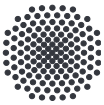

University of Stuttgart
Institute of Industrial Automation and
Software Engineering

Flexible Sensor Data Abstraction in SDVs with Middleware-Driven Approach

Research Project FA-3810

Submitted at the University of Stuttgart by
Aniket Barve

MSc. Electrical Engineering

Examiner: Prof. Dr.-Ing. Michael Weyrich

Supervisor: Akshay Narla, M.Sc.

27.06.2025

Table of Contents

Table of Figures

Table of Tables

Table of Abbreviations

ADAS	**A**dvanced **D**river-**A**ssistance **S**ystems
ASIL	**A**utomotive **S**afety **I**ntegrity **L**evel
AUTOSAR	**A**utomotive **O**pen **S**ystem **A**rchitecture
CASE	**C**onnected, **A**utonomous, **S**hared, and **E**lectric
COVESA	**C**onnected **V**ehicle **S**ystems **A**lliance
DDS	**D**ata **D**istribution **S**ervice
E/E	**E**lectrical/**E**lectronic
ECU	**E**lectronic **C**ontrol **U**nit
FAIR	**F**indable, **A**ccessible, **I**nteroperable, and **R**eusable
HPC	**H**igh **P**erformance **C**omputing
HWAL	**H**ard**w**are **A**bstraction **L**ayer
IoT	**I**nternet **o**f **T**hings
IVN	**I**n-**V**ehicle **N**etwork
LiDAR	**L**ight **D**etection **a**nd **R**anging
MCS	**M**ixed **C**riticality **S**ystems
ML	**M**achine **L**earning
MSOA	**M**ulti-layered **S**ervice-**O**riented **A**rchitecture
OMG	**O**bject **M**anagement **G**roup
OS	**O**perating **S**ystem
OSI	**O**pen **S**imulation **I**nterface
OTA	**O**ver-**T**he-**A**ir
QoS	**Q**uality of **S**ervice
QUIC	**Q**uick **U**DP **I**nternet **C**onnections
RBAC	**R**ole **B**ased **A**ccess **C**ontrol
RTPS	**R**eal-**T**ime **P**ublish-**S**ubscribe

SAL	**S**ignal **A**bstraction **L**ayer
SDV	**S**oftware **D**efined **V**ehicle
V2X	**V**ehicle **t**o **E**verything
VISS	**V**ehicle **I**nformation **S**ervice **S**pecification
VSS	**V**ehicle **S**ignal **S**pecification
WCET	**W**orst **C**ase **E**xecution **T**ime

Glossary

ADAS	Electronic systems that assist the vehicle driver with driving and parking functions to improve car safety and the overall driving experience.
Base64 Encoding	A binary-to-text encoding scheme used to represent binary data in an ASCII string format for transmission over text-based protocols.
Cybersecurity	The practice of protecting systems, networks, and data from digital attacks, ensuring confidentiality, integrity, and availability of information.
gRPC	A high-performance, open-source universal RPC framework that uses HTTP/2 for transport and protocol buffers as the interface description language.
HWAL	A software layer that allows the upper layers of the software stack to access hardware resources in a consistent manner, regardless of hardware differences.
Heterogeneity	The use of different types of hardware, software, or communication protocols within the same system, requiring standardization for smooth integration.
Interoperability	The ability of different systems, devices, or software components to communicate, exchange, and interpret data accurately and consistently across heterogeneous environments.
LiDAR	A remote sensing technology that uses laser pulses to measure distances to surrounding objects and generate high-resolution 3D maps of the environment.
Middleware	Software that connects different components or applications, enabling communication and data management in distributed systems.
Mixed-Criticality	A system setup where tasks with different safety or performance levels run together on shared hardware, with guaranteed isolation and reliability.

OTA	A method of distributing new software, configuration settings, and updates to devices remotely.
QoS	A set of technologies used to manage data traffic to reduce packet loss, latency, and jitter on the network.
RBAC	A security approach that restricts system access to authorized users based on their role within an organization.
Raspberry Pi	A small, affordable single-board computer used for prototyping and educational projects.
Resilience	The ability of a system to continue operating correctly and recover quickly in the face of faults, failures, or unexpected conditions.
RTPS	The wire protocol used by DDS to enable communication between publishers and subscribers in real-time systems.
SAL	A conceptual layer that maps low-level sensor data into semantically meaningful signals using a standardized format.
Scalability	The capability of a system to handle an increasing amount of work, data, or users without compromising performance or requiring fundamental architectural changes.
Validation	The process of evaluating a system or component to ensure it meets defined requirements and performs intended functions correctly and reliably under specified conditions.
YOLO	A real-time object detection algorithm that detects and classifies multiple objects within an image using a single neural network.

Abstract

Modern Software-Defined Vehicles (SDVs) rely heavily on sensor data for real-time decision-making and perception tasks. However, the integration of heterogeneous sensors and communication protocols introduces challenges in semantic consistency, scalability, and data abstraction. Traditional approaches often result in tightly coupled architectures with hardware-specific signal handling, limiting flexibility across platforms. This project addresses the problem of unifying camera sensor data abstraction across different computing platforms, using open-source middleware and standardized semantic layers to align with the evolving needs of SDV architectures.

The proposed solution implements a complete prototype consisting of two independent pipelines—one deployed on a Raspberry Pi and the other on a native Linux laptop. Each pipeline captures raw image data using connected cameras, transmits it using Fast-DDS or Zenoh middleware, and converts it into standardized Vehicle Signal Specification (VSS) signals using Kuksa Databroker via Python-based gRPC clients. Object detection was performed using YOLOv5n and YOLOv8n models, and the detection results were also mapped to VSS signals. The entire system includes real-time signal publishing, static VSS tree integration, and signal synchronization across distributed components. Key implementation challenges, such as C++ build and linking errors, signal parsing issues, and IP mismatches in virtualized environments, were addressed during the development.

This middleware-based architecture demonstrates a modular and scalable framework for semantic signal abstraction in SDVs. The use of VSS and Kuksa enables hardware-independent data representation, improving system interoperability and future integration with ADAS functionalities. The prototype provides a functional baseline for extending support toward dynamic signal updates, secure TLS communication, and integration with more complex driving stacks.

Key Words: Software-Defined Vehicle, Vehicle Signal Specification, Kuksa Databroker, Fast-DDS, Zenoh, gRPC, VSS integration

1 Introduction

This section introduces the concept of Software-Defined Vehicles (SDVs) and outlines their architectural structure, benefits, and challenges. It explains how SDVs enable flexible software deployment while also presenting the technical and integration hurdles associated with their implementation.

1.1 Software Defined Vehicles (SDVs)

1.1.1 Concept of SDVs

The automotive industry is changing from hardware-centric vehicles to SDVs with four major innovations - electrification, automation, shared mobility, and connectivity. The customers expect the vehicles to have features like smartphones including constant software updates and enhanced digital experiences. The SDVs provide the possibility of wireless upgradeability and flexible functionalities, which enable vehicles to evolve over time rather than as a finished product following production [1].

As with the rise of smartphones, software updates became routine; similarly, traditional cars become outdated as new models with better hardware-centric features with costly modifications for upgrades are released. SDVs address this issue by making it possible to update through Over-The-Air (OTA), ensuring that vehicles remain up-to-date without any need of hardware changes. Also, continuous development in the software with Agile methodology enables continuous software improvements, which enhances functionality, performance, and user experience throughout the vehicle’s lifecycle [2].

For conventional vehicles, hardware is the sufficient and necessary condition, and software is an auxiliary factor. But for future vehicles, hardware is merely the basic necessary condition, and software is the sufficient condition that influences the user experience. As contrasted with data-defined vehicles, SDVs more accurately convey the shifting relationship between hardware and software and the evolution trajectory of automotive products [3].

1.1.2 SDV architecture and a classification approach

The shift from domain-based architectures to SDV architecture is an important moment for the automotive industry. In the past, vehicles relied on isolated control domains i.e., engine management and infotainment, leading to inefficiencies and integration problems. These disconnected systems limited responsiveness to critical driving conditions by affecting safety and performance.

SDVs provide a shared software platform that allows easy real-time data exchange between components. This improves vehicle adjustment, real-time data exchange and allows predictive maintenance, and aids decision-making and in turn leading to more secure and effective driving. Furthermore, the transition to SDVs encourages fast development, which leads manufacturers to utilize AI and machine learning, making vehicles intelligent, self-adjusting systems [4].

SDVs can be classified into 5 different development levels:

1. **SDV Level 0:** Software alone assists without intervening in the functioning of the vehicle. Functions such as Park Distance Control or Adaptive Cruise Control have their own hardware and remain unchanged.
2. **SDV Level 1:** The cars are connected with an external IT system and offer smartphone-integrated infotainment and remote control via mobile apps. Software remains the same, but connectivity enhances the user experience.
3. **SDV Level 2:** Software is updated over the air (OTA), allowing for bug fixes and security patches after the car is delivered. Hardware does not change, but this level greatly improves safety and maintenance.
4. **SDV Level 3:** The vehicle can be given software updates even after delivery. Hardware remains the same, but software updates give new functionality, reflecting the direction of a more flexible SDV.
5. **SDV Level 4:** Hardware and software are decoupled so that software updates can be done between various car models. This requires a modular architecture and shared tools for easy updates.
6. **SDV Level 5:** Cars are now open platforms where third-party developers can create and install applications. Just like smartphones, this level brings in new ways of doing business but needs strong security, app separation, and revenue-generating methods [5].

1.1.3 Benefits of SDVs

SDVs are set to bring huge transformation in the automotive sector, and the technologies facilitating this innovation have many major advantages as mentioned below:

1. **Enhanced Performance and Efficiency**: Engine parameters can be monitored continuously through software. Also, it can be used to improve fuel or battery pack efficiency and optimise driving dynamics.
2. **Improved safety**: Serious accidents can be avoided with the assistance of Anticollision systems and driver assistance, which reacts quickly in the critical situations.
3. **Evolving capabilities:** Software updates related to the bug fixes, new features and performance enhancement can be downloaded and installed OTA.

4. **Personalised experience:** Software allows personalization through apps, user profiles, software-defined settings, and integration with cloud services for tailored navigation, entertainment, and diagnostics.
5. **Predictive maintenance**: SDVs can anticipate repairs by monitoring vehicle health in real-time before they become critical issues, saving time and costs for owners.
6. **Increased connectivity:** SDVs support real-time communication with other vehicles, infrastructures, and cloud services. Also, it enhances the safety with the help of Vehicle-to-everything (V2X) technology [6], [7].

1.1.4 Challenges faced by SDVs

Transitioning to SDVs involves complex planning on the part of various stakeholders in the industry. Though SDVs embody landmark developments in car, numerous challenges are to be addressed for a safe and trustworthy tomorrow.

1. **Software complexity:** The quantity of code required to manage any SDV is steep and belies a risk of bugs and vulnerabilities.
2. **Cybersecurity Risks:** Well-designed cybersecurity frameworks should protect vehicles against cyber-attacks.
3. **Data privacy concerns:** In effect, data privacy concerns emerge with the amount of data being collected by the SDVs. Proper regulations along with complete data security practices are required to gain user's trust.
4. **Modular hardware and software:** The present vehicle hardware operates strictly in association with its enabling software. A modular approach would allow software applications to be constructed more independently.
5. **Technical expertise gap:** The automobile industry must attract and harness a generation of talent with combined expertise in software development, cybersecurity, and data management [6][7].

2 Basis

This section provides the foundational background for understanding sensor data abstraction in the automotive domain. It introduces the need for abstraction, reviews existing standards and technologies such as Vehicle Signal Specification (VSS), Data Distribution Service (DDS), and Eclipse Kuksa, and highlights their respective roles and capabilities. Additionally, it examines the challenges of mixed-criticality systems and outlines the middleware requirements necessary for real-time, secure, and scalable data abstraction in SDVs.

2.1 Sensor data abstraction in automotive industry

2.1.1 What is sensor data abstraction?

Sensor data abstraction is the conversion of raw, heterogeneous sensor data (e.g., LiDAR point clouds, camera images, radar signals) into standardized, machine-readable representations. It serves as an interface between physical sensors and applications, abstracting hardware-specific details while retaining contextual and semantic meaning. For sensor data abstraction, we need to understand the implicit properties of the sensors which are used and their setup as it has strong influence on the final resulting data [8].

Sensor data abstraction is the process of converting the raw data from one or more sensors into a common standardized format that is suitable for other applications [8].

In modern vehicles, processing the sensor data is very helpful to derive the environment information for safe driving. This processing includes:

1. To consider how to process the architectures on which data is being collected from sensors and, at what level of the processing chain
2. External perception capabilities such as object detection, classification, range and velocity estimation, which are processing algorithms for external perception capabilities like occupancy detection and occupant alertness
3. Using physics/model/data-driven methods in data processing
4. Performance metrics and validation approaches at the application layer
5. Safety in perception-driven vehicle functions [9].

2.1.2 Importance of sensor data abstraction

Abstraction of sensor data is an extremely useful means for achieving the desired level of interoperability and reusability of data for the IoT and autonomous contexts. Such an abstraction provides a semantic description of data services in adherence with the

FAIR principles. FAIR stands for Findable, Accessible, Interoperable, and Reusable. These principles aim to make data more usable by describing the data services in a way that makes the data services discoverable, accessible, and reusable across systems and over the network easily [10].

In autonomous vehicle operations, sensor data abstraction serves as an interface between sensor data and the machine learning applications that, in turn, enhance the likely transferability of perception models to new setups of sensors [8]. Data abstraction can enhance clustering outcomes for test case generation in the automotive domain, improving clustering validation metrics and reducing [11].

Semantic interfacing allows more expressive representation of sensor data and aids in knowledge acquisition in IoT environments [12]. However, guaranteeing the sensor data quality is still a challenge. And the common issues are absence of data, outliers, bias, and drift. Various strategies are used for error detection and correction in sensor data, such as principal component analysis and artificial neural networks [13].

2.1.3 Benefits of sensor data abstraction

Sensor data abstraction forms the backbone of modern-day vehicle systems, leading to significant advantages in terms of interoperability, efficiency, and perception dependability. Recent studies highlighted these three core benefits:

1. **Algorithm portability across sensor configurations:** Abstraction layers decouple raw sensor outputs (such as lidar point clouds, radar heatmaps) from perception algorithms, enabling ML models to function across heterogeneous hardware setups. This reduces retraining costs by 40-60% when deploying existing models to new sensor arrays. For example, frameworks converting multi-modal data into unified geometric representations allow perception systems to adapt smoothly between vehicles with differing camera placements or radar specifications [8].

2. **Real-time processing efficiency:** The current systems have been characterized by processing of sensory data at scales never thought possible:
    - LiDAR: 2.5 million points/second
    - Radar: 77 GHz sampling rates
    - Cameras: 120 FPS feeds

    This flow is optimized at abstraction which normalizes resolutions by filtering redundant data and prioritizing critical inputs. The authors in [14] provided an AR-enhanced navigation system with sensor abstraction and achieved a latency below 100ms with 34% better lane-keeping accuracy [14].

3. **Enhanced system integration:** Modular architectures using abstraction layers demonstrate:
    - 45% faster V2X communication,
    - 64% improvement in positional accuracy for collision avoidance [14].

2.1.4 Current methods of sensor data abstraction in SDVs

Sensor data abstraction in the contemporary vehicle domain has seen a rapid evolution over the past few years owing to the advancements in software-defined vehicle architecture, modular systems, and high-performance computing. Below are the key methods used for sensor data abstraction in SDVs:

1. **Multi-layered Service-Oriented Architecture:**
   SDVs employ a Multi-layered Software Architecture (MSOA) to abstract sensor data in the most effective way possible. This great approach divides the system into three layers:
    - **Basic Layer:** Basic layer mainly processes raw sensor data and provides compatibility among different sensor types like cameras, radars, and lidars.
    - **Middleware Layer:** Main role of the middleware layer is to accept data fusion processes coming from the sensors and to exchange their results to the upper applications.
    - **Application Layer:** Application Layer consumes the abstracted data in the decision-making processes such as navigation and obstacle detection.

   MSOA enhances the portability of the vehicle software among different vehicle models, reduces complexity, and promotes rapid integration of the new features. This architecture promotes remote updates of optimizations and lifecycle maintenance [15].

2. **Hardware and Operating System Abstraction:**
   HWAL (Hardware Abstraction Layer) and OSAL (Operating System Abstraction Layer) are necessary in the present-day sensor data abstraction framework. These layers can:
    - Establish seamless interfacing with disparate hardware configurations.
    - Manage middleware reuse across various operating systems, thus lowering development cost and allowing greater scalability.

   Thus, software reusability, being mainly charged with the overhead of hardware-specific dependencies, is enhanced when using this approach. This is especially useful in the case of centralized vehicle architectures in which high-performance computing platforms manage the zonal ECUs [15].

3. **Middleware-based Sensor Abstraction:**

Middleware-based sensor abstraction stands out as the SDVs being the heart of the vehicle. It gives a generic interface that is the middleware between vehicle sensors and the application software. Middleware platforms of the type of AUTOSAR, ETAS, or custom frameworks are using hardware abstraction layers for the purpose of unifying the access to different sensor types, which in turn can lead to a new approach of modular, scalable, and updatable vehicle architectures. In this way, developers are able to create applications that can interact with standardized sensor data regardless of the underlying hardware and thus are in the position to do plug-and-play integration and rapid deployment of new features. Not only does middleware decouple software development from hardware changes, but it also stimulates the innovation of SDVs [16], [17].

2.1.5 Challenges in current sensor data abstraction approaches

Sensor data abstraction plays a pivotal role particularly in SDVs and autonomous systems. However, many challenges are faced while implementing effective abstraction framework and those are mentioned below.

1. **Heterogeneity of Sensors and Interfaces:**
   SDVs integrate a wide variety of sensors, including cameras, lidar, radar, ultrasonic sensors, etc, each with their unique data formats, resolutions, and communication protocols. For instance, this lack of standardization across sensor manufacturers and proprietary interfaces creates very considerable barriers in achieving a fully seamless abstraction:
   - Most of the sensors available are limited to a particular brand or are closed systems, therefore access to the primary data is blocked and so the idea of building a common layer of communication between different systems is also blocked.
   - This heterogeneity in turn complicates sensor fusion processes and increases the complexity of middleware design [18].

2. **High Computational demands:**
   The process of abstraction will have to deal with processing real-time multi-modal sensor data in bulk for critical applications like Advanced Driver Assistance Systems (ADAS) with low latencies. One of the challenges here is:
   - The need for high-performance computing (HPC) systems to handle the computational load from abstracting and fusing data from the multiple sensors.

   Centralized E/E architectures, for example with HPCs connected through Gigabit Ethernet, are being adopted. However, these require tremendous costs in hardware and software optimizations [15].

3. **Cybersecurity Concerns:**
   To enable the communication between hardware and software implementations, an abstraction layer must be installed. Such communication becomes the first step into the potential attack path of cyber threats. Key challenges include:
   - Safe communication between sensors and centralized computations.
   - Protection of the OTA changing updates from abstraction layers or sensor configurations.

   In this regard, the deterioration of risk in a nearly all-connected ecosystem of vehicles makes it even harder to avoid tougher encryption and intrusion detection measures [15].

4. **Cost Constraints:**
   Setting up strong frameworks of abstraction needs a considerable amount of research and development spending, especially for computing on high-performance systems, simulation tools, such as NVIDIA Omniverse [19]. Also, centralizing E/E architecture reduces wiring costs but requires upfront investments of HPC hardware too high to make the architecture feasible for low-priced models [20].

2.2 Existing automotive standards in sensor data abstraction

Sensor data abstraction is guided by several international standards that ensure interoperability, modularity, and scalability are taken into consideration while designing and implementing sensor systems. These standards form a basis for integrating onto a single automated driving function framework heterogeneous sensor type. Some details about these standards and their relevance to sensor data abstraction is below:

1. **ISO23150:2023**
   The ISO23150:2023 standard defines the logical interface between various environmental perception sensors (such as radar, lidar, and cameras) and data fusion unit in automated driving vehicle. The standard describes, in a modular-semantic way, levels of sensor data as follows:
   - Object Level: Information about potential moving objects, road-based objects, static objects, and free space.
   - Feature Level: Sensor-specific information such as edges or contours derived from detected camera or lidar measurements.
   - Detection Level: Raw detections from individual sensors, including radar reflections, lidar points.

   Thus, ensuring that there is consistency in abstraction and interpretation of sensor data within the fusion unit, and providing an orderly means to incorporate multi-modal sensors into a single perception pipeline. It excludes raw data

interfaces but emphasizes modularity to permit continuous innovations for evolution of sensor technology, be it for instance 4D imaging radar or thermal cameras [21], [22].

2. **AUTOSAR Adaptive Platform**
   The AUTOSAR Adaptive Platform builds upon the ISO 23150 by specifying the logical interface of sensor abstraction for software-defined vehicle architectures. The platform provides several features including:
   - Standard Interfaces: These are logical representation of information obtained from radar, lidar, and camera data that are compliant with ISO specifications.
   - Modular Architecture: Allows developers to integrate new sensors without a redesign of the entire architecture.
   - Safety and Security Compliance: Ensuring that it meets functional safety standards (for example, ISO 26262) and cybersecurity requirements (for example, ISO 21434).

   The AUTOSAR abstraction layer framework incarnates perfect scaling by decoupling the hardware-specific dependencies from the software application, ensuring low-cost development and better viability across OEMs [23].

3. **Open Simulation Interface (OSI)**
   Alongside ISO 23150, the OSI offers a standard testing and validation framework to enable abstracted sensor data to operate with virtual environments. OSI defines the input-output interfaces for sensor models, thereby allowing simulation-based validation of perception systems under various operating conditions. Which allows for:
   - Modular sensor modelling across various technologies (camera, lidar, etc.).
   - Scenario-based tests to test abstraction fidelity under edge cases such as bad weather or complicated traffic situations.

   This standard is much used together with ISO 23150 to further assure that the abstracted data behaves according to requirements in the real world [24].

4. **ISO 26262**
   Even though not directly targeting data abstraction from sensors, its standard ISO 26262 governs functional safety in vehicle systems. As this standard dictate, abstraction layers must be developed to handle faults effectively while keeping system safety intact. For example:
   - Duplicate paths to vital sensor data provide redundancy against device failure.
   - Validation procedures ensure extracted data meets required safety integrity requirements intended for operations of automated driving [25].

2.2.1 Challenges addressed by these standards

These above-mentioned standards together address some important issues associated with abstracting data from sensors:

- **Interoperability:** Standardized interfaces like ISO 23150 and AUTOSAR enable seamless integration of varied sensors into unified frameworks.
- **Faster Development:** Standardized data communication reduces integration effort and shortens project lead times.
- **Enhanced Safety:** By ensuring data consistency and reliability, ISO 23150 contributes substantially to the safety of autonomous vehicles.
- **Scalability:** Modular designs allow additional sensors to be integrated without requiring a lot of re-engineering [21], [22].
- **Validation:** OSI provides tools for simulation-based testing to ensure that abstracted data meets performance benchmarks under diverse conditions [24].

2.2.2 Important terms associated with sensor data abstraction

There are some terms like VSS, Kuksa databroker and DDS which are not standards related to sensor data abstraction but are often associated with standardization in the automotive industry. Those are briefly described below:

1. **Vehicle Signal Specification (VSS)**
   VSS is an open standard developed by the Connected Vehicle Systems Alliance (COVESA, formerly known as GENIVI Alliance). It defines a hierarchical structure of signals for vehicles in standardized form. The VSS also provides a common data model that supports automobile makers and software engineers in using a common approach towards extracting and managing vehicle signals [26].

2. **Eclipse Kuksa**
   Kuksa databroker is an in-vehicle, open-source data broker that was created as part of the Eclipse Kuksa project. Kuksa databroker provides a VSS model practical application, hence offering secure, standardized access to in-vehicle data. As a middleware component, Kuksa databroker acts as an intermediary among in-vehicle sensors, ECUs, and outside applications [27].

3. **Data Distribution Service (DDS)**
   DDS forms a real-time, publish-subscribe protocol, enabling efficient data exchange between sensors in connected and autonomous vehicle systems. Initially it was developed with industrial and defence applications in mind, but now DDS has become an essential enabler of vehicle sensor data abstraction, particularly in ADAS and autonomous technologies [28].

2.3 Vehicle Signal Specification (VSS) in Sensor Data Abstraction

2.3.1 What is VSS?

The VSS was created in 2016 and it is a protocol-agnostic, open data model developed by the COVESA. It is a standardized syntax and classification scheme for the description of vehicle signals and thereby facilitates a consistent description of vehicle data across numerous Original Equipment Manufacturers (OEMs), applications, and ecosystems. The VSS serves as a "common language" in the description of vehicle data and thereby provides semantic consistency and compatibility among in-vehicle applications, cloud applications, and extrinsic applications. The VSS hierarchical tree diagram is shown in Figure 1 [26], [29], [30].

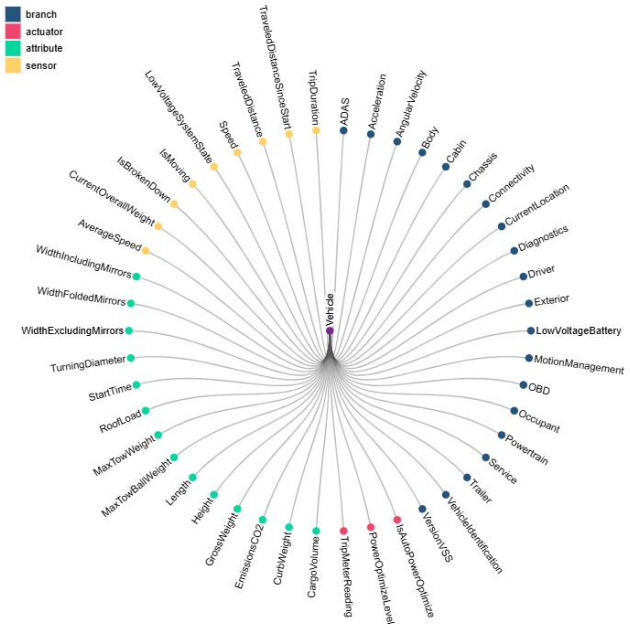

Figure 1. VSS Tree model [30]

VSS has gained broad industry adoption in the automotive sector as a foundational framework for SDVs and connected cars. It solves the problem of fragmented data by building a common construct that simplifies integration and accelerates the pace of innovation [26], [30].

2.3.2 Features of VSS

VSS provides several standardized features which simplifies the vehicle data access and management. These features are mentioned below:

1. **Standardized Syntax and Taxonomy**
   VSS uses a tree model to express vehicle signals as shown in Figure 2. Every signal is addressed by a unique path (e.g., `Vehicle.Drivetrain.Transmission.Speed`) that defines its category, subcategory, and specific feature [29], [30].

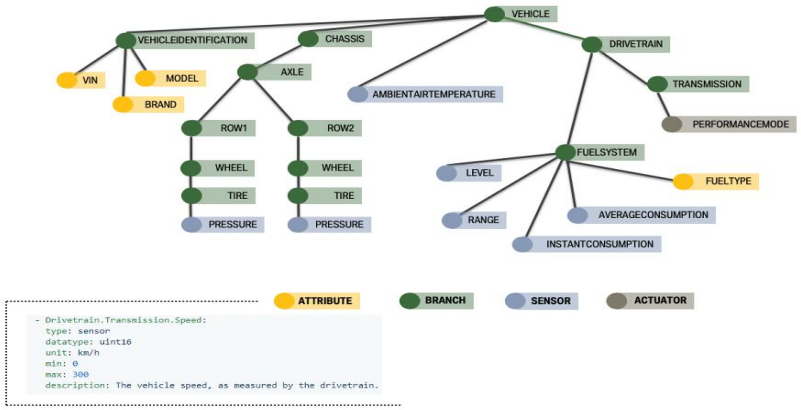

Figure 2. VSS data model structure [30]

Signals are configured using ***vspec*** files, which are human-readable files based on YAML (YAML is not Markup Language). These ***vspec*** files define various attributes, like data type, units, and description [30].

2. **Extensible Signal Library**
   The VSS library defines reusable and standard signals for many domains, including powertrain, chassis, infotainment, etc. The VSS architecture allows for vendor-specific extensions via overlays while still adhering to VSS core syntax rules [30].

3. **Protocol-Agnostic Design**
   VSS is neutral to the underlying protocols (e.g. CAN, Ethernet) and creates a single interface to access various vehicle signals. This provides for future adaptability to varied technologies and deployment scenarios [26], [29].

4. **Interoperability**
   By standardizing the signal names across all the OEMs (e.g., `Vehicle.Speed`), VSS enables seamless communication among cars, devices, and infrastructure across brand and technology boundaries [26], [31].

5. **Developer-Friendly Tooling**
   VSS contains tools for parsing, validation, and translation of ***vspec*** files to other file formats (e.g. JSON) to facilitate development and minimize development costs [30]. The Figure 3 shows the concept of vspec file conversion.

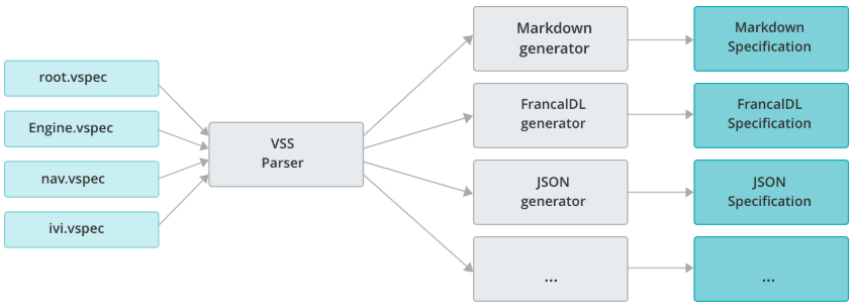

Figure 3. vspec file conversion [26]

6. **Focus on Innovation and Safety**
   VSS gives vehicles the ability to share information independently of brand or proprietary protocol (ex. speed or location), allowing for better coordination with other vehicles or infrastructure to ultimately improve safety. It also provides a basis for innovation by provisioning for developers to build applications across brand applications by passing shared standards [29], [31].

2.3.3 Capabilities of VSS

There are many capabilities of VSS are enabled by the features mentioned above. Some of these capabilities are as follows:

1. **Data Abstraction**
   VSS maps the raw sensor data to standardized signal names which decouples the list of sensor data from adding application logic by providing the abstraction of the data. This will make application development easier and increase the reusability of the data in other applications. VSS ecosystem diagram is as shown in Figure 4 [29], [30].

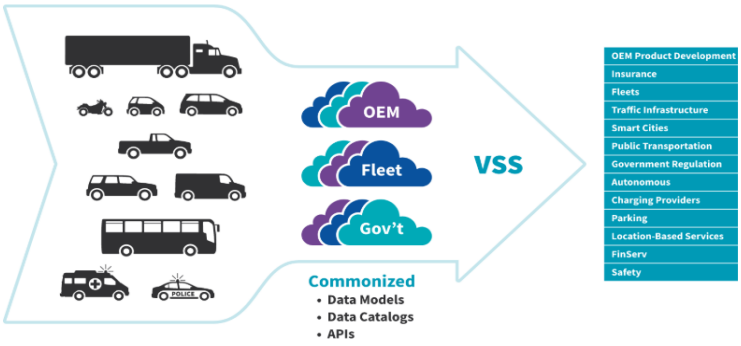

Figure 4. VSS Ecosystem [26]

2. **Improved Interoperability**
   VSS enhances interoperability between systems; it creates a shared understanding of vehicle data to limit fragmentation and assimilate into

connected ecosystems, such as fleet management platforms or smart cities [26].

3. **Operational Efficiency**
   VSS allows OEMs to develop standardized data collection policies across multiple models, while lessening operational burden and providing consistency across product lines [29], [31].

4. **Cost Reduction**
   Utilization of standardized signals destroys the need for customized integrations, and it reduces a large portion of the total time and costs across the development cycle [26], [31].

5. **Scalability for SDVs**
   VSS provides scalability for SDVs by permitting dynamic updating of signal definitions without any changes to the hardware [26].

2.4 Eclipse Kuksa in Sensor Data Abstraction

2.4.1 What is Eclipse Kuksa

Eclipse Kuksa is an open-source framework under the Eclipse Foundation designed to provide standardized access to vehicle data and features over a Vehicle Abstraction Layer (VAL). The principal goal of Kuksa is to create a common platform that creates connectivity of in-vehicle systems to cloud services and third-party applications, thereby enabling interoperability and creating innovation within the vehicles ecosystem.

Kuksa achieves this by adopting the COVESA VSS to ensure signal standardization, while also providing an API for secure and efficient data transfer. Kuksa is ideally suited for SDVs and helps facilitate the development of OEM agnostic applications that may run on a variety of vehicle architectures. Kuksa can be easily integrated with AGL (Automotive Grade Linux) and cloud-based services to provide a rich platform that can support connected vehicles and Autonomy [32], [33].

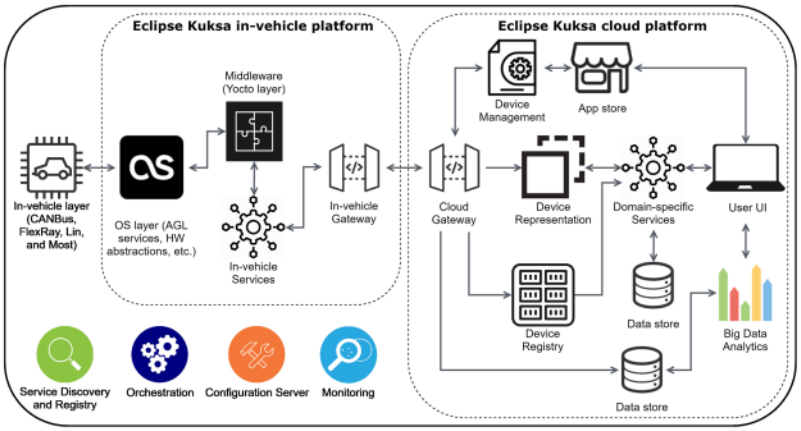

Figure 5. Software architecture of Eclipse Kuksa [27]

Figure 5 shows the software architecture of Eclipse Kuksa. Eclipse Kuksa combines three open-source platforms for *(1) in-vehicle, (2) cloud, and (3) application development.* The *gateways* serve as a communication interface between the vehicles (e.g., CAN-Bus) and the cloud over different data communications protocols, e.g., the MQTT. The in-vehicle platform comprises a *middleware layer* that includes APIs used for vehicle abstraction and communicating with the application, along with an *operating system layer* with core services: bootloader service and hardware abstraction.

The cloud platform controls digital *representations* necessary for devices, and has a *device registry* to enable access to those representations on the cloud platform, and implements domain-specific services for vehicles. It also manages *data storage* on behalf of the devices, enables management of devices (for provisioning, configuration, and monitoring), and provides an app store to host applications for vehicle utilization. The platform has *visualisation* and data analytics tools which will provide capability to also process data and host visualisation of processed data to enable new services [27].

The Eclipse Kuksa software is composed of several core components.

- **Kuksa.val** is a vehicle abstraction layer that provides unified access to data based on the VSS.
- **Kuksa Databroker** is middleware providing communication and management of data flow from the vehicle system to an external application.
- **Kuksa IDE** is a cloud-based development environment that provides everything necessary to developing and testing vehicle apps.

2.4.2 Features of Eclipse Kuksa

Eclipse Kuksa provides some powerful features in sensor data abstraction, and those are mentioned below:

1. **Standardized Data Access**
   Kuksa implements the COVESA VSS for semantic consistency across vehicle signals (e.g., `Vehicle.Drivetrain.Transmission.Speed`). It allows integration through multiple communication protocols. In addition, it can abstract raw sensor data into common formats. Figure 6 describes the basic scope of Kuksa databroker [33], [34].

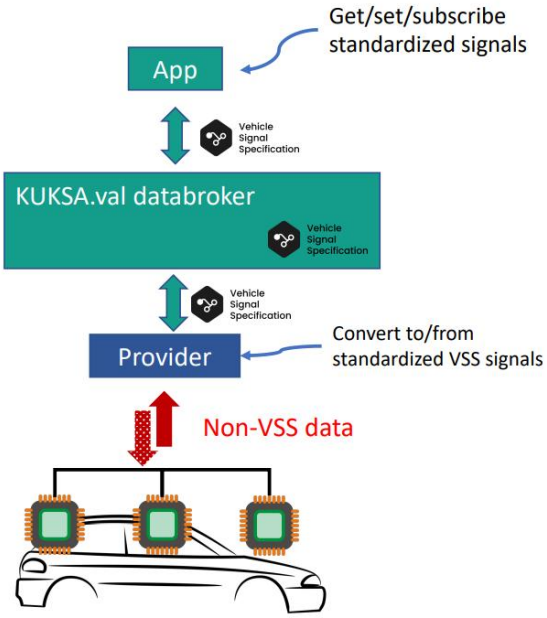

Figure 6. Scope of Kuksa Databroker [35]

2. **APIs for Application Development**
   Kuksa supports API for the W3C Vehicle Information Service Specification (VISS) to allow access via REST/WebSocket, in conjunction with gRPC for real-time data exchange with high-performance needs [32], [36].

3. **Modular Architecture**
   Kuksa has three layers:
   - **Application Layer**: Runs third-party applications with sandboxed space, isolated from other applications.
   - **Middleware Layer**: Abstracts the vehicle's E/E architecture using the API or protocol.
   - **OS Layer**: Integrates with AGL or other Linux platforms to abstract hardware [33].

4. **Cloud Integration**
   It contains a cloud backend which enables OTA updates, fleet management, and remote diagnostics. It also supports two-way communication between vehicles, infrastructure, and 3rd party services via IoT protocols [36].

5. **Developer-Friendly Tools**

Provides Kuksa IDE, based on Eclipse Che, which simplifies application development through shared workspaces, pre-configured environments, and Docker-based deployment. Kuksa supports continuous integration and continuous delivery (CI/CD) to support apps testing and deployment [33], [34].

2.4.3 Capabilities of Eclipse Kuksa

Kuksa has some of the important capabilities which are enabled by its features:

1. **Interoperability Across Platforms**
   Promotes OEM-agnostic development by encapsulating OEM hardware-specific details, through standardized APIs, thus enabling apps to work on any vehicle with Kuksa Databroker support [32].

2. **Real-Time Data Processing**
   The Kuksa Data Broker can facilitate a low-latency data exchange mechanism between car interior systems and external applications; having a low-latency communication mechanism allows Kuksa Data Broker to be applicable to time-dependent use cases, such as autonomy or predictive maintenance [36].

3. **Scalability for SDVs**
   Allows for centralized E/E architecture with only one Data Broker, as well let multiple Data Brokers operate efficiently in a distributed system processing data in a redundant or domain-specific manner (e.g., powertrain vs infotainment) [33].

4. **Enhanced Security**
   Kuksa offers strong security features with encrypted communication channels, role-based access control, and secure OTA updates to protect sensitive vehicle data.

2.5 Data Distribution Service (DDS) in Sensor Data Abstraction

2.5.1 What is DDS

The DDS is a type of middleware protocol and an API standard established by the Object Management Group (OMG) for the purpose of real-time, scalable, and high-performance data distribution. It is utilized across various application areas, including automotive, aerospace, health care, and defence. DDS functions by applying a publish-subscribe model for communication whereby data producers (publishers) and data consumers (subscribers) share data through a collectively understood global data space. This publish-subscribe communication structure decouples the data structure

of the data from the practical implementation, and facilitates interoperability across different software systems, operating systems, and programming languages [37], [38]. Figure 7 shows the DDS architecture.

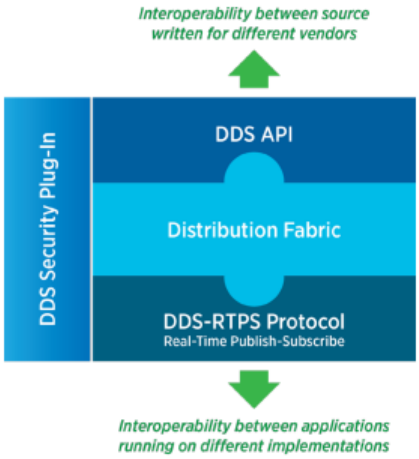

Figure 7. Data Distribution Service (DDS) architecture [38]

In the automotive domain, DDS is critical for establishing real-time relationships between Electronic Control Units (ECUs) and between the ECUs, the sensors, the actuators, and external systems; such as cloud services or V2X systems. As an example, a Software-Defined Vehicle (SDV) generates considerable data on the vehicle and environment, which the vehicle must act upon in a timely and reliable manner [38].

2.5.2 Features of DDS

Figure 8 shows the graphical representation of DDS communication model.

1. **Publish-Subscribe Communication Model**
   With DDS, components in a system can inter-operate without tied or hardwired connections, utilizing topics. Publishers create data for "topics," and subscribers consume data based on their interest for a specific topic. This paradigm makes integration easier by removing fixed connections between applications [37], [39].

2. **Global Data Space**
   DDS allows the concept of a global data space meaning that all nodes participate in sharing consistent, peer-reviewed data that is up-to-date. This means that a subscriber that joins late can access historical data that relates to their actions [39].

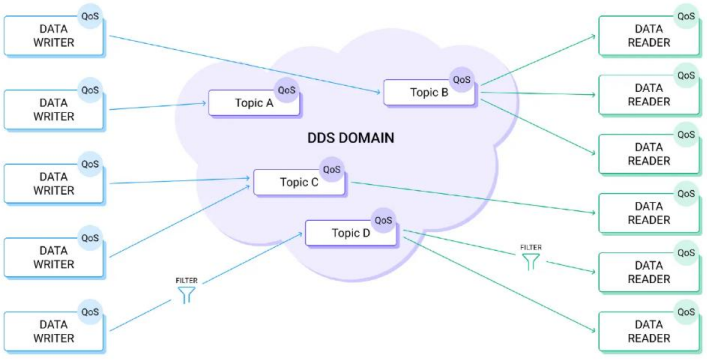

Figure 8. DDS communication model [39]

3. **Quality of Service (QoS) Policies**
   DDS provides a rich set of QoS policies to control various things including reliability, durability, latency, bandwidth consumption, and resource level limits. QoS policies ensure that data transmission satisfies the needs of a specific application. Some examples are:
   - **Reliability**: provides guarantee messages will be delivered.
   - **Durability**: offers the ability for data to persist for late subscribers.
   - **Deadline**: specifies when data should be sent.
   - **Time-based filters**: the ability to only send data if certain time constraints have been met [39], [40].

4. **Dynamic Discovery**
   DDS provides automated, run-time discovery of publishers and subscribers while eliminating the need for pre-configured communication ports, and thus giving applications a "plug-and-play" capability.

5. **Scalable Architecture**
   DDS supports horizontally scaling over many nodes and control domains, making it ideal for distributed systems such as autonomous vehicles or fleets.

6. **Security Features**
   DDS provides strong security mechanisms in the form of encryption, access control, and secure data flow management for protecting sensitive information in real-time applications [39].

2.5.3 Capabilities of DDS

1. **Real-Time Data Exchange**
   For application areas such as collision avoidance or autonomous-driving algorithms, DDS provides latency as low as 30 microseconds and handle millions of messages per second [39], [41].

2. **Interoperability Across Platforms**
   Supports operating systems (e.g., Linux, Windows) and programming languages (e.g., C/C++, Java, Python), helping vehicle companies and organizations successfully implement cross-domain compatibility across varied automotive architectures [37].

3. **Fault Tolerance and Resilience**
   Employs failover functions, to provide continued operation while a vehicle's hardware or communication node fails [41].

4. **Integration with Automotive Standards**
   Integrates with frameworks (e.g., AUTOSAR Adaptive Platform and ROS2) that provide communication continuity and allow for scalability in SDVs [38].

5. **V2X Communication**: Reliable transfer of data occurs between vehicles and between vehicles and the infrastructure [38].

2.6 Mixed-Criticality Concepts and Middleware Requirements

2.6.1 Introduction to Mixed-Criticality in Automotive Systems

The swift advancement of embedded systems, especially in automotive settings, has led to increased consolidation of functions of differing criticality onto shared hardware platforms. Mixed-Criticality Systems (MCS) allow supporting tasks of varying criticality. Safety-critical tasks have strict safety and real-time requirements while non-critical tasks have less strict operational requirements [42].

In automobile systems, criticality is usually specified by standards like the ISO 26262 standard that categorizes functions based on their Automotive Safety Integrity Level (ASIL) from ASIL-A (low criticality) to ASIL-D (high criticality). Providing these diverse requirements appropriately entails diligent resource planning of allocation, isolation, and scheduling of tasks such that the system is reliable and has good performance [43].

Historically, automotive engineers segregated distinct critical functions of systems onto different hardware platforms, primarily to comply with various levels of reliability and safety. Unfortunately, isolating functions increases costs, complexity, and maintenance costs. Practiced engineers understand the push to share applications of varying criticalities and increasingly expect all manufacturers to have middleware that can support temporal and spatial isolation, fault containment, and priority-based scheduling, which will effectively guarantee safety without disturbing MCS architectures [44].

ISO 26262 includes functional safety requirements in automotive systems; it has outlined safety requirements that require appropriate fault handling and isolation mechanisms. Therefore, it is possible to use middleware to develop these types that strictly adhere to temporal and spatial isolation, or other mechanisms, so that lower-criticality software faults or overruns do not adversely affect higher-criticality applications [25].

2.6.2 Examples of Mixed-Criticality Use Cases in Vehicles

Many real-world automotive examples exhibit characteristics like mixed-criticality applications. ADAS is a prominent example, where safety-critical functions like emergency braking and lane-keeping assistance coexist with non-critical features like infotainment and navigation. Since ADAS must ensure that high- and low-criticality tasks are not scheduled at the same time, isolation and resource allocation are especially important [45].

Another example of mixed-criticality applications in the approximate time-baseline of automotive is centralized E/E architectures where a smaller number of physical, high-performance processors replace multiple ECUs. These high-performance, multi-core processors are responsible for a variety of applications, including high-criticality applications such as safety-critical, autonomous driving functionality, as well as the use of the same vehicle's resources for comfort and entertainment applications that are not critical at all. Virtualization technologies enable such mixed-criticality applications as it allows low-criticality applications to operate without negatively affecting performance and safety of high-criticality applications [46].

A somewhat newer example of potential mixed-criticality applications are zone-based automotive architectures. In these architectures, a central computing platform distributes or coordinates and manages the functionality of vehicle features. Zone-based architectures are an improvement in scale, user experience, and reduced wiring complexity. Within any zone-based architecture system, virtualization and middleware technologies will play an important role in ensuring shared resource security, availability and distribution across a number of virtualized functions with varying criticality requirements [46].

2.6.3 Design Goals and Functional Requirements for Mixed-Critical Middleware

- **Temporal and Spatial Isolation:** For automotive middleware to effectively support mixed-criticality workloads, it needs to consider several design goals. First, temporal and spatial isolation is key. It is a must to ensure that the execution of safety-critical functions is not affected by failures or delays in some non-critical software component. This isolation can be done via partitioned scheduling and memory protection, both of which are prerequisites for compliance with ISO 26262 [45].

- **Predictable Real-Time Behaviour:** While ensuring isolation, it is necessary to support predictable real-time behaviour. The middleware needs to provide deterministic communication, especially for time-critical functionality in scenarios where actuators may be controlled, such as braking and steering control, in autonomous or assisted driving systems. The DDS standard can be configured for various QoS policies, with relevance to deadlines and latencies budgets required to ensure consistent real-time behaviour in a variable load environment [47].

- **Scalability and Flexibility:** The middleware also needs to be scalable and flexible. As SDVs mature, the middleware needs to be able to handle new features and applications that will not disturb existing software hardware capabilities. The middleware may need to support service discovery, through modular software and cross-platform capabilities to support multi-ECU or zonal vehicles [43].

- In addition, **Fault Tolerance and Fail-Safety** are the design necessities. Middleware must be able to detect runtime faults, isolate faulty components, and allow for graceful degradation or reconfiguration. Delivering these capabilities allows a vehicle to continue operating when a software or hardware fault occurs [46].

- **Dynamic Discovery and Integration:** Middleware must support service discovery and plug-and-play capability at runtime, which means allowing OTA updates and flexible deployment effectively enabling modularity into SDVs [43].

- Lastly, **Security and Certification Readiness** are also desperately needed in the automotive domain. The middleware is expected to provide authentication, data integrity or tamper proofing and access control appropriately and is also expected to be auditable to be in-line with ISO26262 and ISO 21434 [25], [48].

2.6.4 DDS in Mixed-Criticality Concept

DDS is a key enabler of communication in mixed-criticality automotive systems. DDS is a publish-subscribe communication middleware defined by the OMG (Object Management Group) and is now a key part of the software stacks for automotive systems because of its flexibility, scalability, and ability to support real-time needs.

1. **Topic-based Data Exchange:** DDS uses a topic-based communication model as shown in Figure 9. Topic-based Publish-Subscribe communication in DDS [17] Applications can have topics where they are either publishers (applications generating data) or subscribers (applications receiving data). Topics can be associated with specific data types or channels for control of components that

can be naturally decoupled. This decoupled architecture is well-suited to mixed-criticality systems, where systems may need to exchange data, the critical component (e.g., braking control) can be separated from non-critical (e.g., notification on the dashboard), which may interfere with the behaviour of the critical function. Topics also allow applications to publish data, without having the high coupling of traditional APIs, leaving the subscriber with loosely coupled/decoupled interaction [17].

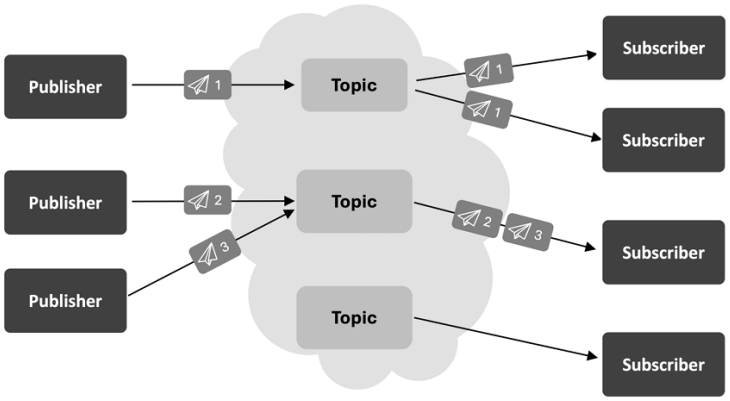

Figure 9. Topic-based Publish-Subscribe communication in DDS [17]

2. **Quality of Service (QoS) Configuration:** One of the most impressive abilities of DDS is its QoS model. Developers can configure QoS policies for latency budgets, reliability (best-effort vs. reliable), history (how much data is retained), deadlines, etc. This feature allows for DDS to provide deterministic delivery of safety-critical messages while relaxing constraints for messages of lesser importance. In this way, DDS allows us to configure ADAS components so that delivery from these can have strict timing constraints, and the infotainment services can have some delay and even loss [17].

3. **Real-time Guarantees and Protocol Efficiency:** DDS uses the Real-Time Publish-Subscribe (RTPS) protocol to guarantee low latency message delivery, through UDP, TCP, and shared memory depending on the deployment environment configuration. In high-performance deployment scenarios, like the centralized automotive architecture, shared memory or even UDP multicast provides for an effectively zero latency and jitter. According to Klüner et al., Fast DDS, an open-source DDS implementation, supports latency requirements <50 µs in optimized automotive deployment environments [17].

4. **Dynamic Discovery and Modularity:** From the outset, DDS provides users with multiple discovery mechanisms, including automatic, static, and manual discovery. Dynamic discovery allows applications that are late-joining in the existing network to seamlessly join the rest of the network. This is a very important advantage in modular automotive platforms where there will be frequent OTA updates or when plug-and-play modules are added to the

network. The discovery mechanisms in DDS allow new safety, or comfort modules to locate existing data producers and consumers; while clearly following appropriate safety protocols with minimal reconfiguration [17].

5. **Security and Mixed-Criticality Isolation**: DDS also provides integral security features, including participant authentication, access control, and data encryption. This is especially evident in use cases where applications of high and low criticality will be using the same in-vehicle network (IVN). Security mechanisms will allow for enforcement of high-security credentials to prevent unauthorized access or otherwise unintentional interference by lower-criticality tasks supporting compliance with ISO 21434 and functional safety standards like ISO 26262 [17], [48].

2.6.5 Existing Standards and Compliance

Middleware for mixed-criticality automotive systems must meet established safety and interoperability standards to ensure a reliable and certifiable approach. This includes compliance with ISO 26262 for functional safety, the AUTOSAR Adaptive Platform for modular and service-oriented vehicle (and other mobile) architectures, and DDS-specific standards for compliance with real-time communication. It is important to be sure that the middleware integrates with the below-mentioned critical standards and frameworks to ensure safe and reliable operation of safety-critical applications alongside the flexible non-safety-critical applications.

1. **ISO 26262 – Functional Safety:** ISO 26262 is the specification for functional safety for automotive electronics. It defines ASIL-A (least critical) to ASIL-D (most critical), and it mandates that safety-critical "software be separated from non-safety-critical" software to demonstrate "freedom from interference." In this context, middleware must be able to provide spatial and temporal partitioning, deterministic behaviour, and fault containment support, so safety cases and traceable verification steps can be developed. DDS and similar middleware frameworks can provide configurable QoS parameters, safety channel abstraction, and deterministic transports that meet the interface requirements for ASIL compliance [49], [50].

2. **AUTOSAR Adaptive Platform:** The AUTOSAR Adaptive Platform embraces the SDV paradigm by establishing a basis of dynamic reconfiguration, high-performance computing, and service-oriented communication. It specifies middleware support for application partitioning (based on the microkernel concept), secure messaging, and dynamic resource management over heterogeneous ECUs. DDS can be implemented as a network binding in the AUTOSAR Adaptive stack, which supports real-time capability, publish-subscribe communication with reliably assured QoS, as well as bounded jitter. The DDS-AUTOSAR interface allows the interoperability of safety-critical and non-critical components coexisting with defined isolation boundaries, adhering

to the expectations of Adaptive Platform, and ensuring standards for mixed-criticality software [51], [52].

3. **DDS Certification and Compliance:** To satisfy ISO 26262 and the emerging ISO 21434 cybersecurity standard, there are certifiable DDS implementations available from vendors (e.g., RTI Connext Cert). Certifiable implementations contain formal development process artefacts, tool qualification data, and safety assurance levels for ASIL-B or ASIL-D applications. The DDS Security supports access control, encryption, and authentication for DDS implementations, enabling safe deployment in heterogeneous environments, thus protecting mixed-criticality communications from malicious interference [50], [52].

2.6.6 Challenges in Implementing Mixed-Criticality Systems

Even though there has been substantial theoretical advancement in the mixed-criticality system design, the translation to practical applications is challenging due to a few technical and industrial obstacles.

1. **Timing analysis and WCET estimation:** One of the important challenges is the Worst-Case Execution Time (WCET) for various tasks. Tasks are highly variable based on sensor readings, vehicle state, and real-time information, so it is challenging to predict WCET. If engineers cannot define it precisely, they end up overestimating to be safe, which results in inefficient use of resources and oversized system designs [53].

2. **Mode Switching and Run-time Guarantees:** Mixed-criticality systems commonly switch between operation modes when timing requirements become breached, for instance, through discarding or downgrading low-criticality tasks to sustain required performance for safety-critical tasks. Safe and predictable mode-switching thresholds and runtime-bounded switching overheads, however, are a non-trivial problem for engineers to identify [53], [54].

3. **Isolation vs. Resource Efficiency**: Standards like ISO 26262 require a strong separation between software with different criticality levels to ensure safety. However, manufacturers aim to maximize hardware usage and cut costs. These two objectives often conflict. Achieving both demands smart design, which may involve middleware, virtualization, or custom scheduling techniques that can be hard to validate [54].

4. **Gaps in Toolchain and Certification:** Most existing development environments were not built with mixed-criticality in mind. Certifying a system where some components are uncertified or have relaxed requirements introduces complications during safety assessments. Proving that high-criticality components will not be affected by the behaviour of lower-criticality

ones takes a significant amount of documentation, testing, and sometimes even formal proofs [53].

5. **Industrial Integration Complexity**: Even when everything is sorted out in theory, actual industrial implementation creates more challenges. ECUs from different vendors, legacy software, asynchronous network messages, and over-the-air update systems all make integration unpredictable. Bridging the gap between research-level models and production-level systems is still a work in progress [54].

2.6.7 Summary of Middleware Requirements

Middleware plays a central role in mixed-criticality software integration for modern automotive systems. It functions as a backbone for software component isolation according to the safety levels and to manage the communication timing, while supporting functional and safety standards. Key requirements for the middleware include enforcement of spatial and temporal isolation; support for time-bound communication by specifying a QoS mechanism; and ensuring that appropriate fail-safety and system resilience are in place, including fault-reporting, fault detection, fault recovery, and upgrade-on-failure.

The middleware is also expected to support scalability, dynamic discovery of services, and transparent OTA update mechanisms for SDV evolution. The integration of middleware solutions with the standards like ISO 26262, ISO 21434, and AUTOSAR Adaptive ensures the technical soundness alongside certifiability. In essence, the middleware in a mixed-criticality environment must guarantee safety, performance, and compliance while maintaining modularity and flexibility of the system.

2.6.8 Lightweight Middleware: Zenoh for Edge Data Communication

2.6.8.1 What is Zenoh?

Zenoh is a recent open-source communication protocol used to combine how data, storage, and compute are unified across the vast diversity of computing environments, from large cloud systems to resource-constrained embedded devices [55], [56]. Zenoh was created by ZettaScale Technology and released under an Apache 2.0 license. The name Zenoh is for the term "Zero Overhead Network Protocol" and it reflects its emphasis on lower communication overhead for better efficiency [55], [57].

Zenoh is fundamentally a data-centric protocol that combines the publish/subscribe and query paradigms, allowing seamless management of data-in-motion (real-time streaming), data-at-rest (persistent data storage), and distributed computations. This all-in-one approach improves on traditional middleware solutions — for example, MQTT, DDS, and SOME/IP — which often compromise flexibility with rigid network topologies to facilitate interoperability or lose the benefits of the protocol in effectively supporting heterogeneous and resource-poor environments [55], [57], [58].

Zenoh's architecture is designed to accommodate various communication models and architectures including peer-to-peer, routed, and brokered layouts; thus, allowing it to be deployed across the spectrum of robotics, automotive, edge, and IoT sectors. Zenoh's protocol stack is cross-platform and can function across the Data Link, Network, and Transport layers, and can utilize communication protocols including TCP/IP, Bluetooth, Serial Links, and CANbus [55], [56].

A key differentiator for Zenoh is its very low wire overhead, which has been reported as low as 4–5 bytes per message, which enables it to operate on equipment with very limited resources [55], [57]. In addition, Zenoh abstracts distributed querying and geo-distributed storage, enabling applications to not only publish/subscribe but also query and persist data across the Cloud-to-Device stack [56], [58].

2.6.8.2 Features of Zenoh

Zenoh offers some great features and those are mentioned below:

1. **Unified Data Abstractions:** Zenoh offers unified publish/subscribe, query, and storage mechanisms in a single protocol, allowing the end-user to handle data in motion (real-time), data at rest (persistent storage), and distributed computing seamlessly. The unified approach cuts down on complexity in the architecture and the development of distributed systems [55], [59].

2. **Minimal Wire Overhead and High Efficiency:** Zenoh achieves extremely low wire overhead, on average, 4–6 bytes overhead per message, on account of the variable length encoding and efficient batching. This results in high throughput and energy efficiency in both the cloud and constrained embedded environments [59], [60].

3. **Flexible Communication Topologies:** Zenoh supports peer-to-peer, routed, and brokered communications (as shown in Figure 10. Zenoh supported topology [61]). And for this it does not require a specific topology. This allows Zenoh to adapt to many different types of deployment from minimal data exchange in a local embedded network to high volume of data and distributed environments [55], [61].

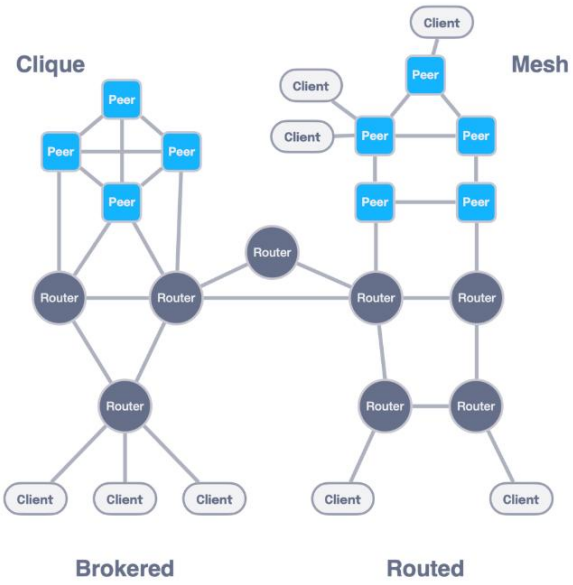

Figure 10. Zenoh supported topology [61]

4. **Cross-Platform and Cross-Layer Compatibility:** Zenoh operates over Data Link, Network, and Transport layers (i.e. TCP/IP, Bluetooth, Serial Links, and CANbus) allowing Zenoh to operate across a wide range of devices such as microcontrollers to data centre servers [55], [61].

5. **Security and Access Control:** Zenoh is implemented in Rust for memory safety and includes support for things like session authentication, pluggable authentication mechanisms, mutual authentication, and secure channels (like mTLS and QUIC). If fine-grained access control is needed, administrator users can define access controls on data operations [61], [62].

2.6.8.3 Capabilities of Zenoh

In addition to the above-mentioned features, Zenoh has some good capabilities, which are described below:

1. **Seamless Data Unification Across Heterogeneous Environments:** Zenoh allows applications to engage with data in motion (real-time streaming), data in rest (persistent storage), and distributed computations all with a single protocol. This capability provides developers with the ability to architect systems that can scale from microcontrollers to cloud data centres without the need for protocol translation or architectural fragmentation [55], [59].

2. **Extreme Performance and Scalability:** Zenoh provides a unique and unusually low latency (13-16 microseconds in peer-to-peer mode) and very high throughput (67 Gbps) which are several times lower and higher than traditional middleware protocols, such as DDS, MQTT, and Kafka. Moreover, these performance characteristics are maintained even in large-scale deployments that are geographically distributed [60], [63].

3. **Extensibility and Integration:** Zenoh allows for an almost limitless capacity for customization using plugins, making it possible to integrate with a variety of databases, data formats, and protocols of communication. The modular design allows for scaling up and integrating with existing technologies and systems, and with new systems and capabilities that arise in the future [60], [62].

2.6.8.4 Comparison: Zenoh vs DDS

Zenoh and DDS are both advanced middleware protocols that have been designed for the purpose of real-time, distributed system communications but differ significantly in their architectural patterns, performance characteristics, and a variety of deployment circumstances.

1. **Performance:** Comparative performance studies have shown that Zenoh has higher throughput than DDS consistently, and in many instances latency. For example, in single machine tests, Zenoh peak throughput was ~67 Gbps as shown in Figure 11, and in multi-machine runs it was ~51 Gbps as shown in Figure 12, compared to Cyclone DDS throughput, seeing similar benchmarks, Zenoh achieved more than double the throughput [63], [64]. Zenoh also exhibited less wire overhead - generally 4-6 bytes per message - allowing for more efficient performance gains, especially when working in bandwidth-constrained approaches. DDS also exhibits lower latency potential in certain configurations (e.g., UDP multicast and conservative resource provisioning on a single machine); however, in the case of typical distributed and wireless approaches for lower latency, Zenoh generally can have lower or comparable latency, particularly with Zenoh peer mode and Zenoh-pico implementation [64], [65].

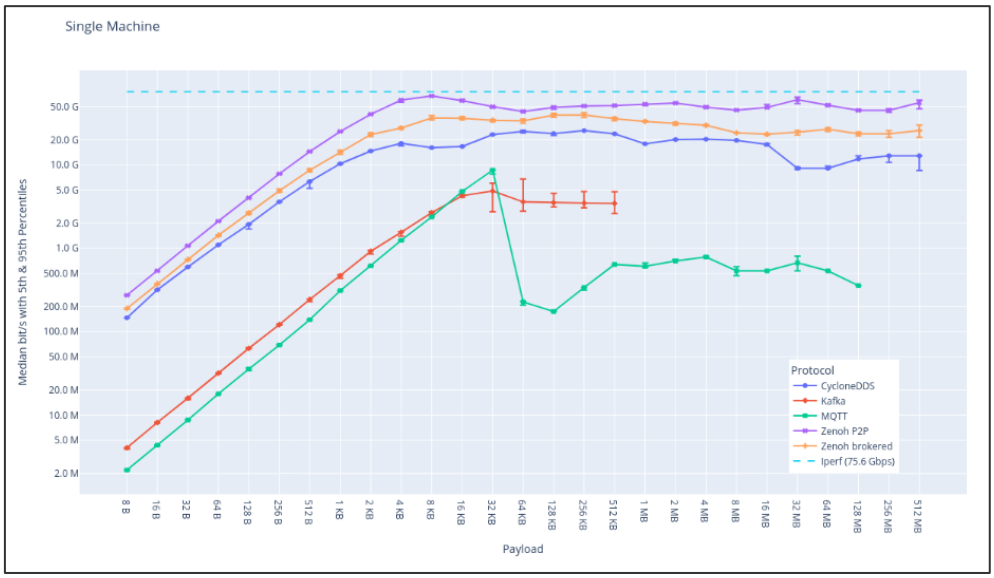

Figure 11. Throughput data in bit/s for the single-machine scenario [63]

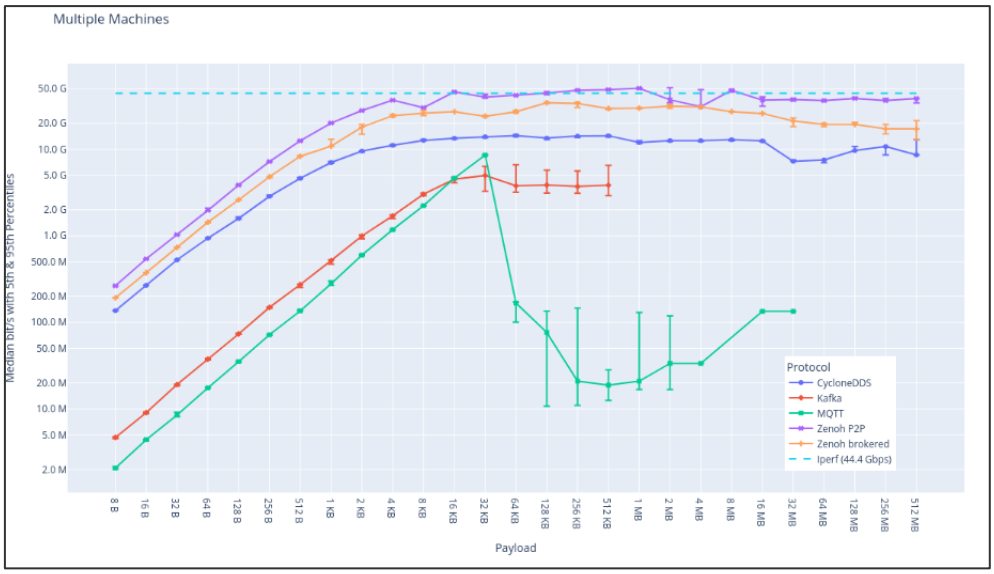

Figure 12. Throughput data in bit/s for the multiple-machine scenario [63]

2. **Discovery and Scalability**: Among the evaluated middleware, Zenoh shows the fastest discovery times, closely followed by FastDDS, while both protocols have better discovery times than legacy DDS and other protocols like vSomeIP. However, for some large-scale robotic fleet scenarios, it may still be faster to use FastDDS Discovery Server v2 in terms of managing metadata exchange, where Zenoh can have a quadratic growth curve in discovery traffic with the number of nodes, potentially increasing network load, especially in very large deployments [66], [67].

3. **Network Topology and Flexibility:** Zenoh accommodates a varied number of network topologies with low latency without imposing a particular topology – peer to peer, mesh, routed, brokered, etc [64]. DDS supports decentralized real-time communication, but it leans heavily on discovery and QoS configuration that are less flexible and possibly confusing, especially in heterogeneous or dynamic networking scenarios [68].

4. **Transport and Environment Adaptability:** Zenoh adopts a much wider range of transport layers (TCP/IP, UDP, Bluetooth, Serial, CANbus, WebSockets) than DDS, which provides a high degree of adaptability from cloud data centers to embedded and edge devices [64]. In comparison, DDS is focused largely on TCP and UDP transports and is commonly used in safety-critical areas like aerospace and automotive, but it is hard to imagine how DDS can scale beyond that due to its 22 QoS policies, which is the area where DDS shines [68].

5. **Real World Application Results:** Zenoh outperforms DDS on Wi-Fi and 4G connections, with lower drift error over mobile robot trajectory tests, while DDS (CycloneDDS) outperformed Zenoh under Ethernet in controlled situations [69].

This indicates that Zenoh is highly suited for distributed, wireless, and resource-constrained deployments.

In conclusion, Zenoh provides strong performance, flexibility, and adaptability for modern edge-to-cloud and distributed IoT systems, while DDS is still a reliable option for safety-critical missions and highly regulated applications when fine-grain QoS control and a willingness to work within an established industry ecosystem are considered.

2.7 Requirements for Data Abstraction using Kuksa Databroker and VSS

2.7.1 Introduction

Sensor data abstraction in SDVs enables interoperability, modularity, and scalability across heterogeneous sensor sources. In the previous sections, SDV concepts, middleware standards, and mixed-criticality-related principles were discussed. This section serves as a bridge from theory to the requirements for implementation of a proof-of-concept.

It is the underlying heterogeneity of the sensors that gives rise to the need for an abstraction layer: sensors generate raw data in different formats and thus require custom processing. Eclipse Kuksa Databroker, along with the VSS strives to provide a powerful solution. Kuksa Databroker is an open-source in-vehicle data broker that grants access to sensor values via a secure and standardized interface, whereas VSS provides a protocol-agnostic data model to ensure semantic consistency for application and platform interchange [26], [27].

This section enumerates the high-level requirements for the integration of Kuksa Databroker and VSS into a functional abstraction system. The requirements are drawn immediately from the aims of our implemented prototype, which transforms raw sensor-data signals into VSS-compliant signals in real-time, serving as a basis for standard-based interoperability of applications throughout the vehicle ecosystem.

The main points of this requirements definition include:

- The importance of consistent signal naming using the VSS hierarchy.
- The requirement for real-time signal update limitations suitable for ADAS-level timing.
- Communication mechanisms between the DDS, the Kuksa data broker, and other instantiations.
- Abstraction mechanisms to mitigate differences in sensor hardware and instantiations under a single common data model.

- Consideration for scalability, extensibility, and signal-level security.

The significance of these requirements is the transition from a conceptual architecture to an operational one. Further, they enable the groundwork for evaluation and testing.

2.7.2 Need for Standardized Signal Mapping

In a multi-sensor environment that is typical of modern SDVs, raw data from different devices must be abstracted into a common semantic structure to allow uniform data handling, interoperability, and simplified application logic. VSS offers a hierarchical signal model that facilitates this standardization by representing signals in a tree-like structure with path-like syntax (e.g., `Vehicle.Camera.RPi.Image`, `Vehicle.Camera.Laptop.ObjectCount`) [26], [30].

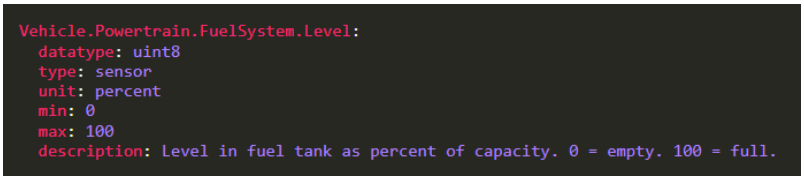

Figure 13. Example of VSS Signal Definition in `.vspec` Format [30]

An example of VSS signal definition in `.vspec` format is shown in Figure 13.

The data abstraction system requires clear signal definitions to be created and registered using VSS. This is done through:

- Authoring `.vspec` files in YAML format, which define signal properties like name, data type, units, and description.
- Translating `.vspec` files to machine-readable formats such as JSON using `vss-tools`, which can be loaded into the Kuksa Databroker [30].

Each sensor's output should be a directly corresponding, uniquely identifying, and semantically meaningful VSS path. For example:

- Data from an encoded binary sensor can be assigned to `Vehicle.Camera.RPi.Image` with a string data type.
- Number of objects as quantitative variables can be assigned to `Vehicle.Camera.Laptop.ObjectCount` with data types such as `int32` [30].

These signal names and formats allow downstream applications to register and interpret sensor data in a uniform manner, without having to worry about the specific sensor hardware or setting up different communication protocols.

Key requirements:

- All sensor outputs will need to be mapped to a VSS-compliant path while using VSS naming conventions.
- Signal definitions will need to be in a standardized scheme that is defined in `.vspec` format.
- Each VSS path represents a known semantic concept (for example: `Value`, `ObjectCount`, `Confidence`).
- The application will need to support the conversion of raw DDS messages into VSS signals [30].

By ensuring all signals will have the same structural and semantic guidelines, this layer of standardization becomes the foundation for modular and scalable vehicle software architecture.

2.7.3 Real-Time Signal Publishing Requirements

In connected vehicle systems, real-time signal publishing is a requirement to ensure accurate, timely, and secure vehicle data is sent to edge and cloud platforms. When using the Kuksa Databroker framework with the VSS low low-latency data transmission across heterogeneous vehicular networks is key. Real-time signals (e.g. engine RPM, speed, steering angle, or more complex streams like camera frames) must meet specific requirements in terms of update frequency, type safety and delivery guarantees. Some of the key requirements are mentioned in detail below:

- **R-RTSP-1 Required Update Frequencies and Latencies:**
  To provide reliable control, important signals must be published at a frequency of 10–100 Hz. To ensure end-to-end latency remains less than 100 ms and hence matching human response times, it is important to consider that latencies greater than this boundary may compromise system responsiveness and safety. Latency in real-time systems is an important consideration. High data rates from high-frequency sensors like cameras and LiDAR used for perception with robots require real-time and efficient offload [70].

- **R-RTSP-2 Data Types, Serialization, and QoS Considerations:**
  Both DDS and Kuksa Databroker use the standard data types defined by VSS, which are integers, floats, booleans, and strings. Kuksa Databroker has enforcement of data types, and units of measurement through its data schema and uses JSON over REST and Protobuf over gRPC to serialize data [30]. DDS utilizes IDL-defined data structures and serializes data via binary CDR, leaving a low overhead and retaining vendor interoperability [71]. Real-time performance is enabled through sampling and event rate management in addition to many configurable QoS polices, including Deadline, LatencyBudget, and Reliability, providing application-specific controls over rate of data, delay of data, and data delivery guarantees [72]. While DDS does support the fast and deterministic ingestion of data (for example, 10–100 Hz with <100 ms latency),

Kuksa Databroker offers VSS-compliance and acts a broker, allowing abstraction and standardization of signal access for SDV applications [70].

2.7.4 Middleware-to-Kuksa Integration Requirements

Transport Bridge Service

A specialized DDS provider micro-service named Kuksa DDS provider links the DDS domain and the Eclipse Kuksa VSS Databroker [73]. Conversely, this Kuksa DDS provider can subscribe to Zenoh key-value topics in Zenoh's lightweight pub/sub/query interface. Zenoh is particularly helpful when used in edge or low-bandwidth environments where it's very low overhead and flexible discovery mechanisms can provide efficient integration [59]. The Kuksa DDS provider service can be deployed as an OCI container or a systemd native service, and subscribes to one or more DDS topics and retains key QoS policies (reliability, deadline, history) and forwards the payloads over Kuksa's gRPC streaming API. Required TLS 1.3 encryption for gRPC and DDS Secure-RTPS mutual authentication [74] ensure confidentiality, while a watchdog heartbeat exposed on a dedicated diagnostic signal (e.g., `Vehicle.Diagnostics.DDSBridgeAlive`) provide functional safety monitoring.

Mapping Layer Service

Semantic harmonization is achieved via a declarative YAML (or JSON) mapping file in which each record is of the form {`topic, ddsField | zenohKey, vssPath, scale, unit`}. The bridge hot-loads this file at the run-time, automatically applies type/unit conversions, and then validates each entry against the active VSS release [75]. Mapping files can include a semantic version (e.g., `mapVersion: veh-0.3.1`) to simplify safe OTA updates, which is an internal convention and not part of the VSS standard.

The transport bridge and mapping layer services together provide a secure, vendor agnostic, and future proof data path from heterogeneous DDS or Zenoh publishers to the standardized VSS tree, meeting the time, scale and inter-operability requirements defined in our project requirements.

2.7.5 Hardware Abstraction Across Sensors

Modern SDV incorporate heterogeneous sensing devices- cameras, radar, LiDAR, GNSS receivers; each delivered by different vendors and attached via diverse physical interfaces (USB, MIPI-CSI, Ethernet, CAN-FD). To prevent this hardware diversity from leaking into application code, the **sensor abstraction layer (SAL)** shall provide a uniform data model and publication mechanism based on Kuksa's Databroker and the VSS. Below are some hardware abstraction requirements mentioned:

- **R-HWAB-1 Interface Decoupling:** The SAL shall isolate driver-level differences (e.g., V4L2 ioctls, vendor SDKs) from higher layers by converting

native frames or messages into canonical datatypes (e.g., base64-encoded JPEG for images, protobuf PointCloud for LiDAR) before they enter the Databroker [76].

- **R-HWAB-2 VSS Compliant Naming:** Every abstracted signal shall occur under `Vehicle.Sensor.<Class>.<Instance>.<Signal>` in the VSS tree, reusing the existing VSS categories where possible (for example, Camera, Radar) and creating vendor-neutral leaf names for new sensor classes [75].

- **R-HWAB-3 Dynamic Extensibility:** The SAL shall support hot-plug registration of new sensor providers via the Databroker plugin API without restart, which is needed to permit on-the-fly growth of the VSS tree [76].

- **R-HWAB-4 Metadata Exposure:** Resolution, updateRate, encoding, and health status must be published as sibling VSS attributes (e.g. `Vehicle.Sensor.Camera.FrontLeft.FrameRate`) so that downstream components can automatically adapt processing pipelines [77].

With these expectations enforced, the architecture will have vendor agnostic sensor representation, and application-level code has constant access to any existing or future sensor through the same Kuksa API and using the same VSS schema.

2.7.6 Scalability and Extensibility

SDV evolve throughout their lifecycle, so the data pipeline must scale in both breadth (more signals) and depth (richer formats). The following requirements ensure the Kuksa + VSS stack grows without downtime or schema fragmentation.

- **R-SCAL-1 Dynamic Tree Growth:** The Databroker shall accept runtime registration of additional VSS leaves via the provider-plugin API [76], with no service restart, and support large signal counts while maintaining query latencies within the performance targets defined in **R-RTSP.**

- **R-SCAL-2 Hot-Reload Mapping:** Field-to-VSS mapping files shall be reloadable on the fly; incompatible entries MUST be rejected with a validation error published under `/Diagnostics/Mapping/*` [75].

- **R-SCAL-3 Topic Flexibility:** The Middleware-to-Kuksa bridge (DDS or Zenoh) shall support wild-card topic subscriptions and new DDS data types using OMG DDS-XTypes' type evolution rules [78].

- **R-SCAL-4 Schema Versioning:** Every new signal batch shall specify `vspecVersion` and `providerId` metadata so subscribers can negotiate

backward compatibility; VISS v2 wildcard queries (`Vehicle.Sensor.*`) enable graceful degradation when a path is missing [79].

- **R-SCAL-4b Schema Agnosticism (Zenoh):** In Zenoh mode, the bridge is to allow signal ingestion based on key-patterns (e.g., `/sensor/cam/front/*`) without requiring the specification of a schema [80]. This means that new signal paths can be added without the restriction of having to first register the schema; this is important for bandwidth constrained or plug-and-play ECUs enabling lightweight and rapid deployment.

- **R-SCAL-5 Performance Baseline:** The data path (DDS → Databroker → VISS v2) shall sustain 5 000 updates/s with < 100 ms p95 latency in a single ECU benchmark, as demonstrated in Kuksa's reference load-test scripts [81]. These performance values are based on Kuksa's official performance benchmarks [81] and are used here just for defining the requirement.

Together, these rules guarantee that new sensor classes, ECU domains, or data formats can be onboarded with zero-downtime and without breaking existing consumers.

2.7.7 Security and Access Control

Safeguarding the confidentiality, integrity, and availability of in vehicle data is essential when the sensor stack starts publishing personal or safety related information. The security controls comprise transport encryption, Role-Based Access Control (RBAC), and key management in both DDS layer and Kuksa layer.

- **R-SEC-1 Role-Based Permissions:** The Kuksa Databroker shall enforce JWT-based RBAC using the `kuksa.permissions` claim; tokens may list wildcard paths (e.g., `read:Vehicle.Sensor.*`) but providers must request only the minimum scope [76].

- **R-SEC-2 Mutual TLS on gRPC:** All Databroker gRPC endpoints shall use TLS 1.3 with server authentication; mutual TLS is required for in-vehicle producers that push data from untrusted domains [82].

- **R-SEC-3 Selective DDS Encryption:** Topics flagged as *critical* in the mapping file (e.g., containing biometric or location data) shall enable DDS Secure-RTPS with AES-GCM encryption and mutual authentication as specified in the DDS Security v1.1 standard [74]. Non-critical high-bandwidth topics may remain unencrypted to save CPU bandwidth.

- **R-SEC-4 Key & Certificate Lifecycle:** All X.509 certificates used for TLS or DDS security shall be rotated at least annually and distributed via an

Uptane-compatible OTA pipeline [83]. Private keys must reside in a hardware protected store (HSM or TPM) when available.

These requirements ensure a layered defence model: data is encrypted in motion, access is limited by explicit roles, and cryptographic material is managed safely throughout the vehicle's lifetime.

2.7.8 Data Logging

End-to-end visibility is required to verify that every message published on DDS reaches its VSS destination within the latency envelope and without silent data loss. The following requirements define the observability layer:

- **R-LOG-1 End-to-End Correlation:** Each DDS sample shall carry a W3C *traceparent* ID; the DDS-to-Kuksa bridge shall propagate that ID as gRPC metadata so a single trace spans DDS → Databroker → VISS clients [84].

- **R-LOG-2 Structured JSON Logs:** Producers, bridge, and Databroker shall emit JSON logs conforming to OpenTelemetry semantic conventions (`timestamp`, `traceId`, `vssPath`, `latencyMs`, `level`, `errorCode`) [85].

2.7.9 Summary of High-Level Requirements for data abstraction

The data-abstraction layer is identified by seven complementary requirement families:

1. **Signal Mapping**: Each output of a sensor maps to a uniquely defined output VSS path in `.vspec` and ensures semantic consistency [26], [30].
2. **Real-Time Signal Publishing**: Time-critical signals are published and forwarded while retaining data integrity characteristics in the presence of dynamic load [70], [71], [72].
3. **Middleware-to-Kuksa Bridge:** A secure, QoS-aware service takes the DDS payload and transforms it into VSS updates and publishes those updates via gRPC for uniform consumer consumption [59], [73], [74], [75].
4. **Hardware Abstraction:** The driver and interface diversity are not surfaced in the sensors' publishers; the sensors have providers that support hot-pluggable drivers and interfaces that publishes the sensor data and metadata in a common format [75], [76], [77].
5. **Scalability and Extensibility:** The VSS tree may change and grow at run-time with the hot-reload of the mapping and the negotiation of schemas with no downtime [75], [76], [78], [79], [80], [81].
6. **Security and Access Controls:** Role-based permissions, encrypted transport, user-selective topic protection and managed certificates all occur when data is in motion [74], [76], [82], [83].

7. **Data Logging:** End-to-end trace IDs and structured logging provides observability of the publish-subscribe in the Kuksa semantics, with export hooks into standards-based monitoring stacks [84], [85].

Together these requirements keep the Kuksa + VSS stack interoperable, performant, secure and maintainable as newer sensors and services join the vehicle ecosystem (applicable to both DDS and Zenoh). Some of these requirements are validated in evaluation phase. All these defined requirements remain targets for future extension and evaluation.

3 Concept

This section presents the conceptual framework developed for sensor data abstraction in SDVs, integrating DDS middleware with the VSS-based Kuksa Databroker. It begins by outlining the design principles followed during the architectural planning, followed by a detailed explanation of the system architecture, signal abstraction layer, and security considerations. The extensibility of the framework and its alignment with modular SDV design are also addressed, concluding with a summary of the overall concept.

3.1 Conceptual Framework Integrating DDS and VSS

3.1.1 Design Principles

The integration of DDS and VSS in SDV is guided by the principle of decoupling transport from semantics. This basically refers to the method of data handling by DDS or Zenoh and the representation of the data (defined by VSS), thus allowing multiple vehicle systems to be flexible and interoperable. DDS provides reliable and real-time communication capabilities, and it is also well suited for messaging high-bandwidth sensor data across heterogeneous ECUs [78]. VSS, on the other hand, provides a common data model for semantic consistency and abstraction across sensor types and vendors [30]. The design distinguishes between the transport layer guarantees provided by DDS and the standardized access to vehicle data provided by Kuksa Databroker (based on VSS) [73].

The decoupled design addresses several contemporary challenges in SDV architectures, including different sensor protocols, no semantic interoperability, and more constrained software stacks. When DDS concerns itself with message delivery, and VSS deals with meaning, it is possible to achieve modular, scalable, vendor-neutral integration without worrying about the low-level details of the sensors and how they transmit DDS messages. The abstraction layer allows developers to work with a common vehicle data model without pinpointing the low-level intricacies of the sensor themselves or the underlying DDS message format.

3.1.2 System Architecture

The architecture of the integration framework between DDS to VSS is structured with a conceptual layered architecture in a three-tier model: (1) Data Producers, (2) DDS Middleware, and (3) VSS Abstraction through Kuksa Databroker. Data is taken from various sensors (cameras, lidar, etc.) as raw data and published as messages to DDS topic/Zenoh key. The framework supports two options of middleware layers (i.e., DDS/Zenoh) to support either scenario based on deployment needs. Although DDS provides reliable distributed message notification in real-time environments with QoS-based communications, Zenoh is intended to support lightweight communications,

such as from sensor edge networks, low bandwidth links, or with highly distributed systems in mind. These publishers are designed to run completely independently, and they only provide the most efficient transport mechanism, with no need to understand the semantic meaning behind their output. The basic architecture is shown in Figure 14.

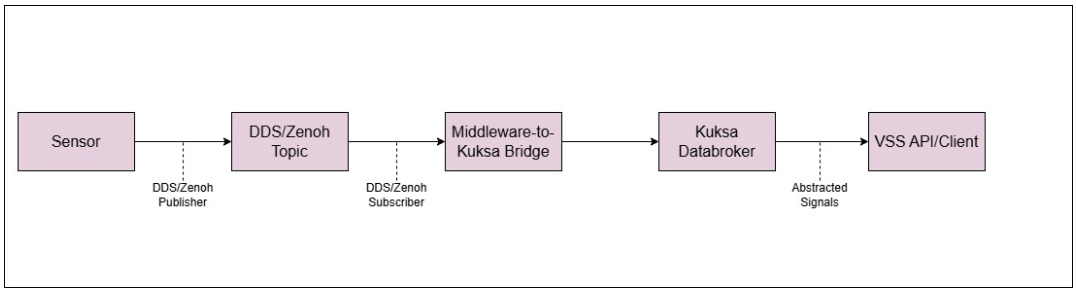

Figure 14. Sensor-to-VSS Signal Abstraction Pipeline using DDS/Zenoh and Kuksa Databroker

A bridge service subscribes to DDS or Zenoh keys, and takes the incoming messages and decodes them using predefined schema mappings. It then pushes semantically rich update to the Kuksa Databroker via its gRPC API. Kuksa Databroker is the final layer of abstraction and takes these messages and creates structured VSS signals using a tree-based hierarchy. This approach allows application developers and consumers to consume real-time standardized vehicle data without exposure to transport layer specifics.

Each component of the system is stateless by-design, except for the Kuksa Databroker, which holds the current signal values and metadata. The decoupling of transport logic (DDS or Zenoh) from semantic abstraction (VSS via Kuksa) renders the system highly flexible, maintainable, and interoperable.

3.1.3 Signal Abstraction Layer

The signal abstraction layer, comprising Middleware-to-Kuksa bridge and the Kuksa Databroker, is responsible for translating transport level messages to semantically rich VSS signals in the DDS/Zenoh-to-VSS abstraction, uses a declarative mapping file (typically in YAML and/or JSON format) which explains how the fields in DDS/Zenoh message will be extracted and put into specific paths in the VSS tree. Each mapping consists of the DDS topic/Zenoh key name, the field(s) of the DDS/Zenoh message to be extracted, the corresponding VSS path, any scaling or unit conversions (optional), and type information.

The Middleware-to-Kuksa bridge service will use this mapping file to parse incoming DDS/Zenoh messages and subsequently push signal updates to the Kuksa Databroker. During this conversion of incoming DDS/Zenoh message types to the VSS schemas, the bridge service validates the data structure according to the current VSS schema, and assures that only valid and expected updates are processed. In the case of mismatching updates (e.g., missing fields, wrong type), error handling mechanisms

trigger either fallback logging, or diagnosis signal updates (e.g., `Vehicle.Diagnostics.MappingError`).

For instance, the DDS message from topic `/camera/rpi_cam` with the field `frame_b64` could be mapped to the VSS signal `Vehicle.Camera.RPi.Image` and the field `object_count` to `Vehicle.Camera.RPi.ObjectCount`. This bridging layer connects raw sensor data and interpreted sensor data in the VSS hierarchy.

This bridging layer is the only access point to the semantics without needing to deal with the complexity of the low-level data structures, so consumers can consistently access the data even though the data may originally come from different sensor hardware and/or transport formats.

3.1.4 Security & Access Design

Security and access control are very important components of the DDS/Zenoh-to-VSS abstraction framework. It ensures that only authorized entities can interact with vehicle signals and in turn the sensitive data remains protected in transit. The security model encompasses both the DDS/Zenoh and Kuksa Databroker layers, enforcing encryption, authentication, and role-based access throughout the pipeline.

At the DDS layer, Secure-RTPS based on the DDS-serenity framework, is used to enable end-to-end encryption and mutual authentication between publishers and subscribers. Topics that contain sensitive data, such as location, identity or camera feeds can be flagged to be encrypted with AES-GCM [74]. AES-GCM is Advanced Encryption Standard in Galois/Counter Mode and it protects our DDS topics by ensuring the data cannot be read or tampered with by unauthorized participants. In addition to providing support for encryption, DDS also provides access control policies which enable us to set which nodes can publish or subscribe to certain topics.

At the Kuksa Databroker, security is managed using JWT-based RBAC [86]. Every client accessing the Databroker through gRPC must provide a valid token with explicitly defined permissions, like `read:Vehicle.Camera.*` or `write:Vehicle.Sensor.LiDAR.*`, enabling strict control over signal access within the VSS tree. This security mechanism is bypassed during the prototype evaluation by running the Kuksa Databroker in `--insecure` mode (i.e., in development mode).

In addition, all communication with the Kuksa Databroker is done over TLS 1.3 to ensure transport-layer security. The process for certificate lifecycle management (including rotation and revocation) is done using OTA-compatible processes, and secrets are kept within a secure hardware-backed store, where supported.

With this layered defence model, it is possible to keep data secure and ensure that access is tightly controlled from the sensor to the signal consumer.

3.1.5 Extensibility Model

As SDVs advance, the framework of underlying data abstraction will explicitly support dynamic scaling and adapting to new signals, ECUs, and sensor types, without any disruptions in the entire system. The DDS/Zenoh-to-VSS architecture is specifically built for extensibility, so that the VSS signal tree can grow dynamically in real-time and new data sources can onboard seamlessly.

At the bridge layer, mapping files can be reloaded during runtime, enabling engineers to change topic-to-signal bindings without restarting the service. This can be achieved by loading the mapping file (e.g., YAML or JSON) inside the Middleware-to-Kuksa bridge's message handler, so the file is re-read each time when a new message is received. Therefore, it is possible to deploy OTA signal changes or quickly iterate in development. Validation will not accept any incompatible mappings, supported by the active VSS schema, and errors will be stored with diagnostic signals.

Kuksa Databroker also supports run-time registration of new VSS paths, through its plugin API. Signals you add this way can immediately be queried and written to, through both the VISS and gRPC interfaces. Further, every update you provide can be paired with a vspecVersion and providerId for backward compatibility and traceability when used in a multi-ECU configuration. However, this capability will not be implemented in this prototype. Instead, required new signals will be added statically to the `.vspec` tree before the runtime.

On the DDS side, extensibility is provided via DDS-XTypes, which supports type evolution (e.g., adding fields or new topics) without breaking existing publishers or subscribers. This assures that heterogeneous nodes in the vehicle, each evolving at its own pace, can continue to interoperate reliably.

Thus, architecture allows for capabilities to be added to the system without architectural refactoring and without service or downtime, providing flexibility for rapidly evolving requirements inherent in SDVs.

3.1.6 Concept Summary

The proposed conceptual framework integrates reliable data transport (DDS/Zenoh) with a semantic abstraction (VSS via Kuksa Databroker) which offers a flexible and extensible approach to vehicle signal representation. The complete end-to-end data flow contains several decoupled, yet co-ordinated segments, each of which serves a different function in the data pipeline.

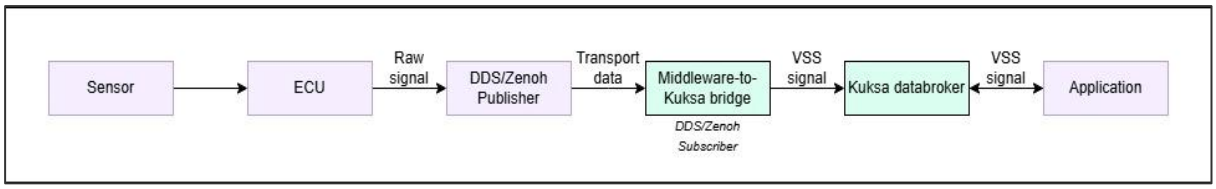

Figure 15. Implementation-based Signal flow from Sensor to Application

The prototype's basic signal flow concept

Signal comes from the sensor-layer; the layer where the sensor such as camera module produces a raw output. The sensor which is connected to an ECU produces a raw output, then immediately publishes that raw output to DDS topics or Zenoh keys using a publisher/subscriber model. DDS or Zenoh transfers all the raw sensor signal reliably and with the assurance of real-time delivery, and needed semantic abstraction (VSS), so that sensor data can simply be streamed to any distributed ECU unit type.

Figure 15 above represents basic idea of this concept and signal flow. The idea is based on the conceptual design in [76], while showing our system's implementation-specific blocks. Middleware-to-Kuksa bridge subscribing to relevant DDS topics or Zenoh keys, converts the raw signals into standardized VSS signals. The bridge pushes the converted signals to the Kuksa Databroker 's gRPC interface. Once entered the Kuksa system, the signals become incorporated into a tree-based, queryable structure defined by the VSS model. This allows for external consumers, such as diagnostics tools, dashboard applications, or cloud services, to get direct access to the signals in a consistent, secure, and semantically meaningful form through either gRPC or VISS API calls.

This layered approach, from DDS/Zenoh to Middleware-to-Kuksa bridge and the Kuksa Databroker, provides a clear separation of real-time data delivery, semantic translation, and access control. Each stage can evolve independently from adjacent layers, while keeping an end-to-end visibility and control capability over the signal flow.

This phase concludes our conceptual design. The detailed conceptual architecture, implementation of this architecture with the sensor interfacing, bridge deployment, and full signal lifecycle validation will be covered in the next Prototype Demonstration phase. It will also incorporate both DDS and Zenoh-based middleware pipelines.

4 Prototype Description

This section describes the prototype developed to demonstrate the practical implementation of the conceptual framework. It begins with the motivation and scope of the prototype, followed by the system architecture detailing the key layers involved: from sensors and middleware to signal abstraction. The signal flow, working principles, and component interactions are explained using a block diagram and layered breakdown, providing a complete view of the end-to-end data processing pipeline.

4.1 Motivation and Scope of the Prototype Demonstration Phase

The prototype demonstration phase implements the prototype based on the research on middleware-driven sensor data abstraction in SDVs. After identifying the requirements for the data abstraction and designing the basic conceptual framework in earlier phases, this stage shifts from theory to hands-on prototype implementation.

The goal is to establish a working prototype that validates the abstraction of sensor data from heterogeneous camera sources (Raspberry Pi camera and USB camera) using middleware technologies (e.g., DDS and Zenoh) and publishing into a unified VSS structure via Kuksa Databroker. This prototype bridges the gap between raw sensor data and standard APIs that facilitate downstream applications like object detection and ADAS.

This report documents the complete implementation pipeline, including:

1. Complete description of the whole system block diagram.
2. The pipeline-wise development of the Raspberry Pi and Laptop subsystems.
3. Step-by-step integration with DDS, Zenoh, Kuksa Databroker, and signal mapping into the VSS.
4. Testing methodologies and performance evaluation.
5. The record of the all the capabilities implemented from the defined requirements for data abstraction using Kuksa Databroker and VSS.
6. Practical issues which were encountered during the development and how they were solved.
7. Practical outlook of the current limitations, including clearly outlined future scope.

By including an authentic, thoroughly documented prototype, this report aims to serve as a practical reference for students and developers working with middleware abstraction in SDVs.

4.2 Concept and System Architecture

4.2.1 Block Diagram

The block diagram of the system architecture for this prototype concept is represented in the Figure 16.

The diagram shows the complete dual-pipeline middleware-based data abstraction layer designed for SDVs. The system has two main pipelines: Raspberry Pi (RPi) and Laptop.

On the Raspberry Pi, PiCamera captures real-time frames, which are then encoded into JPEG format and published as Fast-DDS publisher in C++. Fast-DDS is a C++ implementation of the DDS which is used in this prototype. These messages are then received by a Fast-DDS subscriber which decodes them and forwards the data to the Kuksa Databroker as base64-encoded VSS signals via Kuksa gRPC client.

On the Laptop side, the real-time frames are captured with USB webcam. These frames are transmitted using a Zenoh publisher in Python. A Zenoh subscriber then writes the received data to the Kuksa Databroker in the base64 format, mapped to corresponding VSS paths.

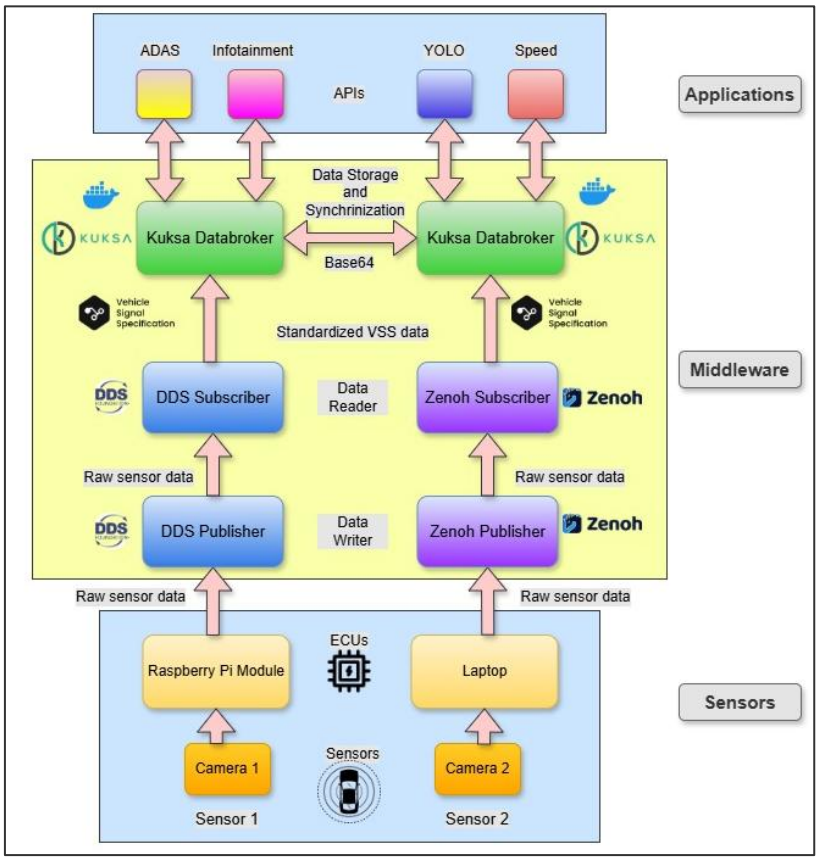

Figure 16. Middleware-Based Architecture for the Sensor Data Abstraction Prototype

Both these pipelines are completely decoupled and separated. However, both pipelines are intended to generate the similar VSS signals into its own Kuksa Databroker following the VSS schema. This approach supports consistent and application-ready signals, regardless of the hardware source.

The Kuksa Databroker is deployed with Docker on both the Raspberry and the Laptop. This allows consistent configuration and portability between the two computing platforms with rapid deployment as well. Docker is advantageous because it allows gRPC-based communication, isolated runtimes, and easy integration with the CLI client (which in each case is also running Docker). In summary, running Docker consistently on both the pipelines speeds up development and testing because discrepancies due to the environment differences.

4.2.2 Description of the Layers (Sensors, Middleware, Applications)

There are three different layers in the block diagram of the prototype in Figure 16. Middleware-Based Architecture for the Sensor Data Abstraction Prototype. Each layer is described briefly below:

1. **Sensor Layer:**
    - Raspberry Pi Camera connected to a Raspberry Pi module (PiCamera2 library) captures the raw real-time camera frames.
    - USB Camera connected to a Laptop (OpenCV) captures the raw real-time camera frames.

2. **Middleware Layer:**
    - **On Raspberry Pi:** A Fast-DDS Publisher sends JPEG frames with Fast-DDS. These published frames are received by another Fast-DDS Subscriber and the frames are written to a shared file. A Python script is then used to read this file, encode it to base64, and then send to Kuksa Databroker using gRPC.
    - **On Laptop:** A Zenoh-based Python publisher publishes JPEGs. A Zenoh subscriber reads these published frames and writes the frames to Kuksa just like the Raspberry Pi.
    - **Kuksa Databroker:** Docker-based Kuksa Databroker in both the pipelines receives these frames and processes VSS-compliant signals.
    - Also, these VSS-compliant signals from both the Kuksa Databrokers (Raspberry Pi and Laptop) are synchronized with shared VSS paths and continuous data exchange. This ensures consistency in the signal values like image data and detected object count, by allowing both the systems to maintain the each other's real-time data.

3. **Application Layer:**
    - Applications, such as object detection (YOLO), use Python scripts to receive image signals from Kuksa Databroker with gRPC.

- After object detection, the same script then writes the detection results (e.g., ObjectCount) back to Kuksa using gRPC.
- This process allows applications with the ability to consume and contribute standardized VSS data through the Kuksa API.

4.2.3 Signal Flow and Working Principle

The end-to-end flow is outlined as follows:

1. **Camera Capture:**
    - Each camera connected to the ECU (Raspberry Pi and Laptop) captures the frame data at discrete time intervals.
2. **Encoding & Middleware Transport:**
    - The image is encoded as a JPEG and published using Fast-DDS (Raspberry Pi) and Zenoh (Laptop) publisher.
3. **Subscriber Ingestion:**
    - Both Fast-DDS and Zenoh have a subscriber. This subscriber receives the JPEG file and stores it locally.
4. **Kuksa VSS integration:**
    - A Python script reads this locally stored JPEG file. The script then encodes it in base64, and sends it to the Kuksa Databroker with the gRPC client interface.
    - Each frame is linked to a specific VSS signal path like `Vehicle.Camera.RPi.Image`
5. **Application Usages:**
    - The VSS signal values can be read by any gRPC-compliant client and used in other ADAS-like applications like object detection.

This middleware-driven, layered design abstracts hardware dependencies, allows modular expansion, and demonstrates vendor-neutral abstractions in practice.

5 Prototype Implementation

This section details the implementation of the middleware-based prototype for sensor data abstraction using VSS and Kuksa Databroker. It covers the complete setup on both Raspberry Pi and Laptop platforms, including hardware, software components, directory structure, and middleware configurations. Key implementation steps such as camera data publishing, middleware transport, and VSS signal conversion are described for each pipeline. The section also explains how the signal mapping was handled in real time and concludes with the output applications including base64 conversion and object detection integration.

5.1 Raspberry Pi Pipeline Implementation

The Raspberry Pi pipeline is responsible for obtaining raw camera frames from the PiCamera module, encoding them into the JPEG file format, and sending them as VSS compliant signals to the Kuksa Databroker. This pipeline has both C++ and Python components, using Fast-DDS for transport and gRPC for the Kuksa aspects. This section describes the step-by-step procedure for implementing this pipeline from hardware setup to signal publication.

5.1.1 Hardware and Software Components

- **Hardware:** Raspberry Pi 4 Model B, Raspberry Pi Camera Module V2
- **OS:** Raspberry Pi OS (64-bit, Debian-based) [87]
- **Middleware:** Fast-DDS (C++), Kuksa Databroker (Docker), Python gRPC client
- **Dependencies:** OpenCV, libjpeg, FastDDS libraries, PiCamera2 library
- **Access and Monitoring:** Use RealVNC Viewer to remotely access the graphical interface of the Raspberry Pi from the Laptop. This is a key advantage when testing out camera output, executing terminal commands, and debugging during the execution of the pipeline without needing a monitor or keyboard for the Raspberry Pi.

5.1.2 Directory Structure and Detailed Setup

The implementation is organized as follows (Figure 17 shows the complete Raspberry Pi pipeline directory structure):

```
~/rpi_pipeline/
├── dds_rpi_publisher/          # Fast-DDS publisher for sending JPEG frames
│   ├── main.cpp
│   ├── RawImageMsg.idl
│   └── CMakeLists.txt
├── dds_rpi_subscriber/         # Fast-DDS subscriber writing JPEGs to file
│   ├── dds_rpi_subscriber.cpp
│   └── CMakeLists.txt
├── vss_sync/
│   └── rpi_sync.py             # Python script to sync VSS signals with Laptop
├── send_to_kuksa.py            # Python script to read image, write base64 to Kuksa
├── kuksa_venv/                 # Python virtual environment for gRPC client
├── kuksa-databroker/           # Kuksa Databroker repository (Docker setup & metadata)
└── images/                     # Folder for saving received JPEGs
```

Figure 17. Raspberry Pi Pipeline Directory Structure

Setup the Raspberry Pi from a clean installation of Raspberry Pi OS 64-bit. To prepare the Raspberry Pi pipeline, follow the below steps:

5.1.2.1 Basic Tools and Python Environment:

- Update the system via: `sudo apt update && sudo apt upgrade`
- Install Python3 and pip: `sudo apt install python3 python3-pip`
- Virtual environment: `python3 -m venv ~/kuksa_venv && source ~/kuksa_venv/bin/activate`
- Install the essential libraries:
  `pip install numpy opencv-python grpcio kuksa-client protobuf setuptools`

5.1.2.2 Installing and Building Fast-DDS:

For installation of the Fast-DDS, follow the official source build guide [88].

- Begin the setup with installing and building `Colcon`, followed by `Foonathan Memory`, `Fast CDR`, `eProsima Fast-DDS`, and `Fast-DDS-Gen` as per the instructions given in the [88].
  These installation steps are to be carried out in the system's global environment rather than inside the Python virtual environment (venv), as Fast-DDS is a C++ middleware and does not require Python venv support. Clone the Git repository and build using CMake as per [88].
- **Dependencies:** `sudo apt install cmake g++`
- After writing the publisher and subscriber scripts, build them with:

```
cd ~/rpi_pipeline/dds_rpi_publisher     #Fast-DDS Publisher
mkdir build && cd build
cmake .. && make
```

```
cd ~/rpi_pipeline/dds_rpi_subscriber   #Fast-DDS Subscriber
mkdir build && cd build
cmake .. && make
```

5.1.2.3 Fast-DDS Message Definitions and Code Structure:

Create the `RawImageMsg.idl` file, which defines the custom Fast-DDS message structure and payload that describes the image data. This includes the timestamp, image format, and payload, which composes the binary JPEG image.

Use Fast-DDS code generation tool `fastddsgen` to process the IDL file to generate the necessary serialization and Fast-DDS interface code. This tool creates files, including:

- `RawImageMsg.hpp`,
- `RawImageMsgPubSubTypes.hpp`, and
- `RawImageMsgPubSubTypes.cxx`,

which are included in both publisher and subscriber C++ implementations to support Fast-DDS communication.

The publisher (`main.cpp`) is implemented to stream JPEG images into Fast-DDS with the message format which is defined using an `.idl` file, while the subscriber (`dds_rpi_subscriber.cpp`) is implemented to listen on the same topic and write the received JPEG data to a file for later use in the data processing pipeline.

5.1.2.4 Kuksa Databroker Setup (Docker-Based):

Clone the Kuksa Databroker from the official GitHub repository [76]. The steps are as shown below:

```
git clone https://github.com/eclipse-kuksa/kuksa-databroker
```

Install the Docker and then verify the Docker setup with:

```
curl -fsSL https://get.docker.com -o get-docker.sh

sudo sh get-docker.sh

docker run hello-world
```

Launch and verify Kuksa Databroker:

```
docker run -it --rm --name Server --network kuksa ghcr.io/eclipse-
kuksa/kuksa-databroker:main --insecure
```

5.1.2.5 VSS Signal Creation and Mapping Configuration

To publish camera signals into Kuksa Databroker, corresponding VSS signals must be defined and included in the Databroker metadata. This is achieved by modifying the `vss_release_4.0.json` file located in the Kuksa Databroker repository. The signals used in this prototype are:

- `Vehicle.Camera.RPi.Image`
- `Vehicle.Camera.RPi.ObjectCount`
- `Vehicle.Camera.Laptop.Image`
- `Vehicle.Camera.Laptop.ObjectCount`

Add all the signal paths shown above manually to the metadata JSON file before starting the Kuksa Databroker container. If these paths are not added to the metadata, the CLI or client scripts would not have the new paths and would be unable to write or read the signals. This is possible after cloning the Kuksa Databroker repository. To add custom VSS signal:

Step 1: Edit the VSS JSON metadata file:

```
nano ~/kuksa-databroker/data/vss-core/vss_release_4.0.json
```

Step 2: Add custom signal entries under the appropriate branch (e.g., `Vehicle.Camera`). And ensure each entry has the correct datatype and description.

Step 3: Once saved the metadata file, start the Databroker (restart if already running) using Docker with `--metadata` flag pointing to the updated JSON file:

```
docker run --network host \
  -v ~/kuksa-databroker/data/vss-core/vss_release_4.0.json:/data/vss.json \
  ghcr.io/eclipse-kuksa/kuksa-databroker:main \
  --insecure --metadata /data/vss.json
```

5.1.2.6 Kuksa CLI Interface:

The Kuksa CLI tool is used to verify and debug the VSS signal flow. The Kuksa CLI tool is provided as a Docker container, and it offers the capability to verify the published signals within the Databroker in real time. The Kuksa CLI uses the default port which connects to the Databroker (localhost:55555), and it has commands like:

```
get Vehicle.Camera.RPi.Image
get Vehicle.Camera.RPi.ObjectCount
```

This tool is very useful during development to confirm signal values, check publishing scripts, and inspect live data without needing to change or restart application logic.

5.1.2.7 Kuksa Python gRPC Client:

Install the kuksa-client Python package from the official PyPI package [89].

- Install the client inside `kuksa_venv` using:

```
pip install kuksa-client
```

- The Python script `send_to_kuksa.py` uses gRPC from kuksa-client and this script:
  - Monitors the received JPEG image (`/tmp/rpi_image.jpg`)
  - Encodes it in base64 format
  - And sends it to the VSS signal `Vehicle.Camera.RPi.Image` using the `set()` method of the Kuksa gRPC client.

5.1.2.8 Other System Setup:

- Create and use a shared directory (`/tmp`) to pass the data between the Fast-DDS subscriber and the Python script `send_to_kuksa.py`.
- Monitor all the logging and debugging outputs via the terminal.

This comprehensive setup guarantees an end-to-end compatibility between the camera, Fast-DDS transport layer, and the Kuksa Databroker with minimal latency and reliable performance.

5.1.3 Camera to Fast-DDS Publisher (C++ implementation)

Implement a Fast-DDS publisher on the Raspberry Pi by using C++ and the Fast-DDS library to publish real-time JPEG frames from PiCamera to a Fast-DDS topic.

First start with defining a custom Fast-DDS message format in `RawImageMsg.idl` file. This defined structure in the `.idl` file should have fields for a timestamp, image format and a binary array to save a JPEG image. Then with the `fastddsgen` tool, generate the corresponding serialization code. This converts the defined structured message into a binary format which can be transmitted over the Fast-DDS and later reconstructed at the subscriber.

To stream the live frames from the PiCamera, use `libcamera-vid` in MJPEG mode. Extract each of the JPEG frames from this stream by finding the start and end markers. Optionally, decode the image using OpenCV to check correctness. And, then encapsulate the binary image data into a Fast-DDS message.

Publish each Fast-DDS message to the topic `rpi_camera_raw`, and use the default QoS settings. For consistent and efficient transmission, maintain a frame rate of around 4-5 frames per second.

This implementation removes the direct tie of our camera hardware from the rest of the system and utilizes Fast-DDS to stream JPEG images in a platform-agnostic format.

5.1.4 Fast-DDS Subscriber to Kuksa Writer (Python)

Set up a Fast-DDS subscriber, which listens to the topic `rpi_camera_raw`. When a JPEG frame is received, write the binary data to a local file path like `/tmp/rpi_image.jpg`. The local file serves as a bridge between Fast-DDS and the semantic abstraction layer.

Run the Python script `send_to_kuksa.py` to listen for this file. Once it detects a new image, script reads the file, encodes it to a base64 string, and update the Kuksa Databroker with the gRPC client. This signal is written to the VSS path `Vehicle.Camera.RPi.Image` with the `set()` method.

This bridge enables downstream applications to have access to the camera feed as a standardized VSS interface, i.e., an object detection script can read this image signal over gRPC, perform inference, and send back the result (e.g., `ObjectCount`) to Kuksa as another VSS signal (`Vehicle.Camera.RPi.ObjectCount`).

5.2 Laptop Pipeline Implementation

Laptop pipeline also works on the same working principle as Raspberry Pi pipeline i.e., abstract the raw camera frames and convert it into VSS-compliant signals. But, in Laptop pipeline, Zenoh is used instead of Fast-DDS. Zenoh is used as a lightweight and a scalable transport layer. This Python-only integration integrates a USB webcam feed into the same abstraction system and highlights the middleware flexibility.

5.2.1 Hardware and Software Components

- **Hardware:** Laptop (with running native Ubuntu), USB webcam
- **OS:** Ubuntu 22.04 or later
- **Middleware:** Zenoh (Python), Kuksa Databroker (Docker), Python gRPC client
- **Dependencies:** Install OpenCV, NumPy, Ultralytics

5.2.2 Directory Structure and Setup

In many aspects, the Laptop pipeline is same as Raspberry Pi pipeline. But this pipeline is entirely implemented in Python using Zenoh for transport and gRPC client for communication with the Kuksa Databroker.

Figure 18 shows the complete Laptop pipeline directory structure:

```
~/laptop_pipeline/
├── publisher.py                # Zenoh publisher sending webcam frames
├── subscriber.py               # Zenoh subscriber saving image locally
├── send_image_to_kuksa.py      # Encodes and sends image to Kuksa
├── detect_from_kuksa.py        # Runs YOLO on image signal, sends ObjectCount
├── laptop_sync.py              # Python script to sync VSS signals with RPi
├── kuksa_venv/                 # Python virtual environment
├── kuksa-databroker/           # Cloned Kuksa Databroker repository
└── zenoh_share/                # Shared folder for image exchange
```

Figure 18. Laptop Pipeline Directory Structure

Before running any of the scripts, install the necessary software tools and other dependencies as mentioned below:

5.2.2.1 Basic Tools and Python Environment:

- Setup native Linux: Download and boot your Laptop with Linux (Preferably Ubuntu official latest version and dual-boot)
- Update the system via: `sudo apt update && sudo apt upgrade`
- Install Python3 and pip: `sudo apt install python3 python3-pip`
- Virtual environment: `python3 -m venv ~/kuksa_venv && source ~/kuksa_venv/bin/activate`
- Install the essential libraries: `pip install opencv-python kuksa-client grpcio eclipse-zenoh numpy ultralytics`

5.2.2.2 Installing Zenoh:

Instead of Fast-DDS, Zenoh is selected for the Laptop pipeline because of its lightweight nature and support for pub-sub patterns over constrained networks. Zenoh allows a flexible means of communication between a Zenoh publisher (sending real-time camera frames) and a Zenoh subscriber (receiving the frames) and writing them to disk. To build both Zenoh publisher and subscriber scripts in this pipeline, install the Zenoh Python client library. Refer the official GitHub repository [90]

If not already installed, run:

```
pip install eclipse-zenoh
```

After installation, Zenoh can be used in Python scripts just by importing it like:

```
import zenoh
```

Note on similar installation of components:

The Kuksa databroker setup, VSS signal creation and mapping configuration, Kuksa CLI interface, and Kuksa Python gRPC client installation steps are similar

to the steps described in sections 5.1.2.4 to 5.1.2.7 and to be used here without any modification.

5.2.3 Camera to Zenoh Publisher (Python)

The Python script `publisher.py` captures raw frames from the USB webcam using OpenCV. Each frame is JPEG-encoded and published to a Zenoh key (`demo/cam/image`). In script `publisher.py`, frames are continuously read using `cv2.VideoCapture()`, and each frame is encoded using OpenCV. This encoded frame then sent via `zenoh.put()`. To ensure low-latency, maintain a frame rate of around 4-5 FPS.

The python script `publisher.py` represents the initial stage in Laptop pipeline and it enables real-time camera data abstraction using Zenoh.

5.2.4 Zenoh Subscriber to Kuksa Writer (Python)

The Python script `subscriber.py` subscribes to the Zenoh key `demo/cam/image`. Once a new frame is received, it writes the image data to a shared file (`/home/aniket-barve/laptop_pipeline/zenoh_share/laptop_image.jpg`) for the next process.

Another Python script `send_image_to_kuksa.py`, monitors this shared file path. It encodes the saved JPEG image into base64, and uses the Kuksa gRPC client to send the signal to the `Vehicle.Camera.Laptop.Image` path. This data flow is similar to the Raspberry Pi pipeline.

This flow creates a full pipeline on the Laptop subsystem, from webcam to Zenoh transport to Kuksa VSS signal abstraction, which is capable of processing images in real-time and synchronized sharing of visual data.

5.3 VSS Signal Mapping and Integration Overview

5.3.1 Real-Time Updates and Mapping File Setup

The detailed steps for creating the custom VSS signals and vspec JSON file editing are described in section 5.1.2.5. This section now focuses on how those custom signals are integrated across both the pipelines and maintained consistently during the runtime.

The four custom VSS signals – `Vehicle.Camera.RPi.Image`, `Vehicle.Camera.RPi.ObjectCount`, `Vehicle.Camera.Laptop.Image`, and `Vehicle.Camera.Laptop.ObjectCount` – are updated in real-time in both pipelines. Both the camera feeds are processed and sent to its respective VSS signal using a Python Kuksa gRPC client script (`send_to_kuksa.py` on Raspberry Pi and `send_image_to_kuksa.py` on Laptop).

The ObjectCount signals are updated similarly by the object detection scripts (`detect_from_kuksa.py`- separate Python scripts for both pipelines with same file name) after inference. The reason for using separate scripts for signal writing is to create a modular approach for updates and debugging. Since all signals have already been pre-declared in the JSON metadata (see 5.1.2.5), no creation of signals at runtime is needed.

The mapping of the signal paths to code logic remains static in this prototype. Each script in this prototype is hardcoded to work with a specific object signal, which ensures proper identification and therefore completely removes uncertainty for each runtime operation. This process is effective for testing environments and small-scale deployments.

Signals are updated whenever a new image frame or detection result is available during execution. The typical refresh rate is dependent upon the pipeline and hardware, but generally falls into a range of 100–250 ms. Here, refresh rate refers to how frequently VSS signals (image and object count) are updated with new input data and not the end-to-end latency of the entire pipeline. This means, refresh rate shows the signal update rate based on camera frame rates and object detection cycles. This ensures VSS values in Kuksa Databroker to always be synchronized with the current input from each subsystem.

5.3.2 Integration Consistency Across Both Pipelines

In order to facilitate consistent and synchronized operation of the Raspberry Pi and Laptop pipelines, the four VSS signals are pre-declared in both JSON metadata files. This allows each system to not only publish its own signals but also read and process and the respective signals from the other pipeline. For example, the object detection script running on the Laptop can read `Vehicle.Camera.RPi.Image` to perform inference on Raspberry Pi frames, and the same goes for the Raspberry Pi being able to access `Vehicle.Camera.Laptop.Image`.

Bidirectional access is possible because of the shared signal naming and identical VSS configurations in each device's Kuksa Databroker instance. Each device has a synchronization script (`rpi_sync.py`, and `laptop_sync.py`) that is running in the background, so that the signals can be copied over to the other pipelines. These synchronization scripts use Kuksa's gRPC client interface to read a value remotely and write it locally so that values stay up-to-date, and visible to each other.

This integration strategy avoids redundant pipelines and supports collaborative processing. The integration design allows signal consumers, such as detection modules, to access the signal in a uniform manner regardless of the source device. The update frequency is synced with the individual pipelines refresh interval, while preserving real-time constraints and maintaining semantic consistency across both the nodes.

5.4 Applications and Output Layer

This section focuses on the usage of the abstracted VSS signals by downstream applications. Specifically, it describes the signal transformation, how YOLO object detection is integrated into applications, and how simulations integrate automotive applications, including ADAS and infotainment applications, which ingest these standardized vehicle signals.

5.4.1 Base64 Signal Conversion

JPEG images delivered by Fast-DDS or Zenoh are written as binary files to shared locations such as `/tmp/rpi_image.jpg` for Raspberry Pi or `/home/aniket-barve/laptop_pipeline/zenoh_share/laptop_image.jpg` for the Laptop. The raw image files cannot be sent over to Kuksa Databroker, as the VSS signal format supports string-based values only. To make it compatible with the Kuksa gRPC client, the images are first encoded into base64 strings in Python. The process of encoding creates a relatively small, text-based version of the binary image. And then the conversion is done using scripts like `send_to_kuksa.py` and `send_image_to_kuksa.py`. These scripts actively listen to the image file, encode the file contents, and send the data to Kuksa using the gRPC client's `set()` method.

This conversion is necessary to maintain the JPEG format within a standard VSS signal to allow downstream applications to decode and reconstruct the image for subsequent analysis. Base64 also guarantees that image data can be transported over network protocols that do not support raw binary payloads.

5.4.2 YOLO-Based Object Detection on VSS Signals

Once the image signals from the camera are exposed in Kuksa Databroker, object detection is performed using the YOLO models (YOLOv8n used on Laptop and YOLOv5n used on Raspberry Pi) that have been implemented using the ultralytics Python package. The application logic is encapsulated in the script `detect_from_kuksa.py`, which takes the standardized image signals using Kuksa's gRPC client. Here each pipeline uses the same name for their own Python script of object detection.

The script continuously reads the image signals from its own camera source- `Vehicle.Camera.Laptop.Image` on the Laptop and `Vehicle.Camera.RPi.Image` on the Raspberry Pi- depending on the device which handles the detection. The script first decodes the base64 image signal back into a regular JPEG using standard Python tools. Then, it sends the image to a pre-trained YOLO model, which checks the image for object detection.

The results of the detections are parsed to retrieve the number of objects detected in the frame. The object count is then written to Kuksa Databroker as another VSS signal:

`Vehicle.Camera.RPi.ObjectCount` or `Vehicle.Camera.Laptop.ObjectCount`, using `set()` from the Kuksa gRPC client.

This implementation has the advantage of operating detections asynchronously from signal ingestion. It also demonstrates the complete signal flow cycle; from hardware abstraction, to semantic VSS signal, to application-level output, confirming the feasibility for real-time ADAS or infotainment applications consuming standardized signals.

5.4.3 Interfacing with ADAS and Infotainment (Simulated APIs)

To demonstrate how this abstracted VSS signals would be ingested by higher-level vehicle functions, this prototype contains simple interfaces that mimic interactions with ADAS and infotainment applications.

In a real automotive system, a lane departure warning, driver monitoring, or infotainment apps would subscribe to VSS standardized signals, provided to them by a Broker. In this prototype, Python scripts act as the Broker, subscribing to the Kuksa Databroker to read image and object count signals and perform object detection in real time. The output signal, such as `Vehicle.Camera.Laptop.ObjectCount`, could be perceived as input to an ADAS module that triggers driver alerts.

This implementation validates a semantic signal flow from the interface to higher-level functions and highlights the modular architecture whereby external applications (simulated APIs) can plug-in without modifying lower-layer pipelines. Future iterations could replace the YOLO detection with real control functions or use a complete infotainment stack to highlight further usability.

6 Evaluation and Testing

This section assesses the prototype's performance in terms of signal interoperability, timing characteristics, and synchronization. It will determine if the system met the requirements defined in the earlier phases, particularly aspects of interoperability and real-time response.

6.1 Signal Interoperability

Four custom VSS signals were declared (`Vehicle.Camera.RPi.Image`, `Vehicle.Camera.RPi.ObjectCount`, `Vehicle.Camera.Laptop.Image`, and `Vehicle.Camera.Laptop.ObjectCount`) in both the Raspberry Pi and Laptop metadata JSON files. This allowed both the pipelines to access and update the data (image and object count) using the common signal names.

Signal synchronization scripts, `rpi_sync.py` and `laptop_sync.py`, allowed each system to pull the values from each other. This enabled a shared view of both the pipelines. Both the scripts fetched the image and object count values from the other device and update them into the local Databroker.

In the final deployment, both Kuksa Databrokers provided a common VSS interface to the application layer. All the four signals were consistently accessed and verified in both the pipelines by using the Kuksa CLI interface as described in section 5.1.2.6. The following commands are used for verifying the signals:

```
get Vehicle.Camera.RPi.Image
get Vehicle.Camera.RPi.ObjectCount
get Vehicle.Camera.Laptop.Image
get Vehicle.Camera.Laptop.ObjectCount
```

Figure 19. Signal Interoperability Check

This showed that signal availability is platform-agnostic. Figure 19 shows the successful interoperability check screenshot. This confirmed the full interoperability between the Raspberry Pi and Laptop pipeline.

6.2 Real-Time Performance

To evaluate the performance in terms of the real-time latency, all the scripts were made to print the timestamp after completion of its operation i.e., publishing image, subscribing it, sending it to Kuksa and object detection result. Also, to track the frames throughout the pipeline, each frame was defined with a unique frame ID. The data was recorded and saved in a CSV file for evaluation purpose. This latency evaluation was limited to the Laptop pipeline. Both the pipelines were started and the data for one minute was saved along with the timestamps. Here, the publisher was publishing the frames at 5 fps rate.

For defining latencies, two key metrics were used:

1. **L1 Latency:** Abstraction layer latency (Zenoh to Kuksa)
   This latency is the time taken to save the image signal in VSS format to Kuksa Databroker after it was published.

2. **L2 Latency:** End-to-end latency (Zenoh to Object detection)
   This latency is the time taken to publish the object detection result of the image frame to Kuksa Databroker after the image frame was published.

Here a sample output and signal flow for Frame ID `2274f0a6` is shown below:

```
[Publisher Timestamp]: 2025-06-16T12:20:02.836675 | Frame ID: 2274f0a6
[Publisher Timestamp]: 2025-06-16T12:20:03.040357 | Frame ID: 3aad5b2c
[Publisher Timestamp]: 2025-06-16T12:20:03.247303 | Frame ID: 15509408
[Publisher Timestamp]: 2025-06-16T12:20:03.451797 | Frame ID: 6a5c5c54
```

Figure 20. Laptop: publisher.py

Figure 20 shows the `publisher.py` script terminal, which shows the frames publishing with the timestamp.

```
[Subscriber Timestamp]: 2025-06-16T12:20:02.837402 | Frame ID: 2274f0a6
[Subscriber Timestamp]: 2025-06-16T12:20:03.041060 | Frame ID: 3aad5b2c
[Subscriber Timestamp]: 2025-06-16T12:20:03.248146 | Frame ID: 15509408
[Subscriber Timestamp]: 2025-06-16T12:20:03.452390 | Frame ID: 6a5c5c54
```

Figure 21. Laptop: subscriber.py

Figure 21 shows the `subscriber.py` script terminal, which shows the frames received and saved in a shared folder with the timestamp.

```
[Latency L1] Zenoh → Kuksa: 0.034s | Frame ID: 2274f0a6
[Send-to-Kuksa Timestamp]: 2025-06-16T12:20:02.871400
[Latency L1] Zenoh → Kuksa: 0.051s | Frame ID: 3aad5b2c
[Send-to-Kuksa Timestamp]: 2025-06-16T12:20:03.091996
```

Figure 22. Laptop: send_image_to_kuksa.py

Figure 22 shows the `send_image_to_kuksa.py` script terminal, which shows the frames sent to Kuksa Databroker in VSS format and the timestamp is printed.

```
0: 480x640 1 person, 1 sink, 4.9ms
Speed: 1.1ms preprocess, 4.9ms inference, 0.7ms postprocess per image at shape (
1, 3, 480, 640)
[Detection Timestamp]: 2025-06-16T12:20:02.879858 | Frame ID: 2274f0a6
[L2 = L1 + Δ] Kuksa → Detection: 0.042s
[Detected Objects]: 2
```

Figure 23. Laptop: detect_from_kuksa.py

Figure 23 shows the `detect_from_kuksa.py` script terminal, which shows the object detection result on the decoded frames and the timestamp is also printed.

After saving the data for one minute, both L1 and L2 latencies were presented in the graphical format. Figure 24 shows the graphical representation of the L1 and L2 latency results.

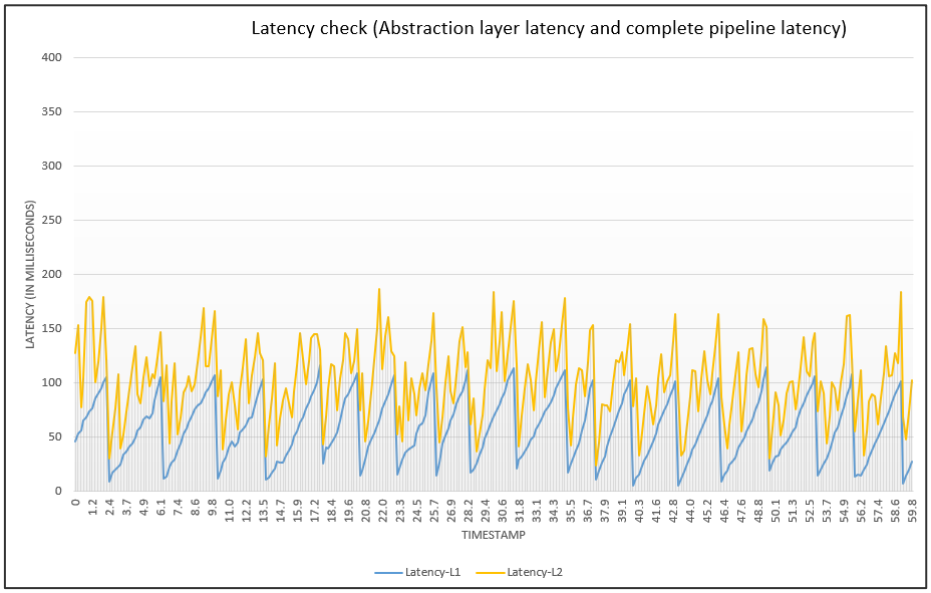

Figure 24. L1 and L2 latency check (Laptop pipeline)

From the CSV logged file, the minimum, maximum and average latency values were extracted and those are shown in Table 1**.** As described, the maximum end-to-end latency was 23 ms, whereas 186 ms latency was maximum end-to-end latency. Overall, the end-to-end latency stayed under 200 ms. Also, our proposed middleware layer (i.e., Zenoh to Kuksa) showed average latency of ~57 ms.

As per author in [91], "*For safety-critical applications like autonomous driving, a sensor-actuator end-to-end latency of 100ms is targeted.*" Logged data for one minute of our prototype showed 102 ms average end-to-end latency which is almost in this range.

Table 1. Latency Values

Range	L1 latency (ms)	L2 Latency (ms)
Minimum	5	23
Maximum	116	186
Average	57.14	102.06

Note: The system used for measuring the latency shown in Table 1 was Intel i5-12450H CPU and 16 GB RAM. Whereas in [91], author mentions about using the system with Intel i7 CPU and 32 GB RAM, which is comparatively powerful.

6.3 Load Testing and Synchronization Checks

The complete system was run for an extended period under continuous image transmission and detection to verify the stability of the VSS synchronization logic. The Python scripts `rpi_sync.py` and `laptop_sync.py` continuously fetched the signals from other system and saved the signal data in the local Kuksa Databroker. Example of this synchronization script is shown in Figure 25.

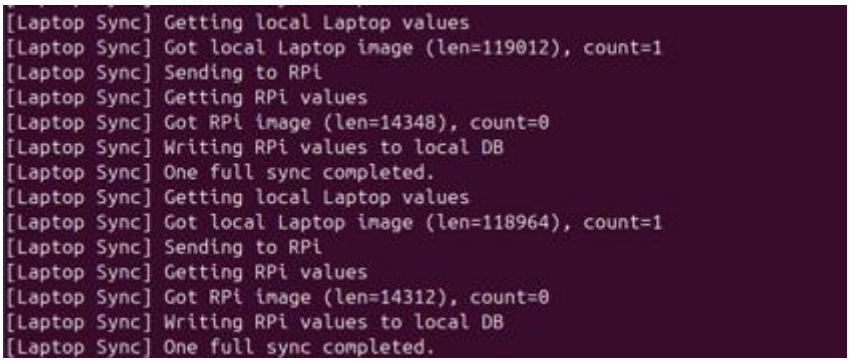

Figure 25. VSS Data Synchronization

No signal mismatch or corruption was observed during the testing. Also, consistently latest values were shown on Kuksa CLI interface of the both devices. The image and object count update was also verified by keeping different objects in the frame.

This confirmed that the synchronization scripts support the cross-device data exchange consistently, without any loss. This also fulfilled the project's real-time and interoperability goals.

6.4 Capabilities Implemented from Requirements (section 2.7)

This section provides a comprehensive review of how the prototype meets the requirements defined in 2.7, with a focus on the implementation and evaluation.

6.4.1 VSS Signal Mapping

VSS-compliant signal names like `Vehicle.Camera.RPi.ObjectCount` and `Vehicle.Camera.Laptop.Image` were implemented using `.vspec` file concerted to JSON. This updated file was loaded into Kuksa Databroker and tested using CLI interface and gRPC scripts to confirm the structural correctness and accessibility.

6.4.2 Real-Time Signal Publishing (R-RTSP-1 and R-RTSP-2)

Image and object count signals from both the Raspberry Pi and Laptop were published and updated in real-time, using some Python-based scripts. Latency was measured using timestamps and was verified to be average L1 (abstraction layer) of ~57 ms and average L2 (end-to-end) ~102 ms.

6.4.3 Middleware-to-Kuksa Bridge

The independent Python scripts served as bridge between Fast-DDS subscribers and Kuksa Databroker, which accepted the JPEG frames, encoded as base64 strings, and then sent them to VSS signal using gRPC client. Each bridge was manually mapped to a single VSS path and was verified using signal logs.

6.4.4 Hardware Abstraction (R-HWAB-1 and R-HWAB-2)

The use of distinct signal formats and naming ensured hardware-agnostic abstraction. Both the Raspberry Pi and Laptop pipelines used image data encoded in base64, so the handling of the signals was consistent, regardless of camera platform. Data exchange with the sync scripts confirmed the interoperability.

6.4.5 Data Logging (R-LOG-1 and R-LOG-2)

Each pipeline stage logged timestamps to the CSV file for the latency analysis. Other print logs also monitored the image length, image sync status, and detected object counts during the runtime. This ensured that the full signal path was traceable and functioning correctly.

6.5 Some Common Issues and Lessons Learned

This section describes key implementation issues that arose during prototyping phase and how those issues were resolved.

6.5.1 C++ Fast-DDS Build and Linking Issue

Repeated errors were encountered upon compilation of C++ Fast-DDS publisher and subscriber due to:

- Missing libraries like `libfastdds` and `libfastcdr`,
- Unresolved symbols due to platform specific differences between the compilers of the Laptop (MSVC) and Raspberry Pi (g++)
- Cross-platform CMake discrepancies

Resolution: The Fast-DDS library was built system-wide on the Raspberry Pi, CMakeLists were made modular to remove wildcard includes, and all the dependencies were installed as per the official installation guide [88].

Lesson learned: A strong understanding of the system-level libraries and Cmake configurations is required to remove the possibility of errors. Use only official installation guides as they are up-to-date.

6.5.2 Fast-DDS IDL Mismatch issues

Updating the `.idl` file without regenerating the bindings caused subscribers to silently lose messages because of schema mismatch.

Resolution: The `.idl` structure was frozen after finalizing the layout, and the bindings were generated every time a change was needed.

Lesson learned: Fast-DDS has a strict schema agreement. Therefore, always regenerate the code whenever an `.idl` definitions is changed.

6.5.3 VSS Signal Parsing in Kuksa Databroker

The edited `vss_release_4.0.json` file with new custom VSS signals caused Kuksa Databroker to crash due to invalid structure (like, misuse of “Branch” as a type). As whole VSS tree is a minified JSON structure, which is one-line format, made it difficult to edit the file manually.

Resolution: The minimized, single-line JSON format was converted into a human-readable, pretty-printed structure. Then the signal addition was made easily. And again, the file was converted back to the minimized structure with the same file name. After editing, the JSON file was validated using `jq`.

Lessons learned: Signal addition needs schema accuracy and clean formatting to avoid the errors.

6.5.4 Unrecognizable Custom JSON File

After successful signal addition in the JSON file, Kuksa Databroker was started. But, in CLI interface, custom signals were not accessible. The edited JSON file was not used by the Databroker by default.

Resolution: The correct path to the metadata file was explicitly passed to the kuksa Databroker by using `--metadata` flag while launching.

Lesson learned: Point out to the correct metadata file with the use of `--metadata` flag.

6.5.5 Fast-DDS Errors on Laptop (Windows)

Fast-DDS subscriber failed compilation on Windows Laptop due to missing .lib files. Also, Cmake configuration errors were observed and resolving these errors was time-consuming and not reliable.

Resolution: After some research, Zenoh was found to be a good option for Laptop pipeline. Zenoh is a simple Python-based setup, which enabled successful communication. So, then Fast-DDS was replaced with Zenoh.

Lesson learned: On windows, Fast-DDS is difficult to configure and maintain. Zenoh provides a lightweight option for rapid prototyping in constrained environments.

6.5.6 IP Conflicts and Environment Mismatch in Windows

In the early phase of the prototype development, Raspberry Pi and Laptop operated in different environments: Raspberry Pi on native Linux, whereas Laptop using PowerShell, WSL, and Python virtual environment. Due to this, the setup assigned different IP to the Kuksa Databroker running in WSL, which caused gRPC communication failure and data exchange between both the pipelines was no longer possible.

Resolution: Linux was installed on the Laptop. This configured both the systems to run on native Linux with static IP addresses in the same subnet. This also eliminated cross-layer IP mismatch and data exchange was successful between both the pipelines with gRPC communication.

Lessons learned: Mixed environments on Laptop like WSL introduces hidden networking layer. Native Linux with static IP offers simple setup for prototyping.

7 Future scope

While the prototype successfully demonstrated sensor data abstraction using middleware, some of the capabilities defined during earlier phases remain unimplemented because of the smaller prototype with just 2 sensors. These capabilities represent significant potential towards production level performance, scalability, and security. The following subsections outline some improvements on unfulfilled requirements and identified limitations.

7.1 Runtime signal extension

The prototype relied on a static `vss_release_4.0.json` file. If one or more signals are changed, restart of the Kuksa Databroker is required. Future work would therefore implement dynamic runtime extension using the Kuksa gRPC client, enabling VSS signals to be added, modified, or removed at runtime (with no downtime), and therefore added flexibility.

7.2 Securing Communication with Mutual TLS

Currently the implementation uses --insecure mode for gRPC communication. This is not suitable in production because a secure communication layer should be defined by using TLS with mutual authentication, allowing the client to verify the server and vice versa. It is reasonable to follow the production-grade data protection and thus allowing integration into secure OTA mechanisms.

7.3 Declarative Signal Mapping via YAML

Currently the prototype uses static hardcoded mappings between Fast-DDS/Zenoh messages and the VSS paths. Moving forward implementations should use declarative YAML files to define mappings to improve maintainability and scalability. These can then be automatically parsed to allow dynamic routing of signals based entirely on metadata, reducing the risk of human coding error.

7.4 Integrating With Real ADAS Stack

Currently object detection is being considered as an independent application module. Future work should be to integrate this output into a complete ADAS software stack to show the end-to-end use of VSS signals for actual vehicle functionality such as perception fusion, decision making and control feedback.

8 Conclusion

This proof-of-concept demonstrated a complete middleware-driven abstraction of camera data from independent sources (Raspberry Pi and Laptop) into a single VSS-compliant data model using Fast-DDS, Zenoh, and Kuksa Databroker. The implementation and verification of all key components like real-time images capturing, Fast-DDS/Zenoh transport of these frames, VSS-compliant signal publishing via gRPC, and object detection was successfully demonstrated.

The work aligns with the modular and data-driven approach of SDVs. Abstracting sensor data via standard VSS signals and gRPC client, provides the decoupling of hardware dependencies, and supports scalable integration into higher-level ADAS or infotainment application integration.

The project successfully bridges raw data transportation and semantic interoperability, forming a foundational structure for middleware-based SDV platforms. With additional improvements in dynamic signal handling, secure communication, and automated mapping, this prototype can be adapted for production-grade SDV use cases.

9 Bibliography

[1] Dirk Slama, Achim Nonnenmacher, Thomas Irawan, "The Software-Defined Vehicle: A Digital-First Approach to Creating Next-Generation Experiences," 2023.

[2] Z. Liu, W. Zhang, and F. Zhao, "Impact, Challenges and Prospect of Software-Defined Vehicles," Automot. Innov., vol. 5, no. 2, pp. 180–194, 2022.

[3] T. S. Madhuri and J. Bala Vishnu, "Software-Defined Vehicles: The Future of Automobile Industry," pp. 192–197, 2025, doi: 10.1109/COMSNETS63942.2025.10885701.

[4] VxLabs. "The Rise of Software-Defined Vehicles (SDVs): A New Era in Automotive Technology." Accessed: Mar. 30, 2025. [Online]. Available: https://vxlabs.de/software-defined-vehicles-sdvs-a-new-era-in-automotive-technology/

[5] Dr. Moritz Neukirchner, "Software-defined Vehicles – A Classification Approach," vol. 19, 2024.

[6] Stellarix. "Software Defined Vehicles: What Do They Offer Us?" Accessed: Mar. 31, 2025. [Online]. Available: https://stellarix.com/insights/blogs/software-defined-vehicles/

[7] Endava. "Software-defined Vehicle (SDV)." Accessed: Mar. 30, 2025. [Online]. Available: https://www.endava.com/glossary/software-defined-vehicle

[8] H. Reichert et al., "Towards Sensor Data Abstraction of Autonomous Vehicle Perception Systems," pp. 1–4, 2021, doi: 10.1109/ISC253183.2021.9562912.

[9] A. Pandharipande et al., "Sensing and Machine Learning for Automotive Perception: A Review," IEEE Sensors J., vol. 23, no. 11, pp. 11097–11115, 2023, doi: 10.1109/JSEN.2023.3262134.

[10] G. Cima, M. Console, M. Lenzerini, and A. Poggi, "A review of data abstraction," Frontiers in artificial intelligence, early access. doi: 10.3389/frai.2023.1085754.

[11] N. Chetouane and F. Wotawa, "Using Data Abstraction for Clustering in the Context of Test Case Generation," pp. 260–271, 2023, doi: 10.1109/QRS60937.2023.00034.

[12] B. Sejdiu, F. Ismaili, and L. Ahmedi, "Integration of Semantics Into Sensor Data for the IoT," International Journal on Semantic Web and Information Systems, vol. 16, no. 4, pp. 1–25, 2020, doi: 10.4018/IJSWIS.2020100101.

[13] H. Y. Teh, A. W. Kempa-Liehr, and K. I.-K. Wang, "Sensor data quality: a systematic review," J Big Data, vol. 7, no. 1, 2020, doi: 10.1186/s40537-020-0285-1.

[14] Anand Kumar Vedantham, "Augmented Reality in Modern Vehicles Enhancing Safety and Driver Experience," International Journal for Multidisciplinary Research (IJFMR), vol. 6, no. 6, 2024.

[15] Dr. Arunkumar M. Sampath. "Software defined vehicles transforming the future of mobility." Accessed: Apr. 1, 2025. [Online]. Available: https://www.telematicswire.net/mobility-in-the-future-will-be-transformed-by-software-defined-vehicles/

[16] S. Kirchner, N. Purschke, C. Wu, M. A. Khan, D. Dixit, and A. C. Knoll, "AUTOFRAME -- A Software-driven Integration Framework for Automotive Systems," 2025.

[17] D. P. Klüner, M. Molz, A. Kampmann, S. Kowalewski, and B. Alrifaee, "Modern Middlewares for Automated Vehicles: A Tutorial," 2024.

[18] J. Siksniene, "Software-Defined Vehicle Support and Coordination Project D2.6 First Technology Forecast Report," 2024. [Online]. Available: https://federate-sdv.eu/wp-content/uploads/2024/09/D2.6-First-Technology-Forecast-Report.pdf

[19] N. Corporation, "NVIDIA Autonomous Vehicles Safety Report | NVIDIA,"

[20] Ondrej Burkacky, Johannes Deichmann, Martin Kellner, Patrick Keuntje, Julia Werra, "Rewiring car electronics and software architecture for the ‘Roaring 2020s’," 2021. [Online]. Available: https://www.mckinsey.org/industries/automotive-and-assembly/our-insights/rewiring-car-electronics-and-software-architecture-for-the-roaring-2020s#/

[21] Hemanth Chakravarthy Mudduluru. "ISO 23150: The Language of Sensor Fusion for Autonomous Vehicles." Accessed: Apr. 1, 2025. [Online]. Available: https://www.linkedin.com/pulse/iso-23150-language-sensor-fusion-autonomous-vehicles-mudduluru-ekyle/

[22] "ISO 23150:2023(en) Road vehicles — Data communication between sensors and data fusion unit for automated driving functions — Logical interface," 2023. [Online]. Available: https://www.iso.org/obp/ui/en/#iso:std:iso:23150:ed-2:v1:en

[23] AUTOSAR, "Explanation of Sensor Interfaces," 2021.

[24] Clemens Linnhoff, Philipp Rosenberger, Simon Schmidt, Lukas Elster, Rainer Stark, and Hermann Winner, "Towards Serious Perception Sensor Simulation for Safety Validation of Automated Driving - A Collaborative Method to Specify Sensor Models," 2021.

[25] "ISO 26262-1:2018(en) Road vehicles — Functional safety — Part 1: Vocabulary," 2018. [Online]. Available: https://www.iso.org/obp/ui/en/#iso:std:iso:26262:-1:ed-2:v1:en

[26] COVESA, "Vehicle-Signal-Specification-Enabling-Ecosystems_20240105," [Online]. Available: https://covesa.global/vehicle-signal-specification/

[27] Ahmad Banijamali, Pasi Kuvaja, Markku Oivo, and Pooyan Jamshidi, "Kuksa: Self-adaptive Microservices in Automotive Systems," 2020.

[28] Object Management Group (OMG). "Data Distribution Service (DDS) for Real-Time Systems." [Online]. Available: https://www.dds-foundation.org/

[29] Adam Konopa. "VSS: The Key to Standardize Automotive Innovation." Accessed: Apr. 4, 2025. [Online]. Available: https://intellias.com/vehicle-signal-specification/

[30] COVESA. "Vehicle Signal Specification." Accessed: Apr. 4, 2025. [Online]. Available: https://covesa.github.io/vehicle_signal_specification/

[31] Yu Fang. "Using COVESA Vehicle Signal Specification (VSS) to accelerate Sonatus' next generation vehicle services." Accessed: Apr. 4, 2025. [Online]. Available: https://www.sonatus.com/blog/using-covesa-vehicle-signal-specification-to-accelerate-sonatuss-next-generation-vehicle-services/

[32] Sven Erik Jeroschewski, Bosch Digital, Vehicle Abstraction with Eclipse Kuksa and Eclipse Velocitas. Youtube: Automotive Grade Linux, 2023. Accessed: Apr. 4, 2025. [Online]. Available: https://youtu.be/LHJnBKb1Ta8?si=vz7JHuCVaGZPmGlz

[33] Sven Erik Jeroschewski, "Eclipse Kuksa,"

[34] Eclipse Kuksa. "Kuksa-InVehicle Repo." Accessed: Apr. 4, 2025. [Online]. Available: https://github.com/eclipse-kuksa/kuksa.invehicle

[35] Sebastian Schildt, "In-vehicle access to standardized VSS Vehicle Signals," [Online]. Available: https://archive.fosdem.org/2023/schedule/event/kuksa/attachments/slides/5650/export/events/attachments/kuksa/slides/5650/KUKSA_Fosdem2023.pdf

[36] P. S. Harri Hirvonsalo, "On deployment of Eclipse Kuksa as a framework for an intelligent moving test platform for research of autonomous vehicles," 2021.

[37] Stephen Goss. "The elevator pitch for Data Distribution Services." [Online]. Available: https://www.noser.com/techblog/introduction-to-data-distribution-services/

[38] Neil Puthuff, "DDS-The-Data-Backbone-of-SDV-Interoperability_V1_1024,"

[39] nanoMQ Team. "DDS and MQTT: Basics, Challenges and Integration Benefits." Accessed: Apr. 4, 2025. [Online]. Available: https://www.emqx.com/en/blog/navigating-dds-basics-limitations-and-integration-with-mqtt

[40] Roberto Menzione. "DDS protocol adoption in Automotive." Accessed: Apr. 4, 2025. [Online]. Available: https://www.linkedin.com/pulse/dds-protocol-adoption-automotive-roberto-menzione-mg2nf/

[41] RTI White Paper, "DDS_in_Autonomous_Car_Design,"

[42] A. Burns and R. I. Davis, "A Survey of Research into Mixed Criticality Systems," ACM Comput. Surv., vol. 50, no. 6, pp. 1–37, 2017, doi: 10.1145/3131347.

[43] Z. Jiang et al., "Re-Thinking Mixed-Criticality Architecture for Automotive Industry," pp. 510–517, doi: 10.1109/ICCD50377.2020.00092.

[44] J. Simó, P. Balbastre, J. F. Blanes, J.-L. Poza-Luján, and A. Guasque, "The Role of Mixed Criticality Technology in Industry 4.0," Electronics, vol. 10, no. 3, p. 226, 2021, doi: 10.3390/electronics10030226.

[45] Jie Zou, "Safety-Aware, Timing-Predictable and Resource-Efficient Scheduling for Autonomous Systems," 2023.

[46] Z. Guo, K. Koufos, M. Dianati, and R. Woodman, "State-of-the-art virtualisation technologies for the centralised automotive E/E architecture," Front. Future Transp., vol. 6, 2025, Art. no. 1519390, doi: 10.3389/ffutr.2025.1519390.

[47] Object Management Group. "Maximizing Data Distribution Service." [Online]. Available: https://www.omg.org/dds/

[48] "ISO/SAE 21434:2021(en) Road vehicles — Cybersecurity engineering." [Online]. Available: https://www.iso.org/obp/ui/en/#iso:std:iso-sae:21434:ed-1:v1:en

[49] Emily Yan. "Achieve Compliance with ISO 26262 Functional Safety Standards." [Online]. Available: https://www.keysight.com/blogs/en/tech/sim-des/data-ip/achieve-compliance-with-iso-26262-functional-safety-standards

[50] TÜV SÜD. "Automotive functional safety according to ISO 26262." [Online]. Available: https://www.tuvsud.com/en/industries/automotive/iso-26262-for-automotives

[51] Vector Informatik GmbH. "Adaptive AUTOSAR and HPC Design with PREEvision." [Online]. Available: https://www.vector.com/int/en/products/products-a-z/software/preevision/autosar-adaptive/#c373477

[52] Fernando Garcia. "Integrating DDS Into the AUTOSAR Adaptive Platform." [Online]. Available: https://www.rti.com/blog/integrating-dds-into-the-autosar-adaptive-platform

[53] Burns, Alan and Davis, Robert Ian, Mixed Criticality Systems - A Review:(13th Edition, February 2022).

[54] E. A. Capota, C. S. Stangaciu, M. V. Micea, and D.-I. Curiac, Towards mixed criticality task scheduling in cyber physical systems: Challenges and perspectives, Journal of Systems and Software, vol. 156, doi: 10.1016/j.jss.2019.06.099.

[55] Census. "Zenoh Protocol Security Analysis." [Online]. Available: https://census-labs.com/news/2025/03/17/zenoh-protocol-security-analysis/

[56] Eclipse Foundation. "Eclipse Zenoh 1.0.0 Debuts, Redefining Connectivity for Robotics and Automotive." [Online]. Available: https://newsroom.eclipse.org/news/announcements/eclipse-zenoh-100-debuts-redefining-connectivity-robotics-and-automotive

[57] Hussein Srour. "DDS and Zenoh: Communication Middleware for Vehicles and Beyond." [Online]. Available: https://www.linkedin.com/pulse/dds-zenoh-communication-middleware-vehicles-beyond-hussein-srour-bfnxf/

[58] Association for Computing Machinery, ACM ICN 2022 - Tutorial: Zenoh (2/5) - The Genesis. Youtube: Association for Computing Machinery, 2022. [Online]. Available: https://www.youtube.com/watch?v=8OHPSffJ8u0

[59] Eclipse Zenoh. "What is Zenoh?" Accessed: Jun. 13, 2025. [Online]. Available: https://zenoh.io/docs/overview/what-is-zenoh/

[60] Jkel. "Zenoh — A Protocol That Should Be on Your Radar." [Online]. Available: https://medium.com/@kelj/zenoh-a-protocol-that-should-be-on-your-radar-72befa697411

[61] Angelo Corsaro. "Zenoh: Unifying Communication, Storage and Computation from the Cloud to the Microcontroller." [Online]. Available: https://www.linkedin.com/pulse/zenoh-unifying-communication-storage-computation-from-angelo-corsaro/

[62] ZettaScale. "Top Most Requested Eclipse Zenoh Features Since Its Beginning." [Online]. Available: https://www.zettascale.tech/news/top-most-requested-eclipse-zenoh-features-since-its-beginning/

[63] William, Circle, and Jerry. "Comparing the Performance of Zenoh, MQTT, Kafka, and DDS." [Online]. Available: https://zenoh.io/blog/2023-03-21-zenoh-vs-mqtt-kafka-dds/

[64] Liang, Wen-Yew, Yuyuan Yuan, and Hsiang-Jui Lin, "A Performance Study on the Throughput and Latency of Zenoh, MQTT, Kafka, and DDS," 2023, doi: 10.48550/arXiv.2303.09419.

[65] Kydos. "Zenoh Performance." Accessed: Jun. 14, 2025. [Online]. Available: https://discourse.ros.org/t/zenoh-performance/30494

[66] D. P. Klüner, L. Hegerath, A. D. Hatib, S. Kowalewski, B. Alrifaee, and A. Kampmann, "Automotive Middleware Performance: Comparison of FastDDS, Zenoh and vSomeIP," 2025. [Online]. Available: https://www.themoonlight.io/en/review/automotive-middleware-performance-comparison-of-fastdds-zenoh-and-vsomeip

[67] ros2torial. "comparison between zenoh vs FastDDS Discovery Server v2." [Online]. Available: https://github.com/eclipse-zenoh/roadmap/discussions/79

[68] Angelo Corsaro, QoS-DDS vs Zenoh, PPT. [Online]. Available: https://speakerdeck.com/kydos/qos-dds-vs-zenoh

[69] J. Zhang, X. Yu, S. Ha, J. P. Queralta, and T. Westerlund, "Comparison of Middlewares in Edge-to-Edge and Edge-to-Cloud Communication for Distributed ROS2 Systems," J Intell Robot Syst, vol. 110, no. 4, 2024, Art. no. 162, doi: 10.1007/s10846-024-02187-z.

[70] Yujia Luo, Time Constraints and Fault Tolerance in Autonomous Driving Systems.

[71] RTI Core Libraries. "Working with Data in Connext." [Online]. Available: https://community.rti.com/static/documentation/connext-dds/current/doc/manuals/connext_dds_professional/users_manual/users_manual/PartCoreConcepts.htm#partcoreconcepts_4109331811_915546

[72] OpenDDS. "Quality of Service." [Online]. Available: https://opendds.readthedocs.io/en/master/devguide/quality_of_service.html

[73] Eclipse Foundation. "Kuksa DDS Provider." [Online]. Available: https://github.com/eclipse-kuksa/kuksa-dds-provider

[74] OMG, "DDS Security Version 1.1," 2018, Art. no. formal/2018-04-01. [Online]. Available: https://www.omg.org/spec/DDS-SECURITY/1.1

[75] COVESA. "Vehicle Signal Specification (VSS) – Release 5.0." [Online]. Available: https://github.com/COVESA/vehicle_signal_specification/releases/

[76] Eclipse Foundation. "Kuksa Databroker." [Online]. Available: https://github.com/eclipse-kuksa/kuksa-databroker

[77] AUTOSAR, "Specification of Sensor Interfaces: Adaptive Platform Release R21-11," 2021. [Online]. Available: https://www.autosar.org/fileadmin/standards/R21-11/AP/AUTOSAR_SWS_SensorInterfaces.pdf

[78] OMG, "Extensible and Dynamic Topic Types for DDS, v1.3," 2020, Art. no. formal/2020-02-04. [Online]. Available: https://www.omg.org/spec/DDS-XTypes/1.3/

[79] COVESA. "Vehicle Information Service Specification (VISS) v2." [Online]. Available: https://github.com/COVESA/vehicle-information-service-specification

[80] Eclipse Zenoh. "Documentation-Abstractions." Accessed: Jun. 13, 2025. [Online]. Available: https://zenoh.io/docs/manual/abstractions/

[81] Eclipse Foundation. "Eclipse Kuksa databroker-perf (Performance)." [Online]. Available: https://github.com/eclipse-kuksa/kuksa-perf

[82] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3: IETF RFC 8446," 2018.

[83] Uptane Alliance, "Uptane Standard for Design and Implementation 2.0.0: Uptane Standard Design," [Online]. Available: https://uptane.org/papers/uptane-standard.2.0.0

[84] W3C. "Trace Context: W3C Recommendation." [Online]. Available: https://www.w3.org/TR/trace-context/

[85] Cloud Native Computing Foundation. "OpenTelemetry Specification v1.46.0." [Online]. Available: https://opentelemetry.io/docs/specs/

[86] R. G. K. Babu, A. Badirova, F. F. Moghaddam, P. Wieder, and R. Yahyapour, "Authentication and Access Control in Cloud-Based Systems," pp. 560–562, 2023, doi: 10.1109/ICUFN57995.2023.10199236.

[87] Raspberry Pi Foundation. "Getting started with your Raspberry Pi." [Online]. Available: https://www.raspberrypi.com/documentation/computers/getting-started.html

[88] eProsima. "Linux installation from sources." [Online]. Available: https://fast-dds.docs.eprosima.com/en/latest/installation/sources/sources_linux.html

[89] Eclipse Kuksa Team. "kuksa-client 0.4.3: KUKSA Python SDK." [Online]. Available: https://pypi.org/project/kuksa-client/

[90] Eclipse Zenoh team. "Eclipse Zenoh: Python API." [Online]. Available: https://github.com/eclipse-zenoh/zenoh-python

[91] T. Betz, M. Schmeller, A. Korb, and J. Betz, "Latency Measurement for Autonomous Driving Software Using Data Flow Extraction," pp. 1–8, 2023, doi: 10.1109/IV55152.2023.10186686.

Declaration of Compliance

I hereby declare to have written this work independently and to have respected in its preparation the relevant provisions, in particular those corresponding to the copyright protection of external materials. Whenever external materials (such as images, drawings, text passages) are used in this work, I declare that these materials are referenced accordingly (e.g. quote, source) and, whenever necessary, consent from the author to use such materials in my work has been obtained.

Signature:

Stuttgart, on the 27.06.2025