the desired segment is produced prior to the opportunity to execute it. This is reflected by the existence of two distinct modules, as shown in Figure 3-5. The module that describes a segment to be executed in the future is called Operation Scheduling. The module that produces that segment once the execution is possible is called Operation Execution. We call this architecture asynchronous because it separates the description of what a next segment to issue should be from its actual execution.

2. Hardware/software co-design: our solution implements operation scheduling entirely in software. A typical Flash package carries one or more Logical Units (LUNs), each of which can perform an operation independently. LUNs are often busy performing internal data movements (to/from the array and page register) that can take tens of microseconds. While a single LUN is busy in a given operation, there is enough time to schedule the following operation in software. Moreover, several LUNs share a channel, which is unavailable during data transfers between the controller and LUN. Similarly, while a data transfer is ongoing, there is enough time to decide in software on the next task to give a particular LUN.

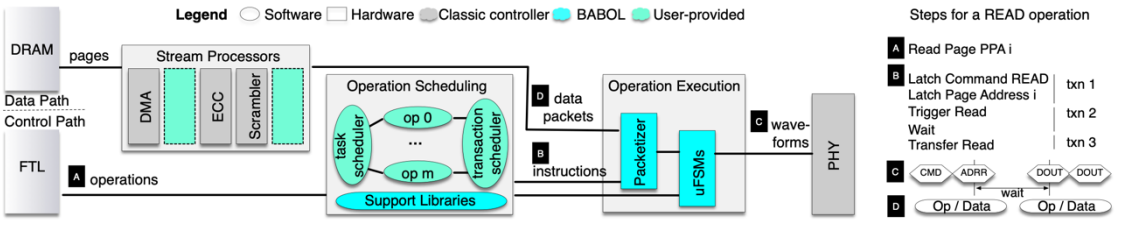


Figure 3-5: (Left) Our new controller comprises three components: stream processors, operation scheduling, and operation execution. Execution has strict time constraints and is implemented in hardware. The implementation, however, does not use hard-coded waveforms. It allows programmatically building them through µFSMs, which are software-configurable waveform segments emitters. Operations such as READ, PROGRAM, and ERASE are written in software and,

with the help of the schedulers, drive the µFSM. (Right) Examples of steps an operation goes through as it is implemented in BABOL.

Thanks to these two new principles, BABOL allows architects to encode standard and nonstandard IO operations easily. It also allows them to define and implement different scheduling strategies.

The basic dialog unit between a controller and the Flash packages is what the ONFI standard calls a Basic Timing Cycle (BTC). Simply put, a BTC is a fragment that establishes one piece of information (e.g., what command to execute, or what address to target, etc.) between a controller and a package. Expressing a full command requires a concatenation of BTCs in a pre-established order that may be unique to each Flash package. Our solution expands the notion of BTCs by replacing them with µFSMs (micro Finite-State Machines), a more powerful way to generate waveform segments to control the reading, writing, and erasing operations of NAND flash memory cells. Every µFSM is parameterized and can issue many variations of the waveform segment. Some µFSMs are a combination of more than one ONFI BTC, while others find no similar BTC in the standard. The idea of parameterizing a µFSM may sound simple, but ultimately, describing segments as patterns rather than constant waveforms is what gives our scheme the expressive power to encode both basic and advanced operations.

Initial results of our new architecture based on a first implementation of µFSMs are given below in Section 3.3.1.

3.2.1.4 Integration with Use-Cases

Our first focus for Declarative Data Exchange is for UC2, though we plan to design solutions that will be as generic as possible and that will be useful for most use-cases. FRIB visited AALTO and CONVEQS in Summer 2023 to understand both the specificities of the use-case and the hardware acceleration that would be possible in that context. Subsequently, CONVEQS visited FRIB in Summer 2024 to further align.

Both the equipment on the road (see Figure 3-6 a, which depicts road units deployed on road pillars) and equipment in instrumented cars (see Figure 3-6 b) support multi-modal sensing (including cameras, LiDARS, and GPS sensors) and could include additional equipment for hardware acceleration (e.g., in the form of low-powered Xiling FPGAs or P4 accelerators).

Figure 3-7 gives an overview of how Declarative Data Exchange integrates with UC2. Offloading data will leverage D-RDMA (with further optimization provided by our CXL extensions and our new flash memory channel controller for those nodes equipped with further hardware). In addition to fast data transfers, we are extending Declarative Data Exchange with filtering capabilities, in order to efficiently identify and remove portions of the raw data that can be cut out prior to transmission (see bottom of Figure 3-7). In the context of UC2, license plate numbers could for example be filtered out prior to transmission (for privacy reasons), or part of a point cloud coming from a LiDAR installed on an instrumented car. The LiDAR extensions are currently being implemented in close collaboration with our work performed in WP4 (see LiDAR Processing Acceleration in D4.2).

Figure 3-6: Some of the nodes and hardware devices available in Helsinki for UC2 on the road (a) and in instrumented cars (b).

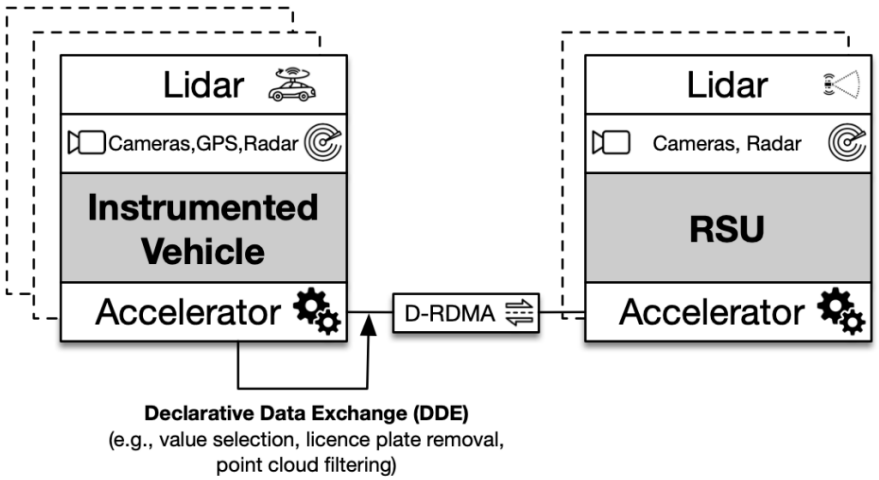


Figure 3-7: An example of a Declarative Data Exchange use for UC2

3.2.2 Offloaded Operators Implementation

Once data has been exchanged between two nodes in SmartEdge, the offload of the operation itself can start. Beyond simply leveraging more powerful or more connected nodes that will accelerate the tasks using better hardware, we provide special support for some of the important operations in SmartEdge that require hardware acceleration. We discuss the design and the implementation of four such operators below, for accelerating face blurring, information gathering, graph operations, and SPARQL query execution, respectively.

3.2.2.1 Offloading Face Blurring

CNIT contributes to this task by implementing a face blur detection and blur algorithm using OpenCV-compatible libraries, aiming to make it run on the ARM processor core available on the BlueField-2 DPU. Previous versions of the algorithm were based on *face_detect*5 (https://github.com/opencv/opencv/blob/4.x/samples/dnn/face_detect.cpp), and they used YuNet detection model to perform the face recognition task.

The YuNet detection model leverages on an open-source library for CNN-based face detection in images. The CNN model has been converted to static variables in C source files. The source code does not depend on any other libraries, and it may be compiled on any platform with C++ compiler.

The most recent version is based on YOLO9, which demonstrated equally good performance for real-time scenarios and allowed for a smoother integration with other algorithms for more complex applications or processing.

After detecting faces, the algorithm blurs the detected faces, granting individual's privacy before any other kind of image processing. A sample image is reported below (see Figure 3-8 below) to provide an example of the algorithm process. Preliminary performance results on our face blurring approach are given in Section 3.3.2.

Default image | Image with detected faces | Image with blurred faces

Figure 3-8: Sample results for the offloaded face blurring operation showing the original image (left), the image with faces detected by the algorithm (center), and the resulting blurred image (right).

3.2.2.2 Integrated Data View

The queryable database for FPGA-accelerated Roadside Units (RSUs) is a critical component in enhancing the real-time responsiveness and operational efficiency of Intelligent Transportation Systems (ITS). This system is designed to address challenges like latency, scalability, and the seamless integration of data from multiple sensors. It achieves this by providing a dynamic, real-time repository of traffic information, which is both accessible and modifiable in real time.

The need for such a database arises from the limitations of conventional RSU architectures, which often rely on cloud or centralized servers for data storage and processing. This traditional approach introduces significant latency, particularly when dealing with time-sensitive tasks like collision avoidance or adaptive traffic light control. A local, queryable database allows the RSU to perform data-intensive tasks at the edge, ensuring immediate access to information and reducing reliance on external networks. The database not only accelerates data processing but also enhances situational awareness by maintaining an up-to-date representation of the local

traffic environment. This is particularly beneficial for autonomous vehicles and advanced driver-assistance systems, which require precise, low-latency data for navigation and decision-making.

The database is accessible to various stakeholders in the ITS ecosystem. Smart vehicles equipped with Vehicle-to-Everything (V2X) communication technology can query the database to obtain real-time information about nearby objects or traffic conditions. This enables vehicles to optimize navigation, avoid collisions, and enhance fuel efficiency. Traffic management centers can use the database to monitor and manage traffic flows across intersections, dynamically adjust signal timings, and respond effectively to incidents. Additionally, other RSUs in a connected network can exchange queries to coordinate actions, such as rerouting vehicles or managing regional traffic congestion.

The architecture of the database is designed with hardware efficiency in mind, leveraging the capabilities of FPGA technology to manage memory and processing tasks. The data pipeline starts with the reception of raw sensor data, which is processed through a series of modules. These include data filtration (to remove outdated or irrelevant information), transformation (to standardize data formats), and association (to map sensor-specific identifiers to globally unique IDs). The database continuously updates itself with the latest information from connected sensors, ensuring that it remains a reliable source of real-time data.

Memory structure plays a pivotal role in the functionality of the database. The database uses a hierarchical memory architecture, divided into four categories based on the type of data being stored. The associations category, represented by the M_Assoc module, handles the mapping between global IDs and sensor-specific IDs, facilitating quick lookup of associated objects across multiple sensors. The detections category, which includes M_Det and M_Sub, stores data for newly detected objects. M_Det maintains information about all detections, while M_Sub focuses on the subset relevant to association and matching algorithms. The objects category, comprising M_Flag, M_Pre, and M_Obj modules, is dedicated to tracking objects that have been assigned global IDs. These modules manage various aspects of the object lifecycle, from flagging and prediction to storing current attributes. Finally, the costs category, encapsulated in the M_Cost module, holds the cost matrix used for associating detections with tracked objects. Among them, association-related and object-related databases are directly accessed when being queryed.

When a query is made, the database retrieves the required information based on specific keys. These keys include object identifiers (global IDs), spatial attributes (bounding box coordinates or positional data), motion characteristics (velocity and acceleration), and temporal markers (timestamps of the last update). Additionally, status flags indicate the validity and update status of objects, while predicted states offer forward-looking insights based on Kalman filter predictions. For diagnostic purposes, the database can also return cost values associated with object-detection matching.

By integrating sensor fusion algorithms with real-time data management, the database not only serves immediate needs but also lays the groundwork for future ITS applications. For example, it could enable predictive traffic management, where RSUs anticipate congestion patterns and adjust traffic signals preemptively. The modular and hierarchical design ensures that the database can adapt to increasing traffic demands and evolving technological requirements.

3.2.2.3 Offloading Graph Operations

As introduced in Deliverable D5.1, the first broad type of operation offload we consider is based on generic graph operations accelerated on powerful P4 nodes. The main accelerators we investigate in that context are programmable switches supporting P4 operations (but note that the techniques we design below can take advantage of further P4 accelerators, like DPUs or smart NICs that are also leveraged in other SmartEdge tasks).

As a reminder (see D5.1 for details), a programmable hardware switch is a platform unlike any other. It is divided into two semi-independent units, a control plane and a data plane, as Figure 3-10 depicts. The control plane is responsible for management tasks, e.g., bringing switch ports up or down. It usually consists of an x86 machine, an Intel Xeon in most cases, and it can run a common Linux distribution. The control plane functionality is available through C and Python libraries provided by the switch manufacturer.

The data plane is the component that receives packets from the network ports and forwards them back to their destination ports. The forwarding decision is the result of a computation— a networking protocol. In a programmable switch, the networking protocols are expressed as programs. These switches come with SDKs that can compile such programs into binaries they can run.

To explain how to program the data plane, we need to introduce a few concepts. The reason is that the programming model the switch supports is quite unique. The data plane consists of shared-nothing units called Match-Action Units (MAUs or, interchangeably, stages) arranged in a pipeline. An MAU is for a switch what a core is for a general-purpose x86 CPU. MAUs, however, have many constraints. Chief among them is that they can only send their results to the next MAU in the pipeline and can only receive input from the previous one.

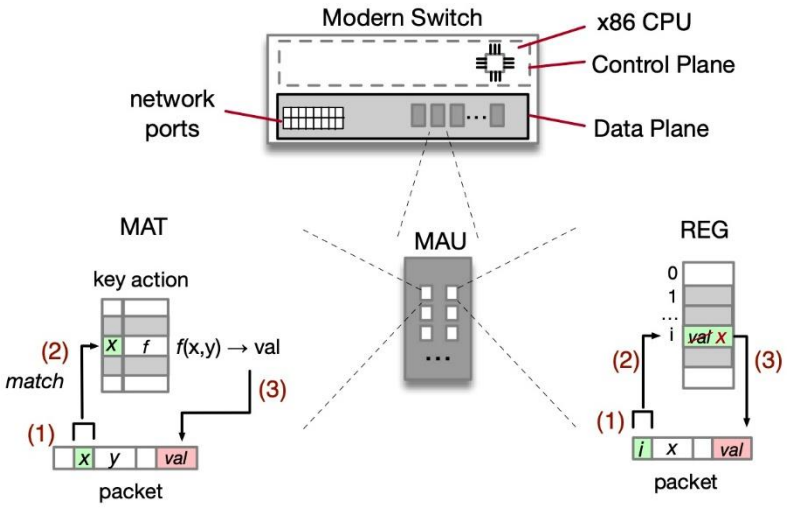


Figure 3-10: (Top) The switch is composed of a control plane and a data plane. The data plane has a pipeline of Match-Action Units (MAUs) with two types of storage: Match-Action Tables (MATs) and Registers (REGs). (Bottom) A MAT can read and alter a packet by: (1) selecting the field(s) to match; (2) performing the match, e.g., via equality comparison, and if an entry is found; (3) executing the matched entry's action, altering the packet's contents. A register works similarly, although the access to registers is positional.

We designed and implemented the offload of graph-based operations, most specifically Graph Pattern Mining (GPM), which is an important class of graph analysis. It finds all subgraph occurrences that match specific patterns. We decided to start with offloading GPM operations for two reasons: i) because GPM operations are, we believe, a very good match with the (limited) expressivity of P4 and hence were a great choice as a first offloading experiment and ii) because GPM operations are prominent in many applications, including listing cliques, finding motifs, and in mining frequent subgraphs.

A common approach to execute GPM algorithms is to iterate over all possible subgraphs and check if they match the desired pattern. This is done using a two-step process. First, subgraph enumeration extends subgraphs by adding one more node. This generates intermediate candidate subgraphs. Second, pattern analysis examines these intermediate subgraphs and looks for the pattern. Successful candidates are the ones that meet the pattern's matching criteria. This approach is iterative; meaning that the successful subgraphs that passed the pattern analysis are then extended again by adding one more node, and so on. The process ends when no further subgraphs can be extended.

We pioneered a brand-new technique to offload GPM tasks in the context of SmartEdge using a powerful node with advanced P4 capabilities. The overall idea behind the offloading is illustrated in Figure 3-11 below. Numbers in white illustrate the process from the edge nodes perspective: (1) a node syncs with the orchestrator (in this case running on the control plane) and solicits some work. (2) The requesting node is assigned a fragment of the problem, and (3) starts processing that fragment. (4) The resulting patterns are output by the nodes. The workflow on the accelerator, in red, is somewhat similar, even though the individual steps are performed differently: (1) the switch registers to the orchestrator to announce that it can accelerate part of the process. (2) the switch accelerator is assigned specific graph fragments (3) The nodes send the corresponding data (a subgraph) by D-RDMA to the accelerator (4) The subgraph is handled on the P4-accelerated node and the desired patterns are emitted as results.

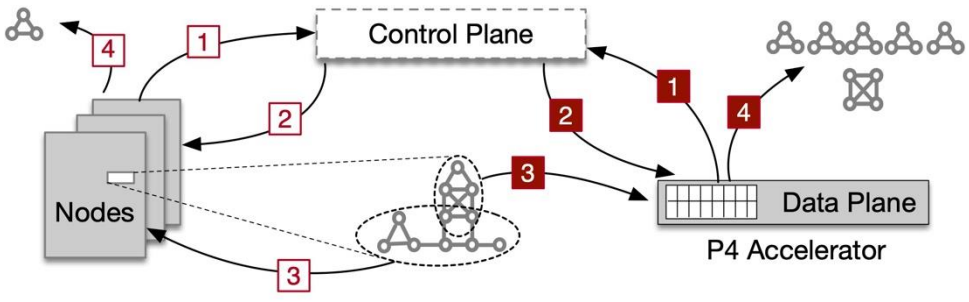


Figure 3-11: Our offloading framework main workflow. The edge nodes and a P4 accelerator (in this case a powerful and programmable P4 switch) operate independently but can offload GPM computations dynamically. In red, part of the problem is offloaded to the powerful P4 accelerator: (1) Resources or the P4 switch's data plane become available and (2) the switch accelerator is assigned specific graph fragments (3) The nodes send the corresponding data (a subgraph) by RDMA (4) The subgraph is handled on the P4-accelerated node and the desired patterns are emitted as a result.

While conceptually simple, the whole process is technically complex. It was the subject of full research paper presented at SIGMOD (the top venue for data-intensive systems) in 2023 (see

[Hussein23] for details). This complexity stems from two points: first, while we picked a relatively simple task (GPM) to offload, the programming model of P4 is limited and reformulating GPM in a P4-compatible, feed-forward fashion was technically challenging. Second, the P4 switch we used is actually very powerful, and optimizing the overall offloading process to take full advantage of the accelerator was technically challenging. We report on a number of new results on this task in Section 3.3.2 below.

3.2.2.4 Offloading SPARQL Execution

Our second focus in terms of offloading generic operations in SmartEdge leveraging P4 focuses on SPARQL. SPARQL is a very generic language, that is heavily used to describe both data and operations in SmartEdge. Offloading SPARQL operations to P4-accelerated devices could considerably accelerate the SmartEdge workload, but is technically challenging since SPARQL is a full-fledged, expressive language with many different and powerful operators.

We focus on accelerating a subset of SPARQL in P4, focusing on key operations for SmartEdge. Our first interest is in conjunction and disjunction of triple patterns: a triple pattern is a basic query structure in RDF/SPARQL that matches a subject-predicate-object relationship, potentially using variables for flexible data retrieval. The support of triple patterns allows us to support a wide variety of use-cases, without having to implement all the complexities of the language. As follow-up goals, we plan to focus on Property paths using advanced path expressions supported by SPARQL 1.1 (e.g., *AlternativePath or ZeroOrMorePath*), and complex aggregate operations supporting *GroupBy* and *Having*.

We have generalized the design that we took for our first offloading operation (illustrated in Figure 3-11) for those points. In terms of techniques, we are basing our offload on part of our GPM framework (since both GPM operations and advanced path operations require exploring parts of graphs in an iterative manner). For triple patterns, we are extending our initial, previous work on offloading relational operators [Lerner2019].

The design and implementation of this offloading solution is a collaboration between FRIB and TUB that started in mid 2024 and will continue until the end of the project.

3.2.2.5 Integration with Use-Cases

The face blurring component integrates with Use Case 3 (Collaborative Robotic Moves), which aims to offer autonomous robots controlled with a swarm intelligence system able to grant superior reliability, efficiency and security in a smart factory scenario. Robots, in this scenario, aim to be autonomous in decision-making procedures and will be able to interact with other robots and with human operators. Human-device interaction must occur granting full respect of an individual's privacy on top of any other data analysis and exchange performed by the SmartEdge service. Image processing necessary to grant privacy should not impair communication rate within the swarm intelligence system. To offer respect of privacy, any acquired image and/or video must be processed with proper face-detection and blur algorithm before any further analysis or communication performed by the swarm.

The other offloaded operators have been designed with UC2 in mind, though they could be easily integrated with further use-cases as they offload generic operations that could be useful to most use-cases. Figure 3-12 gives an overview of how our offloaded operators integrate with UC2. As explained above, lighter SmartEdge nodes exchange data through Declarative Data Exchange to more powerful nodes, like in intelligent Road Side Unit (RSU). The RSU can then

take over complex computations more efficiently using hardware acceleration (called *Declarative Processing [DP]* in Figure 3-12). Advanced operators such as those described above (e.g., face blurring, complex graph or SPARQL operations) can then run either on a DPU or on a hardware component with advanced P4 functionalities.

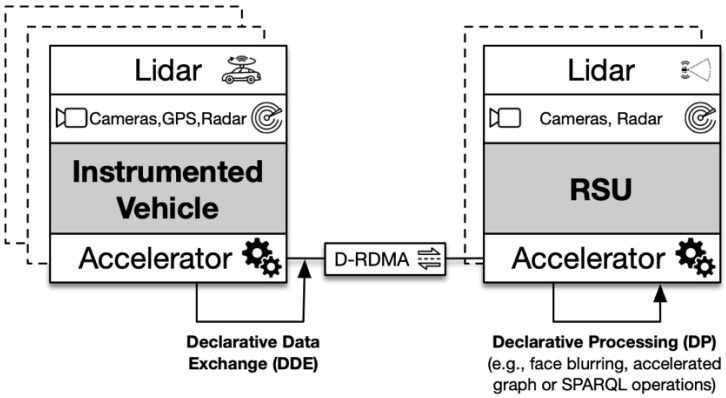


Figure 3-12: Integration of offloaded operators (through Declarative Processing) with UC2

3.2.3 Runtime Optimizer Implementation

Finally, we designed a component running on the SmartEdge orchestrator to optimize the offloading process of complex, data-driven operations. The overall design of this runtime optimizer is given below in Figure 3-13.

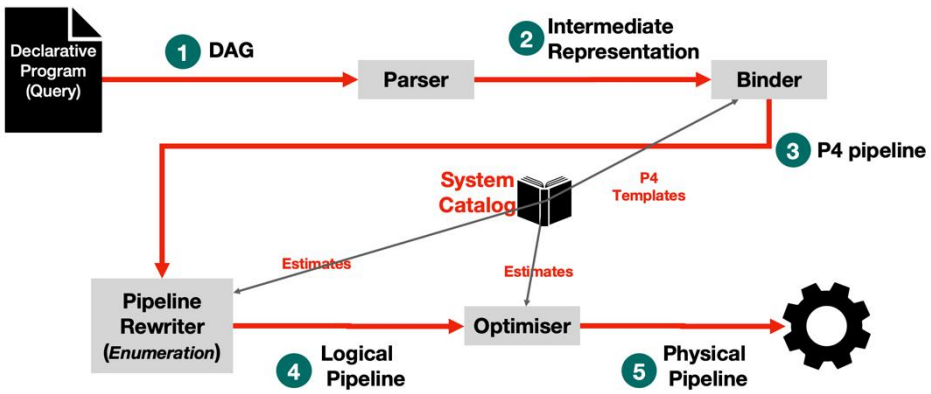


Figure 3-13: The design of our runtime optimizer for offloading operations in SmartEdge; as most components from this task, the optimizer will take a declarative specification of complex operations to offload (1), will translate this high-level representation of the complex operation into some intermediate representation (2) that can be translated into P4 and optimized before instantiating the optimized pipeline that needs to be run on a one or several accelerated node.

As with most other components designed in the context of this task, the optimizer uses high-level, low-code declarative constructs as input, more specifically a declarative program describing the complex operations to offload. The optimizer then represents this program into a Directed Acyclic Graph of operations, that are then translated into some intermediate representation to be optimized. The optimization process itself leverages a system catalog

containing set of rules to rewrite and optimize the offloading operation into an efficient physical pipeline that can be run on one or several P4-accelerated nodes (in a way similar to the optimization of complex declarative queries in data-intensive systems). The optimization process spans three different layers, as illustrated in Figure 3-14: 1. physical optimizations (e.g., based on statistics to pick the most efficient physical plan to run the complex offloading), 2. logical optimization (e.g., based on logical rules on how to move various operators to obtain more efficient offloading plans) and 3. hardware-specific optimizations (e.g., fusion of operators that can be jointly run in a common P4 pipeline, for example fusing a join and a group-by operation).

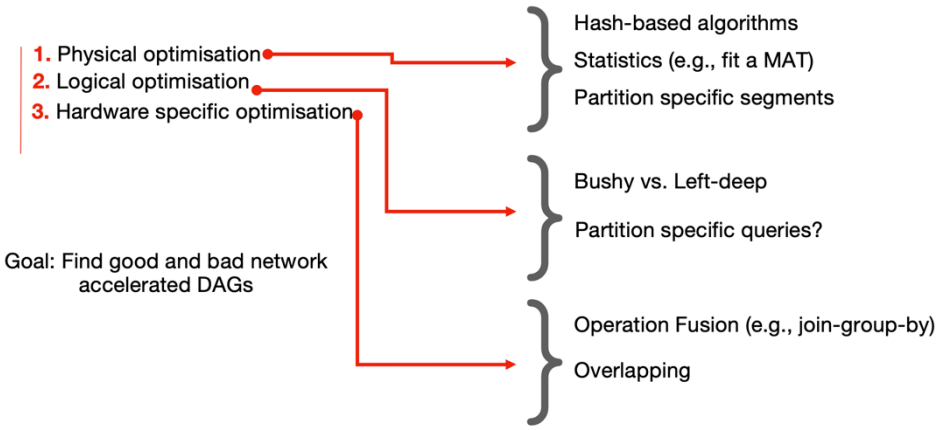


Figure 3-14: the three layers of optimization that will be supported by our runtime optimizer.

We describe below a series of techniques we have designed and started implementing in this context, including:

3.2.3.1 Optimized Resource Allocation on P4 Hardware

A key challenge of offloading complex operations to a P4 hardware accelerator is the allocation of resources. Programmers need to interact with the compiler to define memory needs. Unfortunately, coding P4 hardware accelerator like programmable switches involves a steep learning curve. Managing the logic and memory requirements is often accomplished through access to NDA-protected documentation or trial-and-error coding lacking proper tool support. We exemplify the allocation of memory for executing a join operation using the Intel Tofino2 switch. This switch allows a total of 48 SRAM blocks per stage.

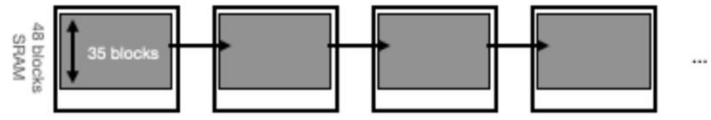
Suppose that we want to execute a program with one join operation and we build the hash table of the base relation of the join with one register, but the base relation R requires the whole SRAM of the processor stage to build its hash table. The open-source documentation says that a single register can use a maximum of 35 SRAM blocks plus one additional block for control, as shown in Figure 3-15 (a). The intuition is to code the maximum number of allowed blocks, but we may find two problems using this approach.

First, if we try to use the remaining 12 blocks for building a second hash table, the logic of our whole operation becomes more complex, as the hash tables would have different sizes. The current extension of the P4 compiler to implement registers cannot increase the complexity that

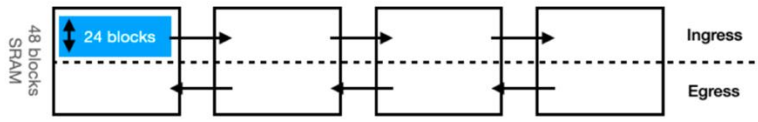
much. The implementation of registers is restricted to executing two if-else pairs and read-modify-write operations on a pair of registers. Second, contrary to the open-source documentation, the current P4 compiler does not allow the use of all 35 documented blocks, rejecting the program. Figure 3-15 (b) illustrates that a register is limited by the P4 compiler to allocate a maximum number of 24 blocks. It also shows that the implementation of registers only uses a single stage, another limitation in the allocation of memory space that must be treated by the programmer.

Our solution is to implement multiple identical size registers in the same pipeline to have access to more SRAM space as illustrated by Figure 3-15 (c). To facilitate the allocation of resources with less human interaction, we extend the P4 compiler to generate hardware code for optimizing specific operations. These operations create registers to run the operations in a specific pipeline and occupy all available blocks in the processor stages.

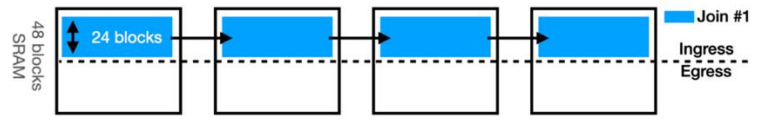
This extension offers three advantages. First, we generate a simpler code logic. The code is generated in only one control (e.g., code is generated in the ingress control). Second, we can occupy all the available memory. Third, we enhance parallelism with multiple equally sized registers within the same stage and pipeline, ultimately improving performance. For example, in our switch, we generated code to occupy a total of 40 registers in the ingress pipeline divided into 2 registers with 94,200 entries per stage. Figure 3-15 (d) illustrates this with an allocation of 24 blocks for each register with two registers allocated per stage.



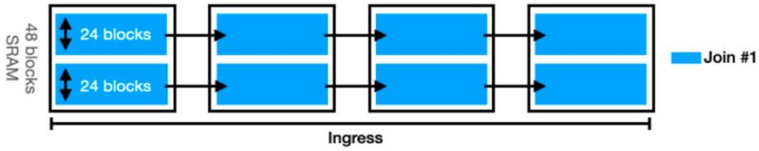
(a) Maximum documented SRAM allocation per register for data: 35 out of 48 blocks per stage [15]



(b) Naive SRAM allocation: 1 register of 24/48 blocks in the ingress pipeline



(c) Allocation of multiple stages: 1 registers of 24/48 blocks per stage in the same pipeline (eg., ingress)



(d) Allocation of the entire SRAM allows parallelism: 2 registers of 24/48 blocks per stage in the same pipeline (eg., ingress)

Figure 3-15: One-operation query SRAM allocation on an Intel Tofino2 architecture: naive vs. optimized resource allocations.

3.2.3.2 Operator Fusion

We designed and implemented optimizations involving very common operations, i.e., joins and group-by operations. We introduce the fusion operation with notation $R \bowtie S$ that combines the network execution of join and group-by operations. The fusion operation optimizes the execution of these operations by eliminating the need for a separated group-by hash table and the external drain procedure proposed in previous work. This leads to simplified logic and reduces the memory footprint compared to other approaches.

Algorithm 1: Fusion query plan

```
1  stage 1..N(tbl : int[SIZE/2], agg : int[SIZE/2]):
2      upon receiving packet, metadata m do
3          match pkt.build ∧ ¬m.drop
4              if tbl[m.hash] = ∅ then
5                  tbl[m.hash] ← pkt.join_key
6                  agg[m.hash] ← pkt.value
7                  m.drop ← true
8              else if tbl[m.hash] = pkt.join_key then
9                  agg[m.hash] ← agg[m.hash] + pkt.value
10                 m.drop ← true
11             else
12                 m.drop ← false
13         match pkt.probe ∧ ¬m.found
14             if tbl[m.hash] = pkt.join_key then
15                 pkt.value ← agg[m.hash]
16                 m.found ← true
17             m.drop ← ¬m.found
18     return
```

Figure 3-16: the optimized fusion query plan with two distinct phases (building and probing the fusion table)

Algorithm 1 in Figure 3-16 describes the fusion operation. The fusion operation is divided into two phases: a build and probe the fusion table, respectively. In the first phase, we built the fusion table with tuples from the relation where the join key also serves as the group key (line 5). The fusion hash table uses a collision chain structure to occupy more memory across the processor stages. The incoming tuples from the build table S read the hash table entry in the collision chain. If the entry is empty or the key matches, the fusion and aggregation tables are updated. Otherwise, if the entry is occupied and the key does not match, the packet moves to the next stage until the key gets inserted. In the aggregation hash table, the values are summed and indexed by the same hash (lines 6 and 9). We mark the packet as dropped after updating both fusion and aggregation tables. The algorithm guarantees that the collision chain of both hash tables extends throughout all processor stages, maximizing memory utilization.

Following the creation of the fusion table, we probe the fusion table. The tuples of the probe table check the collision chain and, in the case of a match, their packet moves forward to the

group-by stage. If a key is found in the fusion table (line 14), its packet is updated with the aggregated value (line 15), and the final result is sent to the server without requiring any external drain procedure. However, the implementation of the fusion operation requires two adaptations. We adapt the packet format to accommodate the aggregated value. We also adapt the wire format with MAUs that can read and write the aggregated values to the packets.

Figure 3-17 illustrates the fusion operation step by step. On the left side of the figure, Source S has 5 tuples and is sent to the network switch to build the fusion hash table. In the middle, the fusion hash table entries are indexed by a hash key calculated from column $S.c$. The fusion hash table also stores the aggregation values, which combine data from column $S.d$ based on the same hash key. Finally, on the right side, the other source R has 5 tuples and is sent to the switch for probing the fusion table. If a row in R matches an entry in the fusion table, its packet gets updated with the corresponding aggregation value and is forwarded to the server. Otherwise, any rows in R that do not find a match are eventually dropped.

Build(S)

c	d
1	20
1	30
2	50
3	60
5	70

Fusion Group by Hash Table

S.c	sum(S.d)
1	50
2	50
3	60
5	70

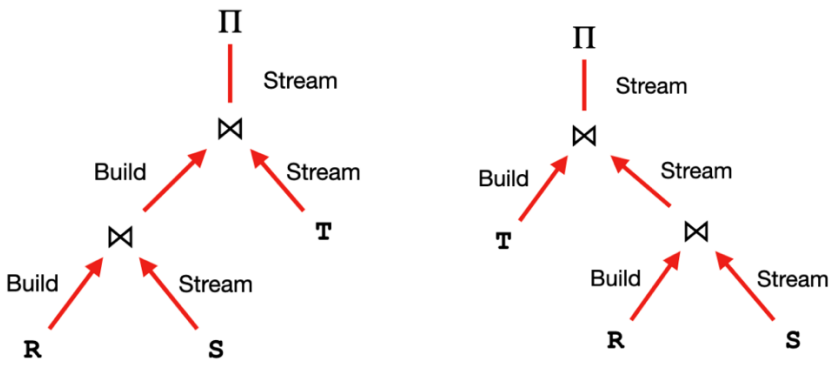
Probe(R) Updating values

a	b	i+2	drop
1	x	50	0
2	x	50	0
3	x	60	0
4	x	-	1
5	x	70	0

Figure 3-17: Query executing the fusion operation. We build consider a hash-table with relation (i.e., source) S and aggregation attribute is $S.d$. We stream source R. If there is a key match $S.c = R.a$, we update the packet with the aggregation value and forward it, otherwise we drop the packet.

3.2.3.3 Optimizing Join-Join Topologies

Another important case we successfully optimize on a P4 accelerator is complex queries including several joins. Figure 3-18 illustrates two different cases with a sequence of two join operations. We introduce two different ways to build the hash tables and optimize the join operations. In the left-deep plan, the hash tables are built from the result of each join. In the right-deep plan, we build the hash tables out of the relations R and T, and we stream relation S to probe the hash tables in a pipelined fashion.



(a) Left-deep plan **(b) Right-deep plan**

Figure 3-18: Different execution strategies for query ($R \bowtie S \bowtie T$). Left-deep plan builds the hash tables with R and the output of the join operation. The right-deep plan builds the hash tables with the relations R and T. It streams S and the join output.

Figure 3-19, left (Algorithm 2) describes the execution of the left-deep query plan. We build the first hash table with the base relation R. For that, the packet metadata from relation R is marked as build (line 3). We call this hash table tbl_{lo} because it is the lowest table in the query plan tree (line 4). The hash table tbl_{lo} is built within the processor stages 1..K, where K is a parameter defined according to the amount of space required. When the packets from relation R arrive in the switch, tuples that find an empty place in tbl_{lo} are inserted and their packet are dropped (lines 4-6). Otherwise, the packets move to the next stage in the collision chain as discussed in the previous section. The tuples of the relation S probe the relation R on lines 9-12. Tuples with a matching key move on in the pipeline building the second hash table and having their packet dropped (lines 16-21). The second table and the highest one in the query plan tree is called tbl_{hi}. Finally, we mark the packets from the last relation T only as probe packets (lines 9 and 22). These tuples probe both hash tables tbl_{lo} (lines 9-12) and tbl_{hi} (lines 22-26). If a tuple matches its key, its packet is forwarded to the server (line 25), otherwise it is dropped (line 26).

Figure 3-19, right (Algorithm 3) describes the execution of the right-deep query plan. We assume the two base tables R and T have functional dependencies with relation S. In the algorithm, we differentiate the packets of the relations by the build and probe metadata fields. We mark packets of relation R only with build, while we mark the packets of relation S only with probe. As in the left-deep algorithm, we call the first hash table tbl_{lo} and the second table tbl_{hi} representing where they are placed in the query plan tree. The hash tables can be built at the same time (lines 3 and 9). In case of a matching key, we flip the memorization bit in lines 17 and 19. At the end, we run a special operation in both memorization bits to decide between forwarding or dropping the packets (lines 20 and 21). Our next steps include a full experimental validation on those optimization schemes.

Algorithm 2: Left-deep query plan

```
1  stage 1..K(tbl_lo : int[SIZE]):
2      upon receiving packet, metadata m do
3          match pkt.build ∧ ¬pkt.probe ∧ ¬m.drop
4              if tbl_lo[m.hash] = ∅ then
5                  tbl_lo[m.hash] ← pkt.join_key
6                  m.drop ← true
7              else
8                  m.drop ← tbl_lo[m.hash] = pkt.join_key
9          match pkt.probe ∧ ¬m.found_lo
10             if tbl_lo[m.hash] = pkt.join_key then
11                 m.found_lo ← true
12             m.drop ← ¬m.found_lo
13     return
14 stage K + 1..N(tbl_hi : int[SIZE]):
15     upon receiving packet, metadata m do
16         match pkt.build ∧ pkt.probe ∧ ¬m.drop
17             if tbl_hi[m.hash] = ∅ then
18                 tbl_hi[m.hash] ← pkt.join_key
19                 m.drop ← true
20             else
21                 m.drop ← tbl_hi[m.hash] = pkt.join_key
22         match ¬pkt.build ∧ pkt.probe ∧ ¬m.found
23             if tbl_hi[m.hash] = pkt.join_key then
24                 m.found_hi ← true
25             m.found ← m.found_lo ∧ m.found_hi
26             m.drop ← ¬m.found
27     return
```

Algorithm 3: Right-deep query plan

```
1  stage 1..N(tbl_lo : int[SIZE], tbl_hi : int[SIZE]):
2      upon receiving packet, metadata m do
3          match pkt.build ∧ ¬pkt.probe ∧ ¬m.drop
4              if tbl_lo[m.hash] = ∅ then
5                  tbl_lo[m.hash] ← pkt.join_key
6                  m.drop ← true
7              else
8                  m.drop ← tbl_lo[m.hash] = pkt.join_key
9          match pkt.build ∧ pkt.probe ∧ ¬m.drop
10             if tbl_hi[m.hash] = ∅ then
11                 tbl_hi[m.hash] ← pkt.join_key
12                 m.drop ← true
13             else
14                 m.drop ← tbl_hi[m.hash] = pkt.join_key
15         match ¬pkt.build ∧ pkt.probe ∧ ¬m.found
16             if tbl_lo[m.hash] = pkt.join_key then
17                 m.found_lo ← true
18             if tbl_hi[m.hash] = pkt.join_key then
19                 m.found_hi ← true
20             m.found ← m.found_lo ∧ m.found_hi
21             m.drop ← ¬m.found
22     return
```

Figure 3-19: Optimized P4 algorithms for the execution of join-join query plans

3.2.3.4 Integration with Use-Cases

Our runtime optimizer is generic and can be used to optimize a wide range of complex operations. We illustrate how it integrates with UC2, along with the other component of Task 5.2 described above, in Figure 3-20.

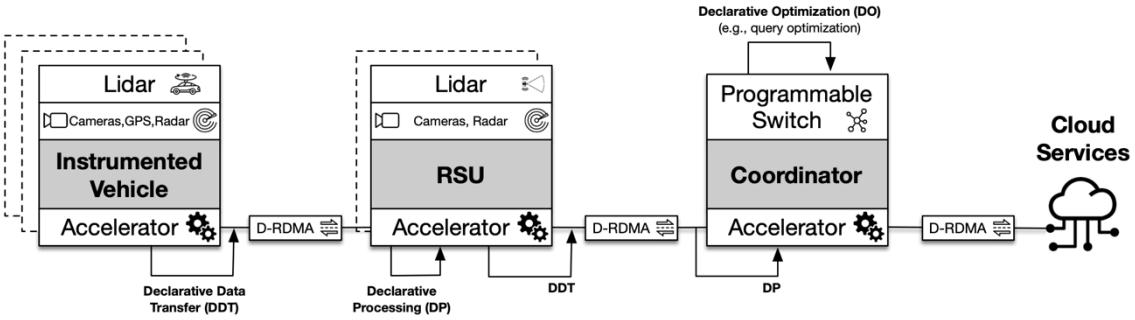


Figure 3-20: integration of the various components described in Task 5.2 with use-case UC2

Several instrumented vehicles (left of the figure) capture local data e.g., through LiDARs. They stream their data to (one or several) nearby Road Side Units (RSUs) having more computing capabilities and better connectivity. Some processing takes place at the RSUs, in our example typically point cloud filtering and processing to extract relevant data from the point cloud data streamed by the vehicles. The RSUs transmit their processed LiDAR data to a coordinator (which can run on the RSU also, or on the orchestrator itself depending on its connectivity and capabilities). The coordinator performs more complex operations on the various data feeds it

receives, e.g., performing scene understanding from the processed LiDAR streams. Finally, it runs a complex query to integrate (i.e., join) the resulting data (taking advantage of our runtime optimizer to optimize the query, see Declarative Optimization on top of Figure 3-20). It can then make sensible traffic decisions based on the integrated, high-level semantic data that is produced from the overall process. Optionally, it can also save part of the semantic model by efficiently transferring a subset of the results to the cloud (right side of Figure 3-20).

3.3 Empirical Results and Demonstration

We describe below the first results we obtained for the most advanced components described above in Section 3.2. As our components often include hardware programming, the implementation efforts involved are substantial and we are still in the process of finalizing several of them. We will report additional empirical results in D5.3.

3.3.1 Declarative Data Exchange

We started by running several base experiments to verify the viability of our Declarative Data Exchange stack (see Section 3.2.1 above). In a first experiment, we programmed an FPGA-based NIC to empirically assess the performance of D-RDMA operations and evaluate its efficiency under varying transfer sizes. Those experiments were originally reported in [Ryser 22]. We used a repurposed version of Corundum, a high-performance network card logic that supports several FPGA-based platforms. The reads were sequential and target a large 1GB memory region. Figure 3-21 shows the results, both in terms of latency and throughput, for DMA operations reading increasingly large chunks of data from the network card.

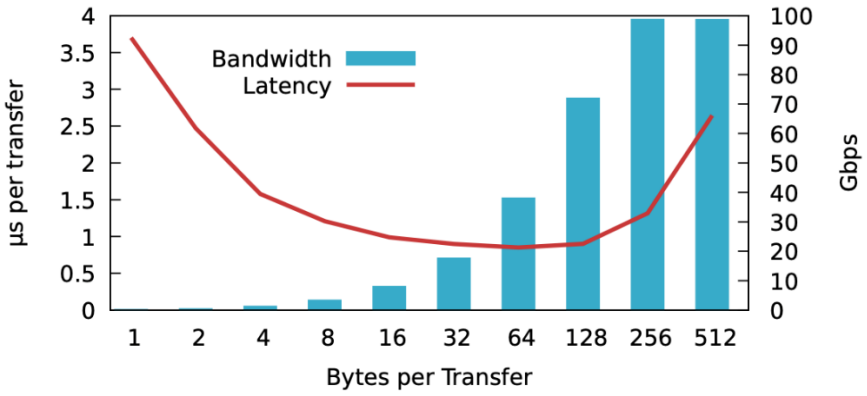


Figure 3-21: D-RDMA Latency and bandwidth results of reading increasingly large chunks of data from a network card.

The throughput results are as expected: the larger the transfer, the better the throughput. To reach the peak throughput, the operation needs to move at least 256 bytes, which is also the maximum payload size of the PCIe link. In terms of latency, we note that very small reads can be higher than that of medium ones. The added latency for larger transfers is due to the transfer of multiple packets when the maximum payload size of the PCIe link (256 bytes) is reached.

To better understand the transfer performance of non-contiguous regions, we measure the latency of reading increasingly large chunks of data with different strides. Figure 3-22 shows the results of this experiment. We observe that for most strides (i.e., more than 8 bytes between transfers) the latency times remain minimal.

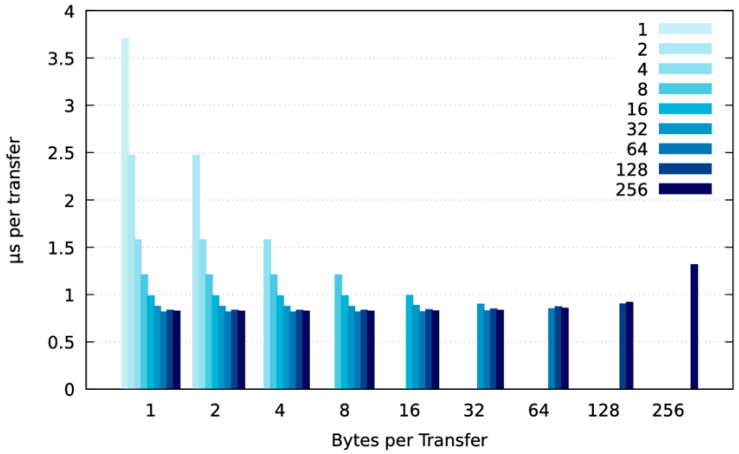


Figure 3-22: D-RDMA Latency of reading increasingly large chunks of data with different strides.

3.3.2 Face Blurring

The Face Blurring component is designed to be light-weight, fast and accurate, granting an accuracy between 0.77 and 0.89. It allows to recognize face of pixel between around 10x10 to 300x300 pixels due to the training scheme and can perform on multiple faces in the same picture. Performances of the CNN-based face detection method on intel CPU are reported online (https://github.com/ShiqiYu/libfacedetection).

The algorithm has been tested on images and videos. Preliminary performance results of the face blur algorithm have been tested using the perf tool available on Linux, and results are reported below:

Performances for face_blur algorithm tested on an image

FSP		25.7
Time elapsed		0.0675 s
	#	
Task-clock	364,47	5.40% CPU utilized
Context-switches	373	1.15 /sec
CPU-migrations	19	45.35 /sec
Page-faults	19210	46.09 /sec

The most recent version is based on YOLO9, which demonstrated equally good performances for real-time scenarios and allowed for a smoother integration with other algorithms for more complex applications or processing. The data in the table are median results of five consecutive tests.

The algorithm has then been assessed in a communication scenario, emulating the cooperation/communication among different edge computing nodes, leveraging RDMA-based communication for data exchange instead of traditional TCP-based communication. A testbed encompassing two DELL PowerEdge servers equipped with Bluefield DPU at 25G has been implemented. Prometheus is used to collect real time metrics on CPU load from both servers and DPUs.

Results are depicted in Figure 3-23. As expected, the CPU load is significantly lower when using RDMA compared to TCP, with RDMA requiring nearly half the CPU resources

Figure 3-23: (top) testbed setup for RDMA assessment. (middle) Graphana view of RDMA performance. (lower left) Performance using traditional TCP data transfer. (lower right) performance using RDMA.

3.3.3 Accelerated Graph Operator

We ran a number of preliminary experiments to show the viability of our P4 offloading technique for accelerating graph operations. Our setup is as follows: we run our experiments on a cluster of 16 nodes. Each node contains two sockets, each with an 8-cores 2.1 GHz Intel Xeon CPU E5-

2620v4, and a total of 128 GB of main memory. The servers run Ubuntu Linux 22.04.1. The servers are interconnected through a programmable hardware switch with 100 Gbps ports based on the Tofino 1 chip. We use Mellanox's ConnectX-5 100 Gbps network cards on all the servers.

We use several standard graph datasets in our experiments. The main properties of these graphs are given in Figure 3-24. We tested our approach on two standard graph operations: listing k-cliques (where cliques are complete subgraphs of size k) and finding k-motifs (where motifs are connected patterns of size k).

Graph	—V(G)—	—E(G)—	$\bar{d}$	4-nodes		5-nodes	
				cliques	subgraphs	cliques	subgraphs
Mico	100k	1.08M	22	515M	984M	19B	36B
Skitter	1.7M	11M	13	149M	2.1B	1.1B	27B
Wiki	2.3M	4.6M	3	65M	4.8B	383M	49B
Patents	3.7M	16M	10	3.5M	68M	3M	44M

Figure 3--24: main properties of the standard graphs used for our experiments

The results of our offloading approach are extremely promising: we outperform the state of the art (Fractal [Dias19]) by more than an order of magnitude (i.e., more than 1000%) on average, as Figure 3-25 below illustrates.

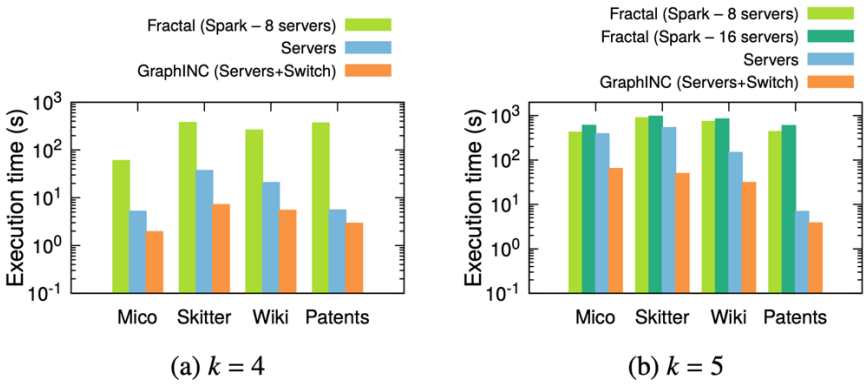


Figure 3--25: Comparison of running a graph pattern mining task (k-cliques) entirely on nodes (servers) using a state-of-the-art framework (Fractal [Dias2019]) versus offloading the workload to the switch using our acceleration approach called GraphINC.

The results on our second task, k-motifs, are impressive as well (see Figure 3-26), as our approach outperforms the state-of-the-art (Fractal) by more than 80x in the case of k = 3, and by more than 500% in the case of k = 4.

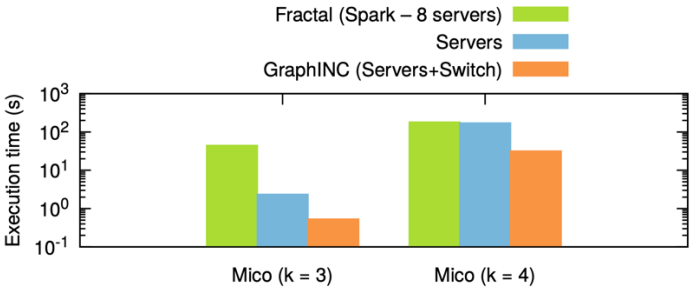


Figure 3-26: Comparison of running a graph pattern mining task (k-motifs) entirely on nodes (servers) using a state-of-the-art framework (Fractal [Dias2019]) versus offloading the workload to the switch using our acceleration approach called GraphINC.

4 SWARM COORDINATION AND ORCHESTRATION

This section presents ongoing work on the initial implementation of the orchestration, adaptive coordination, and optimization mechanisms for SmartEdge swarms. The primary focus here is on our progress within Task T5.3.

The content of this section is organized as follows:

Section 4.1 provides an overview of the main components delivered by Task T5.3 and their features. It introduces the fundamental principles behind the adaptive coordination and dynamic orchestration mechanisms applied to SmartEdge swarm intelligence. This section also examines how components from Task T5.3 integrate with artifacts developed by other tasks and work packages.

Section 4.2 details the implementations of each component:

> **Section 4.2.1** presents the finalized architectural design and initial implementation of the Swarm Adaptive Coordinator artifact (A5.3.1). This artifact enables adaptive coordination within swarms, allowing them to dynamically adjust their behavior in response to environmental changes.
> **Section 4.2.2** describes the final architecture and initial implementation of the Dynamic Task Orchestration artifact (A5.3.2). This component supports orchestration across the swarm, enabling efficient task distribution among swarm nodes and coordinating their collaborative interactions.
> **Section 4.2.3** reviews the current progress on implementing the Optimizer artifact (A5.3.3). Designed to enhance swarm efficiency, this artifact optimizes task allocation and resource usage, further supporting the adaptive capabilities of the SmartEdge swarm.

Section 4.3 demonstrates the features of the orchestration and coordination mechanisms.

4.1 MAIN COMPONENTS AND FUNCTIONALITIES

Task 5.3 provides SmartEdge swarm intelligence with mechanisms to autonomously form the swarm, construct processing pipelines, deploy and execute applications, and optimize performance in response to dynamic changes in swarm context.

In the initial phase, applications are created using the Low-code IDE (A5.4.4), which converts application logic into processing pipelines composed of interdependent tasks. This pipeline is structured as a semantic program, implemented through CQELS-RL, as described in Section 5.4.1 of Deliverable D5.1. To deploy and execute applications within the SmartEdge swarm the semantic program is registered with a designated SmartEdge coordinator. This coordinator manages the orchestration, task execution, and coordination throughout the swarm.

Figure 4-1 illustrates the architectural layout of the components that facilitate coordination, orchestration, and optimization within the SmartEdge swarm. On the left side of Figure 4-1, the architecture of a SmartEdge coordinator node is depicted, while the right side represents a SmartEdge swarm member node. In the coordinator node, the Orchestrator is responsible for processing the semantic program, compiling it into executable plans, and directing these plans

across a network of SmartEdge nodes. This ensures that tasks are allocated and coordinated among swarm members, following the specified logic of the application. The Coordinator is responsible for forming and maintaining the swarm. It keeps track of available nodes, managing their status and ensuring that all eligible nodes are ready and able to participate in application execution. The Optimizer continuously monitors the resources and performance of swarm members to adaptively optimize the execution plan. It ensures efficient resource utilization, balancing workload across nodes based on their availability and capacity.

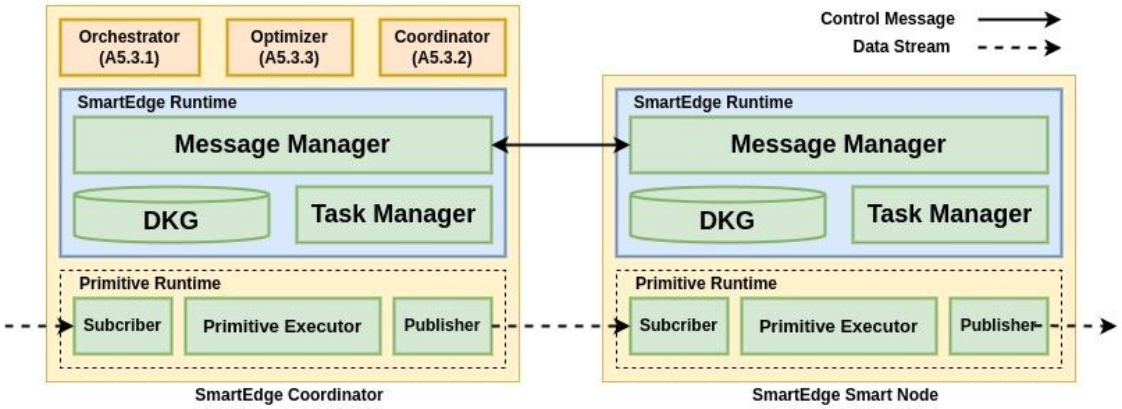


Figure 4-1. Overview of the building blocks for adaptive coordination, dynamic orchestration, and optimization within the SmartEdge swarms.

The functionalities of the components implemented in Task T5.3 are built upon core components developed in Task T5.4. The communication between components and swarm nodes is managed by the Message Manager within the SmartEdge Runtime (A5.4.1), which handles control messages essential for orchestrating tasks across the swarm. The Dynamic Knowledge Graph (A5.4.2) acts as a contextual knowledge base, storing key information about the swarm, such as node availability and resource capacity for each node. By providing a shared, real-time view of the swarm's status, the DKG enables the Optimizer, Coordinator, and Orchestrator to dynamically adjust their actions in response to changing conditions, ensuring efficient and adaptive task management throughout the swarm.

The execution of a *semantic program* in SmartEdge is a processing pipeline of interconnected SmartEdge processing primitives (as outlined in Section 5.4.3 of Deliverable 5.1). These primitives are fundamental operations within the data processing pipeline defined by the semantic program, assigned to and executed by each member of the swarm. The Primitive Runtime, which is responsible for executing these primitives, includes three main components: Subscriber, Primitive Executor, and Publisher. The Subscriber component receives data streams or updates from other nodes or external sources, serving as the entry point for incoming data that is subsequently processed according to the application's requirements. Acting as the core of the processing pipeline, the Primitive Executor handles the execution of each primitive task, with oversight by the Task Manager to ensure alignment with the overall execution plan. Once data processing is complete, the Publisher component sends the results to other nodes or back to the SmartEdge Coordinator, enabling continuous data flow and collaboration across the swarm.

Figure 4-2 illustrates a sequence diagram representing the process of registering, compiling, and assigning tasks within the SmartEdge swarm, where the Orchestrator, Coordinator, and Dynamic Knowledge Graph (DKG) collaborate to distribute tasks across nodes based on their capabilities.

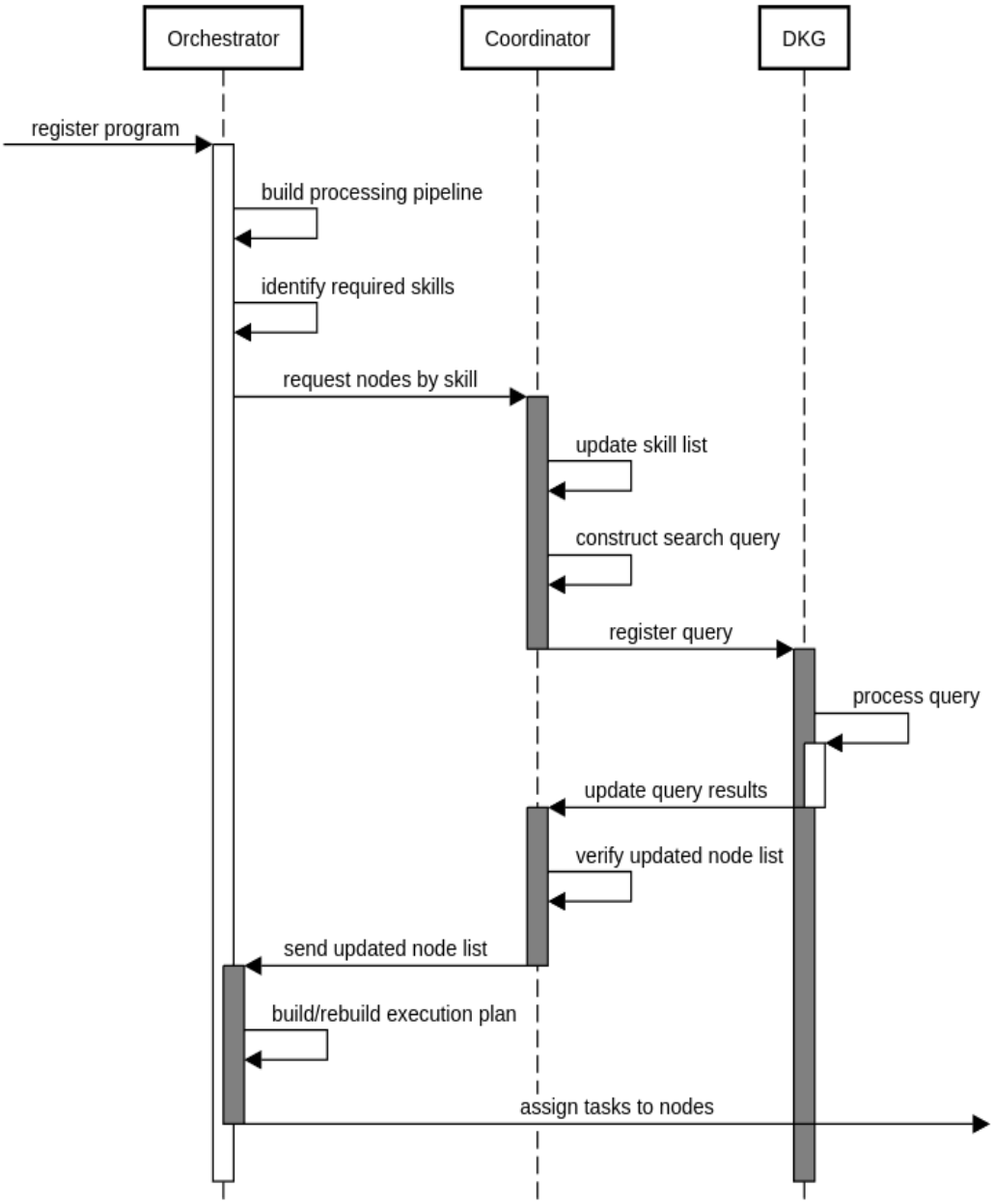


Figure 4-2. Sequence diagram of coordination and orchestration process in the SmartEdge system

The process begins with the Orchestrator registering a program and activating itself to initiate task management. It then compiles the program and constructs a processing pipeline. Following this, the Orchestrator identifies the specific skills needed to execute the tasks defined in the program and sends a request to the Coordinator to obtain a list of nodes capable of performing these tasks.

The Coordinator updates and maintains the skill list. It constructs a continuous query and registers it with the DKG to search for and keep updated records of which nodes possess the required skills and are available. The DKG responds with an updated list of nodes that meet these criteria. The Coordinator then forwards this list of eligible nodes to the Orchestrator, providing the necessary information for task allocation.

With the list of available nodes, the Orchestrator builds a detailed execution plan that aligns tasks with the skills and capacities of each identified node. Finally, the Orchestrator assigns and

distributes tasks across the swarm for execution, ensuring that each task is matched to the nodes best suited to complete it. This coordinated approach optimizes resource usage and enables efficient, distributed task execution across the SmartEdge swarm.

Changes in the state of the swarm are captured by the DKG, which then updates the query results and forwards them to the Coordinator. The Coordinator verifies these changes to determine if the Orchestrator should be notified. If needed, the Orchestrator rebuilds the execution plan for the processing pipeline based on the updated list of devices and their capacities.

4.2 COMPONENTS IMPLEMENTATION

This section presents the initial implementation of the Swarm Adaptive Coordinator and Swarm Dynamic Orchestrator, both developed in Java 11. To process RDF data, we integrate essential components from the RDF4J library for efficient RDF handling. Additionally, to enable interaction with the Message Manager implemented in ROS2 Python, we use the JPype library, allowing seamless communication between Java and Python modules. This setup supports real-time message exchange and coordination across the SmartEdge swarm.

4.2.1 Swarm Adaptive Coordinator

The formation of a SmartEdge swarm relies on the features of Artifact A4.2 (MS4.2), the Swarm Coordinator, a networking layer component provided by WP4. When SmartEdge smart nodes enter the swarm area, they connect to the Wi-Fi network established by the Access Points. Once connected, each smart node is assigned an IP address, and the Swarm Coordinator adds the node's data to the Address Resolution Table, making it discoverable within the swarm and facilitating the exchange of swarm communication messages. At this stage, the P4 program operating at the network layer restricts communication to interactions with the Swarm Coordinator only, preventing direct communication between smart nodes. This controlled approach ensures that onboarding is secure and that only authorized nodes are integrated into the swarm. The Swarm Adaptive Coordinator component further supports the ongoing management of SmartEdge swarm members. Developed as an advanced feature within the application layer of SmartEdge smart nodes, it enables nodes to function as coordinators, overseeing the integration and coordination of swarm members and ensuring an efficient, dynamically adaptable swarm structure.

The discovery and formation mechanisms are based on the semantic descriptions of SmartEdge nodes and network information stored in the Dynamic Knowledge Graph (DKG). The coordination process begins with the Orchestrator providing a list of required skills needed to execute tasks and applications across the swarm. Using this list, the coordination component identifies and selects suitable SmartEdge nodes by referencing their semantic descriptions in the DKG. These descriptions offer a contextual understanding of each node's capabilities, network status, and operational conditions. This dynamic discovery and formation mechanism enables the swarm to align its resources efficiently, ensuring that each task is assigned to nodes with the necessary skills and capacity.

Beyond the initial setup, the Swarm Adaptive Coordination component continuously monitors the swarm's state, observing changes in the environment and the performance of individual nodes. This enables the component to anticipate or respond to disruptions and resource variations. If a node becomes unavailable or if resource demands shift, the coordinator can reconfigure the swarm in real-time, dynamically reallocating tasks and adjusting network

connections as necessary to maintain seamless operation. This proactive management capability not only improves resilience but also maximizes the efficiency and reliability of the swarm.

The Swarm Adaptive Coordination component's functionality is further enhanced by a set of specialized sub-components that address key requirements for swarm formation and management:

Semantic-based Node Discoverer: This component enables the coordinator to select nodes based on specific task requirements, utilizing SPARQL queries directed at the DKG to locate nodes that offer the required skills and capabilities. By returning the appropriate APIs or protocols for communication, it simplifies node selection and integration.

P4-based Network Information RDFizer: To facilitate robust communication links between nodes, this sub-component collects and translates P4-based network metadata into RDF format, making it accessible through the DKG. By transforming raw network data into semantic representations, it enables complex analyses and queries about network status, enhancing the swarm's understanding of its connectivity landscape.

Semantic-based ROS Communication: This sub-component provides interoperability with ROS (Robot Operating System) data, utilizing semantic technologies to describe, annotate, and process ROS topics and messages. By converting ROS messages into RDF and enabling real-time Graph Stream Processing, it allows the swarm to integrate and interpret diverse ROS data sources, making them available for collaborative processing within the SmartEdge framework.

4.2.1.1 Semantic-Based Node Discoverer

The semantic-based node discoverer is a central component of the swarm coordinator, responsible for dynamically identifying and selecting nodes in the SmartEdge swarm that possess the required skills for specific tasks. Upon receiving a list of required skills from the Orchestrator or another component, the semantic-based node discoverer constructs a SPARQL query to search for nodes that match these requirements. This SPARQL query is then registered with the Dynamic Knowledge Graph (DKG), which stores semantic descriptions of all nodes in the swarm, including their capabilities, status, and environmental context.

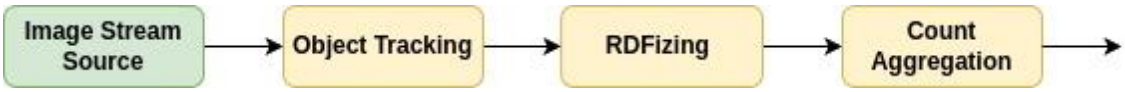


Figure 4-3. Processing Pipeline for Counting Vehicles in an Observation Zone

For example, to calculate the number of cars within a designated observation zone, as required in Use Case 2 (UC2), the system employs a processing pipeline as depicted in Figure 4-3. This pipeline starts by receiving a video stream from a camera positioned to monitor the area. The video feed is passed to an object tracking task, which typically uses a deep neural network (DNN) model, such as YOLO or SSD, to track each car as it enters or exits the observation zone. The RDFizing task follows, converting the tracking data into a structured RDF (Resource Description Framework) format. This transformation enables seamless integration, querying, and interpretation of the data across various components of the system. In the final stage, count

aggregation is performed through an RDF stream processing task that continuously computes the aggregate count of cars in real-time. Alternatively, a graph matching pattern can be applied to analyze the data, as outlined in Deliverable D5.1, Section 5.4.1. Each task in this pipeline is defined by the Orchestrator and then passed to the Coordinator, which is responsible for identifying the appropriate SmartEdge nodes capable of fulfilling these tasks.

To identify devices capable of performing a specific task, it is first to define the required skills for that task. For instance, if the task involves monitoring an option zone, a camera with the "skill" to observe that zone is needed (Figure 4-4). For object detection tasks, the device must support deep neural network (DNN) inferencing, which typically requires a GPU or specialized hardware for efficient processing. In contrast, tasks such as RDFizing (converting data into RDF format) or count aggregation depend more on standard CPU capabilities, as these tasks leverage general processing power rather than specialized AI features.

```
1  {
2    "@context": {
3      "smart-edge": "http://smart-edge.eu/onto/",
4      "geo": "http://www.w3.org/2003/01/geo/wgs84_pos#",
5      "rdfs": "http://www.w3.org/2000/01/rdf-schema#"
6    },
7    "@type": "smart-edge:ImageObserver",
8    "smart-edge:field_of_view": {
9      "@type": "smart-edge:Area",
10     "smart-edge:coordinates": {
11       "smart-edge:topLeft": { "geo:lat": "60.16500", "geo:long": "24.91230" },
12       "smart-edge:topRight": { "geo:lat": "60.16500", "geo:long": "24.91340" },
13       "smart-edge:bottomLeft": { "geo:lat": "60.16400", "geo:long": "24.91230" },
14       "smart-edge:bottomRight": { "geo:lat": "60.16400", "geo:long": "24.91340" }
15     }
16   }
17 }
```

Figure 4-4. JSON-LD of the semantic description for a required skill to observe an option zone.

As an example, Figure 4-4 illustrates a JSON-LD representation of a required skill, "Image Observer," which enables the observation of a specified option zone. This skill includes the observes_zone field, outlining the boundaries of the option zone using geo:lat and geo:long coordinates for each corner (topLeft, topRight, bottomLeft, bottomRight). By utilizing the SmartEdge data schema and RDF data format to define and standardize device capabilities, this approach enables efficient task-to-device matching based on both required skills and hardware specifications.

Figure 4-5 provides a JSON-LD snapshot of a camera located at Junction 266, detailing its location, video streaming endpoint, and field of view. The camera is modeled as a Thing using the Web of Things (WoT) ontology. Lines 7 to 16 specify the camera's metadata, including its title, unique identifier within the SmartEdge ecosystem (urn:uuid:9489991a-7622-45b6-8437-f859835d4), and geographical coordinates (geo:lat and geo:long). The video streaming mechanism is described in lines 12 through 19, where a wot:EventAffordance is used to define the streaming endpoint. The RTSP URL (RTSP://helsinki.fi/camera/266_1/) and content type (video/mp4) are specified, enabling direct access to the camera's live feed. Lines 17 to 27 describe the camera's skill, Image Observer, which is modeled as a subclass of the smart-edge:Skill ontology. This skill includes the camera's field of view, represented as a smart-

edge:Area with a unique identifier (fov_id: cam_266_1). The field of view is further defined with specific boundary coordinates (topLeft, topRight, bottomLeft, bottomRight) using geospatial properties (geo:lat and geo:long) to delineate the area monitored by the camera. This JSON-LD structure integrates the WoT and SmartEdge ontologies, providing a semantically rich and interoperable representation. It enables precise context definition, facilitating effective task coordination and enhancing functionality within the SmartEdge ecosystem.

```
{
  "@context": [{
      "smart-edge": "http://smart-edge.eu/onto#","geo": "http://www.w3.org/2003/01/geo/wgs84_pos#",
      "rdfs": "http://www.w3.org/2000/01/rdf-schema#","wot": "https://www.w3.org/2019/wot/td#",
      "dc": "http://purl.org/dc/elements/1.1/"}],

  "title": "Camera number at Junction 266",
  "id": "urn:uuid:9489991a-7622-45b6-8437-f859835d4", "@type": "wot:Thing",
  "geo:lat": "60.16453", "geo:long": "24.912846",
  "events": {
    "traffic_images": {
      "@type": "wot:EventAffordance",
      "forms": [{
        "wot:href": "RTSP://helsinki.fi/camera/266_1/",
        "wot:contentType": "video/mp4"}]}
  },
  "smart-edge:hasSkill": {
    "@type": "smart-edge:ImageObserver",
    "smart-edge:field_of_view": {
      "@type": "smart-edge:Area",
      "fov_id": "cam_266_1",
      "coordinates": { "bottomLeft": { "geo:lat": "60.16420", "geo:long": "24.91250" },
                       "bottomRight": { "geo:lat": "60.16420", "geo:long": "24.91320" },
                       "topLeft": { "geo:lat": "60.16480", "geo:long": "24.91250" },
                       "topRight": { "geo:lat": "60.16480", "geo:long": "24.91320" }}
    }
  },
  "smart-edge:ImageObservation": {
    "@type": "rdfs:Class", "rdfs:subClassOf": "smart-edge:Skill"
  }
}
```

Figure 4-5. JSON-LD Snapshot of Semantic description for a camera at Junction 270, Helsinki

To identify the appropriate camera, the device must provide images of the option zone, meaning we need a camera with a field of view that overlaps with the option zone's coordinates. This process involves querying for cameras whose field of view intersects with the specified option zone. The SPARQL query shown in Figure 4-6 demonstrates how to retrieve cameras meeting these criteria by matching their field of view with the option zone's boundaries.

```
PREFIX smart-edge: <http://smart-edge.eu/onto#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
PREFIX wot: <https://www.w3.org/2019/wot/td#>

DESCRIBE ?camera
WHERE {
    # Retrieve cameras with the ImageObserver skill and their FOV coordinates
    ?camera a wot:Thing ; smart-edge:hasSkill ?skill .
    ?skill a smart-edge:ImageObserver ; smart-edge:field_of_view ?fov .
    ?fov smart-edge:coordinates ?fovCoords .
    ?fovCoords smart-edge:topLeft ?fovTopLeft ;smart-edge:bottomRight ?fovBottomRight .
    ?fovTopLeft geo:lat ?fovTopLeftLat ; geo:long ?fovTopLeftLong .
    ?fovBottomRight geo:lat ?fovBottomRightLat ;geo:long ?fovBottomRightLong .

    # Define the option zone coordinates directly as constants
    BIND(60.16500 AS ?ozTopLeftLat)
    BIND(24.91230 AS ?ozTopLeftLong)
    BIND(60.16400 AS ?ozBottomRightLat)
    BIND(24.91340 AS ?ozBottomRightLong)

    # Compute overlapping rectangle coordinates using IF expressions
    BIND(IF(?fovTopLeftLat <= ?ozTopLeftLat, ?fovTopLeftLat, ?ozTopLeftLat) AS ?intersectTopLat)
    BIND(IF(?fovBottomRightLat >= ?ozBottomRightLat, ?fovBottomRightLat, ?ozBottomRightLat) AS ?intersectBottomLat)
    BIND(IF(?fovTopLeftLong >= ?ozTopLeftLong, ?fovTopLeftLong, ?ozTopLeftLong) AS ?intersectLeftLong)
    BIND(IF(?fovBottomRightLong <= ?ozBottomRightLong, ?fovBottomRightLong, ?ozBottomRightLong) AS ?intersectRightLong)

    # Ensure there is an actual overlap between FOV and option zone
    FILTER(?intersectTopLat > ?intersectBottomLat && ?intersectRightLong > ?intersectLeftLong)

    # Calculate areas
    BIND((?fovTopLeftLat - ?fovBottomRightLat) * (?fovBottomRightLong - ?fovTopLeftLong) AS ?fovArea)
    BIND((?ozTopLeftLat - ?ozBottomRightLat) * (?ozBottomRightLong - ?ozTopLeftLong) AS ?optionZoneArea)
    BIND((?intersectTopLat - ?intersectBottomLat) * (?intersectRightLong - ?intersectLeftLong) AS ?intersectionArea)

    # Calculate Intersection over Union (IoU)
    BIND(?intersectionArea / (?fovArea + ?optionZoneArea - ?intersectionArea) AS ?iou)

    # Filter for significant overlap (e.g., IoU > 0.5)
    FILTER(?iou > 0.5)
}
```

Figure 4-6. JSON-LD Snapshot of Semantic description for a camera at Junction 270, Helsinki

To generate the SPARQL queries, a query template is defined for each required skill. These queries are then subscribed to the Dynamic Knowledge Graph (DKG) to listen for updates. Each skill maintains its own list of suitable nodes discovered from the DKG, and each list is assigned to a corresponding skill. The lists are organized by relevant categories, ensuring that the most capable devices are prioritized.

For example, for cameras, the list is sorted by Intersection over Union (IoU), prioritizing those with the highest overlap with the option zone. For nodes that handle tasks like object detection, the list is sorted based on available CPU and GPU resources, with nodes offering higher available resources ranked at the top.

When there is a change in the swarm's state, such as a device nearing battery depletion or a camera going offline due to environmental factors like strong winds, the DKG detects these updates and sends the new information back to the Coordinator. The Coordinator then updates the lists, accordingly, identifying devices at high risk of becoming unavailable by observing those that fall toward the bottom of the list. If a device's risk level exceeds a certain threshold, the Coordinator alerts the Orchestrator, allowing it to adjust the task assignments or reconfigure the swarm as needed to maintain optimal performance and stability.

This artifact is built upon the Recipe model described in Work Package 3 (WP3). Moving forward, we maintain alignment with the template specifications and track updates to the Recipe model in WP3 to ensure compatibility and improvements in functionality. Additionally, we will explore criteria for detecting node failure, allowing for more proactive management within the swarm. This includes identifying specific signals that indicate a node's reduced functionality or risk of failure, such as low battery levels, connectivity issues, or hardware malfunctions.

4.2.1.1.1 P4-based network information RDFizer

To help the Coordinator understand the network context, this sub-component collects and translates P4-based network metadata into RDF format, making it accessible via the Dynamic Knowledge Graph (DKG). By converting raw network data into semantic representations, it enables advanced analyses and complex queries about network status, improving the swarm's awareness of its connectivity landscape. For instance, which connections can provide bandwidth up to 10 MB per second.

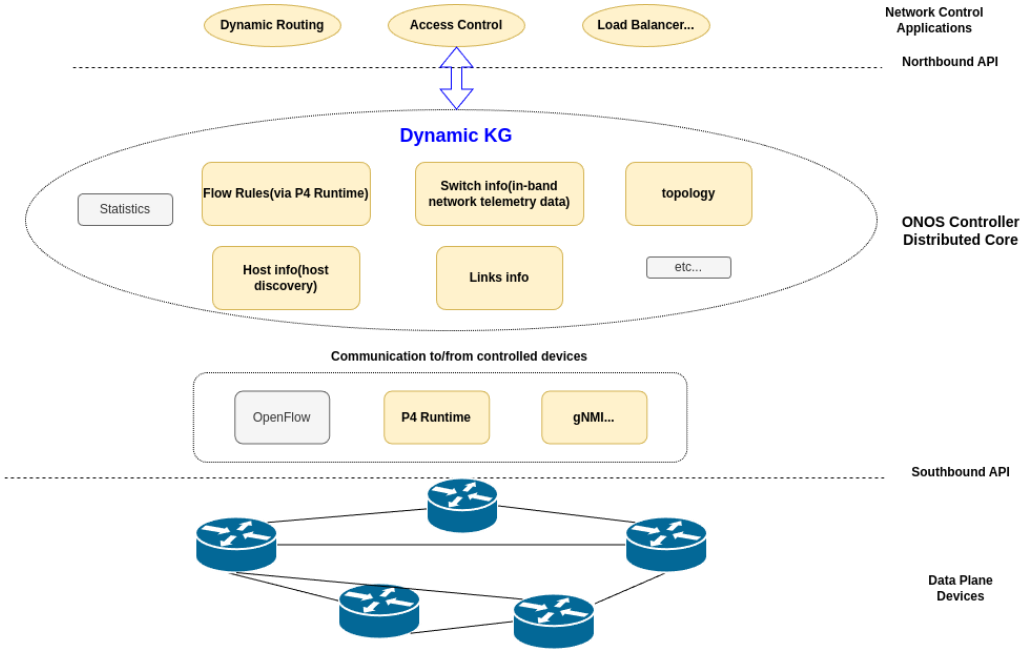


Figure 4-7. Overview of the integration DKG with ONOS control plan

We adopt the Software-Defined Networking (SDN) approach, with ONOS (Open Network Operating System) serving as the control plane implementation. The data plane can be implemented using P4, ensuring compatibility with P4 devices in the SmartEdge project. Figure 4-7 illustrates how the Distributed Knowledge Graph (DKG) mechanism integrates into a traditional ONOS-based SDN controller:

- *Southbound API:* Enables communication between the ONOS controller and data plane devices using protocols like P4 Runtime.
- *ONOS Distributed Core:* The dynamic knowledge graph integrates with the distributed core, facilitating the storage and management of semantic annotations for network state information.
- *Northbound API:* Provides interfaces for network-control applications to interact with the controller, leveraging DKG insights for tasks like dynamic routing, access control, and

load balancing. Applications can query and update states using the knowledge base, enabling adaptive and informed network management.

The workflow of integrating ONOS with components such as the ONOS controller and P4-enabled devices in the data plane is illustrated in Figure 4-8. Key components include the knowledge generator (referred to as the RDFizer), the dynamic knowledge graph (DKG), and its SPARQL engine. The ONOS controller interacts with real-time data sources in the data plane, including packets and P4Runtime control entities. This data is abstracted into Plain Old Java Objects (POJOs) by the controller. The RDFizer then transforms these POJOs into semantically uniform Resource Description Framework (RDF) structures, ensuring consistent and standardized data representation for reasoning and decision-making processes. This RDF-based approach introduces an innovative method for managing network data and leveraging it for higher-level applications.

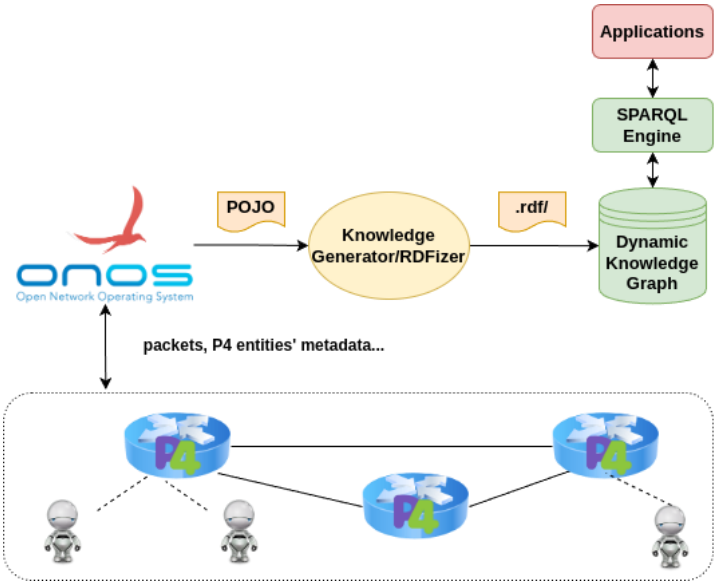


Figure 4-8. Overview of the workflow of P4-based RDFizer

This module is planned for the second release. At the current stage, we have successfully extracted P4-based metadata from packets traversing the network using P4. Figure 4-9 showcases the RDF-annotated P4-based metadata within the ONOS environment. The testing was conducted on a simulated network using Mininet, and the codebase is accessible in the GitLab repository of the SmartEdge project. The next step involves integrating these components into the SmartEdge runtime instance, establishing communication with the DKG via the DDS protocol.

<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://eu.smartedge/PortStatistics> .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/serialNumber> "unknown" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/swVersion> "Stratum" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/packetsRxDropped> "0" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/packetsRxErrors> "0" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/packetsTxDropped> "0" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/packetsTxErrors> "0" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/durationNano> "1900521" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/bytesSent> "11562" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/type> "COPPER" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/deviceId> "device:leaf2" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/portNumber> "[leaf2-eth2](2)" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/hwVersion> "BMv2 simple_switch" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/manufacturer> "Open Networking Foundation" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/chassisId> "0" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/bytesReceived> "13258" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/packetsReceived> "115" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/portSpeed> "10000" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/packetsSent> "94" .
<http://smartedge.eu839b7e0a-8e2d-49b6-b4ca-12cfbf62c069> <http://smartedge.eu/durationSec> "289" .

Figure 4-9. Screen shot of RDF annotation of P4-based metadata on MiniNet.

4.2.2 Swarm Dynamic Orchestrator

The Swarm Dynamic Orchestrator manages the collaborative actions of SmartEdge nodes within a SmartEdge swarm, orchestrating tasks and linking nodes to create a processing pipeline. In the SmartEdge toolchains, a declarative programming approach is used, where each application is represented as a Semantic Program (detailed in Section 5.4 of Deliverable 5.1). Upon receiving a Semantic Program, the Orchestrator decomposes it into sub-tasks to construct a federated execution plan. This plan links specific data sources to processing operators and defines where processed data should be routed. Additionally, the Orchestrator adapts dynamically to changes in the swarm environment. If a single node lacks sufficient computational resources to handle a complete task, the Orchestrator partitions the task into smaller segments (when feasible) and distributes them across other capable nodes within the swarm. This adaptive strategy ensures efficient utilization of resources and robust task execution across the SmartEdge network.

4.2.2.1 Semantic Program Parser

The execution of a semantic program begins by parsing it into a structured data representation that encapsulates all the operators, and their relationships defined in the program. In the initial implementation, the Orchestrator is designed to parse and process key elements of a semantic program, including data input sources, data outputs, and essential operations for processing semantic data (RDF), such as sub graph matching, aggregation, and filtering.

The parser is developed by extending the SPARQL parser from the RDF4J library. Since CQELS extends the SPARQL syntax by introducing keywords like STREAM and WINDOWS for processing streaming data, these additions enable the parser to handle window operations seamlessly. Existing implementations for parsing RDF nodes, aggregation functions, and other SPARQL elements are reused to maintain efficiency and consistency. To support the extended syntax, the underlying JavaCC grammar files from RDF4J are extended to include the CQELS-specific keywords. These extended grammar files are processed through the same compilation pipeline to generate essential components for the parser, such as a Lexer and token recognizer.

In its current version, the Orchestrator can process semantic programs for specific use cases, such as counting all vehicles at a junction. This capability demonstrates the initial functionality

of the system, laying the foundation for more complex stream processing operations in future iterations.

4.2.2.2 Task Skill Mapper

Before constructing an execution plan for a semantic program, the system must first identify the specific operations required to execute the program. This process is conceptually similar to how a database management system processes a query. In such systems, the query is parsed into a structured format, and then a logical query plan is generated. Similarly, in the SmartEdge runtime, the parsed semantic program serves as the foundation for building a logical query plan.

This component is integral to the workflow, as it parses the semantic program and generates a logical query plan that organizes the required operations into a sequence of interconnected operators. The logical query plan acts as a blueprint, detailing how data flows between operators and how tasks are to be executed. Each operator in the plan represents a discrete computational task, such as filtering, aggregation, or stream processing.

For each operator in the logical query plan, the system generates a detailed specification of the computational requirements needed to execute it. This specification includes the skills required for the operator, such as the ability to perform deep learning inference, graph processing, or basic data transformations. Additionally, the system identifies the hardware resources needed, such as CPU, GPU, memory, or specific software libraries, to ensure optimal execution.

Once the skill requirements for the operators are identified, the Task Skill Mapper sends the compiled list of required skills to the Coordinator. The Coordinator queries the available SmartEdge nodes to find devices capable of fulfilling the specified skill requirements. This interaction involves using the Dynamic Knowledge Graph (DKG) to retrieve information about each node's current state, capabilities, and resources.

The Coordinator returns a list of devices that match the required skills and are available for task execution. The Execution Plan Builder then use this information to map each operator in the logical query plan to specific devices within the swarm. By ensuring that tasks are assigned to nodes capable of executing them efficiently, this process enables distributed, optimized execution of the semantic program across the SmartEdge ecosystem.

4.2.2.3 Execution Plan Builder

The Execution Plan Builder is the final stage in the task orchestration process, where the system translates the logical query plan and device capabilities into a concrete execution plan. This component assigns tasks to specific devices within the swarm and establishes the necessary links between devices to create a complete processing pipeline. The resulting execution plan ensures that tasks are distributed efficiently and data flows seamlessly across the SmartEdge swarm.

The Execution Plan Builder starts by receiving the logical query plan, which contains a detailed representation of the operators and their interconnections. For each operator in the plan, the required skills and resources (e.g., computational power, memory, GPU availability) are matched against the list of available devices provided by the Coordinator.

Using the skill-to-device mapping generated by the Task Skill Mapper, the Execution Plan Builder assigns each operator to a specific device. For example, a deep learning inference task may be assigned to a device with a GPU and DNN capabilities, while a simpler data filtering task may be allocated to a device with basic CPU resources.

The Execution Plan Builder accounts for real-time changes in the swarm environment, such as devices becoming unavailable or experiencing resource constraints. If a device is no longer suitable for a task, the plan is dynamically adjusted by reassigning the task to an alternative device. This adaptability ensures resilience and continuity in task execution.

```
{
  "tasks": [
    { "device_id": "jetson_01",
      "assigned_operations": [{
          "operation": {"type": "ObjectTracking"...},
          "inputs": [{
              "protocol": "RTSP",
              "endpoint": "RTSP://helsinki.fi/camera/266_1/"
            }],
          "output": {
            "protocol": "DDS",
            "endpoint": "/jetson_01/object_tracking_01"
          }
        },{
          "operation": {"type": "RDFizing"...},
          "inputs": [
            {
              "protocol": "DDS",
              "endpoint": "/jetson_01/object_tracking_01"
            }
          ],
          "output": {
            "protocol": "DDS",
            "endpoint": "/jetson_01/rdf_object_tracking_01"
          }
        }]},
    { "device_id": "raspberry_pi4_01",
      "assigned_operations": [{
          "operation": {"type": "RSP"...},
          "inputs": [{
              "protocol": "DDS",
              "endpoint": "/device_01/rdf_object_tracking_01"
            }
          ],
          "output": {
            "protocol": "DDS",
            "endpoint": "/raspberry_pi4_01/pipeline_01/result"
          }}]
    }]
}
```

Figure 4-10. Example of an execution plan generated by the Orchestrator in JSON format.

For example, consider a processing pipeline involving object tracking, RDFizing, and RDF Stream Processing (RSP) (Figure 4-10). The pipeline begins with a device, such as jetson_01, performing object tracking using the Yolov8 model on a live video feed from a camera endpoint (RTSP://helsinki.fi/camera/266_1/). The tracking results are then converted into RDF format (RDFizing) on the same device. These RDFized results are passed to another device, such as raspberry_pi4_01, which processes the data using RDF Stream Processing to execute a query and aggregates the results. The Execution Plan Builder maps each operation to the appropriate device by evaluating the computational requirements of tasks (e.g., GPU-intensive object tracking vs. lightweight RSP queries) and the devices' available capabilities. It also defines the

communication protocols and endpoints, such as RTSP for video feeds and DDS for inter-device communication, ensuring seamless data flow across the pipeline. By dynamically adjusting to the swarm's state and monitoring device performance, the Execution Plan Builder maintains optimal resource utilization and robust task execution, even in dynamic environments. This modular approach allows for scalable, efficient, and resilient distributed task management within the SmartEdge system.

4.2.3 Swarm Optimizer

This component empowers SmartEdge smart nodes to enhance their performance by dynamically adjusting runtime parameters within the SmartEdge Runtime hosted on each node. The Optimizer evaluates resource utilization and workload efficiency, providing recommendations to Coordinators about task distribution. For example, it can determine whether a task currently running on a node should be offloaded to a more capable node, such as one with higher computational power or access to cloud infrastructure, ensuring optimal resource usage across the swarm.

The Optimizer plays a pivotal role in enabling intelligent decision-making within the SmartEdge ecosystem. By analyzing the computational demands of tasks and the resource constraints of nodes, it helps maintain balance and responsiveness in the swarm. This capability is particularly critical for scenarios involving resource-intensive tasks like deep learning inference, where efficient allocation can significantly impact performance and energy consumption.

The development of this component is planned as part of the SmartEdge Solution 2 release, with the current implementation in its initial stages. At this stage, the foundational mechanisms for runtime parameter tuning and basic task distribution are being established. The corresponding use case for testing and validating the Optimizer is under development as part of Task T5.3. Future iterations aim to fully integrate the Optimizer with the SmartEdge Coordinator and Dynamic Knowledge Graph (DKG) for real-time monitoring and adaptive task allocation within the swarm.

4.3 Empirical Results and Demonstrations

This section provides a detailed overview of the demonstration of the Swarm Orchestrator and Coordinator in two use cases: UC2 - counting vehicles at junction 266 and UC3 - collaborative observation in a smart factory environment. These demonstrations emphasize the system's capability to handle node discovery and task distribution across the SmartEdge swarm.

For UC2, the demonstration involves deploying a processing pipeline to count vehicles passing through a designated observation zone. The pipeline comprises tasks such as object tracking, RDFizing, and count aggregation, which are distributed across multiple SmartEdge devices. The Swarm Orchestrator processes a semantic program to generate an execution plan, while the Coordinator ensures that tasks are assigned to suitable devices based on their capabilities and resource availability.

For UC3, the demonstration features a ground vehicle robot equipped with an Intel RealSense D435i camera. As the robot navigates through the operational area, it builds a 2D occupancy map using RTAB-Map (Real-Time Appearance-Based Mapping) and simultaneously performs real-time object detection and segmentation with a YOLO model. All components, including the map-building and object detection algorithms, are implemented as ROS2 (Robot Operating

System 2) nodes. This pipeline represents a "Primitive Runtime" within the SmartEdge Runtime system, with the robot acting as a smart node in the SmartEdge swarm. The demonstration showcases how mobile robots can achieve real-time mapping and scene understanding, enhancing navigation and operational efficiency in a smart factory environment.

These use cases demonstrate the collaborative and adaptive capabilities of the Swarm Orchestrator and Coordinator in managing distributed tasks across a heterogeneous network of SmartEdge devices. The demonstrations underline the scalability, robustness, and versatility of the system in handling complex workflows in dynamic edge environments.

4.3.1 Object tracking and counting demo of UC2

As a part of implementation of Use Case 2, this demo demonstrates the SmartEdge system's ability to process real-time video streams from a camera monitoring a junction and count the number of vehicles passing through (Figure 4-11). This use case utilizes specific components of the SmartEdge toolchain:

- Semantic Programs: Define the workflow for vehicle counting, detailing the tasks and data flows required for execution.
- Swarm Coordinator: Identifies and manages SmartEdge nodes, ensuring that resources are allocated based on task requirements.
- Swarm Orchestrator: Distributes and executes tasks across the swarm nodes, managing their execution order and resource utilization.
- Dynamic Knowledge Graph (DKG): Tracks the real-time state of nodes and tasks, enabling efficient coordination and task updates.

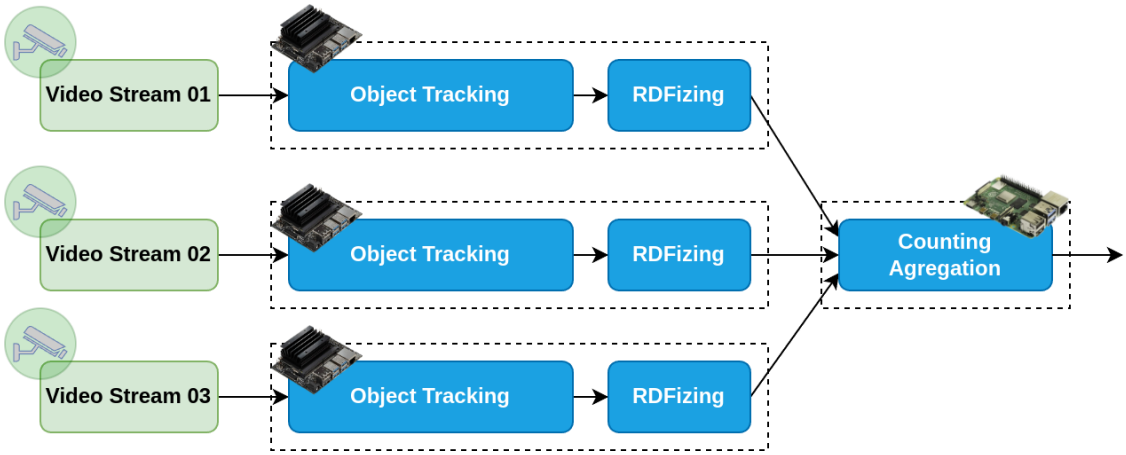


Figure 4-11. Working Pipeline

In this demonstration, SmartEdge components will be deployed on a Jetson Nano, while another PC will be used to send task requests. Both devices are connected to the same local network. To launch a SmartEdge swarm node, a single command is required: ***`./smartedge_launch.sh 1 nano true true`.*** In this command, 1 specifies the swarm node ID, nano indicates deployment on a Jetson Nano device, the first true denotes that this is a swarm node with both the coordinator and orchestrator, and the second true enables debug mode. Once the command is executed, two Docker containers will be built, and the ROS2 nodes will start running (as shown in Figure 4-12).

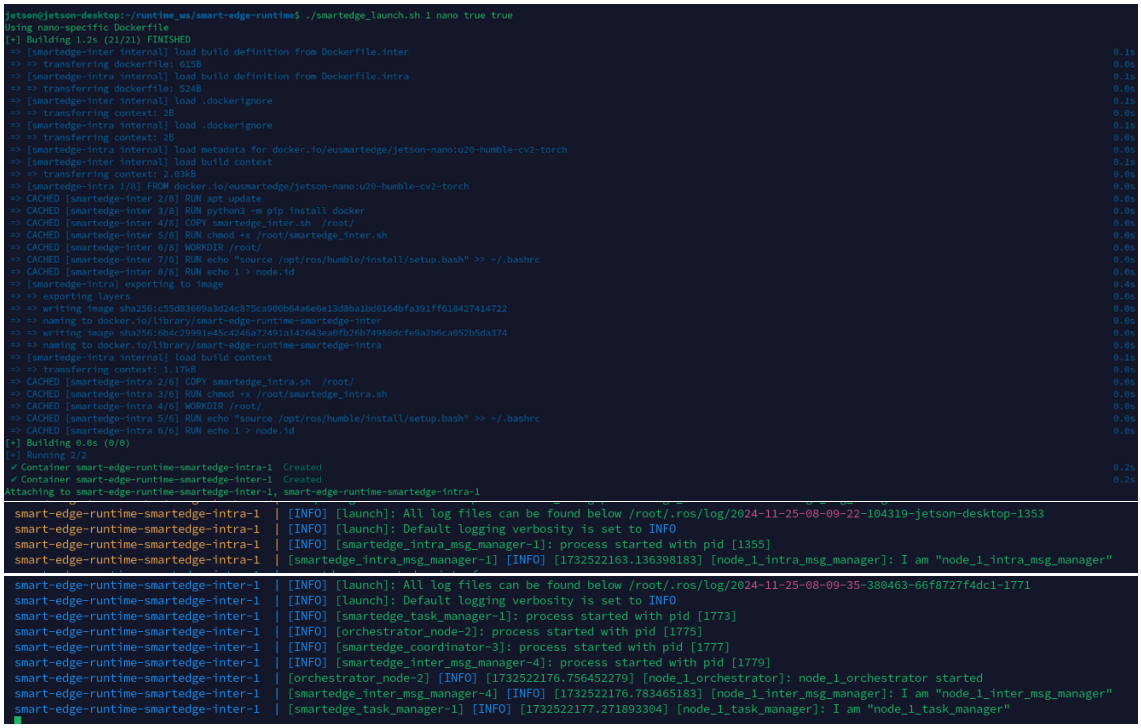


Figure 4-12. Building process of the swarm node

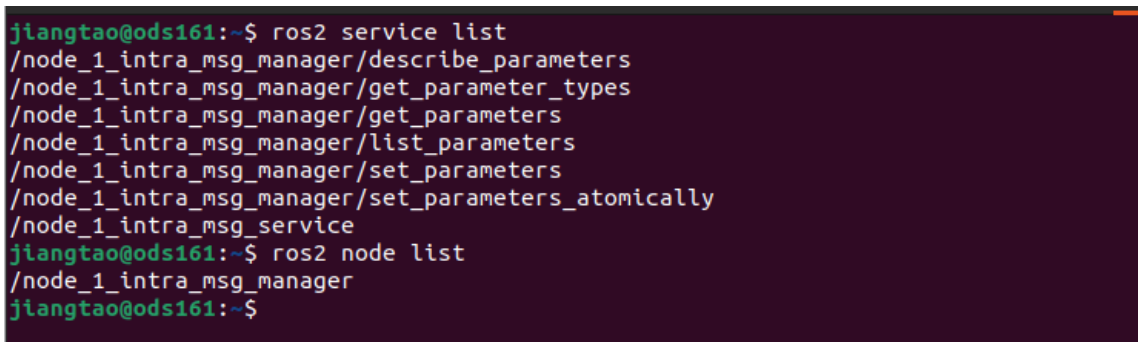


Figure 4-13. The PC can only discover the intra-message manager ROS2 node

On the PC, only the intra-message manager ROS2 node, located in the intra-communication Docker container on the Jetson Nano, is visible. Conversely, on swarm node 1, all ROS2 nodes and services are discoverable (as shown in Figure 4-13 and Figure 4-14).

```
root@jetson-desktop:~# ros2 service list
/node_1_coordinator/describe_parameters
/node_1_coordinator/get_parameter_types
/node_1_coordinator/get_parameters
/node_1_coordinator/list_parameters
/node_1_coordinator/set_parameters
/node_1_coordinator/set_parameters_atomically
/node_1_coordinator_service
/node_1_inter_msg_manager/describe_parameters
/node_1_inter_msg_manager/get_parameter_types
/node_1_inter_msg_manager/get_parameters
/node_1_inter_msg_manager/list_parameters
/node_1_inter_msg_manager/set_parameters
/node_1_inter_msg_manager/set_parameters_atomically
/node_1_inter_msg_service
/node_1_intra_msg_manager/describe_parameters
/node_1_intra_msg_manager/get_parameter_types
/node_1_intra_msg_manager/get_parameters
/node_1_intra_msg_manager/list_parameters
/node_1_intra_msg_manager/set_parameters
/node_1_intra_msg_manager/set_parameters_atomically
/node_1_intra_msg_service
/node_1_orchestrator/describe_parameters
/node_1_orchestrator/get_parameter_types
/node_1_orchestrator/get_parameters
/node_1_orchestrator/list_parameters
/node_1_orchestrator/set_parameters
/node_1_orchestrator/set_parameters_atomically
/node_1_orchestrator_service
/node_1_task_manager/describe_parameters
/node_1_task_manager/get_parameter_types
/node_1_task_manager/get_parameters
/node_1_task_manager/list_parameters
/node_1_task_manager/set_parameters
/node_1_task_manager/set_parameters_atomically
/node_1_task_manager_service
root@jetson-desktop:~# ros2 node list
/node_1_coordinator
/node_1_inter_msg_manager
/node_1_intra_msg_manager
/node_1_orchestrator
/node_1_task_manager
```

Figure 4-14. On the Jetson Nano, all the ROS2 nodes are discoverable

Now, let's call the /node_1_intra_msg_service to initiate the task using the following command in Figure 4-15:

```
ros2 service call /node_1_intra_msg_service smartedge_interfaces/srv/RequestMessageService \
  "{ requester :  node_0,
     request_type : REQ_INIT_01,
     request_description: counting car quantity on junction 266
  }"
```

Figure 4-15. Sample service request of initializing a task

Upon execution, the response will be received as follows (Figure 4-16):

```
response:
smartedge_interfaces.srv.RequestMessageService_Response(response='primitives are running')
```

Figure 4-16. Example of a service response after initializing a task

We will observe a new Docker container: " primitive_task_1", being built and running (as shown in Figure 4-17). This primitive container publishes object detection data to the topic "/node01", making it accessible externally. For example, the PC can discover this topic and subscribe to it to receive the resulting bounding box data (as shown in Figure 4-18).

```
jetson@jetson-desktop:~$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED              STATUS              PORTS     NAMES
d2cae831a0e2   4916f5480c08                         "/bin/bash -c 'sourc…"   About a minute ago   Up About a minute             primitive_task_1
66f8727f4dc1   smart-edge-runtime-smartedge-inter   "bash smartedge_inte…"   51 minutes ago       Up 51 minutes                 smart-edge-runtime-smartedge-inter-1
ed003bc26610   smart-edge-runtime-smartedge-intra   "bash smartedge_intr…"   51 minutes ago       Up 51 minutes                 smart-edge-runtime-smartedge-intra-1
```

Figure 4-17. Docker container list after distributing the task

```
jiangtao@ods161:~$ ros2 topic list
/node01
/parameter_events
/rosout
jiangtao@ods161:~$ ros2 topic echo /node01
data: '[{"label": "train", "bbox": [365, 387, 842, 872]}, {"label": "person", "bbox": [1760, 896, 1816, 1056]}, {"label": "car", "bbox"...'
---
data: '[{"label": "train", "bbox": [366, 387, 842, 872]}, {"label": "person", "bbox": [1760, 896, 1816, 1056]}, {"label": "car", "bbox"...'
---
data: '[{"label": "train", "bbox": [353, 387, 843, 872]}, {"label": "person", "bbox": [1760, 895, 1816, 1056]}, {"label": "car", "bbox"...'
---
data: '[{"label": "train", "bbox": [295, 389, 843, 872]}, {"label": "person", "bbox": [1760, 896, 1816, 1056]}, {"label": "car", "bbox"...'
---
data: '[{"label": "train", "bbox": [363, 384, 842, 871]}, {"label": "person", "bbox": [1760, 896, 1816, 1056]}, {"label": "car", "bbox"...'
---
data: '[{"label": "train", "bbox": [365, 388, 842, 872]}, {"label": "person", "bbox": [1760, 896, 1816, 1056]}, {"label": "car", "bbox"...'
---
data: '[{"label": "train", "bbox": [365, 383, 842, 871]}, {"label": "person", "bbox": [1760, 895, 1816, 1056]}, {"label": "car", "bbox"...'
---
data: '[{"label": "train", "bbox": [365, 383, 841, 871]}, {"label": "person", "bbox": [1760, 895, 1816, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [365, 383, 842, 872]}, {"label": "person", "bbox": [1760, 895, 1816, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [365, 385, 842, 872]}, {"label": "person", "bbox": [1760, 896, 1816, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [365, 384, 842, 872]}, {"label": "person", "bbox": [1760, 895, 1816, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [365, 386, 842, 872]}, {"label": "person", "bbox": [1760, 895, 1816, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [363, 388, 841, 873]}, {"label": "person", "bbox": [1760, 895, 1816, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [365, 386, 842, 872]}, {"label": "person", "bbox": [1759, 895, 1815, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [364, 388, 842, 872]}, {"label": "person", "bbox": [1760, 895, 1815, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [363, 389, 841, 872]}, {"label": "person", "bbox": [1759, 895, 1815, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [365, 389, 841, 872]}, {"label": "person", "bbox": [1760, 895, 1815, 1056]}, {"label": "person", "bb..."
---
data: '[{"label": "train", "bbox": [364, 393, 845, 874]}, {"label": "person", "bbox": [1760, 896, 1815, 1056]}, {"label": "person", "bb..."
---
```

Figure 4-18. Object detection results subscribed from the ROS2 topic '/node01'

To terminate the task, simply call the "/node_1_intra_msg_service" again using the following command (as shown in Figure 4-19):

```
ros2 service call /node_1_intra_msg_service smartedge_interfaces/srv/RequestMessageService \
  "{ requester :  node_0,
     request_type : REQ_STOP_01,
     request_description: '{ \"task_id\": [1] }'
  }"
```

Figure 4-19. Example of a service request to stop a task

This will return the response: "primitives are stopped." In the running log of swarm node 1, the task termination procedure will be visible, showing that the primitive container is deleted at the end of the process (as shown in Figure 4-20). Verifying with "docker ps" will confirm that the "primitive_task_1" container has indeed been removed (as shown in Figure 4-21).

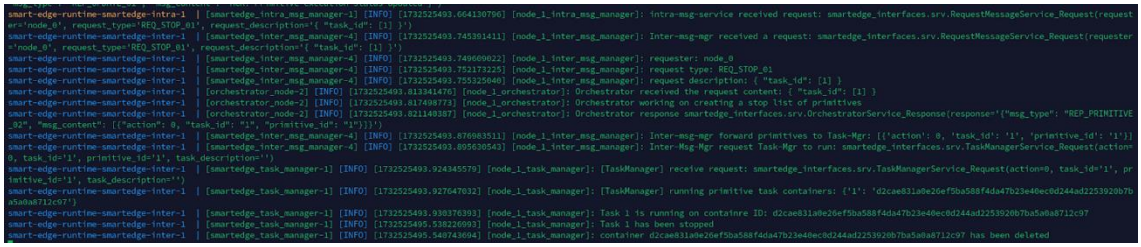


Figure 4-20. Logs for terminating a task

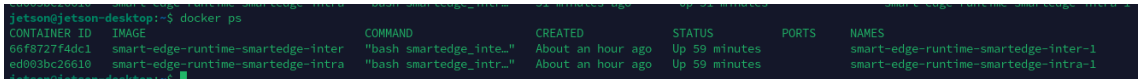


Figure 4-21. Docker container list after terminating the task

Figure 4-22 shows the detection and counting of vehicle results being visualized.

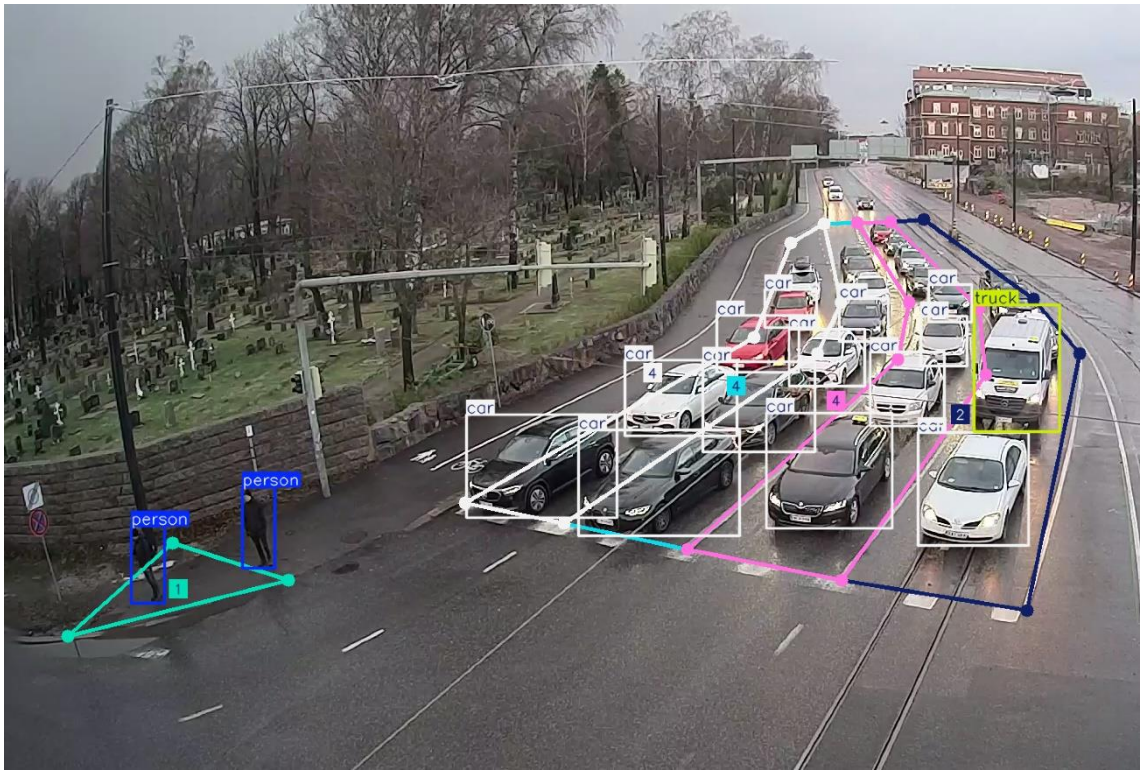


Figure 4-22. Demonstration of Detection and Counting Visualized at Conveqs Junction 266, Helsinki, Finland

4.3.2 Demonstration Semantic SLAM map builder of UC3

In this demo, we have implemented key components of Smart Factory Use Case 3 by deploying a ground vehicle robot (as shown in Figure 4-23) equipped with an Intel RealSense D435i camera. The robot navigates through the operational area while simultaneously building a 2D occupancy map using RTAB-Map (Real-Time Appearance-Based Mapping) and performing real-time object detection and segmentation using YOLO model. All components, including the map builder and object detection algorithms, are implemented as ROS2 (Robot Operating System 2) nodes. This setup demonstrates how mobile robots can achieve real-time mapping and scene understanding, enhancing navigation and operational efficiency in a smart factory environment.

This pipeline is a "Primitive Runtime" within the SmartEdge Runtime pipeline. The ground vehicle robot is a smart-node in the SmartEdge swarm, executing the tasks above.

Figure 4-23. Ground vehicle robots equip with sensors.

4.3.2.1 Working Pipeline

- **Application Creation:** The application logic for mapping and object detection is defined and converted into a semantic program (processing pipeline).
- **Program Registration and Compilation:**
  - The semantic program is registered with the SmartEdge Coordinator.
  - The Orchestrator compiles the program into an executable plan, identifying required tasks and associated skills.
- **Node Discovery and Task Assignment:**
  - The Coordinator queries the DKG to find nodes with the necessary skills (e.g., SLAM, object detection).
  - The robot is identified as a capable node, and tasks are assigned accordingly.
- **Primitive Runtime Execution:**
  The robot's Primitive Runtime components execute the assigned tasks:
  - **Subscriber:**

- The robot's sensors, including the Intel RealSense D435i camera, act as data generator by capturing RGB images, depth data and IMU from the environment.
- These data streams subscribed by this Subscriber then serve as the input for processing tasks within the robot's Primitive Runtime.

- **Primitive Executor:**
  - The SLAM map builder using RTAB-Map and the object detection and segmentation using YOLO are implemented as Primitive Executors.
  - These executors process incoming data to generate a 2D occupancy map and identify objects with bounding boxes and segmentation masks.
- **Publisher:**
  - Processed outputs, such as the occupancy map and object detection results, are published to specific topics within the swarm.
  - Other smart-nodes can subscribe to these topics, enabling data sharing and collaborative processing across the swarm.

4.3.2.2 Experiment and Demonstration

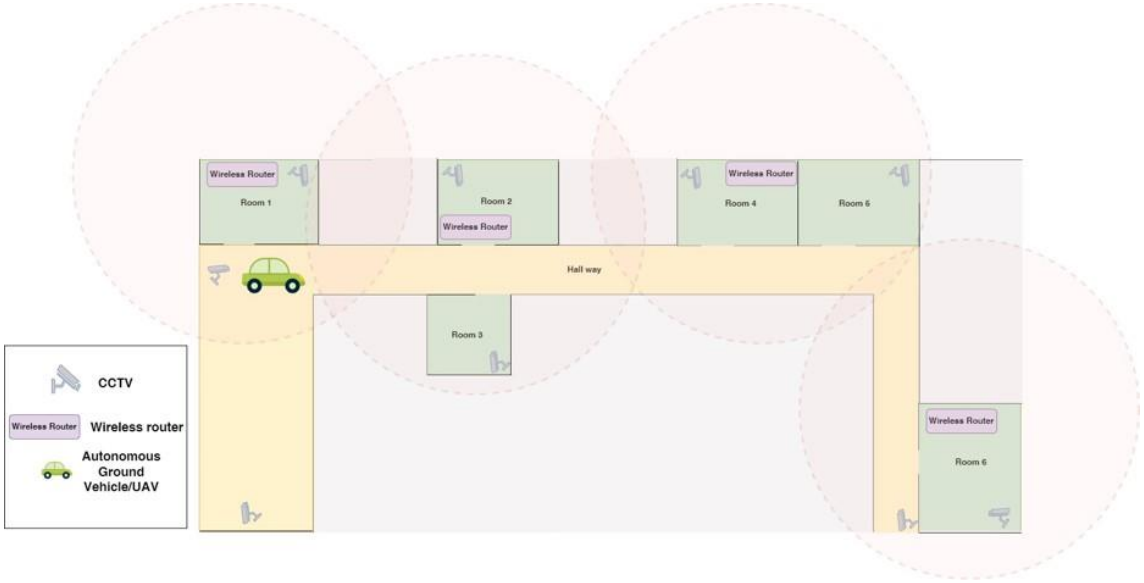


Figure 4-24. The Actual Floor Plan

This demonstration showcases the advanced capabilities of a ground vehicle robot equipped with the Intel RealSense D435i camera, running SmartEdge Runtime to generate a SLAM map and achieve environmental understanding. Starting in Room 4 (as the actual floor plan shown in Figure 4-24), the robot navigates outward into the hallway, then progresses through a long hallway, and dynamically generates a detailed 2D occupancy map of its surroundings (as shown

in Figure 4-26). The system concurrently performs semantic segmentation (as shown in Figure 4-25), providing rich contextual labels for each detected area.

Figure 4-25. Semantic Segmentation

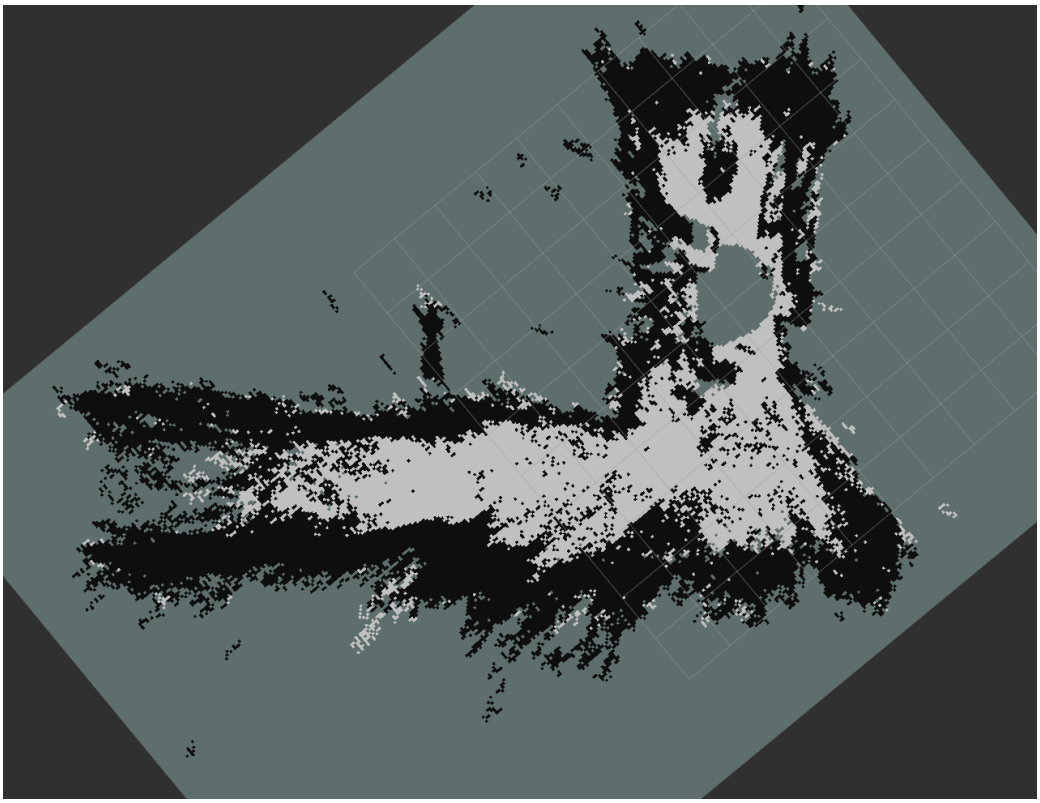


Figure 4-26. Part of a 2D occupancy map generated by a robot moving from Room 4 toward the long hallway.

5 CROSS-LAYER TOOL CHAIN FOR DEVICE-EDGE-CLOUD CONTINUUM

5.1 MAIN COMPONENTS AND FUNCTIONALITIES

Figure 5-1 illustrates the architecture of the SmartEdge toolchain, a cross-layer framework designed to support the development, deployment, and execution of distributed applications across the Device-Edge-Cloud Continuum. The toolchain consists of interconnected modules that enable low-code application creation, dynamic task allocation, and optimized runtime execution across heterogeneous devices. It bridges the gap between edge nodes and cloud resources, ensuring seamless operation and scalability within a distributed environment.

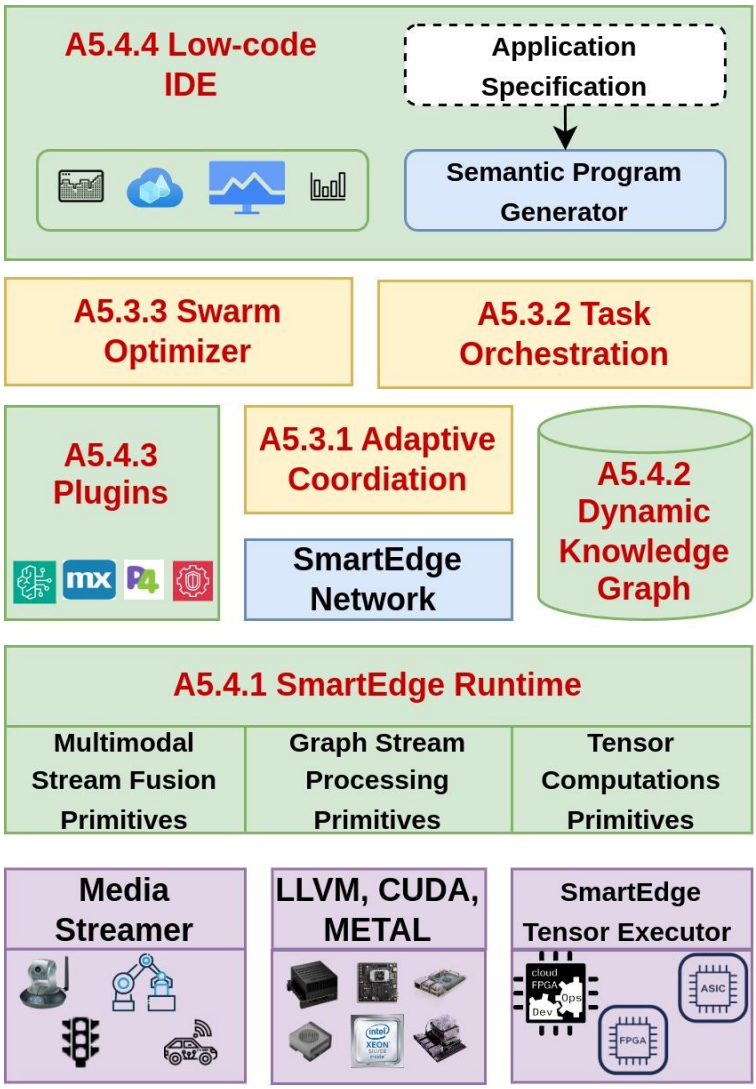


Figure 5-1. Overview of the design and initial implementation of the SmartEdge toolchain.

At the top layer, the Low-Code IDE (A5.4.4) handles the creation and specification of SmartEdge applications, offering a user-friendly interface that allows developers to design applications using low-code development techniques. By abstracting underlying complexities, the IDE enables users to visually create applications without extensive coding. Within this interface, developers specify application requirements and logic, which are then processed by the

semantic program generator. This generator translates the application logic into a structured semantic program, creating a processing pipeline of interdependent tasks that is ready for execution within the SmartEdge system.

The Swarm Coordination (A5.3.1) and Task Orchestration (A5.3.2) layer includes modules for task distribution, coordination, and optimization across the swarm of devices. The Swarm Optimizer (A5.3.3) continuously monitors resource availability and workload distribution, optimizing task allocation to ensure efficient resource utilization. By balancing workloads based on each node's current capacity, the optimizer enhances performance and responsiveness. Task Orchestration manages the execution sequence of tasks, ensuring that tasks are distributed and processed according to the semantic program's specified logic. Adaptive Coordination dynamically forms and maintains the swarm, monitoring available nodes, assessing their status, and ensuring that only suitable nodes are used for task execution. This module adjusts the swarm composition in response to real-time conditions, allowing for continuous adaptation.

The Dynamic Knowledge Graph (DKG) (A5.4.2) functions as a contextual knowledge base, storing real-time information on node availability, resources, and task requirements. Serving as a shared repository, it enables the Optimizer, Coordinator, and Orchestrator to make informed, data-driven adjustments based on the current state of the swarm.

Additionally, various plugins are being implemented to extend SmartEdge's functionalities, supporting integration with external systems and frameworks such as machine learning libraries and network management tools. These plugins enhance the toolchain's adaptability, allowing it to meet specific application requirements more effectively.

The SmartEdge Network layer (developed in WP4) underpins communication and data exchange among swarm nodes, facilitating reliable connectivity for task coordination, data sharing, and control messaging.

The SmartEdge Runtime and Hardware Abstraction layer focuses on providing an efficient runtime environment for diverse devices. The SmartEdge Runtime executes tasks across swarm nodes, leveraging processing primitives designed for specific computation types:

- Stream Fusion Primitives: Combine data from multiple sources, such as video, audio, or sensor feeds.
- Graph Stream Processing Primitives: Handle graph-based data streams, supporting the processing of complex data structures.

This layer also includes components for hardware abstraction, enabling tasks to run on various devices within the swarm. Support for heterogeneous hardware (e.g., LLVM, CUDA, METAL) allows SmartEdge to execute tasks on GPUs and specialized processors across platforms, including NVIDIA and Apple devices.

Overall, the SmartEdge toolchain integrates a low-code development environment, adaptive task management, and optimized runtime execution on diverse edge devices. This architecture supports flexible, scalable, and resilient application deployment, making it ideal for real-time, data-intensive applications in dynamic environments where resources and device capabilities vary.

5.2 COMPONENTS IMPLEMENTATIONS

5.2.1 SmartEdge Runtime

The SmartEdge Runtime enables the execution of semantic programs by managing data inputs, outputs, and the runtime processing of SmartEdge primitives. These primitives include core operations such as data fusion, stream queries, and tensor computations, as outlined in Section 5.4.3 of Deliverable D5.1. To enhance functionality and optimize performance, the runtime supports plugin integration, such as the Runtime (A5.4.1), which augments network capabilities for efficient task execution.

As depicted in Figure 4-1 (Section 4.1), the SmartEdge Runtime is composed of four primary components:

- Message Manager: Handles control communication between SmartEdge nodes, ensuring seamless coordination.
- Dynamic Knowledge Graph (DKG): Maintains metadata and contextual information about nodes and the swarm for dynamic decision-making.
- Task Manager: Oversees resource allocation, manages task distribution, and executes operations through the Primitive Runtime.
- Primitive Runtime: Executes the specific primitives defined in semantic programs, enabling distributed processing.

These components collectively ensure efficient task execution and robust coordination across the SmartEdge system.

5.2.1.1 Message Manager

To implement the Message Manager, we utilize ROS2, a middleware framework based on the Data Distribution Service (DDS) standard. The design rationale for choosing ROS2 is its ability to enable each component of the system to operate as an independent ROS node. These nodes can run on the same device or across multiple devices, communicating seamlessly via ROS2's DDS-based communication mechanism. This approach ensures modularity and flexibility, as each component can function independently while remaining part of a larger interconnected system.

One of the key advantages of using ROS2 is its Intra-process Communication (IPC) mechanism, which allows ROS nodes running on the same device to pass messages as efficiently as reading from a shared memory buffer. This could avoid unnecessary latency for on-device communication.

Another benefit of ROS2 is its wide support across SmartEdge UC devices, making it a natural choice for implementation. Moreover, to accommodate integration with other communication protocols, we can incorporate Zenoh, a DDS-based middleware that serves as a bridge between diverse communication protocols. Zenoh enables the system to interoperate with non-DDS-based components, further extending the flexibility and scalability of the SmartEdge system.

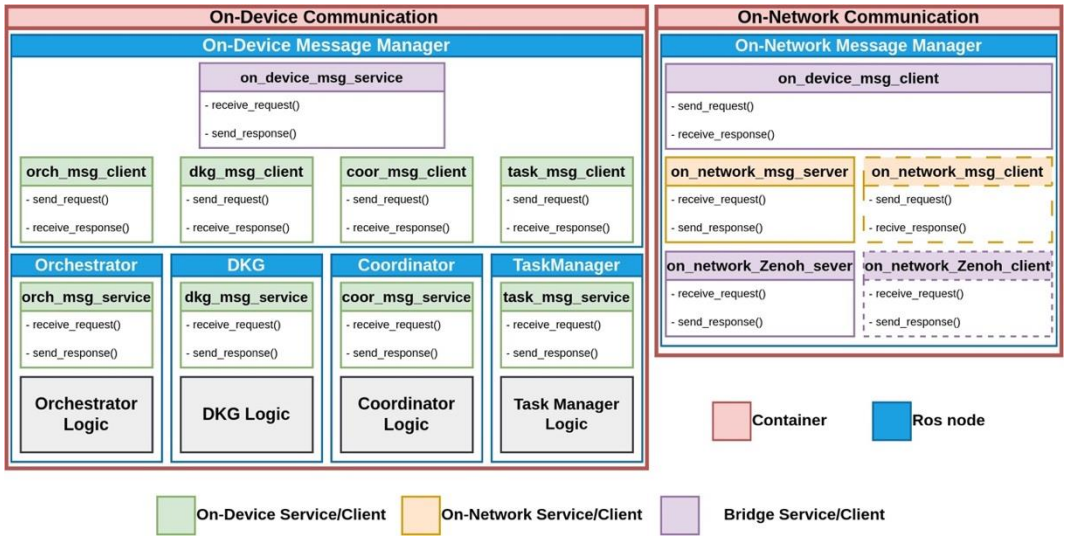


Figure 5-2. Overview of architecture of the Message Manager

Figure 5-2 illustrates the architecture of the Message Manager, divided into on-device communication components and on-network communication components, each responsible for facilitating communication either within the components on the same node or across multiple nodes:

- Red Boxes: Represent Docker containers, which encapsulate ROS nodes and related services.
- Blue Boxes: Represent ROS nodes responsible for handling message routing.
- Green Boxes: Represent ROS services and clients dedicated to on-device communication.
- Orange Boxes: Represent ROS services and clients facilitating on-network communication.
- Purple Boxes: Represent the Zenoh bridge, which enables integration with external devices or protocols outside of ROS2.

Control messages received from the on-network communication components are routed to the appropriate on-device components for further processing.

The messages are organized using the following ROS syntax:

- string requester: Specifies the ID of the node initiating the request.
- string request_type: Defines the category or type of the request.
- string request_description: Provides a detailed explanation of the request, including any necessary parameters or context in JSON format.
- string response: Contains the response to the request, which may include results, acknowledgments, or error messages.

The request_type field in the message syntax identifies the component or service that should handle the request. This allows messages to be routed within the system. The following table provides details about the currently defined request types:

Name	Description
ORC	**Orchestrator**: Requests of this type relate to Orchestration tasks.
COO	**Coordinator**: Requests of this type relate to Coordinator tasks.
DKG	**Dynamic Knowledge Graph**: Request of this type related to DKG tasks.
TMN	**Task Manager**: Request of this type realted to Task Manager tasks.

Figure 5-3. Table of request type supported in the current implementation.

An example of the message can be found in the demonstration described in Section 4.3, with additional details provided in the README file available in our GitLab repository.

5.2.1.2 Dynamic Knowledge Graph

The Dynamic Knowledge Graph (DKG) functions as a central repository and query engine for managing metadata, task states, and contextual information within the SmartEdge ecosystem. Its primary purpose is to enable seamless interaction between SmartEdge nodes by providing real-time updates and supporting both static and dynamic queries. This section outlines the DKG's implementation, covering its integration with existing RDF frameworks, query capabilities, APIs, and message routing mechanisms.

The current version of the DKG operates in embedded mode, allowing deployment on any SmartEdge smart node that supports the Java Virtual Machine (JVM). This design ensures compatibility across diverse devices within the SmartEdge ecosystem. Implemented in Java, the DKG integrates with ROS2 using JPype to facilitate communication with other components in the system, which predominantly rely on ROS2 Python. JPype acts as a bridge, enabling seamless interaction between the Java-based DKG and ROS Python nodes, allowing for the exchange of queries, responses, and real-time updates.

The DKG supports two query types: One-Shot Queries and Continuous Queries. One-shot queries execute a single SPARQL query to retrieve specific information, such as the current state of a node or resource availability in the swarm. Continuous queries, on the other hand, register monitoring tasks within the knowledge graph to provide real-time updates to applications. For instance, a continuous query can track the availability of nodes with specific skills and notify relevant components when changes occur.

To manage RDF data and execute SPARQL queries, the DKG uses the RDF4J framework, a robust and feature-rich library. For storage, it employs the RDF4Led approach, a lightweight RDF storage solution optimized for resource-constrained edge devices, ensuring efficient data handling even in limited environments. Furthermore, the DKG leverages the CQELS codebase to extend SPARQL querying capabilities for continuous queries, allowing it to monitor real-time data streams and dynamically notify subscribers of updates.

To interact with the DKG, messages are structured as follows:

```
{
  "requester": "coordinator_01",
  "request_type": "DKG",
  "request_description": {
    "operation" : "query",
    "query" : "\"SELECT ?node WHERE { ?node :hasSkill :ObjectDetection. }\""
  }
}
```

The DKG provides a set of APIs to interact with the knowledge graph. These APIs facilitate data insertion, deletion, querying, and registration of continuous queries. Key API functions include:

- Insert: Add new RDF triples to the knowledge graph.

```
{
  "operation": "insert",
  "data": {
    "content": "<node_01> <hasSkill> <ObjectDetection> .",
    "format": "n-triples"
  }
}
```

- Delete: Remove specific RDF triples from the knowledge graph.

```
{
  "operation": "delete",
  "data": {
    "content": "<node_01> <hasSkill> <ObjectDetection> .",
    "format": "n-triples"
  }
}
```

- Query: Execute a one-shot SPARQL query.

```
{
  "operation": "query",
  "query": "query_string"
}
```

- Register Query: Register a continuous query and receive updates when changes occur.

```
{
  "operation": "register",
  "query": "query_string"
}
```

5.2.1.3 Task Manager

The Task Manager is a core component of the SmartEdge runtime, responsible for managing and coordinating tasks assigned by the Orchestrator on a SmartEdge smart node. Each task involves executing a pre-packaged SmartEdge Primitive, which encapsulates a specific computational operation or function required to achieve the task's objectives. These primitives are executed within the Primitive Runtime, a flexible containerized environment that is dynamically provisioned to execute the primitive.

The Task Manager handles the lifecycle of the Primitive Runtime, including initialization, execution, and termination. During initialization, it ensures resources, configurations, and dependencies are in place for successful execution. During execution, the Task Manager monitors performance and ensures operations meet requirements. After task completion or when no longer needed, the Task Manager terminates the Primitive Runtime, freeing resources to maintain system efficiency.

The Task Manager is implemented as a ROS2 node in Python and utilizes the Docker library to manage Docker containers that host primitives. It maintains a mapping of tasks to their corresponding Docker containers, enabling efficient tracking and management of tasks throughout their lifecycle. This mapping simplifies operations such as starting, monitoring, and terminating tasks while ensuring optimal resource utilization.

The Task Manager provides a ROS2 service for on-device communication, named "*node_<node_id>_task_manager*", where "*<node_id>*" represents the swarm ID of the node. For instance, in the demo discussed in Section 4.3, the service name is "*node_1_task_manager*". When the on-device message manager receives a primitive message, it creates a Task Manager service client to call the service, either to execute or terminate the primitive(s). To call the Task Manager service, the client must send a request with the following four fields:

- ***action***: An integer value of 1 or 0, indicating whether the Task Manager should execute (1) or terminate (0) the specified primitive.
- ***task_id***: A string representing the unique task ID to which the primitive belongs. Note that a single task may consist of multiple primitives.
- ***primitive_id***: A string representing the unique ID of the primitive.
- ***task_description***: A JSON-encoded string describing the primitive, which contains the following sub-fields:
    - ***inputStream***: Specifies the endpoint and protocol for the input data stream that the primitive will subscribe to.
    - ***primitive***: Describes the operation and model to be used in the primitive execution.
    - ***outputStream***: Specifies the endpoint and protocol for the output data stream that the primitive will publish to.

```
{
    "action" :  1,
    "task_id" : '1',
    "primitive_id" : '1',
    "task_description":
    {
        "inputStream":{
            "protocol": "rtsp",
            "endpoint": "rtsp://<your-token>/camstream/jatkasaari_263_2"
        },
        "primitive": {
            "op": "ObjectDetection",
            "model" : "yolo_objectdetection:yolov8n.pt"
        },
        "outputStream": {
            "protocol": "ros2-dds",
            "endpoint": "/node01"
        }
    }
}
```

Figure 5-4. An example of a Task Manager service request

When a task is requested to start, the Task Manager follows a structured workflow to ensure efficient execution. First, it checks whether a Docker image for the task already exists. If the image is available, the Task Manager directly creates a container from it. If no pre-existing image is found, the Task Manager dynamically generates a Dockerfile based on the task description. This process includes selecting a base image that aligns with the task's requirements, such as compatibility with ROS2 or specific operating system configurations, specifying the required libraries and dependencies, and defining the input and output streams' protocols and endpoints. Once the Dockerfile is generated, the Task Manager builds the Docker image and creates a container to execute the task. The Task Manager tracks details such as the task ID, primitive ID, Docker image ID, and Docker container ID, maintaining an updated mapping between them to streamline lifecycle tracking and management.

When a request to stop a task is received, the Task Manager efficiently handles the termination process. It identifies the associated container using the task-to-container mapping, stops the container, and removes it to free up system resources. The mapping is then updated to reflect the task's completion and removal from the system.

Once the Task Manager completes the execution or termination of a primitive, it responds to the service client with two fields: a code and a message, both as strings. The code reflects the Task Manager's status for the request and is categorized into three types: success, failure, and warning.

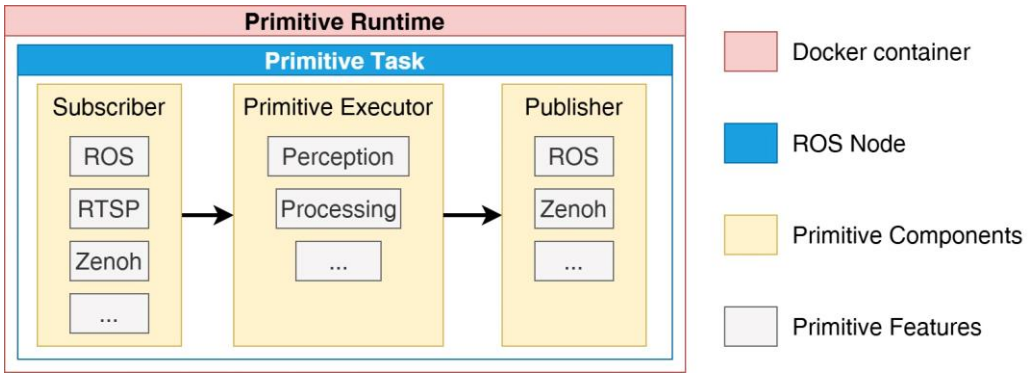
By managing task initialization, execution, and termination within Docker containers, the Task Manager ensures robust and efficient task management, meeting the dynamic and distributed demands of the SmartEdge ecosystem. This modular, containerized approach enhances scalability, portability, and overall system performance, enabling seamless operation across heterogeneous environments.

5.2.1.4 Primitive Runtime

The Primitive Runtime is a core component of the SmartEdge Runtime, responsible for executing task primitives on SmartEdge smart nodes. Primitives represent the essential building blocks of the system's computational capabilities, each performing a specific operation critical to the execution of distributed tasks within the device-edge-cloud continuum. These operations include, but are not limited to, object detection, object tracking, and graph stream processing, addressing a wide range of application requirements.

For instance, in the context of a smart factory use case, an object detection primitive might analyze real-time video streams to identify and classify objects within a production area. Similarly, an object tracking primitive could monitor the movement of identified objects across frames, providing insights into dynamic processes. For applications involving complex data relationships, a graph stream processing primitive might handle tasks like matching patterns or aggregating information in real-time.

The Primitive Runtime ensures efficient execution of these operations by leveraging available hardware resources on SmartEdge nodes, such as CPUs, GPUs, or accelerators. It isolates each primitive within a controlled environment, often utilizing containerized deployments, to prevent conflicts and optimize resource utilization. This modular design allows the Primitive Runtime to adapt to the heterogeneous capabilities of edge devices while maintaining high performance and scalability. Through its ability to execute diverse primitives, the Primitive Runtime empowers the SmartEdge system to meet the computational demands of various applications in real-time distributed environments.



3

Figure 5-4. Primitive Runtime

When a task is assigned, the Task Manager establishes a Docker containerized environment to serve as the execution platform for the Primitive Runtime. The configuration of this container depends on the task requirements. it may be isolated for internal communication or configured for external access. The container includes all necessary components, such as required libraries, dependencies, and environment variables, ensuring the runtime is tailored to the specific task's operational needs.

The Primitive Runtime operates within this container and comprises three core components, as illustrated in Figure 5-5:

- Subscriber: The Subscriber functions as the entry point for incoming data streams or messages from other nodes or external sources. It supports various input protocols and methods, including ROS (Robot Operating System), Zenoh, and RTSP (Real-Time Streaming Protocol). For instance, in a typical scenario, the Subscriber might receive video streams via RTSP using OpenCV, where the data consists of video frames intended for further processing.
- Primitive Executor: At the heart of the Primitive Runtime, the Primitive Executor processes the incoming data based on the assigned task. This component executes computational or transformative operations such as object detection, object tracking, image recognition, sensor data fusion, data aggregation, or event detection. In the provided example, the Primitive Executor utilizes the YOLO model to perform Object Detection on video frames received from the Subscriber. The processed data is then forwarded to the Publisher for dissemination.
- Publisher: After completing the processing, the Publisher handles the transmission of output data to the next node in the pipeline or back to the SmartEdge Coordinator. It supports multiple output protocols and methods, including ROS, Zenoh, and DDS (Data Distribution Service). In the aforementioned example, the Publisher converts the output data from Object Detection into DDS format and publishes it to the designated topic, facilitating seamless integration with downstream processes.

Detailed demonstrations of the SmartEdge Runtime's primitives are provided in Section 4.3, showcasing their practical applications. Examples include vehicle counting at Junction 266 (Use Case 2) and collaborative observation in a smart factory environment (Use Case 3). These use cases illustrate the capabilities of the Primitive Runtime in real-world scenarios, highlighting its flexibility and efficiency in executing diverse computational tasks.

5.2.2 SmartEdge Plugins

SmartEdge plugins enhance the functionality of the SmartEdge Runtime by enabling seamless integration with external tools, libraries, and services. These plugins are designed to improve the system's adaptability, performance, and versatility. They support specialized processing tasks, leverage hardware acceleration, and enable interoperability with third-party systems, making the SmartEdge Runtime a flexible and modular platform capable of meeting diverse application requirements within the device-edge-cloud continuum.

The SmartEdge ecosystem includes several key plugins, such as the P4 Runtime Plugin for advanced network control, in-network ML plugins for distributed machine learning inference, and a Security Plugin to enhance system resilience and protect data integrity. Mendix is a graphical low-code IDE. With components developed in WP3, it enables the implementation of recipes within Mendix. The Mendix plugin for the SmartEdge runtime allows seamless integration of SmartEdge nodes managed by the runtime into these recipes. Chunks and Rules plugin supports cognitive artificial intelligence workflows. These plugins collectively expand the capabilities of the SmartEdge Runtime, allowing it to address complex, distributed computing scenarios with efficiency and scalability.

5.2.2.1 P4 Runtime Plugin

The P4 Runtime Plugin (see Figure 5-6) is a software module that handles the configuration of the P4 switch, it offers an API that allows P4 controllers to interface remotely with the P4 switches within the swarm to access the functional services offered by the switch in order to

read the state of the P4 switch including the tables, counters, registers, and other relevant configuration parameters. Furthermore, it allows the controller to make changes to the switch configurations, such as adding or modifying table entries, or reading and writing from/to registers. The plugin is composed of a control server and an agent, where the server runs on the P4 switch and the agent connects over a TCP connection.

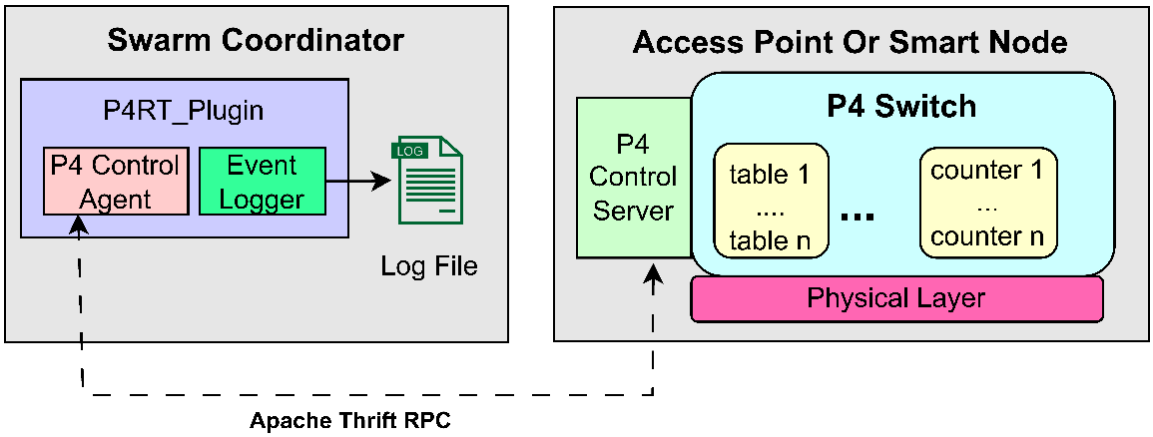


Figure 5-5. P4 Runtime Plugin between coordinator and AP or Smart Node

The swarm coordinator implements a control agent and is responsible for keeping track of the state of the P4 switches in the network and updating the match-action entries every time a node joins or leaves the swarm. The BMv2 software switch that is currently running in the access point offers an API for Remote Procedure Call (RPC API) over both Apache Thrift protocol and Google Remote Procedure Call (gRPC). The current P4 Runtime Plugin builds on the available python BMv2 thrift library from the GitHub repository of the P4 group to implement its functionality. There can be multiple P4 control agent software modules which keep an open connection to the P4 switch control server, and whenever a change in the state of swarm is detected or a new configuration need to be written by the coordinator or to be forced by the coordinator, the P4 control agent reads and verifies the state of the affected swarm nodes before sending the new P4 configuration to the relevant nodes. This plugin constructs a higher abstraction layer that is established on top of the original P4 thrift library.

The plugin is written as a python module and is imported by the main script of the coordinator, it contains the necessary functionality that help to translate and communicate the high level commands issued by the coordinator to the P4 switch without the need of the coordinator to know the exact implementation of the P4 program or the exact communication protocol. It is served as well by an event-logging module that keeps track of all the operations performed by the plugin that serves the following purposes:

i. Keeping track of the swarm state evolution during the swarm lifetime.
ii. Debugging purposes for the improvement of the SmartEdge workflow.
iii. Following changes to the swarm state, that can be used in conjunction with other telemetry data to help with the detection of security issues.

This plugin also offers the connectivity between various Access Points within the swarm, allowing fast updates of other access points without the need for the coordinator to intervene in order to propagate the change. For example, if a smart node disconnects abruptly from its associated access point without sending the Leave message, this disconnection is detected by

the access point manager, and immediately transmitted as a P4 configuration update to access points simultaneously, as well as an update to the coordinator to update the state of the swarm.

This plugin in this release implemented a few changes that have been made with respect to the release reported in D5.1:

- Addition of the logging functionality.
- Collection of P4 telemetry data, has been made to be optional and can be activated for certain nodes and certain types of packets.
- The Join and Leave messages in case the process is initiated by the swarm node, have been extended to reach the coordinator which takes the final decision if a node is accepted in the swarm or not.

The Switch message is deprecated, and smart nodes can move freely between Access Points without the need to inform the coordinator.

5.2.2.2 In-Network Machine Learning

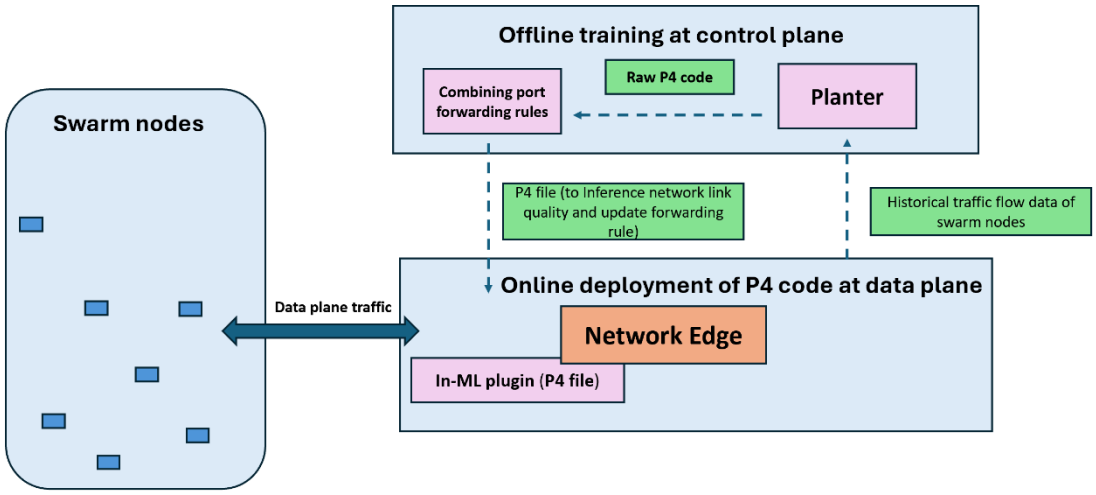


Figure 5-7. In-Network Machine Learning Plugin

The purpose of this plugin is to deploy In-network machine learning components (e.g., intelligent services based on Planter [Zheng24]), on edge servers within the swarm network. This allows for real-time monitoring and necessary configuration of network quality and swarm nodes connected to the network. For instance, by dynamically updating data forwarding rules to schedule connectivity to different swarm nodes in the P4 data plane, the plugin aims to achieve fast and stable communication transmission quality, as well as accurate application-specific objectives.

Specifically, deployment of the In-network machine learning plugin at the swarm edge can be divided into two phases: offline training and online inference (see Figure 5-7).

Offline Phase: In this phase, collected data from network links and swarm nodes can be input into the Planter component within the control plane. Customized machine learning training algorithm can be executed to infer various potential task objectives, such as determining whether a node's position within the wireless network possesses sufficient network resources

for the required data transmission, or whether the current link quality is experiencing congestion. Based on the training results, customized P4 code will be generated. This code includes logic for making inferences within the P4 data plane based on real-time data. Subsequently, according to the inference results, port forwarding rules can be updated to exclude nodes with weak signal quality or those connected to congested links from the forwarding path, thereby preventing unnecessary transmission delays or failures.

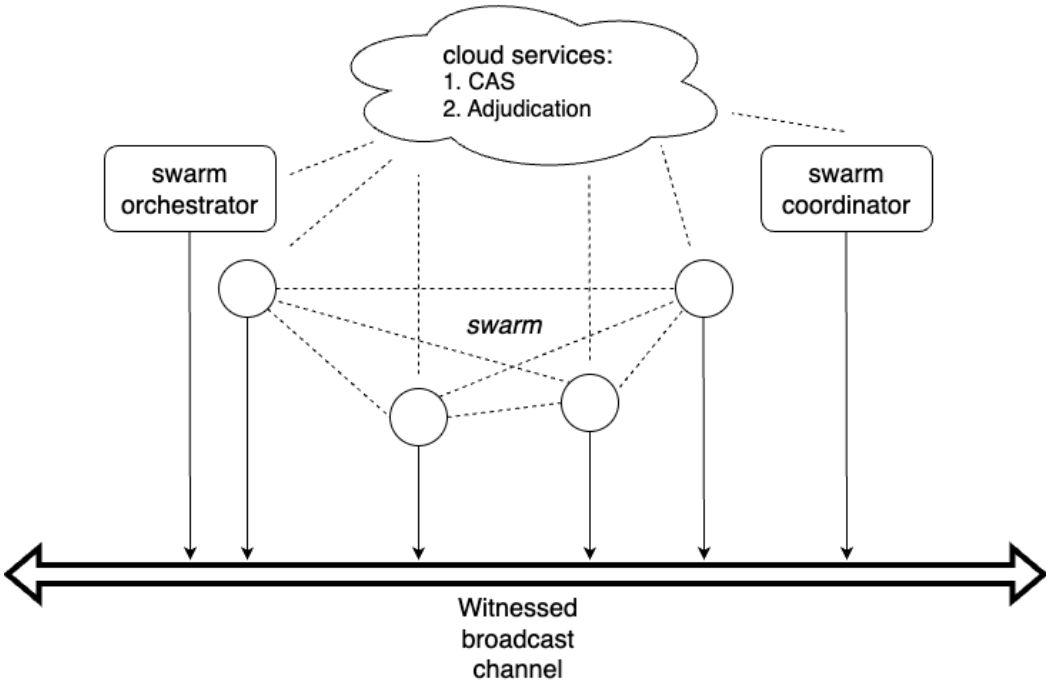
Online Phase: During this phase, the generated P4 code can be integrated into existing P4 data plane code and deployed onto the edge device where it operates in real-time, processing incoming swarm data packets. Once a specific swarm node is detected to be in a suboptimal link state based on pre-trained models, the corresponding port forwarding rules are dynamically updated (for example, avoiding forwarding to swarm nodes with weak signal quality). Similarly, when a moving node regains good signal quality for real-time data exchange, it will be added back to the forwarding table. At the same time, the P4 program continues to perform the necessary inferences directly on the data plane, leveraging pre-trained models or logic to make decisions at line rate. As these inferences occur, associated performance metrics and statistical data—such as packet processing times, path selections, and load distribution—are continuously recorded. This real-time data is then fed back into the control plane, where it can be analyzed to further improve and refine the offline training models. This feedback loop helps to optimize the system by allowing for adaptive learning based on actual network conditions and traffic patterns, ensuring that future deployments are more efficient and responsive to changing dynamics.

5.2.2.3 Security

IMC has completed security requirement gathering with the use-case partners, which was a very successful exercise that helped to define, refine and generalize security expectations inherent in the swarm computing paradigm of SMARTEDGE. As a result, we posed two problems: (i) security support of a recipe computation by a swarm and (ii) security support of swarm crowd sensing. Problem (i) is typical for swarms operating in a smart factory, while problem (ii) naturally arises in sensor-swarms in smart cities, where sensors are co-located with humans through either wearing them (as in medical use cases) or keeping them in a personal vehicle (as in traffic control use cases).

Both problems were studied, and novel security protocols were proposed. Based on those, software development of demonstrator security systems was initiated and is currently underway. We expect to create two software suites (Artifact 1 and 2 below) that implement the protocols.

5.2.2.3.1 Light-weight ledger for robot swarms



IMC has developed a low-cost security infrastructure for robot swarms to run recipes securely. At the core of the proposal is a distributed ledger maintained using an extended version of the Wintenitz Stack Protocol (WSP) [Shafarenko24]. The swarm orchestrator acts as the ledger custodian by pushing frames on a stack according to the protocol. The smart nodes send state-transition messages and additional protocol data that guarantee that

1. Each message is genuine
2. It has been received intact by all swarm members so that the ledger is consistent across the swarm

The protocol operates on a zero-trust basis, which means that the content of all communications can be proven to be genuine (or counterfeit) without trusting any member including the orchestrator. The only requirement is that at least one node in the swarm is not compromised (otherwise the fully compromised swarm would be able to collude and produce an alternative ledger after the fact).

The ledger guarantees **non-repudiation** of any transactions based on the messages placed on the ledger stack. We assume that the recipe is a state transition system. We do not care about the precise semantics of the transitions as long as all of them depend only on the messages received so far, since the ledger guarantees that those messages are genuine and have been received by all swarm members in exactly the order they appear on the ledger. Since messages can be of a variable size, they are not broadcast as such; they are stored in Content-Addressed Storage (CAS) and their fixed-size hash value is broadcast.

Due to the low-latency requirement for robot swarm communications, the consensus mechanism cannot be any one of the conventional arrangements (such as Proof of Work). Since all messages are broadcast and since the broadcast is typically over the radio, we have proposed that the consensus mechanism be optimistic (i.e. aiming to detect, rather than prevent the ledger split) and that it is realized in the form of a Witnessed Broadcast Channel (WBC). The WBC can be implemented as Wi-Fi broadcast with extra nodes (“witnesses”) preset at the scene, which listen on the channel but never transmit on it. The witnesses use a protected side channel

to “compare notes” and to detect any discrepancies. No knowledge of the protocol is required of a witness; all that witnesses are trying to establish is that broadcast messages are not jammed and replaced as they travel across the premises. An attacker would not be able to suppress any of the witnesses unnoticeably since their location is unknown not only publicly, but even to swarm members themselves (thus eliminating the inside threats).

IMC will implement the protocol as a software suite with four components:

1. Orchestrator WSP driver (a Python library)
2. Smart node WSP driver (a Python library)
3. Adjudicator (a Python app)
4. CAS

At this time the protocol design is complete, and one of the four components (CAS) has been implemented.

5.2.2.3.2 Anonymous Reputation Management System

In a crowd sensing swarm, smart nodes need to be trusted to provide genuine sensor data. This is difficult, since the data includes the time and location of the sensor, which, in the case of traffic control, is typically embedded in a car driven by a human. Registering the driver with the purpose to authenticate the sensing report has an unfortunate side effect of disclosing personal data, namely the location of the driver at a certain time. The uptake of the crowd-sensing traffic control system could be impeded by users (drivers) needing to trust the system servers not to disclose the time- and location-referenced identity to any third party as this would violate EU GDPR and might create an opportunity for a criminal.

IMC have proposed a novel security protocol based on semi-blind signatures and Guy-Fawkes linkage of the sensing report with the user’s anonymous certificate and reputation update coupon. The algorithms involved no modular exponentiation by a swarm node due to the use of a small public exponent; the protocol is thus low latency. It is important from the point of view of swarm universality, i.e. suitability of the security infrastructure to swarms of any kind, including low-footprint, energy-limited ones.

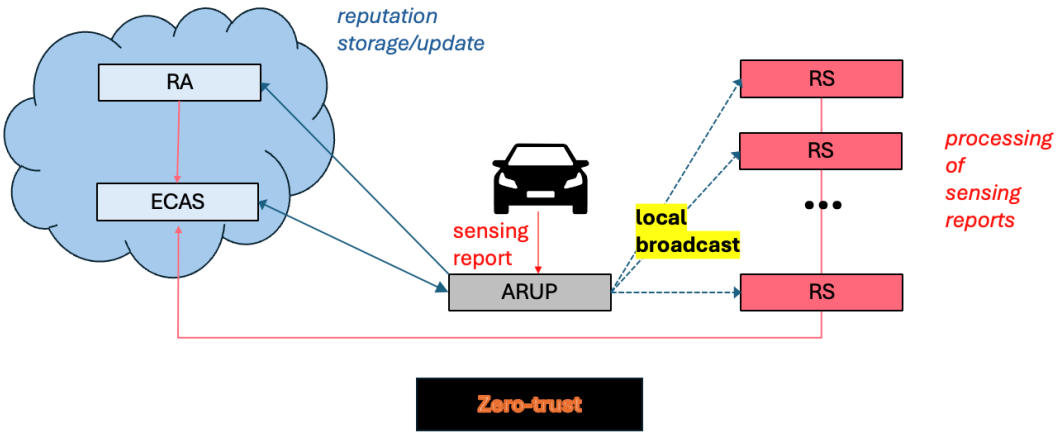


Figure 5-7. Overview of Autonomous Reputation Management System

Implementation. There are three components to implement: the Registration Authority (a server), RA, the Reputation Server (RS) and the driver for the ARU (Anonymous Reputation

Update) Protocol. There is also the need for an extended version of CAS, but this component has already been implemented by us.

5.2.2.4 Mendix Plugin

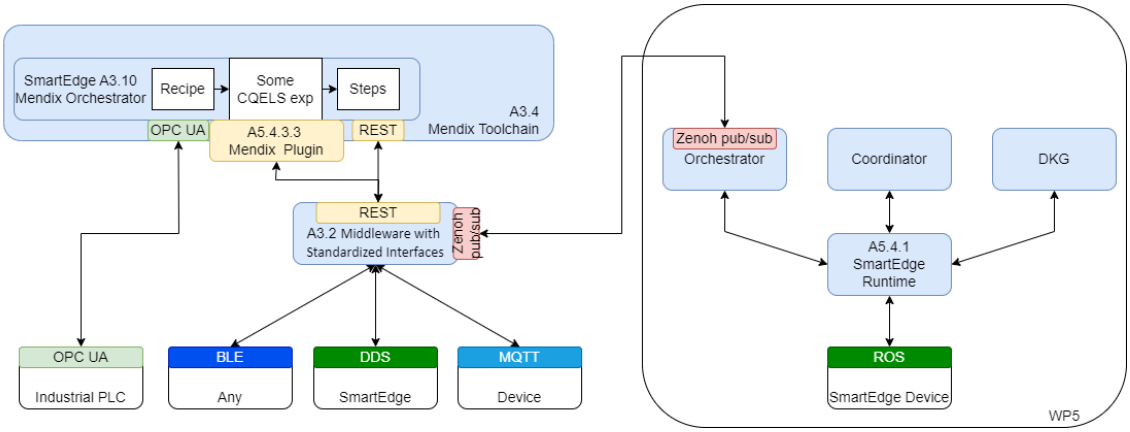


Figure 5-8. Mendix Plugin for SmartEdge Runtime

The Mendix plugin enables interaction between the SmartEdge nodes, which are under control of the SmartEdge runtime (A5.4.1) and the recipes developed within the Mendix toolchain (A3.4). This integration expands Mendix's functionality, making it possible to design recipes and execute applications that include the nodes that are controlled by the SmartEdge runtime (A5.4.1).

Communication between the Mendix plugin and the swarm dynamic orchestrator is enabled by Zenoh middleware. A Zenoh client will be implemented within the Mendix toolchain, enabling the plugin to directly interact with the dynamic swarm orchestrator. The plugin uses Zenoh's publish-subscribe model to send CQELS-QL expressions for executing specific recipes within the orchestrator. By subscribing to Zenoh topics, the plugin retrieves back the results from the SmartEdge runtime.

Mendix IDE will be extended with the SmartEdge CQELS-QL Editor to allow users to create CQELS expressions as part of their recipes.

5.2.2.5 Chunk and Rules

Chunks & Rules [https://w3c.github.io/cogai/chunks-and-rules.html] applies research in the cognitive sciences to simplify IoT application development, layering on top of the hardware abstractions provided by digital twins and devices using the robot operating system (ROS). Event driven concurrent threads of behaviour are described with facts and rules using a convenient easy to learn syntax at a higher level than RDF.

The cognitive architecture is inspired by John Anderson's ACT-R, mimicking characteristics of human cognition and memory, including spreading activation and the forgetting curve. The rule engine operates on chunk buffers associated with cognitive modules, where each buffer holds a single chunk, i.e. a collection of properties forming a unit of knowledge. This provides for much higher performance compared to rules that operate directly on knowledge bases. Rule actions are asynchronous, enabling distributed cognition and real-time control.

Perception builds live models of the environment for situational awareness, including events that trigger appropriate behaviours. A built-in suite of operations can be used on chunk graphs. Applications can define their actions in terms of intents, i.e. aims, purposes, goals or objectives. An example is an intent for controlling a robot arm. The intent specifies the position and orientation of the arm's gripper, but not the details of the various actuators in the arm, nor the trajectory for smoothly changing them from the current state to the target state.

This mimics the brain where decisions formed by sequential cognition are delegated to the cortico-cerebellar circuit. This determines which muscles to use and generates real-time control signals using feedback via perceptual models in the cortex. An example is reaching out to grab a mug of coffee on your desk. This uses a combination of dead reckoning and corner of the eye perception to fine tune motions. This architecture decouples reasoning from real-time control.

ROS is an open-source framework for robots with a strong developer community. It is message based with hardware abstraction. Applications can subscribe to topic-based streams and invoke services using request/response pairs. Chunks & Rules is a good fit to controlling ROS devices. ROS streams can be used to update chunk models of the robot and its environment, as well as to trigger events. Chunk & Rules can be used to invoke ROS services with delegation for planning and execution.

A single cognitive agent may be used to control multiple devices. In the web-based factory demo, a single Chunks & Rules agent controls two conveyor belts, one robot arm, a bottle filling station and a bottle capping station. The bottles are placed into boxes holding two rows of three bottles. Another demo simulates a smart home with control over lighting and heating, along with presence-based implementation of personal preferences.

Cognitive rules can respond in milliseconds and can be complemented by faster reactions using simple reflex responses implemented at a lower level. Application development is a collaboration between the people maintaining the low-code description of high-level behaviour and system programmers responsible for the lower-level code needed to integrate digital twins for sensors and actuators. Development starts using a simple approach and iteratively refines it as new requirements come to light, e.g. when something unexpected occurs at run-time and needs to be handled. This may necessitate improvements to digit twins, e.g. to sense error conditions. In the robot example, unusual situations could include a bottle falling over, being only partially filled or badly capped.

Chunks & Rules is well suited to swarms of agents. They can take on different roles as needed, and communicate by name, role, topic or proximity. Agents can further leave messages in the environment as a form of stigmergy. This involves a high level of abstraction that hides the details of the underlying protocols and communication technologies. Agents can work together to form collective (hive) minds in relation to situational awareness and hive goals. This allows agents to collaborate on combining diverse sources of information from different sensors into a unified model of the environment, and likewise to collaborate on solving goals.

In respect to implementation, open-source JavaScript libraries are available for the Chunks & Rules engine, and for interfacing to ROS and dealing with the lower-level messaging protocols. These libraries can be used with NodeJS. Work is underway to study the feasibility of integrating Chunks & Rules within Mendix. The concept of "recipes" in Mendix is essentially the same as "intents" for Chunks & Rules, in that they specify what needs to be done, but not how. This also relates to the Web of Things, which uses JSON-LD to describe the affordances of digital twins in respect to an object model with properties, actions and events.

5.2.3 Low-code IDE

To minimize programming efforts, the artifact is implemented with a GUI that allows users to describe application specifications using natural language or drag-and-drop functionality (via the Mendix Plugin). This user-friendly interface enhances accessibility and simplifies the process of specifying applications, enabling users to easily interact with the system and create complex programs without extensive coding knowledge.

The Low-code IDE also includes a compiler responsible for transforming application specifications into semantic programs. These semantic programs are subsequently interpreted by SmartEdge coordinators to assemble SmartEdge swarms. In this phase, the compiler also prepares scripts for the SmartEdge Runtime to deploy SmartEdge primitives required by the application. This feature ensures seamless integration and execution of application functionalities within the SmartEdge environment, optimizing performance and user experience.

5.2.3.1 Model Builder

The Model Builder is an intuitive graphical user interface designed for the training and deployment of customized computer vision models, as shown in Figure 5-9. It streamlines the development process by integrating user inputs as based on a SPARQL interface, facilitating efficient access and exploration of pre-trained models through both SPARQL queries and natural language inputs.

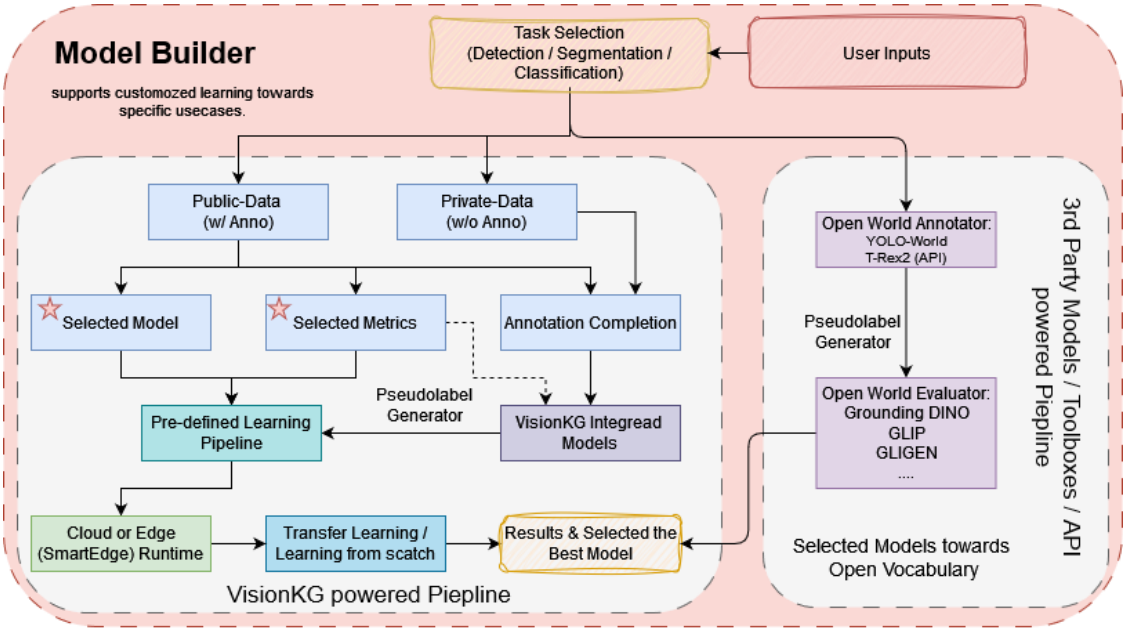


Figure 5-9. Pipeline of the Model Builder

This dual-input capability significantly empowers users by articulating their specific requirements or criteria through interactive graph query patterns, thereby enhancing its usability towards desired use cases or tasks, such as detection and classification, segmentation, or visual relationship detection. To ease the deployment of trained learning systems towards diverse edge devices, in this part, we will introduce a cross-device inference engine build, as shown in Figure 5-10. In this way, we will utilize computing power either from the cloud or edge to find the best-fit operators (trade-off between the inference speed and metrics of evaluation) for target devices, aiming to accelerate the inference.

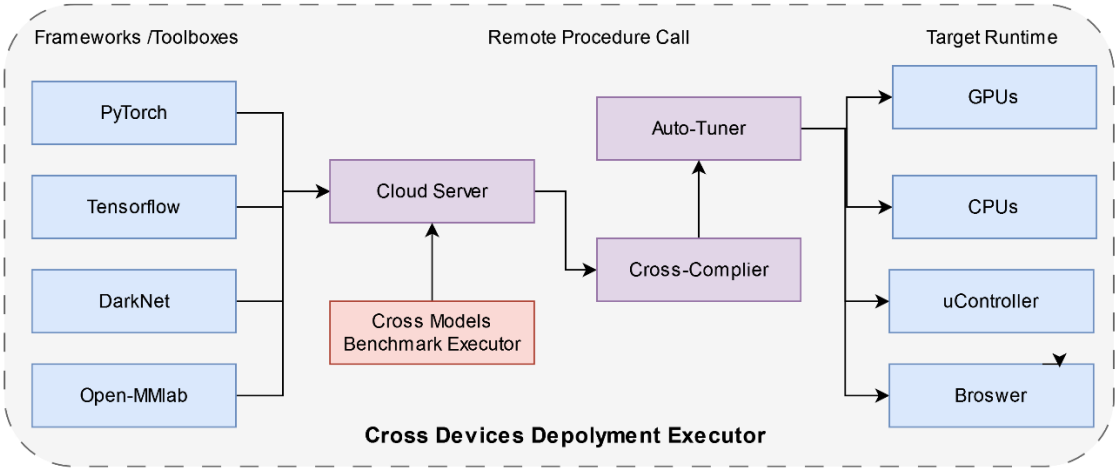


Figure 5-10. Cross-Devices Inference Engine Builder

Following the selection of a task, the system will showcase a comprehensive list of all models compatible with the selected task. Furthermore, the integration of Automated Machine Learning (AutoML) will ease the learning curve of potential users in various algorithms, architectures, and configurations. It also simplifies the process of model ensemble and optimizes the selection of models tailored to specific use cases, as shown in Figure 5-11.

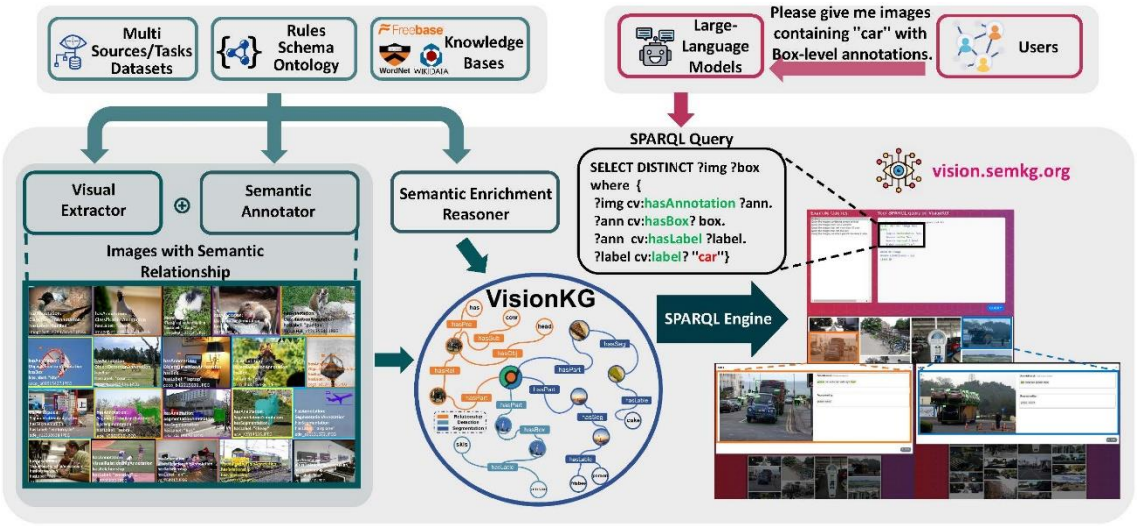


Figure 5-11. Overview of the construction of learning / evaluating pipeline using VisionKG

With model builder, users can initiate a training pipeline by writing queries to construct models targeted at diverse composite visual datasets. Specifically, users can retrieve images and annotations using a few lines of SPARQL code to employ RDF-based descriptions and obtain desired data, such as images with box-level annotations of *car* and *person* from interconnected datasets in VisionKG. Combined with popular frameworks like PyTorch and TensorFlow, or toolboxes such as MMDetection and Detectron2, users can leverage the retrieved data to construct customized learning models using minimal Python code effortlessly. Benefiting from the interconnected datasets and extensive model zoo, users only need to specify the model they wish to use and the hyperparameters they intend to set. Figure 5-12 presents a simplified example of code for constructing an object detector using VisionKG data.

```python
# Import VisionKG utilities and integrated training pipeline
from vision_utils import semkg_api
from torch_model_zoo import utils
from torch_model_zoo.train_eval import train_eval_pipeline

def prepare_vkg_pipeline(query_string):
    # Execute the query
    rels = semkg_api.query(query_string)
    params = utils.prepare_for_training(rels)
    return params

# SPARQL query to VisionKG for images with specific objects
query_string = ''' SPARQL query for object detection '''
params = prepare_vkg_pipeline(query_string)
params['MODEL'] = 'fasterrcnn_resnet50_fpn'
train_eval_pipeline(params)
```

Figure 5-12. A simplified example of VisionKG pipeline serving to the Model Builder

In this way, based on VisionKG built-in model builder, users can carry out more complicated MLOps steps, from dataset exploration, and distillation to training execution, including composing visual datasets with unified access and taxonomy through SPARQL queries, automating training and testing pipelines, and expediting the development of robust visual recognition systems. The proposed model builder's features enable users to efficiently manage data from multiple sources, reduce overheads, and enhance the efficiency and reliability of machine learning models.

5.2.3.2 Metric Reporting & Visualization

The metric reporting and visualization solution allows collecting any system data, storing it, and evaluating various metric types in streaming media applications. With this SmartEdge components and use cases can be monitored and visualized. The aim is to improve the behavior of the monitored components and to detect error states.

5.2.3.2.1 Main components and Functionalities/Features

The metrics solution is made up of three conceptional components: metrics providers, a metrics server and metrics consumers. Figure 5-13 gives an overview of the structures. It also shows that multiple providers and consumers are possible whereas a default metrics consumer as Grafana Dashboard is provided.

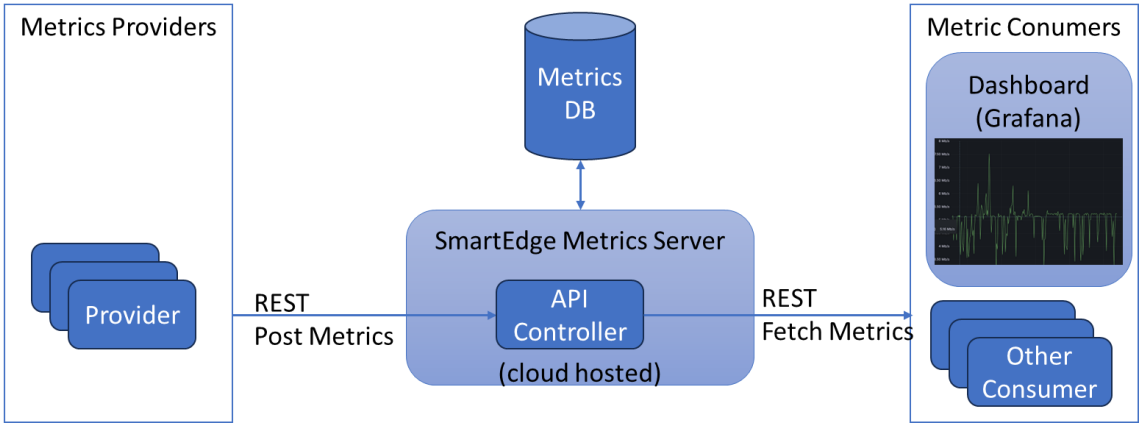


Figure 5-13. Metric Reporting and Visualization Components

The metrics client is about reporting any network or application specific metrics, for example the current video stream bitrate, the client-side playback framerate and the round-trip time

between client and server if a video stream playout is monitored. If the target is to monitor the hardware state of a specific machine the metrics client may collect data about the system platform such as CPU and GPU performance, memory utilization and storage space available. It should be noted that the set of metrics can be extended depending on the requirements of the metrics client.

All collected metrics are sent to the metrics server, which exposes its functionality through a REST interface. The data is sent in JSON format, the schema for which is shown in the following section.

Lastly, the metrics stored on the metrics server can be continuously visualized. For visualizing the data, a Grafana dashboard is provided. The dashboard allows real-time system monitoring so that changes in the monitored system are directly reflected in the dashboard. In addition to the dashboard all collected data can be requested by external applications so that specific analytics can be made for cases where the dashboard is not sufficient.

5.2.3.2.2 Component implementations

Message Schema

Any metric report must be sent to the server in JSON format. This section describes the message schema expected by the metrics collection server through one example for each side. Mandatory key-value pairs are highlighted with bold text. Other pairs are optional.

The following listing shows an example JSON as currently reported by the remote rendering artifact:

```
[
  {
    "Body": {
      "rsi": "ec0934f2-72cc-4b1f-8e5c-cd46f7cca34a",
      "ts": "1731925738",
      "mt": "PerformanceMetricType",
      "ge": "unity",
      "sn": "Remote Warehouse V100",
      "vc": "VP8, rtx, VP9, H264, AV1, red, ulpfec",
      "ac": "opus, red, multiopus, ILBC, G722, PCMU, PCMA, L16, CN, telephone-event",
      "gm": "Tesla V100-PCIE-32GB",
      "cm": "Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz",
      "cpu": "144",
      "gpu": "79",
      "sysMem": "7500",
      "gpuMem": "3465",
      "hn": "hostname"
    }
  }
]
```

The JSON consists of a list with a single object inside. This object has the attribute “Body”, which is mapped to an object holding all reported values. “rsi” and “ts” are mandatory fields, the first of which holds the unique identifier of the renderer while the second is the timestamp for which data is reported. The following list explains each subsequent optional key-value pair:

Key	Value Description
rsi (mandatory)	Unique identifier of the reporting instance (as uuid)
Ts (mandatory)	timestamp as seconds since 1.1.1970 (unix epoch timestamp)
mt	Metric type (currently only performance)
ge	Rendering engine (unity, unreal engine)
sn	Scene name (identifier for the rendered scene)
vc	List of supported video codecs
ac	List of supported audio codecs
gm	GPU model
cm	CPU model
cpu	CPU utilization in %
gpu	GPU utilization in %
sysMem	Absolute CPU memory utilization in MB
gpuMem	Absolute GPU memory utilization in MB
hn	Hostname

Table 5-1 optional default reporting values

The following listing shows an example JSON as currently reported by a WebRTC client which consumes a video stream. As previously, mandatory fields are highlighted with bold letters.


```
[
 {
  "Body": {
   "rsi": "f900d288-0127-4a8d-9476-5e9ad77df5f2",
   "cid": "ab077924-c19b-4b0d-87c9-e5b7f9b7d140",
   "ts": "1697627086",
   "mt": "WebRTCMetricType",
   "ua": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/127.0.0.0 Safari/537.36",
   "rvc": "H264",
   "rvr": "1244x1080",
   "rtt": "27",
   "rvb": "49239",
   "minvb": "10000",
   "maxvb": "80000",
   "vfr": "60",
   "maxvfr": "60"
  }
 }
]
```


The “rsi”, “cid” and “ts” are mandatory values. “rsi” is the unique identifier of the rendering server the client is connected to. This value is used to match the client metric to the correct rendering instance. “cid” is the unique identifier of the client and “ts” is the timestamp for which data is reported.

Key	Value Description
rsi (mandatory)	Unique identifier of the reporting instance (as uuid)
cid (mandatory)	Unique identifier of the client (as uuid)

ts (mandatory)	timestamp as seconds since 1.1.1970 (unix epoch timestamp)
mt	Metric type (currently only WebRTCMetricType)
ua	user agent of the clients web engine
rvc	Video codec
rvr	Display resolution (height x width)
rtt	Round-trip time (in ms)
rvb	Video Bitrate
minvb	Minimum bitrate (specified initially)
maxvb	Maximum bitrate (specified initially)
vfr	Video framerate (number of images per second)
maxvfr	Maximum framerate (specified initially)

Table 5-2 optional default values for clients

Reporting to the Metrics Server

Using the message schemas above, metrics can be reported to the server. For this project, a REST interface is provided at the following URL:

https://metaverse-sand.servicebus.windows.net/smartedgehub/messages

The interface allows POST requests (the url does not show content in a web browser) that contain formatted messages in the request body. The correct format to send messages requires escaped quotation marks. The following shows a minimal example of a correctly formatted message that would result in the entry being added to the database:

```
[{"Body": "{\"ts\": 1697627086, \"cid\": \"cdc051e9-f19e-42ca-8ca0-1d69aba6b65b\", \"mt\":
\"WebRTCMetricType\", \"rtt\": 45, \"rsi\": "b923a654-967e-4bfe-9571-117f33c25e5e"}"}]
```

Additionally requests to the metrics server require the following HTTP headers:

- “Content-Type” with the value “application/vnd.microsoft.servicebus.json”
- “Authorization” with the value “SharedAccessSignature sr=metaverse-sand.servicebus.windows.net%2Fmetaversehub&sig=<SIG>se=<SE>&skn=MetaversePublish”
    - <SIG> and <SE> need to be replaced by the correct authorization values that are shared through secure channels and can be requested from the artifact owner

5.2.3.2.3 Experiment and Demonstration

The data reported to the metrics server is visualized in close to real-time on the SmartEdge dashboard available at the following URL:

http://metaverse.westeurope.cloudapp.azure.com:3000/d/adb16c20-6333-4f35-9a24-0276b57b95ef/uc1-remote-rendering?orgId=1&refresh=5s

The user name and password that is needed to access the dashboard can be requested from the Artifact Owner.

Figure 5-14. Screenshot of the SmartEdge Metrics Dashboard

Figure 5-14 shows an example Dashboard as currently used by the remote rendering artifact. For use case 1 a metrics provided is currently in development. For use case 2 a dashboard template was created based on the example metrics provided in D5.1.

5.2.3.3 Remote Rendering

The 3D Remote Rendering and Streaming framework allows accessing photorealistic interactive 3D environments on commodity devices. This is achieved by offloading graphical computation from the user device to a nearby GPU-enabled cloud or edge compute node.

5.2.3.3.1 Main components and Functionalities/Features

Rendering 3D environments elsewhere and using user-devices purely for playback opens up a wide variety of possibilities for industrial applications, such as utilizing cloud resources for running physics-based simulations while providing the possibility for real-time observation. 3D environments can be instantiated and stopped on cloud infrastructure on-demand, reducing cost. Coupled with 5G, ultra-low latency connections between the user, the graphical compute node, and physical systems enable real-time bidirectional interactions between virtual and physical systems, allowing remote work where it was not possible before.

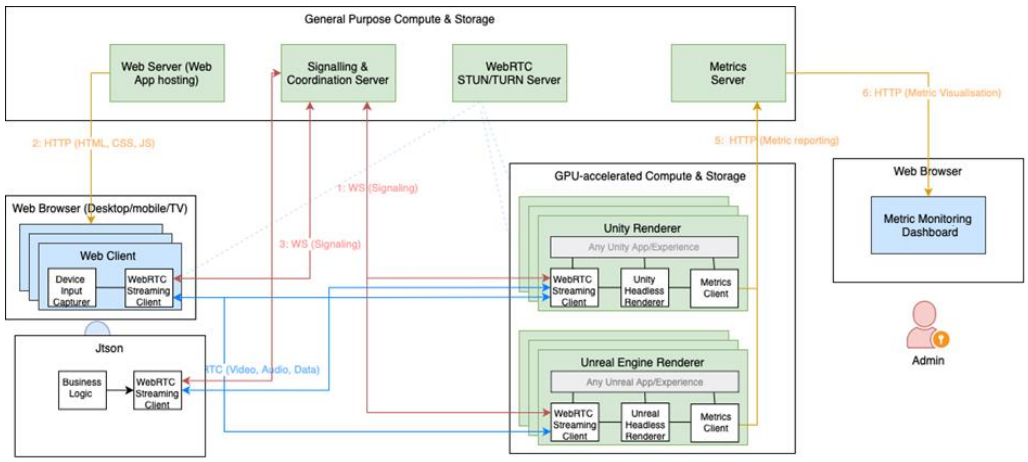


Figure 5-15. Remote Rendering Architecture

Figure 5-15 shows the components of the remote rendering system. The architecture consists of three distinct segments, which are clients (left), renderers (right), and a central coordinator responsible for connecting the two (top).

Both renderers and clients connect to the central coordination server upon being instantiated. Whenever a client wants to establish a connection to a renderer, the coordination server facilitates that the connection offers a channel for message exchange between the two components. After this connection negotiation, a direct connection is established between a client and a renderer. This connection is used to send control data from the client to the renderer and video data from the renderer to the client.

At their core, renderers are instances of either Unity or Unreal Engine that are run in headless mode, meaning that compared to the default execution, the visual output of the game engine is not shown on a connected device but rather encoded and forwarded to a different display location. This game engine instance expects to receive control data (i.e., data used to manipulate the 3D environment or the virtual camera within) from a remote location via a WebRTC DataChannel. Control data can be button presses on a keyboard, movement of a VR controller, mouse clicks or any other data, such as IoT data or remote-control data for physical devices, as long as it can be handled by the 3D application. Once received, the control data is read and mapped to an appropriate change in the 3D environment, which could be changing camera position, moving an avatar, or mirroring the state of an IoT device. Subsequently, the output of a virtual camera placed within the 3D environment is encoded as a video and sent to the client, where it can be viewed by the interacting user. Clients are connected to renderers in an n:1 relationship.

Before a renderer can establish a connection and exchange information with a client, the WebRTC signaling procedure needs to be performed. This process is facilitated by the Coordination Server, to which a WebSocket connection is established as soon as the renderer is instantiated. This connection is open throughout the entire lifecycle of the renderer, in case a new connection needs to be negotiated.

Renderers, including the game engine instance and all other supplementary components, such as encoding, message exchange, and configuration, are packaged as Docker containers, allowing streamlined deployment on any public or private compute infrastructure, as long as it utilizes NVIDIA GPUs.

In the remote rendering architecture, clients are responsible for forwarding control data to renderers and playing back the returned video stream. They are implemented using web technologies, resulting in high compatibility with consumer devices, such as computers, mobile phones, and VR devices. A client is connected to a renderer via a direct WebRTC connection, where control data is sent to the renderer via a DataChannel and video data is received via a media connection.

As is the case with each renderer, every client opens a WebSocket connection to the coordination server upon being instantiated. This connection is used to perform the WebRTC signaling procedure necessary to build a peer-to-peer communication to a renderer. This connection stays open throughout the client’s entire lifecycle, in case a new connection needs to be negotiated.

Upon being opened, the coordination server sends a list of connected renderers to the client's WebSocket connection. The client can then choose a renderer it wants to connect to from this list, which initialized the WebRTC signaling process.

The main task of the coordination server is facilitating the connection process between clients and renderers. As mentioned in previous sections, both clients and renderers first need to establish a WebSocket connection to the coordination server before they can build a direct WebRTC connection. Any messages necessary to negotiate an appropriate connection are sent to the signaling server by both parties and are then forwarded to the message target. In general, whenever a client wants to connect to a renderer, it sends an initial connection message via the coordination server. The message contains an identifier for the target renderer and additional information, such as a username. The coordination server maps the identifier to the WebSocket connection belonging to that renderer and forwards the message without change. Responses are handled the same way. The channel is then used to perform WebRTC signaling. Besides this, the coordination server includes a STUN and a TURN component, bundling all WebRTC related services in one place.

5.2.3.3.2 Component implementations

In the context of this project, a dedicated remote rendering system was set up and is accessible to the partners. It is based on the Unity Gaming Engine and can be accessed at https://fameverse.fokus.fraunhofer.de/smartedge/. The credentials were made accessible within the project and can be requested from the artifact owner.

Both clients and renderers continuously report metrics to the Metrics Server, which is described in detail in chapter Metric Reporting & Visualization. Metrics include application metrics, such as the renderer's rendering framerate, and CPU, GPU and memory utilization, but also network metrics, such as the streaming bitrate, jitter, and round-trip time. On the client side, the video framerate is reported to the metrics server. Visualizations then allow to identify deployment issues and find the optimal deployment for a specific use case.

5.2.3.3.3 Experiment and Demonstration

The default instance of the Remote Rendering presents a selection screen of the client to the user. Here a scene that is available on the server can be selected (for example a scene that relates to a specific use case). After clicking it, the user can decide on a username and press the appearing "Join" button, which will connect the client to the renderer, displaying the 3D environment (see Figure 5-16). Currently test scenes like the factory scene in the image as well as a Use Case 1 scene with virtual streets of the Helsinki test area of Use Case 2 are available.

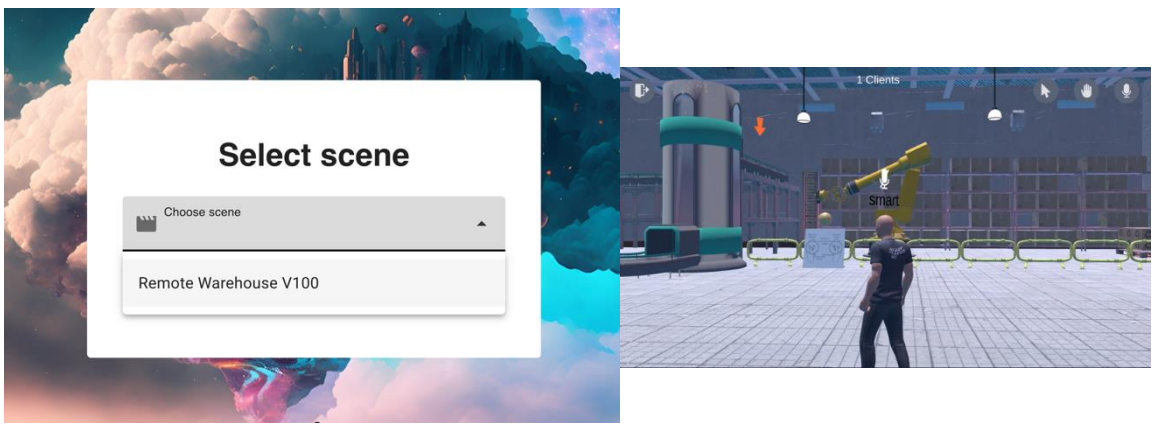


Figure 5-16. Renderer Selection (left) and Environment View (right)

For the next version of the artifact it is planned to add W3C Thing Description over MQTT support to control objects in the scenes the same way as physical objects would be controlled using W3C Thing Descriptions. The aim is to use virtual objects using recipes and the thing description directories as defined in work package 3.

The remote rendering pipeline can be applied to existing 3D environments made with Unity 3D. This requires supplementing an existing 3D application with all necessary libraries, including input mapping and control, metric reporting, video streaming, configuration, and coordination packages.

Also, the Remote Rendering architecture integrates the Metrics Reporting and Visualization Artifact and thus acts as example application for the usage of the Metric Reporting and Visualization.

5.2.3.4 Ego-vehicle Visualization (A5.4.4.4)

5.2.3.4.1 Main components and Functionalities/Features

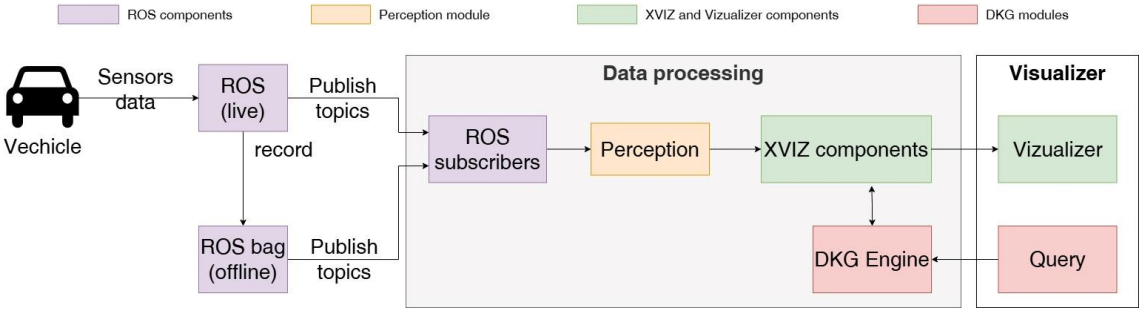


Figure 5-17. Visualizer Breakdown Architecture

The Visualizer is a tool developed to help visualize and interact with data from vehicles. The goal is to simplify how data from autonomous vehicles (AVs) is collected, processed, and understood. The tool provides a comprehensive way to view data from multiple sources, including lidars, cameras, and radars, all in one synchronized view.

The main aim of this system is to make it easier to monitor and analyze data from AVs. Working with data (both offline and live data) from multiple sources can be very challenging, especially when the data needs to be processed and visualized quickly and accurately. The Visualizer solves this problem by providing a system that can visualize this data in a clear and interactive way, improving tasks like debugging and testing AV systems.

To enhance portability and scalability, the entire system is containerized using Docker. This approach ensures consistency across different environments and simplifies deployment processes.

Figure 517 illustrate the architecture overview of the artifact Ego-vehicle Visualization. The Visualizer has two major components:

- The **Data Processing** component, which handles data collection, synchronization, and preparation.
- The **Visualizer** component, which allows users to view and interact with the data on a web interface.

System requirements:

- Operating System: Linux (recommended Ubuntu 20.04 LTS or later for compatibility and stability).
- Containerization: Docker installed and configured for managing containerized applications efficiently.
- GPU Support: Dedicated GPU(s) with CUDA support enabled to accelerate deep learning tasks.
- Graphical User Interface (GUI) Support: GUI enabled on the system to support visualization components and interactive interfaces.

5.2.3.4.2 Component implementations

The Data Processing component is the backend of the system. It manages and prepares data from various sensors across multiple vehicles. It's built on a customized version of the XVIZ server, originally developed by Uber, and uses Node.js for high performance and scalability.

ROS compatible: Data acquisition from various sensors, including cameras, lidar, radar, and other autonomous vehicle sensors are managed through the Robot Operating System (ROS). ROS operates on a publish-subscribe model where data is continuously published by different sensors and subscribed to by the Data Processing component. This mechanism allows for real-time synchronization of sensor data from multiple vehicles, which is essential for accurate visualization. Synchronizing data across multiple sources and vehicles is crucial to provide a clear and consistent picture of how different vehicles interact with their environment and with each other.

Perception Module for High-Level Data Extraction: A key feature of the Data Processing component is the Perception Module, which processes raw sensor data to extract higher-level information. This module utilizes deep learning frameworks such as TensorFlow, PyTorch or other open-source models, implemented in Python, to perform tasks like object detection and semantic segmentation. By processing camera and lidar data, the Perception Module generates 3D object information and environmental context, which is crucial for understanding the surroundings of the vehicle. The use of GPU acceleration and optimized neural network architecture ensures that the perception tasks are performed efficiently.

XVIZ components and Synchronization for Real-Time Multi-Data Handling: This part plays a crucial role in managing and processing data before visualization. The data from previous steps are then routed to the XVIZ components, which encode the information into a standardized, real-time streaming format optimized for autonomous systems visualization. The synchronizer within the XVIZ module ensures that data from different ROS bags (each potentially with its own timestamp and rate) is time-aligned, creating a coherent, unified data stream. This synchronized data is then passed to the visualizer, where it can be displayed in real time, allowing for accurate and interactive analysis of multi-source data in a simulated live environment.

Dynamic Knowledge Graph for Real-Time Data Correlation: In addition to this, a Dynamic Knowledge Graph is integrated into the Data Processing component. This knowledge graph is used to model and link data points in real-time, allowing for dynamic interactions. Users can query this knowledge graph using SPARQL, a specialized query language that enables specific questions to be asked about the relationships between data points from multiple vehicles and sensors. This feature provides a deeper level of insight, allowing users to explore correlations

between sensor readings, compare data across different vehicles, and better understand complex situations.

To ensure consistent and easy deployment across different environments, the entire Data Processing component is containerized using Docker

The Visualizer component provides an interactive interface for viewing and interacting with the data that has been collected and processed by the backend. This component is built using the open-source platform Streetscape.gl, which was also developed by Uber. The interface of the visualizer is designed to be simple yet powerful, helping users explore and analyse complex data from multiple vehicles. The visualizer is divided into two main sections that make interaction intuitive as shown in Figure 5-18.

Semantic Query Input for Customized Data Display: On the left side, there is a SPARQL query text box. This text box allows users to enter specific queries to control what is displayed on the visualization screen. Users can request data about a particular vehicle, filter the data based on sensor types, or analyze interactions between different vehicles. This interactive element makes the visualizer flexible and user-driven, allowing users to decide which parts of the data they want to focus on and how they want it displayed.

Dynamic Data Visualization Interface: On the right side of the interface, the visualizer displays the visual representation of the data. This area shows point clouds, detected objects, and other geospatial information gathered from various sensors across multiple vehicles. The visualizer can handle both real-time data, which allows users to monitor live data streams from different vehicles simultaneously, and offline data, where recorded information can be replayed for analysis. This dual capability is particularly helpful for both real-time monitoring and retrospective debugging or testing.

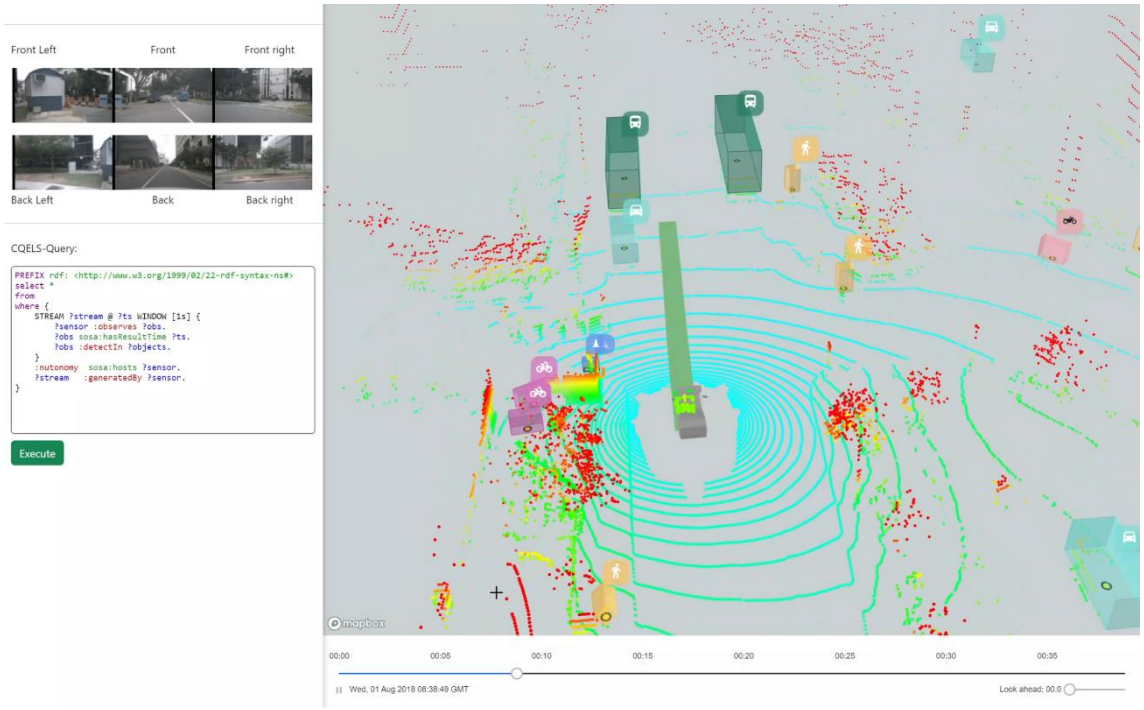


Figure 5-18. Demonstration Visualizer

Multi-Source Data Integration for AV System Analysis: The Visualizer component's ability to combine data from multiple sources and vehicles makes it an effective tool for engineers and developers working on AV systems. For example, engineers can replay a recorded scenario and analyze how each vehicle's sensors reacted to the same object or situation. The visualizer supports this analysis by allowing users to query and visualize data flexibly, which is useful for identifying problems and making improvements. It is particularly effective in urban settings where multiple AVs may be operating together, offering insights into vehicle interactions and helping improve the systems for real-world conditions.

5.2.3.4.3 Experiment and Demonstration

Hardware setup

NVIDIA Jetson Orin 64GB:

- Memory: 64GB LPDDR5 RAM with 204.8 GB/s bandwidth, supporting large datasets and multiple data streams simultaneously.
- CPU: 12-core ARM Cortex-A78AE, optimized for parallel processing and efficient task distribution across cores.
- GPU: NVIDIA Ampere architecture with 2048 CUDA cores and 64 Tensor Cores, delivering powerful AI acceleration for 3D rendering, real-time data analysis, and deep learning.

Software setup

- Docker platform.
- Operating System: The visualizer operates on Ubuntu 22.04, specifically tailored with NVIDIA's JetPack SDK
- ROS 2 Framework: The visualizer uses ROS 2 (Humble), which enables real-time data handling and reliable inter-process communication through the DDS middleware.
- Data Preparation: Live ROS data is sourced from a car/robot or ROS bag files containing standard ROS messages collected from the UC2 Cars setup, which operated across the city of Helsinki.

Due to the limited availability of vehicles, we used recordings from three locations with the same vehicle, adjusting the timestamps to a common starting point. This simulates multiple live-streaming vehicles, allowing us to visualize the data in our Visualizer.

Demonstration

This Figure 5-19 displays our visualizer actively monitoring data streams from three synchronized vehicle sources in real time. In this view, you can see multi-vehicle data seamlessly integrated, providing a live, comparative look at each vehicle's sensor outputs and location data.

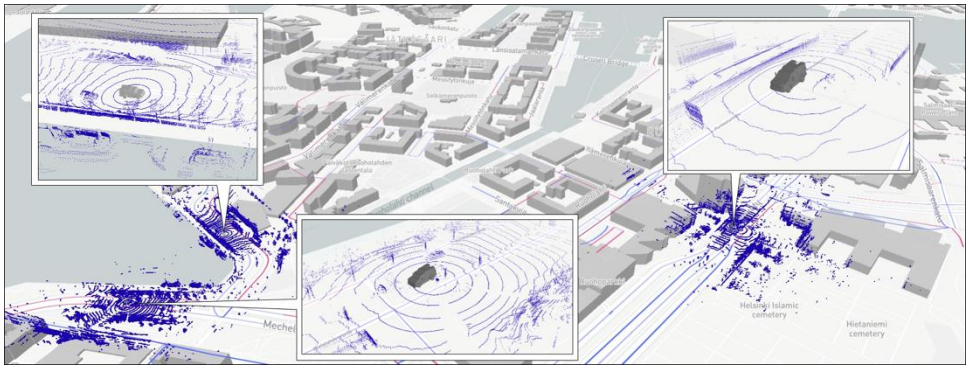


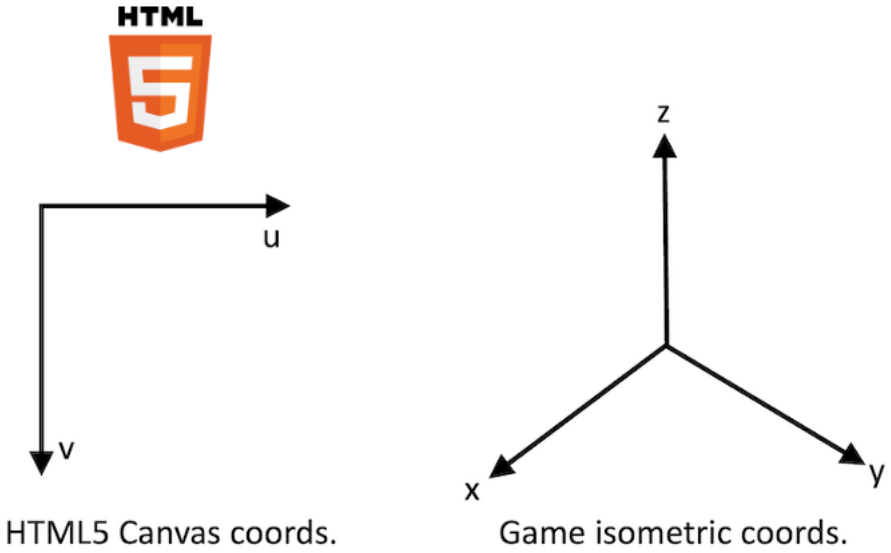
Figure 5-19. Simultaneously visualize data streaming from 3 vehicles

5.2.3.5 Swarm Visualization

This artifact focuses on the use of Web browsers to monitor and control swarms. It assumes that the current state of the swarm is available to a server running at the edge or in the cloud. The server hosts the web page and associated resources, as well as supporting Web Sockets for streaming the swarm state to the web page, along with messages for controlling the swarm. A particularly simple approach to implementing the server is to use NodeJS.

The web page uses HTML5's CANVAS element to render a visualization of the swarm state, either as a 2D, 2.5D or 3D presentation. We will focus on a 2.5D isometric presentation in the current description. The web page script uses the *window.requestAnimationFrame* method to render each frame at a rate dependent on the speed of the computer the browser is running on. The *CANVAS2D API* makes it straightforward to draw text and graphics on the screen area provided by the CANVAS element. Bitmapped image resources can be loaded from PNG resource files.

2.5D isometric presentations are often used in computer games such as *Sim City,* as well as in traditional Chinese scroll paintings. The approach is akin to viewing a scene from a very long distance so that parallax effects are negligible. This has the advantage that the size of an object is unchanged regardless of its position in the scene. To ensure a convincing composition, you need to render objects in sequence, so that objects closer to the viewer are drawn after objects further from the viewer. This involves mapping the object locations (x, y, z) to the HTML5 CANVAS coordinates (u, v).



$$u = 0.5 \cdot (y - x) \qquad x = v - u + z$$
$$v = 0.5 \cdot (x + y) - z \qquad y = u + v + z$$

Figure 5-20. Coordinate transformations

This is a slight simplification, as you will probably want to provide users with the means to pan and zoom scenes. The next step is to sort the scene components into the order in which they need to be rendered. A simple yet efficient algorithm considers the bounding box for each object in respect to the ground plane (x, y).

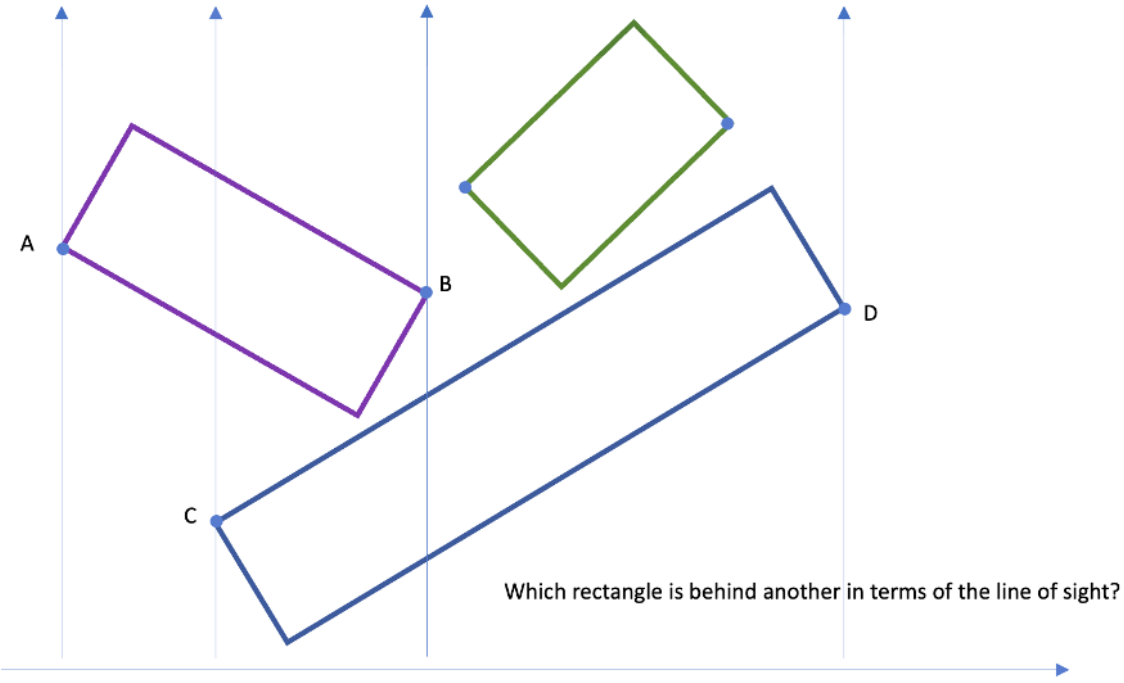


Figure 5-21. Rendering order

The scene components are associated with bitmap images showing them from multiple points of view corresponding to different rotations. The following example shows a truck rendered at 45-degree intervals. To speed loading the bitmaps for the different rotations are concatenated into a single PNG image resource, e.g.

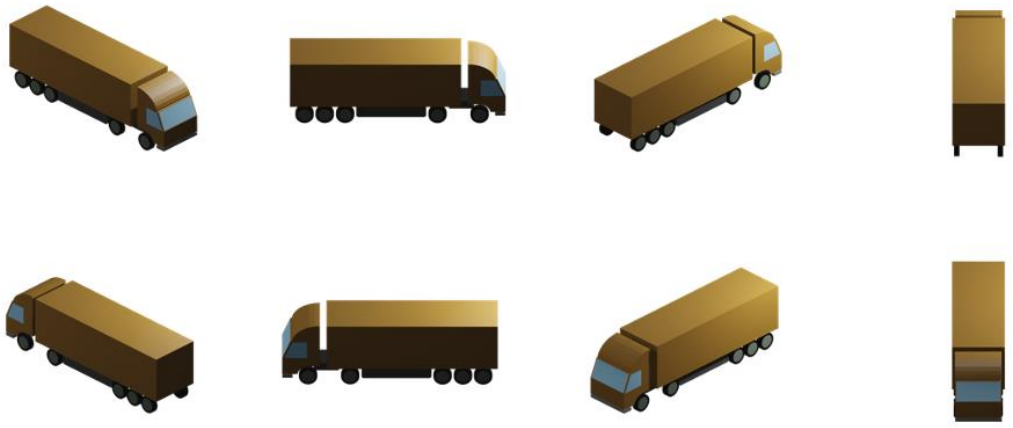


Figure 5-22. Object orientations

The images can be generated by imaging a 3D model created in an editing tool like Blender in conjunction with a Python script.

The swarm state is modeled as a set of objects with their (x, y, z) cartesian coordinates, scale and rotation. The object's velocity allows the web page to compute the current position and orientation at the exact time for rendering each frame.

Some complications arise when one object overlaps another, e.g. when the forks of a forklift truck are moved into the base of a pallet prior to moving some goods around in a warehouse. This is handled by decomposing the 3D model into smaller components that can be rendered in the correct order to provide the desired visualization.

Another technique involves offscreen compositing along with the means to apply image masks. This can be applied, for instance, to forklifts moving goods into and out of a truck positioned at a loading bay in a warehouse.

W3C/ERCIM can provide coding and design assistance to use-case owners, as the details will depend on the needs of each use-case. We have a proof-of-concept demonstrator *SimSwarm* that simulates a warehouse where a swarm of robot forklifts transport pallets of goods between incoming and outgoing trucks, using shelving as temporary storage as needed. The demonstrator allows users to hover the mouse pointer over the forklifts to see pop-ups showing information on the pallet's contents and the job number.

- *SimSwarm*: https://www.w3.org/Data/demos/chunks/warehouse/

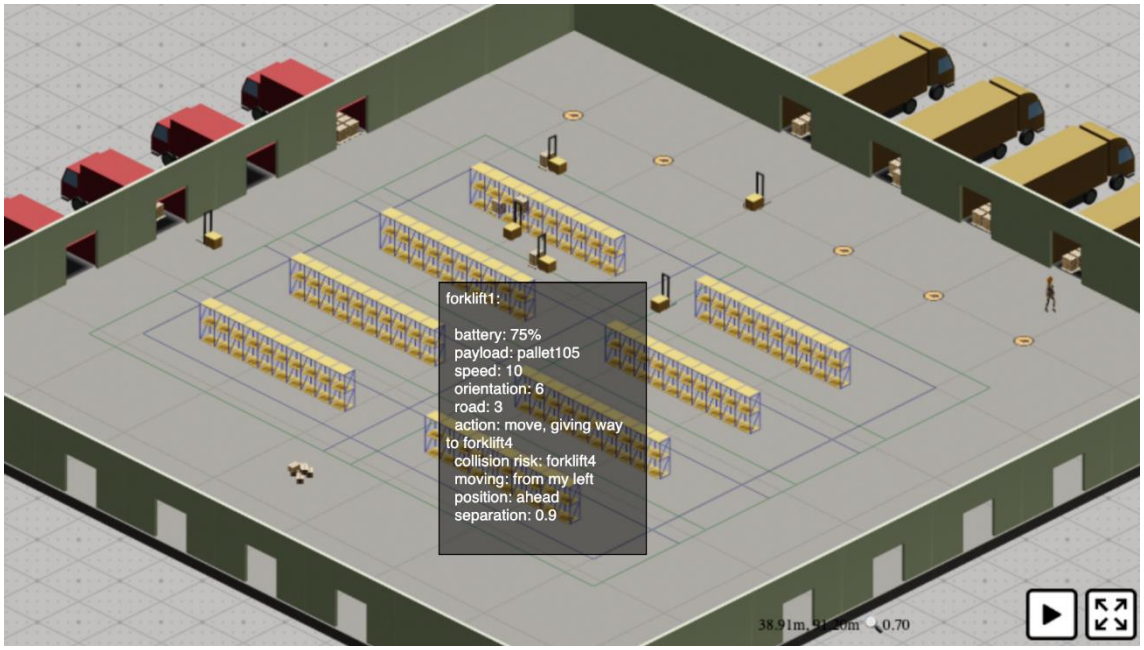


Figure 55-23. SimSwarm smart warehouse

The demonstrator routes forklifts along a predefined grid close to the shelves and point to point elsewhere. The forklifts need to be sent to recharge stations when their battery levels demand. The approach uses collision avoidance rules from the perspective of each forklift, along with freezing when a human is nearby (see the above example). The truck manifests are generated stochastically from statistical distributions over pallet types. Job control allocates forklifts to jobs taking into account their current battery levels.

W3C/ERCIM looks forward to collaborating with use case owners to apply web-based monitoring and control to the needs of their specific use cases. We will re-use the algorithms developed for *SimSwarm* and offer help with the web page scripts and associated resources6.

6 CONCLUSIONS

This deliverable D5.2 presents the first implementation documentation for goals in Objective 5 of the SmartEdge project: *Low-code Programming Tools for Edge Intelligence* providing. We divided this component into four main modules: (1) semantic driven multimodal stream fusion for Edge devices; (2) swarm elasticity via Edge-Cloud Interplay; (3) adaptive coordination and optimization; and (4) cross-layer toolchain for Device-Edge-Cloud Continuum. In each module, we presented the main (sub-)components and their functionalities, following up with the detailed implementations. Additionally, we also presented initial experimental results and/or demonstrations.

The first implementation details were aligned with our design presented in D5.1 and carefully mapped to corresponding artifacts in WP6. However, this deliverable does not cover artifacts, which are planned to be released in the future, i.e. after the milestone of this deliverable, as detailed in Section 1.

REFERENCES

[Aberer01] K. Aberer, "P-Grid: A self-organizing access structure for P2P information systems", in Cooperative Information Systems: 9th International Conference, CoopIS 2001 Trento, Italy, September 5–7, 2001 Proceedings, vol. 9, Springer Berlin Heidelberg, pp. 179-194.

[ACT-R] "ACT-R Research Group, CMU", last modified 2023, URL: http://act-r.psy.cmu.edu/

[Anh18] A. Le-Tuan, C. Hayes, M. Wylot, and D. Le-Phuoc. Rdf4led: An rdf engine for lightweight edge devices. In IOT '18, 2018.

[Anh19] A. Le-Tuan, D. Hingu, M. Hauswirth, and D. Le-Phuoc. Incorporating blockchain into rdf store at the lightweight edge devices. In Semantic '19, 2019

[Anh21] Le Tuan, Anh, et al. "VisionKG: Towards A Unified Vision Knowledge Graph." ISWC (Posters/Demos/Industry). 2021.

[Auer17] Auer, Sören, et al. "Dbpedia: A nucleus for a web of open data." international semantic web conference. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

[Balazinska04] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In NSDI'04, 2004

[Biffi20] Biffi, Leonardo Josoé, et al. "ATSS deep learning-based approach to detect apple fruits." Remote Sensing 13.1 (2020): 54.

[Bowden22] Bowden, David, and Diarmuid Grimes. "Intelligent Image Compression Using Traffic Scene Analysis." Irish Conference on Artificial Intelligence and Cognitive Science. Cham: Springer Nature Switzerland, 2022.

[Carion20] Carion, Nicolas, et al. "End-to-end object detection with transformers." European conference on computer vision. Cham: Springer International Publishing, 2020.

[Chen21] Chen, Qiang, et al. "You only look one-level feature." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2021.

[Chen23] Chen, Qiang, et al. "Group detr: Fast detr training with group-wise one-to-many assignment." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2023.

[Chen19] Chen, Tianshui, et al. "Knowledge-embedded routing network for scene graph generation." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019.

[Chunks and Rules] "Chunks and Rules Specification", W3C Cognitive AI Community Group, last modified 04 January 2024, URL: https://w3c.github.io/cogai/chunks-and-rules.html

[Cong23] Cong, Yuren, Michael Ying Yang, and Bodo Rosenhahn. "Reltr: Relation transformer for scene graph generation." IEEE Transactions on Pattern Analysis and Machine Intelligence (2023).

[Cudre-Mauroux13] Philippe Cudré-Mauroux, Iliya Enchev, Sever Fundatureanu, Paul Groth, Albert Haque, Andreas Harth, Felix Leif Keppmann, Daniel P. Miranker, Juan F. Sequeda, Marcin Wylot: NoSQL Databases for RDF: An Empirical Evaluation. ISWC (2) 2013: 310-325.

[Cui22] Cui, Yu, and Moshiur Farazi. "VReBERT: a simple and flexible transformer for visual relationship detection." 2022 26th International Conference on Pattern Recognition (ICPR). IEEE, 2022.

[Dai16] Dai, Jifeng, et al. "R-fcn: Object detection via region-based fully convolutional networks." Advances in neural information processing systems 29 (2016).

[Danh11] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In ISWC'11, pages 370–388, 2011.

[Danh15] D. Le-Phuoc, M. Dao-Tran, C. Le Van, A. Le Tuan, T. T. N. Manh Nguyen Duc, and M. Hauswirth. Platform-agnostic execution framework towards rdf stream processing. In RDF Stream Processing Workshop at ESWC2015, 2015.

[Danh17]D. Le-Phuoc. Operator-aware approach for boosting performance in RDF stream processing. J. Web Sem., 42:38–54, 2017.

[Danh18] D. Le-Phuoc. Adaptive optimisation for continuous multi-way joins over rdf streams. In Companion Proceedings of the The Web Conference 2018, WWW '18, pages 1857–1865, 2018.

[Danh21] Le-Phuoc, Danh, Thomas Eiter, and Anh Le-Tuan. "A scalable reasoning and learning approach for neural-symbolic stream fusion." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 35. No. 6. 2021.

[Dell17]D. Dell'Aglio, D. L. Phuoc, A. Le-Tuan, M. I. Ali, and J.-P. Calbimonte. On a web of data streams. In DeSemWeb@ISWC, 2017.

[Denny14] Vrandečić, Denny, and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase." Communications of the ACM 57.10 (2014): 78-85.

[Dias19] Vinicius Dias, Carlos HC Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A general-purpose graph pattern mining system. In Proceedings of the 2019 International Conference on Management of Data. 1357–1374.

[Dominik23] Kreuzberger, Dominik, Niklas Kühl, and Sebastian Hirschl. "Machine learning operations (mlops): Overview, definition, and architecture." IEEE Access (2023).

[Duc21] Manh Nguyen Duc, Anh Lê Tuán, Manfred Hauswirth, Danh Le Phuoc: Towards autonomous semantic stream fusion for distributed video streams. DEBS 2021: 172-175.

[Fumero19] Juan Fumero, et al. "Dynamic application reconfiguration on heterogeneous hardware". VEE 2019: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution (2019): 165–178

[Girshick15] Girshick, Ross. "Fast r-cnn." Proceedings of the IEEE international conference on computer vision. 2015.

[Grubenmann18] T. Grubenmann, A. Bernstein, D. Moor, and S. Seuken. Financing the web of data with delayed-answer auctions. In WWW '18, 2018

[Haller19] A. Haller, K. Janowicz, S. J. D. Cox, M. Lefran¸cois, K. Taylor, D. Le-Phuoc, J. Lieberman, R. Garc´ıa-Castro, R. Atkinson, and C. Stadler. The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. Semantic Web, 10(1):9–32, 2019.

[Hong21] Hong, Jinyung, and Theodore P. Pavlic. "Representing Prior Knowledge Using Randomly, Weighted Feature Networks for Visual Relationship Detection." arXiv preprint arXiv:2111.10686 (2021).

[Hornung13] A. Hornung, K.M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees" in Autonomous Robots, 2013; DOI: 10.1007/s10514-012-9321-0.

[Hussein23] Rana Hussein, Alberto Lerner, André Ryser, Lucas David Bürgi, Albert Blarer, Philippe Cudré-Mauroux: GraphINC: Graph Pattern Mining at Network Speed. Proc. ACM Manag. Data 1(2): 184:1-184:28 (2023).

[ISD24] [dataset] https://www.ncdc.noaa.gov/isd

[Jia23] Jia, Ding, et al. "Detrs with hybrid matching." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023.

[Jiang23] Jiang, Bowen, and Camillo Taylor. "Hierarchical Relationships: A New Perspective to Enhance Scene Graph Generation." NeurIPS 2023 Workshop: New Frontiers in Graph Learning. 2023.

[Jicheng23] Yuan, Jicheng, et al. "VisionKG: Unleashing the Power of Visual Datasets via Knowledge Graph." arXiv preprint arXiv:2309.13610 (2023).

[Jung21] Jaehoon Jung, et al. “SnuRHAC:ARuntimeforHeterogeneousAcceleratorClusters withCUDAUnifiedMemory”. HPDC '21: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (2021): Pages 107–120

[Kien21] Kien-Tran, Trung, et al. "Fantastic Data and How to Query Them." NeurIPS (Workshop on Data-Centric AI}, 2021.

[Kirillov23] Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., … Girshick, R. (2023). Segment Anything. arXiv:2304. 02643.

[Kundu23] Kundu, Sanjoy, and Sathyanarayanan N. Aakur. "IS-GGT: Iterative Scene Graph Generation With Generative Transformers." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023.

[Kurt08] Bollacker, Kurt, et al. "Freebase: a collaboratively created graph database for structuring human knowledge." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. 2008.

[Lee24] Sangjin Lee, Alberto Lerner, Philippe Bonnet, Philippe Cudré-Mauroux: Database Kernels: Seamless Integration of Database Systems and Fast Storage via CXL. CIDR 2024.

[Lerner19] Alberto Lerner, Rana Hussein, Philippe Cudré-Mauroux: The Case for Network Accelerated Query Processing. CIDR 2019.

[Lin17] Lin, Tsung-Yi, et al. "Focal loss for dense object detection." Proceedings of the IEEE international conference on computer vision. 2017.

[Liu22] Liu, Shilong, et al. "Dab-detr: Dynamic anchor boxes are better queries for detr." arXiv preprint arXiv:2201.12329 (2022).

[Zheng24]Zheng, Changgang, et al. "Planter: Rapid prototyping of in-network machine learning inference." *ACM SIGCOMM Computer Communication Review* 54.1 (2024): 2-21.

[Liu23] Liu, Shilong, et al. "Grounding dino: Marrying dino with grounded pre-training for open-set object detection." arXiv preprint arXiv:2303.05499 (2023).

[Manh19] Nguyen-Duc, M., Le-Tuan, A., Calbimonte, J.P., Hauswirth, M., Le-Phuoc, D.: Autonomous rdf stream processing for iot edge devices. In: JIST 2019, pp. 304–319. Springer, Cham (2019)

[Manh22] Nguyen-Duc, Manh, et al. "SemRob: Towards Semantic Stream Reasoning for Robotic Operating Systems." arXiv preprint arXiv:2201.11625 (2022).

[Mar23] Hirzel, Martin. Low-Code Programming Models. Commun. ACM 66(10): 76-85 (2023)

[Munshi09] A. Munshi, The OpenCL specification, 1, IEEE, Stanford, CA, USA, 2009.

[Nozal20] Raul Nozal, et al. “EngineCL: Usability and Performance in Heterogeneous Computing”. Future Generation Computer Systems 107 (2020): Pages 522-537

[Naphade19] Naphade, Milind, Zheng Tang, Ming-Ching Chang, David C. Anastasiu, Anuj Sharma, Rama Chellappa, Shuo Wang et al. "The 2019 AI City Challenge." In CVPR workshops, vol. 8, p. 2. 2019.

[Onos24] https://opennetworking.org/onos/

[Pang19] Pang, Jiangmiao, et al. "Libra r-cnn: Towards balanced learning for object detection." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019.

[Park24] Kibin Park, Alberto Lerner, Sangjin Lee, Philippe Bonnet, Yong Ho Song, Philippe Cudré-Mauroux, and Jungwook Choi. BABOL: A Software-Defined NAND Flash Controller. In 57th IEEE/ACM International Symposium on Microarchitecture (MICRO 2024).

[Ranftl20] Ranftl, René, et al. "Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer." IEEE transactions on pattern analysis and machine intelligence 44.3 (2020): 1623-1637.

[Redmon15] Redmon, J., Divvala, S. K., Girshick, R. B., & Farhadi, A. (2015). You Only Look Once: Unified, Real-Time Object Detection. CoRR, abs/1506.02640. Retrieved from http://arxiv.org/abs/1506.02640

[Ren15] Ren, Shaoqing, et al. "Faster r-cnn: Towards real-time object detection with region proposal networks." Advances in neural information processing systems 28 (2015).

[Ren24] Ren, T., Liu, S., Zeng, A., Lin, J., Li, K., Cao, H., … Zhang, L. (2024). Grounded SAM: Assembling Open-World Models for Diverse Visual Tasks. arXiv [Cs.CV]. Retrieved from http://arxiv.org/abs/2401.14159

[Rezatofighi19] Rezatofighi, Hamid, et al. "Generalized intersection over union: A metric and a loss for bounding box regression." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019.

[Robot Chunks Demo] “Robot demo using Chunks & Rules”, last modified 02 Jul 2020, W3C Cognitive AI Community Group, URL: https://www.w3.org/Data/demos/chunks/robot/

[Ryser22] André Ryser, Alberto Lerner, Alex Forencich, Philippe Cudré-Mauroux: D-RDMA: Bringing Zero-Copy RDMA to Database Systems. CIDR 2022.

[Schneider22] Patrik Schneider, Daniel Alvarez-Coello, Anh Le-Tuan, Manh Nguyen Duc, Danh Le Phuoc: Stream Reasoning Playground. ESWC 2022: 406-424.

[Shaoqing15] Ren, Shaoqing, et al. "Faster r-cnn: Towards real-time object detection with region proposal networks." Advances in neural information processing systems 28 (2015).

[Speer17] Speer, Robyn, Joshua Chin, and Catherine Havasi. "Conceptnet 5.5: An open multilingual graph of general knowledge." Proceedings of the AAAI conference on artificial intelligence. Vol. 31. No. 1. 2017.

[Tang19] Tang, Kaihua, et al. "Learning to compose dynamic tree structures for visual contexts." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019.

[Tian19] Tian, Zhi, et al. "Fcos: Fully convolutional one-stage object detection." Proceedings of the IEEE/CVF international conference on computer vision. 2019.

[Tommasini19] R. Tommasini, D. Calvaresi, and J.-P. Calbimonte. Stream reasoning agents: Bluesky ideas track. In AAMAS, pages 1664–1680, 2019

[VanAssche21] Van Assche, Dylan, et al. “Leveraging Web of Things W3C recommendations for knowledge graphs generation”, Proceedings of the 21st International Conference on Web Engineering, 2021.

[Vrandečić14] Vrandečić, Denny, and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase." Communications of the ACM 57.10 (2014): 78-85.

[W3C24] “Web of Things” [Online], https://www.w3.org/WoT/documentation/, Retrieved 02/2024

[Xu17] Xu, Danfei, et al. "Scene graph generation by iterative message passing." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[Zellers18] Zellers, Rowan, et al. "Neural motifs: Scene graph parsing with global context." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

[Zhang22] Zhang, Hao, et al. "Dino: Detr with improved denoising anchor boxes for end-to-end object detection." arXiv preprint arXiv:2203.03605 (2022).

[ZhangJi18] Zhang, Ji, et al. "An interpretable model for scene graph generation." arXiv preprint arXiv:1811.09543 (2018).

[Zheng23] Zheng, Changgang et al. "DINC: toward distributed in-network computing", ACM CoNEXT and Proceedings of the ACM on Networking (PACMNET), December 2023.

[Zong23] Zong, Zhuofan, Guanglu Song, and Yu Liu. "Detrs with collaborative hybrid assignments training." Proceedings of the IEEE/CVF international conference on computer vision. 2023.

[Lin14] Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., ... & Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13* (pp. 740-755). Springer International Publishing.

[Chen21] Chen, Q., Wang, W., Huang, K., De, S., Coenen, F.: Multi-modal generative adversarial networks for traffic event detection in smart cities. Expert Systems with Applications 177, 114,939 (2021).

[Gao20] Gao, J., Li, P., Chen, Z., Zhang, J.: A survey on deep learning for multimodal data fusion. Neural Computation 32(5), 829–864 (2020)

[Khadanga19] Khadanga, S., Aggarwal, K., Joty, S., Srivastava, J.: Using clinical notes with time series data for icu man- agement. In: Proceedings of the 2019 Conference on Em- pirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pp. 6432–6437 (2019)

[Yang 21] Yang, H., Kuang, L., Xia, F.: Multimodal temporal- clinical note network for mortality prediction. Journal of Biomedical Semantics 12(1), 1–14 (2021)

[Wang24] C.-Y. Wang, I.-H. Yeh, H.-Y. M. Liao, YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information, 2024. doi:10.48550/arXiv.2402.13616. arXiv:2402.13616.

[Wojke17] Wojke N, Bewley A, Paulus D. Simple online and realtime tracking with a deep association metric. In2017 IEEE international conference on image processing (ICIP) 2017 Sep 17 (pp. 3645-3649). IEEE.

[Zheng24] Zheng, Changgang, et al. "Planter: Rapid prototyping of in-network machine learning inference." ACM SIGCOMM Computer Communication Review 54.1 (2024): 2-21.

[Shafarenko24] Shafarenko, A. “Winternitz stack protocols for embedded systems and IoT.” Cybersecurity 7, 34 (2024).