Architecture and Transfer

[WP4; T4.3; Deliverable: D4.3 Version 1.1]

Non-confidential

AI training using Simulated Instruments for Machine Optimization and Verification

Document Information

Project	ASIMOV
Grant Agreement No.	20216 ASIMOV - ITEA
Deliverable No.	D4.3
Deliverable No. in WP	WP4; T4.3
Deliverable Title	Architecture and Transfer
Dissemination Level	public
Document Version	1.1
Date	2024.05.16
Contact	Lukas Schmidt
Organization	NorCom
E-Mail	Lukas.schmidt@norcom.de

The ASIMOV-project was submitted in the Eureka Cluster AI Call 2021
https://eureka-clusters-ai.eu/

Task Team (Contributors to this deliverable)

Name	Partner	E-Mail
Lukas Schmidt	NorCom	Lukas.schmidt@norcom.de
Niklas Braun	AVL	Niklas.braun@avl.com
Bram van der Sanden	TNO	Bram.vandersanden@tno.nl
Ezra Tampubolon	NorCom	ezra.tampubolon@norcom.de
Thomas Kotschenreuther	RA Consulting	t.kotschenreuther@rac.de
Holger Kohr	Thermofisher	holger.kohr@thermofisher.com

Formal Reviewers

Version	Date	Reviewer
1.0	2024.05.16	Mehrnoush Hajnorouzi (DLR); Sebastian Moritz (TrianGraphics)

Change History

Version	Date	Reason for Change
0.2	2024.05.07	Taking document offline to update figures and bibliography
0.3	2024.05.13	Including last missing parts
1.0	2024.05.15	Corrections
1.1	2024.05.16	Corrections

Abstract

This document is part of the deliverables of the ASIMOV project. It concentrates on the technical aspects on realizing an ASIMOV solution with the aim to facilitate the reader to realize digital twinning for other use cases and industries. It gives an overview of technologies, standards and methods that were used within the project and describes a possible architecture while accentuating aspects for the use in production that go beyond the prototypes that were realized during the course of the project.

Contents

Table of Figures

Table of Tables

1 Introduction

The motivation to use a digital twin to optimize physical systems in comparison to traditional methods is often to reduce time and costs.
Being able to reduce time and costs, however, is tightly connected with a technical realization that follows an efficient, scalable architecture and keeps development efforts reasonable.
While deliverables of WP 2 looked at an ASIMOV solution from a more theoretical point of view, this deliverable focuses on technologies, possible architectures and matters of consideration when designing and implementing an ASIMOV solution for other problems.
It will give an overview of technology and standards that are related to the use-cases of the consortium in section 2. Section 3 will be then about the architecture that may be reused by other industries, pointing to aspects to consider.
However, while the document also handles requirements that typically come up at later stages of development, the reader has to keep in mind that the content of this document was created with the experience from the prototypes that the consortium members implemented which was not a solution in production.

2 State of the art of technology and methods in the ASIMOV context

This section describes existing technology and methods that form the building blocks for the ASIMOV solution. The ASIMOV solution will be presented in the following chapter as the combination of those with emergence effects.

2.1 Technology and Methods for Infrastructure

2.1.1 Data Storage

When building an Asimov solution, the aspect of how to realize data storage must be addressed.
In the following section it is shortly shown what data may exist, what storage systems exist and what aspects should get attention.

Data can fall into these categories:

Input Data:
- Raw data utilized for pre-training the model.
- Configuration: Parameters defining system behavior.
- Templates: Configurable input files that change seldomly
- External Data: Information sourced from third-party APIs or databases.

Data During Use:
- Logs: Continuous streaming of system activities that may be important to save centrally for monitoring, documentation, debugging and validation purposes.
- State Information: Current system state, including active runs, current metrics, system load.
- Data Exchange Between Components: Messages and intermediate results for debugging purposes.

Output Data:
- Model: Trained machine learning models.
- Aggregation of Logs: Summarized system logs.
- Validation Information: Results of data validation processes.
- Report: Analytical summaries or presentations.

Taking into account the different form that the mentioned data is probably going to have, it is very likely that different storage solutions have to be used to efficiently retrieve, process and store data.
In addition to having a system that "just works" there might be additional requirements for later use in production based on the company policies for operational use.

For example, the following aspects might have to be considered:

- **Protection**: Implementation of encryption, access controls, and authentication mechanisms to safeguard data integrity, confidentiality, and compliance with regulatory requirements.
- **Access Control:** Granular control over user permissions and privileges, regulating data access, manipulation, and sharing to mitigate risks and ensure accountability.
- **Localization**: Storage and access of data within specific geographical regions or network segments, optimizing latency, compliance, and regulatory requirements.
- **Sharing**: Facilitation of controlled data sharing among authorized users, applications, or systems, while preserving data integrity, confidentiality, and traceability.
- **Replication**: Duplication of data across multiple storage nodes or locations to enhance data availability, fault tolerance, and disaster recovery capabilities.
- **Fast Search**: Implementation of indexing, caching, or search optimization techniques to enable rapid and efficient data retrieval, querying, and analysis, enhancing user experience and productivity.

For the realization of the Asimov solution as a prototype and for later production there is a variety of storage solutions to the diverse requirements and use cases:

- **Local Filesystem:** Traditional storage on local disks, suitable for small-scale applications, temporary storage, or individual system configurations.
- **Network Fileshares:** Network-based file storage systems enabling shared access to files across multiple clients or systems, commonly utilized in enterprise environments for centralized data management and collaboration. Examples: NFS (network file system), SMB (Server Message Block)
- **Distributed Filesystems**: Distributed filesystems leverage multiple storage nodes across a network to provide scalable, fault-tolerant storage for large volumes of data. These systems distribute data across multiple nodes, ensuring redundancy and availability even in the event of node failures or network partitions. Distributed filesystems offer horizontal scalability, data replication, and seamless integration with cloud-based environments for storing, managing, and analyzing big data at scale. Examples: HDFS (Hadoop Distributed File System), Ceph
- **Blob Storage**: Scalable, durable, and cost-effective storage services optimized for handling unstructured data, commonly deployed in cloud environments for data lakes, backups, content delivery, and archival purposes. Examples: MinIO, AWS (Amazon Web Services) S3
- **Indexation of Metadata and Content:** Association of metadata with data objects to facilitate efficient searching, filtering, and retrieval based on custom attributes, properties, or tags, enhancing data discoverability and usability in content management, digital asset management, or document repositories. Example: Elasticsearch, Solr
- **SQL (Structured Query Language) and NoSQL Databases:** Structured and unstructured database solutions offering diverse capabilities and trade-offs in terms of data consistency, scalability, performance, and flexibility:
  - **SQL Databases**: Relational database management systems (RDBMS) providing ACID (Atomicity, Consistency, Isolation, Durability) compliance, robust transaction support, and SQL-based querying capabilities for structured data with well-defined schemas and relationships. Example: Postgres, MSSQL
  - **NoSQL Databases**: Non-relational database systems offering schema flexibility, horizontal scalability, and high throughput for handling semi-structured or unstructured data with varying consistency models, including key-value stores, document databases, column-family stores, and graph databases, tailored to specific use cases such as real-time analytics, content management, or social networking. Examples: MongoDB, Redis
- **Message Brokers**: Message brokers provide reliable, asynchronous communication channels for exchanging data between distributed systems or microservices. These software platforms facilitate message queuing, pub/sub messaging patterns, and event-driven architectures, enabling decoupled, scalable, and fault-tolerant communication between heterogeneous components. Examples: Kafka, RabbitMQ

2.1.2 Computing

For the Asimov solution, as it concerns simulations and Artificial Intelligence (AI) applications, the demand for fast results leads to the necessity of parallelization and distributed computing.

When speed is crucial, relying solely on a single Central Processing Unit (CPU) core becomes impractical. Therefore, parallelization becomes essential. The challenge lies in effectively orchestrating tasks across multiple cores or machines to minimize communication overhead and maximize efficiency.

A key consideration is how to segment the code to enable seamless parallel or distributed processing. This necessitates a thorough understanding of the problem domain and the ability to architect solutions that leverage parallel processing without compromising coherence.

Efficient communication between parallel processes or distributed nodes is crucial for optimal performance. Techniques such as data compression, batching, and intelligent routing can help mitigate the impact of communication delays.

To avoid reinventing solutions for each project, it's crucial to adopt architectural patterns that facilitate scalability and deployment across different environments.

Parallel data access is fundamental in many computing scenarios, necessitating architectures that support simultaneous access from multiple processing units without sacrificing data integrity or performance.

Architectural patterns such as the Queue Worker pattern and microservice architecture offer robust solutions to these challenges. The Queue Worker pattern enables asynchronous task processing for efficient resource utilization and scalability. Meanwhile, microservice architecture decomposes complex systems into smaller, loosely coupled services, enabling independent scaling and deployment.

In addition to parallelization, Graphics Processing Units (GPUs) play a significant role in computing for simulations and AI applications. GPUs are highly specialized processors designed to handle parallel tasks efficiently, making them ideal for computation-intensive tasks such as matrix operations and neural network training. Leveraging GPUs can significantly accelerate tasks, leading to faster results and improved performance.

Container platforms have emerged as a crucial component of modern computing infrastructure, providing standardized environments for packaging, and deploying applications. Containers offer portability and consistency, reducing the effort required to port solutions to different environments.

In conclusion, computing for simulations and AI applications necessitates a focus on parallelization, distributed computing, and scalable architectures, alongside strategic utilization of GPUs. By leveraging architectural patterns, GPU acceleration, and container platforms, developers can address challenges of speed, scalability, and portability effectively, fostering advancements in these domains.

2.1.3 Deployment

For implementing the Asimov solution, we recommend using containerization as the underlying deployment strategy.

Containerization offers significant advantages over traditional deployment methods where software and configurations were installed imperatively on single hardware machines. By encapsulating software within containers, portability and modularity are greatly enhanced. Each containerized application comprises an image containing not only the software itself but also the necessary operating system environment and dependencies. Unlike full virtual machines (VMs), which require a fully configured environment, containerized applications minimize overhead by reusing components of the host system.

In the containerization process, developers build an image, explicitly defining the desired runtime environment for the software. Once an image is constructed, multiple instances, known as containers, can be launched. Each container operates independently, with its own configuration, ensuring no interference with others.

Kubernetes, often abbreviated as K8s, has emerged as a leading orchestration platform for managing containerized applications at scale. It automates various aspects of container deployment, scaling, and management, providing a robust infrastructure for modern cloud-native architectures. By abstracting underlying infrastructure intricacies, Kubernetes empowers developers to concentrate on application development and deployment, alleviating concerns about infrastructure management. Its functionalities, including auto-scaling, load balancing, and self-healing capabilities, streamline the handling of containerized workloads across various environments, spanning from on-premises data centers to public cloud services.

2.1.4 Monitoring

Monitoring refers to the continuous measurement and observation of system metrics. It is usually done on different levels:

- Operating system / environment of the solution: System load, system log scanning, …
- Service level: Availability, performance metrics, …
- Application level: System resource usage, log scanning, …
- End-user: Status and metrics of a job/workflow

Making use of monitoring allows to optimize the involved system, services and applications and is important for fast issue detection and trouble shooting.

The selection of libraries, frameworks, and applications for implementing a monitoring solution depends on the specific use-case, such as the scale of the system, the technology stack employed, and the desired metrics to be monitored.
Two prominent open-source examples are Prometheus and Grafana. While Prometheus is commonly used for collecting and querying metrics from distributed systems, Grafana can be used to visualize the collected metrics in created dashboards. Grafana offers the possibility to integrate not only Prometheus but also other data sources like InfluxDB or Elasticsearch.

2.1.5 Logging

Logging is the systematic recording of events, actions, and data points that occur at runtime. It serves as a critical component for effective monitoring and troubleshooting, enabling the identification of root causes for issues detected during monitoring. Well-executed logging allows for the reconstruction of the system's state at the time of an incident, aiding in diagnosis and resolution.

Most programming languages offer logging libraries, facilitating the implementation of logging functionality. Logs typically incorporate levels such as "INFO," "WARNING," and "ERROR," allowing for the filtering of relevant messages based on severity.

In distributed systems, centralizing logs is advantageous for gaining a holistic understanding of system behavior. This practice aids in forming a comprehensive view of system performance and troubleshooting distributed issues efficiently.

While logging is commonly associated with collecting messages from applications, it serves a broader purpose in the context of machine learning. In this domain, logging becomes a valuable tool for documenting ML experiments, facilitating reproducibility and experimentation tracking. Notably, frameworks like MLflow, discussed in section 2.2, offer robust logging capabilities tailored specifically for machine learning workflows.

2.2 Technology and Methods for AI

2.2.1 Model Logging

When utilizing a machine learning (ML) model that requires training, it's essential to employ a framework for logging and tracking experiments throughout the training process. This is crucial because such a framework enables:

- Reproduction of specific parameter sets used in each experiment, facilitating detailed examination of configuration settings. This can help identify parameter ranges that enhance model performance.
- Reproduction of additional experiment details, such as Key Performance Indicators (KPIs) and explanations of the models (e.g., through AI Explainability Frameworks like SHAP), aiding in building trust in the black-box model.
- Preservation of historical models from past experiments, providing valuable references for understanding the evolution of the environment and/or the Reinforcement Learning (RL)Model, even if they're not directly related to the current problem environment.
- Optimization of hyperparameters through comprehensive tracking, allowing for pinpointing of the best-performing hyperparameters across the entire experiment. This facilitates effective refinement and enhancement of models.
- Analysis of learning progress over time by continuously monitoring learning experiments. This facilitates comparison of results over time, revealing trends and progressions.

Having this functionality not only allows for transferring learning results across different timeframes and various learning agents but also empowers domain experts to conduct in-depth analysis of learning outcomes. Additionally, it enables the reproduction of learning experiments for further analysis or validation when needed, contributing to the robustness and transparency of the RL process. A suitable framework for such purposes is provided by MLFlow [1].

2.2.2 Model Serving through Frameworks

Once a trained model is prepared for application usage, it can be deployed into production by encapsulating it within a Docker container. This deployment strategy ensures that the model operates in an isolated environment, mitigating potential dependency conflicts. MLFlow, for example, offers functionality to facilitate this process. Tasks to be performed by the model can then be requested through HTTP requests.

2.2.3 Model-Libraries

When developing machine learning (ML) models, it's strongly advised to make use of established libraries. For example, in the domain of Deep Reinforcement Learning, TensorFlow [2] and PyTorch [3] are highly recommended for building neural networks along with specific implementations for the reinforcement learning such as MPO-package [4].

For other types of ML models, scikit-learn (SKLearn) [5] is an outstanding option, offering a wide array of preprocessing and postprocessing functions along with diverse ML models.

The advantages of employing these frameworks are plentiful. They contribute to fewer errors and faster progress. Additionally, by furnishing fundamental architectures, they allow us to focus on optimizing performance. These frameworks feature optimized implementations, resulting in quicker execution and training times.

Furthermore, they streamline the model development and deployment process, often providing user-friendly APIs and interfaces. They also promote collaboration among developers by offering a common framework and language. Moreover, they seamlessly integrate with other popular frameworks like MLFlow, as mentioned earlier.

2.2.4 Further ML-Libraries

Another valuable aspect to consider in model building is the utilization of hyperparameter tuning frameworks. One notable approach is Bayesian optimization [6] [7] which stands as a state-of-the-art strategy for optimizing black-box functions without assuming any predetermined form. Given that ML models comprise intricate and frequently evolving components, and thus can be likened to black-box functions, Bayesian optimization proves particularly suitable for this task. There are specific implementations available, such as the HyperOpt package [8] and the Optuna package [9]. These frameworks offer efficient tools for tuning hyperparameters, thereby enhancing the performance of ML models.

2.3 Technology and Methods for Digital Twinning

The general architecture of DTs in the ASIMOV context is explained in D2.3 [10]. It is important to not see the DT part of ASIMOV as a standalone component but as a highly integrated subsystem of the whole solution architecture. The tools that are used for Twinning therefore need to fulfill certain requirements:

- **Modularization**
  - DTs may require cross domain integration of multiple tools. It has to be ensured that the complexity of the physical system can be handled by the tool used for digital twinning.
- **Variety in interfaces**
  - As DT and PT need to be connected, it has to be ensured, that the respective communication protocols of the PT are also supported by the DT. In addition to that, the DT also has to be coupled with the RLA, which may support a different subset of communication technologies and furthermore has to operate on a different time scale than the actual twinning. Timescale, the data size and types therefore need to be considered when choosing a technology for digital twinning.
- **Real-Time coupling**
  - The PT may provide a continuous stream of data that can be used by the DT to adapt itself to the PT. Should DT and PT run in sync, real-time computation capabilities are necessary.
- **Simulation interfaces**
  - To gain any value from the DT, it has to be ensured that sufficient simulation tools can be integrated by the DT toolbox.

In the Unmanned Utility Vehicle (UUV) use case, these requirements were fulfilled by Model.CONNECT [11].

2.4 Standards and data formats

Using standards has several advantages:

- Enables **interoperability** between components (e.g., DT and RLA components) created by different parties.
- Improves **interchangeability**, to exchange a component for an updated version or a component from a different party.
- Improves **reusability** of components.
- Improves **developability**, meaning that an ASIMOV solution can be developed faster, as less time is spent on customizing and integrating systems.

This is important especially for complex environments, like those covered by the ASIMOV solution. Thus, the use of standards allows an easy adaptation of the ASIMOV solution to upcoming projects and by that ensures the usability of the project results.

2.4.1 ASAM Standards

Within the automotive industry several standards around measurement and diagnostics are hosted by the Association for Standardization of Automation and Measurement Systems (ASAM) [12].With the development of the autonomous driving functions, a full set of new standards are now hosted and maintained by the ASAM. These standards aim to support the development and validation of autonomous driving functions. These descriptions can be used with real driving scenarios, but also to run simulations as e.g. done within the ASIMOV UUV demonstrator. Due to their free accessibility the standards are named OpenXXX-standards:

- OpenSCEANRIO
- OpenDRIVE
- OpenCRG
- OpenLABEL
- OpenSimulationInterface
- OpenODD

In the following those standards are mentioned that were used by the ASIMOV solution or could be relevant for derived work. Besides the relevant OpenXXX-standards also classical standards are mentioned.

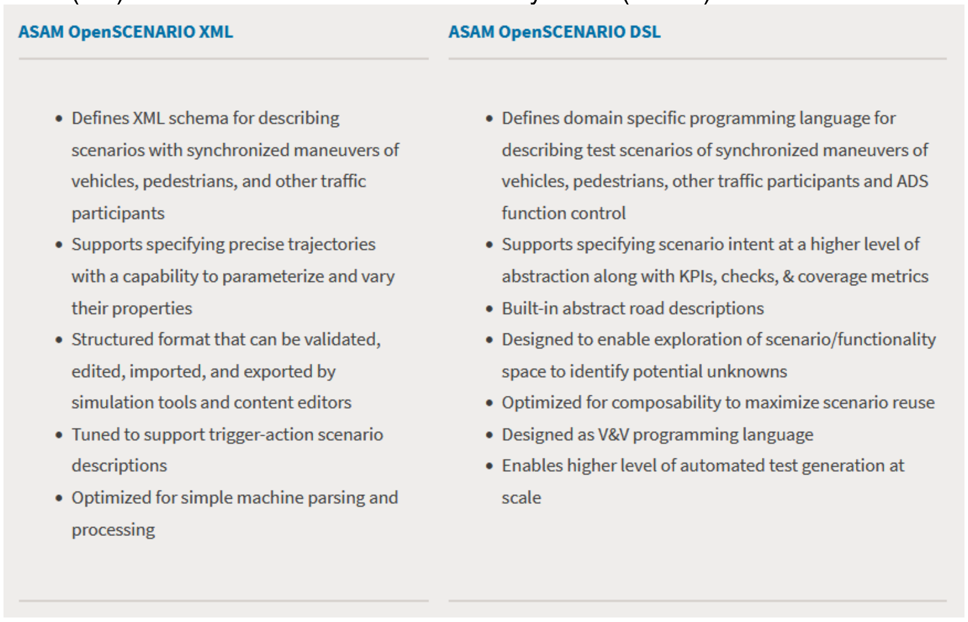
2.4.1.1 OpenSCENARIO

The ASAM OpenSCENARIO standard [13] [14] specifies a file format for describing the dynamic content of driving and traffic simulators. Due to the huge number of use-cases for this standard, it is developed in two flavors: The ASAM OpenSCENARIO XML specifies an XML-based file format for the description of traffic scenarios, while the ASAM OpenSCENARIO DSL focuses on the utilization of a domain specific language (DSL) for the scenario description.

Within ASIMOV, the XML variant of the OpenSCENARIO standard has been used with the UUV demonstrator in WP4. It is used by the scenario generation process to hand over the scenario to the simulation environment, see Figure 18 in D1.3 [15].

The two variants of OpenSCENARIO are focused on different use-cases as depicted in Figure 1 [14]:

- ASAM OpenSCENARIO XML: Predictable highly precise scenarios that may be used with an external test specification for V&V
- ASAM OpenSCENARIO DSL: Large-scale V&V of the safety and functionality of autonomous vehicles (AV) and advanced driver-assistance systems (ADAS)



ASAM OpenSCENARIO XML

- Defines XML schema for describing scenarios with synchronized maneuvers of vehicles, pedestrians, and other traffic participants
- Supports specifying precise trajectories with a capability to parameterize and vary their properties
- Structured format that can be validated, edited, imported, and exported by simulation tools and content editors
- Tuned to support trigger-action scenario descriptions
- Optimized for simple machine parsing and processing

ASAM OpenSCENARIO DSL

- Defines domain specific programming language for describing test scenarios of synchronized maneuvers of vehicles, pedestrians, other traffic participants and ADS function control
- Supports specifying scenario intent at a higher level of abstraction along with KPIs, checks, & coverage metrics
- Built-in abstract road descriptions
- Designed to enable exploration of scenario/functionality space to identify potential unknowns
- Optimized for composability to maximize scenario reuse
- Designed as V&V programming language
- Enables higher level of automated test generation at scale

Figure 1: Comparison of ASAM OpenSCENARIO XML [14] vs. OpenSCENARIO DSL [13]

Describing coordinated operations involving numerous entities, such as vehicles, pedestrians, and other traffic participants, is the main use-case for ASAM OpenSCENARIO. A maneuver can be described using trajectories (e.g., derived from a recorded driving move) or driver actions (e.g., conducting a lane shift). The standard also includes other details, such as the ego vehicle's description, the driver's appearance, pedestrians, traffic, and environmental circumstances.

The maneuver description data in ASAM OpenSCENARIO XML is serialized in an XML file format and arranged in a hierarchical structure. The standard is supplied with the schema. Simulation tools and content editors can simply validate, edit, import, and export the XML file. The format is independent of vendors and technologies.

The OpenSCENARIO standard is capable of being utilized with road surface profiles from ASAM OpenCRG (section 2.4.1.3) and is used in conjunction with road network specifications from ASAM OpenDRIVE (section 2.4.1.2). The full description of the static and dynamic content of in-the-loop vehicle simulation applications are addressed by all the three standards in combination.

2.4.1.2 OpenDRIVE

ASAM OpenDRIVE [16] allows to define a road network, where the scenarios encoded in OpenSCENARIO can take place. The road network consists of road-segments, that can be concatenated and crossings to connect the different roads. Each road segment specifies also the different lanes available in both directions.
Each road segment can be fully described to form a 3D curvature so that also curves and hills can be modeled.
With the OpenDRIVE information, the traffic participants, defined in OpenSCENARIO can be placed and dynamically simulated. Thus, the OpenDRIVE content is referenced by an OpenSCENARIO description. Several scenario-descriptions might reference the same OpenDRIVE file.
OpenDRIVE is also used within the UUV use case in ASIMOV as an output of the scenario generator.
As the time of writing, the version 1.8.0 is the most current version of the standard, released in October 2023.

2.4.1.3 OpenCRG

The ASAM OpenCRG [17] standard is also used in combination with the OpenDRIVE/OpenSCENARIO standards. As an addition to the roads and lanes specified in OpenDRIVE, OpenCRG allows to specify the road surface as a curved regular grid (CRG). The roads surface is described as the elevation along the road reference line organized as a grid to both sides of that line.

With the elevation information, the dynamic simulation can be far more detailed, according to e.g. the torque between road and tyre. OpenCRG is not used within the ASIMOV UUV simulation.

2.4.1.4 Measurement Data Format (ASAM MDF)

The Measurement Data Format, or MDF for short [18], is a binary file format used to hold measured or recorded data for offline analysis, long-term storage, or post-measurement processing. Although it is utilized in many other application areas, the format for measurement and calibration systems (MC-systems) has come to be considered the de facto standard.

The MDF format allows to store large volumes of measurement data efficiently and with highly performance. MDF is designed as a compact binary format for flexible and high-performance writing and reading. It is divided into loosely connected binary blocks. Sorting the data in a loss-free manner allows for quick index-based access to every sample. Direct writing of sorted MDF files is even possible with distributed data blocks.

The distributed data blocks enable also the direct writing of sorted MDF files. This file format can store both raw measurement values and their conversion formulas, ensuring that the raw data can be properly interpreted and analyzed using post-processing tools. To allow the separation of the different types of information, MDF is based on a linked block structure (s. [Figure 2]), where future extensions of the data format might introduce new types of blocks. With this separation of meta information and measured raw data the recording can directly write down the received data frames and thus delegate the interpretation of the data to the post-processing tools.

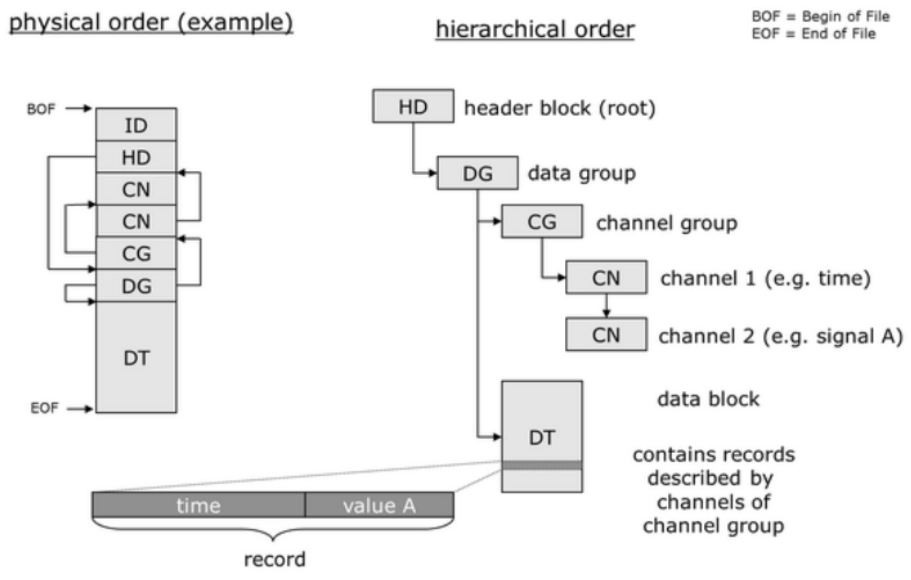


Figure 2: Example of a simple MDF-block structure [19]

As the time of writing the current standard version of MDF is version 4.2, released on 2019-09-30.

With the upcoming new technologies around the ADAS systems, also additional use-cases require to record more complex data than just signal values. Thus, currently the standard is extended to allow also the recording of image or video data, radar and lidar information, such as e.g. point cloud data or even object lists. With this extension it covers similar functionality as the ROSBAG format used with the ROS framework (section 2.4.2.1).

The release of the next version of MDF is planned for July 2024.

2.4.1.5 Open Simulation Interface

The ASAM Open Simulation Interface (OSI) [20] [21] has started as a project defining standards for the integration of sensor models into simulation systems. With the upcoming OpenXXX-Standards it was extended, so that also other entities could be described needed for the simulation of a scenario driven environment.
Thus, the project now also defines several data sets, clustered into interfaces:

- Ground Truth
- SensorData
- SensorView
- FeatureData
- Traffic participants (TrafficUpdate)
- TrafficCommand

To allow the utilization of the data format specifications, these specifications are realized as Protobuf [22] models. As the standard is open, the data specification is hosted on a public github-repository [21].
Some of the OSI data models are also used within the simulation environment of the UUV demonstrator.

2.4.1.6 OpenODD

An Operational Design Domain (ODD) [23] specifies the context boundaries for which a technical system is designed and developed. This gets important when the system is finally validated to be fit for purpose, i.e. it runs without problems inside its ODD.

To enable an automated validation, the specification should specify this ODD to be

- Searchable
- Exchangeable
- Extensible
- Machine readable
- Measurable and verifiable
- Human readable (by utilization of ISO 34503 [24])

Parallel to the developments on OpenODD, on the ISO side there's a preparation activity within the Standard ISO 34503: ISO/AWI 34503 – Road vehicles – Taxonomy for Operational Design Domain for Automated Driving Systems. [24]

ASAM made a concept paper (the link is given here: [23]) for specifying a standard for such an ODD description. On base of this concept result, currently a project is developing the first version of the OpenODD Standard within ASAM is expected to the end of March 2024 [25].

2.4.1.7 OpenLABEL

OpenLABEL is another standard for describing recognized objects from ADAS (Advanced Driver Assistance Systems) data, such as the ROSBAG format (see section 2.4.2.2). OpenLABEL defines the annotation format and the labeling methods for objects and scenarios.

The annotation format and labeling techniques for objects and scenarios are defined by the ASAM OpenLABEL standard. It also includes guidelines for using labeling techniques and the respective terminology.

Working with many clients led to a significant fragmentation in the way different organizations classify and characterize the elements that make up the driving environment. These classifications and descriptions serve as the core building blocks of an autonomous driving (assistant) system's (ADAS) perceptual stack, as they allow an ADAS to gain both a basic and in-depth awareness of the environment in which it operates.

It is intended with OpenLABEL to provide guidelines on how to use the annotation methods and definitions. This means that OpenLABEL should be used to store the results of AI algorithms for object recognition. Therefore OpenLABEL provides predefined object types and classes in a JSON format to be used within the labelling of respective ADAS data.
OpenLABEL is available as a standard in version 1.0.0 since 2021-11-12 [26]

2.4.1.8 ASAM-ODS Big Data Format

As the volume of data continues to grow, it becomes useful to adopt a specific data format conducive to efficient data management, especially when leveraging resource management technologies like Spark that rely on parallelization frameworks to enhance analysis performance.

One such standard format is the ASAM-ODS format [27] proposed by the ASAM Organization. The ASAM-ODS Mass Data Format employs a Parquet packed scheme with compression, allowing for possible distribution across multiple files. This format is well-suited for storage in systems like Hadoop.

For analysis purposes, NorCom has developed several libraries tailored to work with ASAM-ODS data:

- ASAM ODS Analytics INGEST: Facilitates the ingestion of data from MDF / ATFX files into the ASAM-ODS format.
- ASAM ODS Analytics ANALYZE: Provides tools for analyzing time series, objects, and conducting various analyses on the data.
- ASAM ODS Analytics EGRESS: Enables the export of data from ASAM-ODS format to MDF files.

The adoption of this data format and the associated products is particularly beneficial in handling the increasing amount of data generated by solutions like ASIMOV. These tools are instrumental in providing analysis, such as validation and model review, essential for leveraging the insights derived from the data.

2.4.2 Practical Quasi-Standards

Sometimes technical developments gain importance because they represent a solution to current development problems. If this is the case, the utilization of those developments increases and further developments attach around. Then those developments become a quasi-standard, as their usage supports to solve comparable problems.

One such development is the following system, heavily used e.g. in the area of prototyping of autonomous driving functions.

2.4.2.1 Robot Operating System (ROS)

The Robot Operating System (ROS) [28] is a framework to easily build up distributed systems, as e.g. in vehicle networks. Many prototyping systems are developed on the base of ROS as it allows to combine functionalities as sensors, actuators and computing nodes via network.

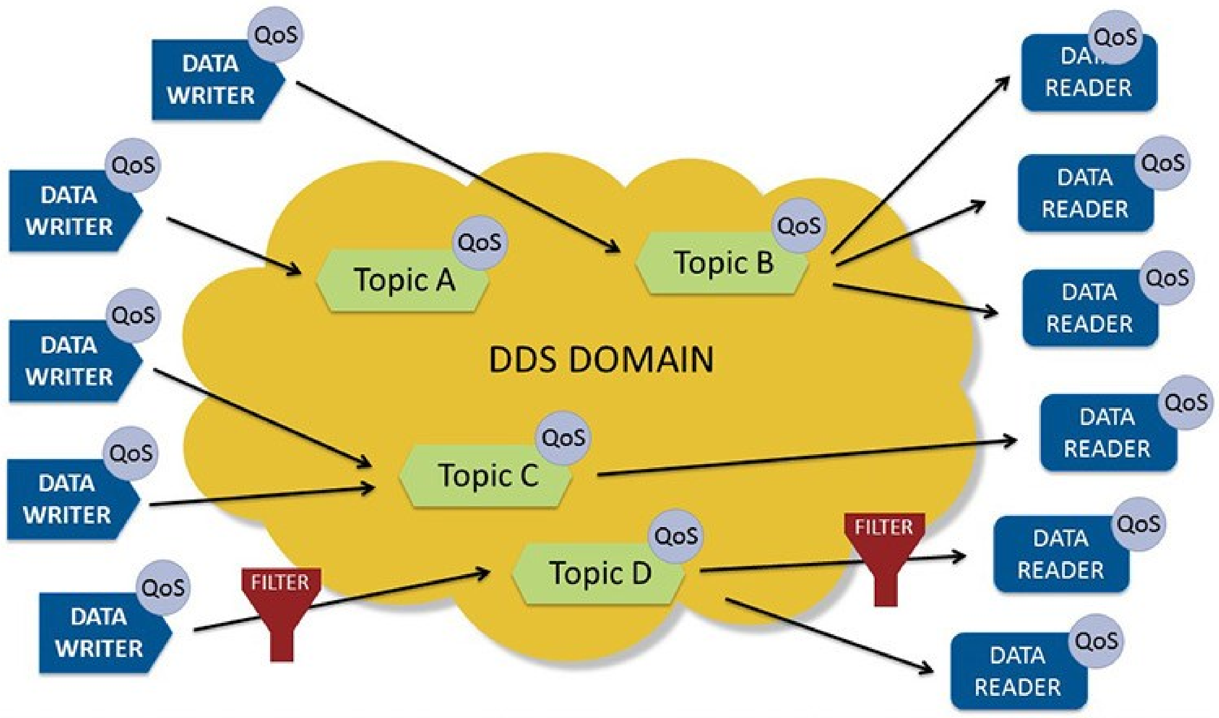


Figure 3: DDS - Structure of the DDS-System. Source: DDS-Foundation [29]

As of version 2 ROS also provides real-time capabilities and by utilizing the OMG Data Distribution Service (DDS - see Figure 3) Standard it is even compatible to the Adaptive AUTOSAR standard.
The ROS-system is extensible with custom message types which describe the format of the topics exchanged. To provide access to all message types used within a system, ROS is designed as a white

box system, i.e. for the development of a node, it needs access to all referenced message type specifications, which are normally part of the sources of the respective node implementations.
The ROS framework does not only provide the technology necessary to implement the functional nodes, but it also provides a set of tools supporting the development process. There are also tools to be used to inspect, measure and control a ROS based system.

2.4.2.2 ROSBAG

Tied to ROS is the natively supported file format for recording and playback, the ROSBAG file format [30]. As ROS supports a lot of different data types by default, and it is extensible towards custom datatypes, the measurement needs the flexibility to record all these different datatypes into a ROSBAG file.
By design a ROSBAG file is a potentially filtered message trace of the exchanged topics including the timing information. Thus, a ROSBAG file can be used to trace a system, but also to replay the topics into another system. This is especially interesting for the development of a new node or subsystem.
ROSBAG files can keep data of Signals, Point clouds, Trajectories, Positions, Directions and even images, video data or map information.

3 Architectural Views of the ASIMOV solution

In this section, we introduce the elements to think about in an ASIMOV solution architecture. The architecture is described using four views: the requirements, functional, logical, and technical view. Each view represents the system from the perspective of a related set of concerns [31]. The architecture adheres to the ASIMOV reference architecture described in [32]. Compared to the reference architecture, we illustrate the concrete architectures for the TEM use case and UUV use case of the ASIMOV project

3.1 Requirements View

When it comes to the requirements for the entire architecture of the ASIMOV solution, the requirements of the DT itself must be kept in mind. They have been identified in ASIMOV deliverable D2.3 [10]. The following requirements for the overall architecture complement the DT.

When building a system, where DTs are used to train an AI and where there additionally, is a connection to the PT required, the system will most probably end up being relatively complex. Such a complex system can involve multiple parties and tools for development. To aid such a development process, it can be beneficial to aim for a modular structure, defining clear interfaces between tools of different partners or subdivisions as well as different functional components. The clear separation provokes extensive interface description work, which – depending on the number of involved parties – can be significant work upfront, which pays dividends later in the development process. The benefits of such a modular architectural and development approach are:

- Responsibilities
    - By separating the entire architecture into an ensemble of functional components, that can be tested standalone, it becomes easier to assign teams and responsibilities to specific components.
- Well-thought-out Interfaces
    - As the interfaces become the start and end point for every functional component, it becomes very important to think sufficiently about these aspects. Ideally, those interfaces should cover standards.
- Modularity
    - If components are clearly separated by their Interfaces to other components, it becomes easy to swap them out or work on them in parallel during the development phase or even later. This can make a solution grow without having to think about significant side effects on the remaining architecture.
- Scalability
    - A scalable solution including more than one DT and/or AI training instance, also benefits from the possibility to have clear-cut components, that can be instantiated multiple times.

- Explainability
  - Interfaces between the components can easily be tracked, which makes explainability of the toolchain easier compared to a more integrated system design.
- Generalization
  - One of the goals of the architecture is its applicability to a wide variety of use-cases. Clear separation of modular components makes it also easy to transfer the architecture to a different domain.

3.2 Functional View

The functional view of the system describes the system's functionality, i.e., what the system should do. It defines the **system functions** in the system's design with their responsibilities. It also defines the interfaces between internal system functions and functions of external systems. The key functional elements together deliver the required system functions, based on the requirements described in the requirements view. The high-level functional view considering DT and AI has been defined in ASIMOV deliverable D2.3 [10].

3.2.1 Functional model for the TEM use-case

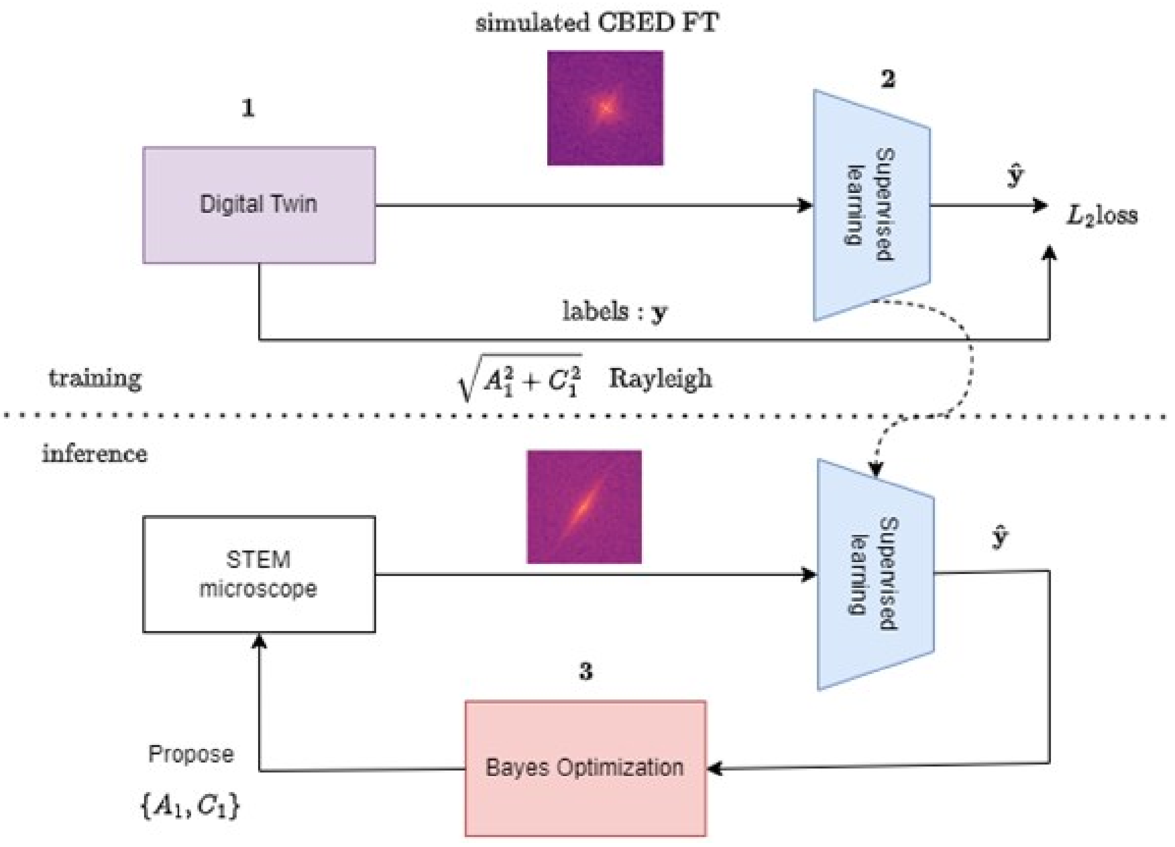


Figure 4: Functional model of the TEM use case.

Since the main focus of the TEM use case is method development and usage of domain knowledge, we took a lightweight and flexible approach to architectural design. The architecture is split into components running on the microscope (below the dashed line in Figure 4) vs. components running on a workstation or cluster.

- The microscope PC is in the lead. It hosts a Python environment with packages for controlling the microscope hardware, acquiring images, and communicating with the workstation.
- The workstation hosts a Flask server that encapsulates the estimation and control logic. In Figure 4 these components are divided into a pretrained network (blue box) and a Bayesian Optimization method. Alternatively, estimation and control can be combined in a single RL agent (not depicted).
- The Flask server on the workstation receives images from the microscope, processes them, and returns control actions to the microscope. The microscope PC applies these actions.

3.2.2 Functional model for the UUV use-case

The UUV use case leverages a container architecture, where many different components are managed in separated instances, that can talk to each other via a virtual network. Through functional modularization and organizational requirements, the UUV architecture consists of the following components:

- **Reinforcement Learning**
  This container contains the reinforcement learning agent itself. It communicates with its environment via the conventional RL interfaces, i.e. it outputs actions in the form of a Json file, which comprises information about the scenario that needs to be created in the next step. Its inputs are the state and reward, stored in a *.pkl file.
- **Scenario Generator**
  The Scenario Generator can be seen as a front desk for actually creating the 3D environment. It receives the Json file and forwards it to the TrianBuilder. As soon as the TrianBuilder is ready building the Scenario in a proprietary format, the Scenario Generator forwards this information to the Carla Exporter, which creates the Carla files.
- **TrianBuilder**
  The TrianBuilder takes the Json scenario variation description as input and creates a 3D environment based on that information. Details are described in D2.2 [33].
- **Carla Exporter**
  The Carla Exporter takes the 3D environment built by the TrianBuilder as input and exports it to a different format, making it compatible with the Carla environment simulator, which is used as Sensor Engine in this project.
- **Simulation Handler**
  The simulation handler is the front desk of the simulation. It takes the request for running a simulation and loads the newly generated 3D environment in the Carla Runtime. After this, it initiates the execution of the simulation through the Vehicle Dynamics and Scenario Engine.
- **Vehicle Dynamics and Scenario Engine**
  This is the co-simulation running Model.CONNECT, which includes the vehicle dynamics model, as well as the driving function and the scenario engine, which effectively executes the movements of all traffic participants, defined in an ASAM OpenSCENARIO file. Hereby it closely interacts with the Carla runtime, where all these movements are projected into the 3D environment, where the sensor models are executed. This communication happens through an OSI Sensorview, as defined in the ASAM OpenX Standards. The output of this block is an *.osi log file, that contains traces of the movements of all traffic participants for all timesteps in the scenario.
- **Carla Runtime**
  The Carla runtime is the 3D environment execution. It closely interacts with the Scenario Engine. All 3D-geometry-based sensors are rendered in this environment. The environment itself is the one exported from the Carla Exporter.
- **Feature Engineering**
  The Feature Engineering takes the *.osi simulation results and analyzes them to calculate criticality KPIs. These KPIs are then exported as *.pkl file and used by the Reinforcement Learning as state and reward.

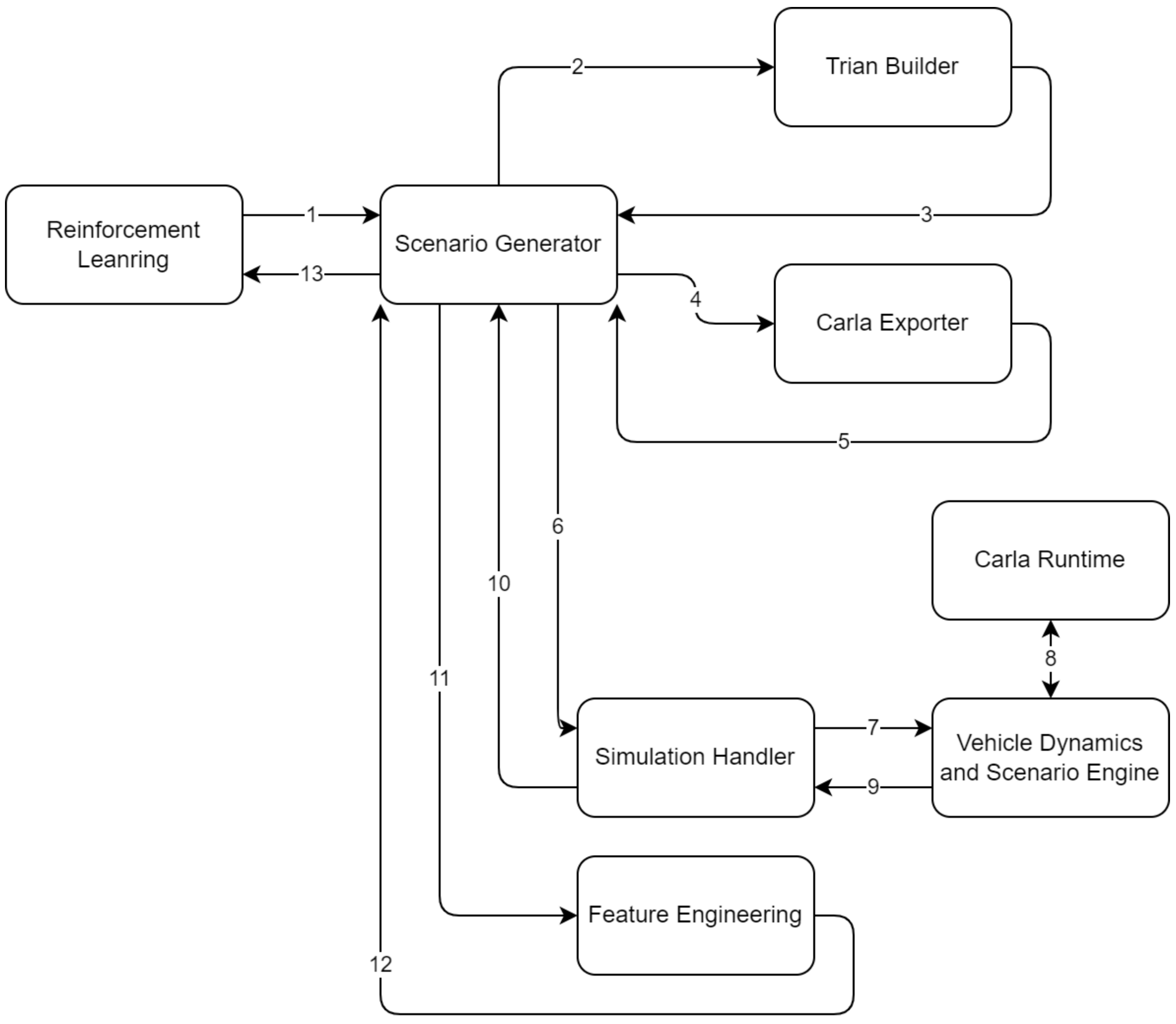


Figure 5: Overall Architecture UUV Use Case

The architecture created can be seen in Figure 5, with the numbers in the connections representing the execution order. It reflects the before mentioned functional dependencies. This version of the architecture was very much focused around the core idea of the RLA interacting with an environment. The Scenario Generator represented this environment and therefore handled the organization of all other containers centrally. This proved to be suitable for an easy implementation, but has its limitations in terms of scalability, as the whole process is very much sequential and individual components cannot easily be multiplied for faster execution.

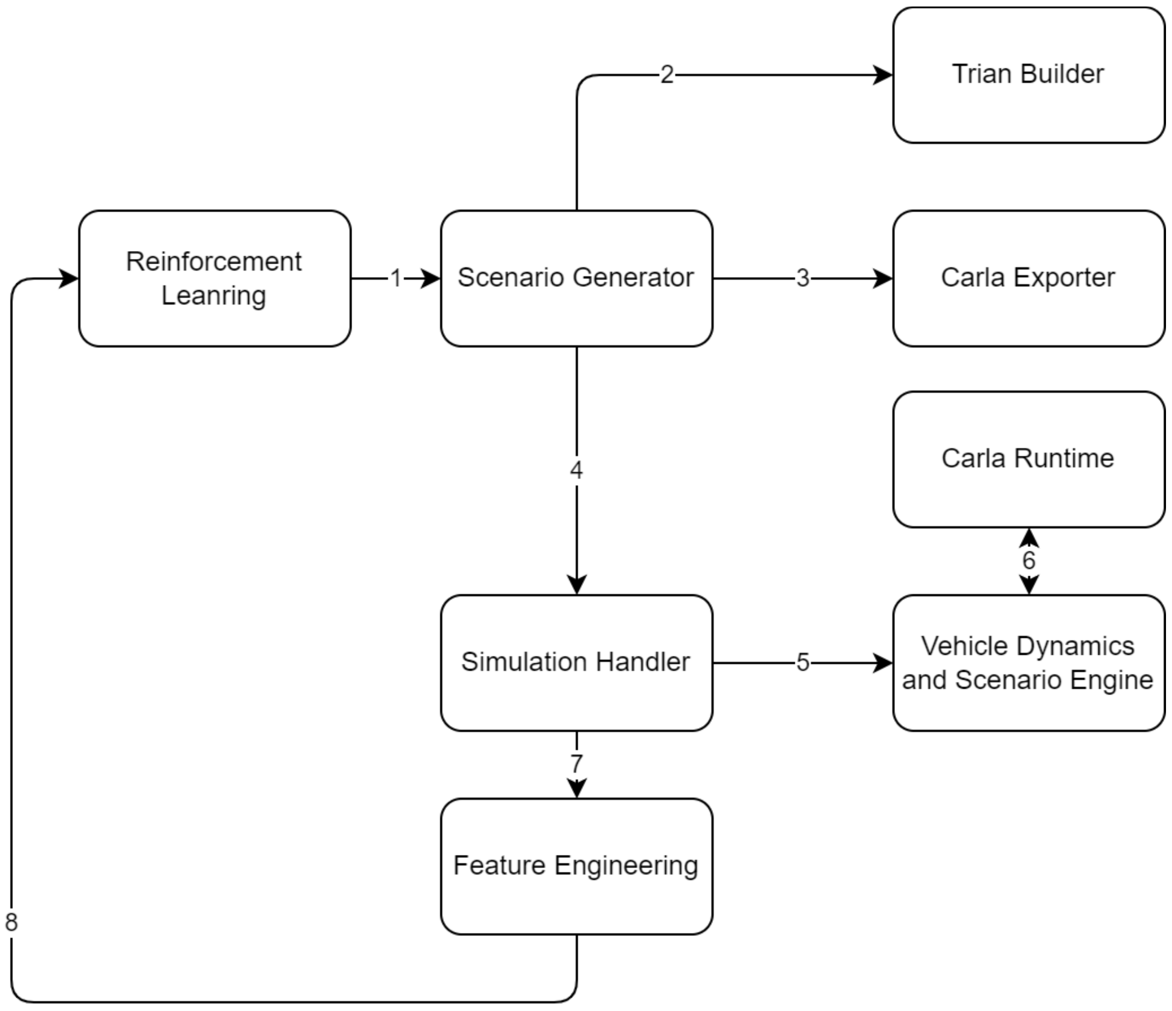


Figure 6: New Overall Architecture UUV Use Case

To solve the disadvantages of this architecture, a second version of the architecture was planned. This can be seen in Figure 6. Although this architecture contains the same elements, it has several benefits. As its workflow is better structured, following a loop configuration, no central component exists. Instead, the elements follow a clear sequential order. This has the clear benefit that the loop cannot only be a loop of containers, but a loop of queues, with containers attached to them as workers. That way, performance bottlenecks can be identified and fixed by multiple workers. This functionality of course requires an RLA that can handle parallel streams. If the RLA supports this, scalability can be massively increased by simply adding new workers to queues. This architecture therefore also benefits heavily from the modularity of the container architecture.

3.3 Logical View

The logical view describes the structure of the system in an implementation-agnostic way. It focuses on the system decomposition into **logical components** and their communication. A logical component provides one or multiple functions specified in the functional view.

The set of logical components of the ASIMOV solution adheres to the reference architecture, as specified in [32] and aligns with the components identified in [10]. The composition of elements in the logical view depends on the specific ASIMOV development phase. There are specific logical views for the training phase, operational phase, and fine-tuning phase.

3.3.1 Logical view in the training phase

In the training phase, the system architecture contains the following logical components as shown in Figure 7:

- **Simulation environment:** contains everything required to generate data for the RLA training, including data management, pre- and postprocessing, and twinning capabilities.
- **Controller:** orchestrates the training process and provides an interface to the human users.
- **Reinforcement Learning Agent:** responsible for learning from the behavior provided to it, and for suggesting new settings in order to learn an optimal strategy.

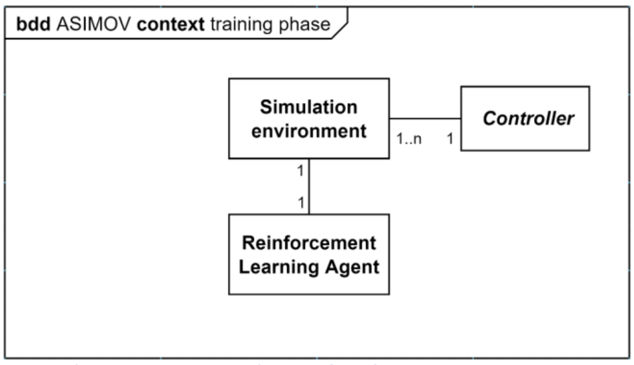

bdd ASIMOV context training phase
Simulation environment
1..n
1
Controller
1
1
Reinforcement Learning Agent

Figure 7: Block definition diagram of the ASIMOV solution during the training phase

3.3.2 Logical view in the operational phase

In the operational phase, the simulation environment is connected to the cyber-physical system (or also called physical twin) that needs to be optimized. Depending on the implementation, either the CPS requests its optimization, or the optimization is managed by a higher-order management system that controls the state of multiple systems. Therefore, the cardinality between the Simulation environment and Controller is left open compared to the training phase. A Block definition diagram of it can be seen in Figure 8.

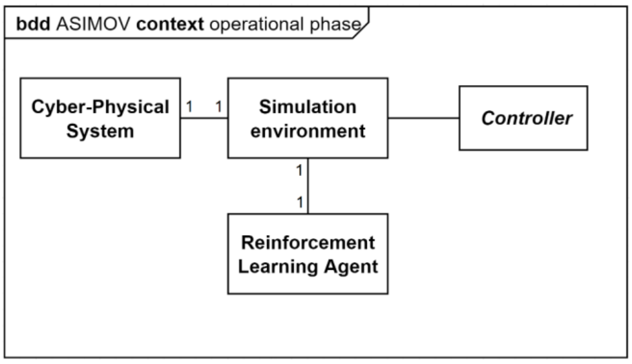

bdd ASIMOV context operational phase
Cyber-Physical System
1
1
Simulation environment
Controller
1
1
Reinforcement Learning Agent

Figure 8: Block definition diagram of the ASIMOV solution during the operational phase.

3.3.3 Logical view in the fine-tuning phase

In the fine-tuning phase, the focus is on tuning the DT as well as the optimization AI. The tuning is done based on the data that is collected during the operational phase. The high-level logical view in the fine-tuning phase is the same as in the operational phase. Special emphasis is put on the adaptation model (as introduced in [34]), that is part of the digital twin. This adaptation model compares the DT and the PT behavior, and adapts the DT if needed.

3.4 Technical View

The technical view describes possible technical implementation of the elements described in the logical view. It focusses on the interfaces, data formats, and standards to exchange data between components, as well as the deployment of components in terms of software and hardware. Regarding deployment, it is important to also consider performance-related aspects.

3.4.1 Architecture

The proposed architecture for an ASIMOV solution is a microservice architecture where each logical component is realized as a mostly independent unit (compare to section 3.3). This approach offers several advantages in software development. According to Fowler et al. [35], microservice architecture enhances scalability, maintainability, and deployability by breaking down complex systems into smaller, loosely coupled services.

Each microservice is realized via one or more containers, with one container serving as an interface accessible by other components, where responsibilities, expected input and expected output are well-documented (compare to Figure 9). The interface container encapsulates the functionality of the microservice, ensuring that only the interface is exposed to other components. This design principle, as advocated by Newman [36], promotes encapsulation and modularity, facilitating easier maintenance and evolution of the system.

The dependencies of each component are minimized, with the interface serving as the primary point of interaction. This design strategy aligns with the principles of high cohesion and low coupling, as outlined by Martin [37] fostering flexibility and resilience within the system.

Moreover, this approach facilitates code reuse. By standardizing interfaces and decoupling implementation details, microservices and containers can be developed independently, allowing for the reuse of functionality across different projects and teams. This promotes collaboration and accelerates development efforts, particularly in large-scale software projects.

To implement the prototype effectively, REST interfaces have been adopted due to their simplicity and widespread adoption. RESTful APIs enable easy testing of individual components, as emphasized by Richardson and Ruby [38].

In cases where long-running tasks are involved, asynchronous jobs can be initiated, with the option to specify a callback endpoint or a function. This design choice enhances system responsiveness and enables efficient resource utilization, in line with the principles of event-driven architecture [36].

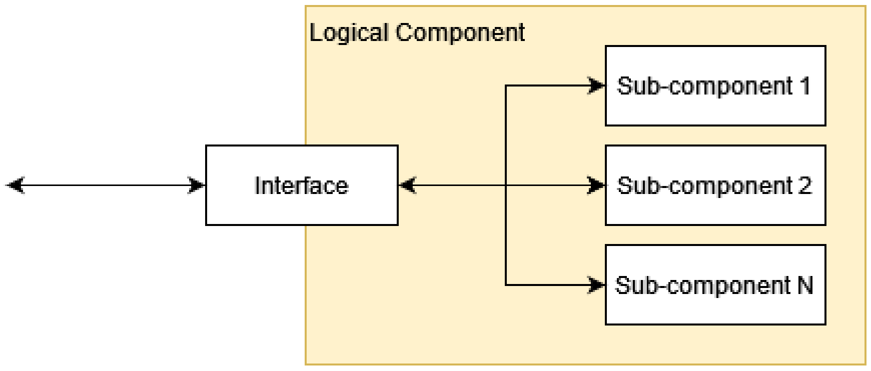


Figure 9: Structure for the implementation of logical components with the interface sub-component that adapts to components within and outside of the logical component.

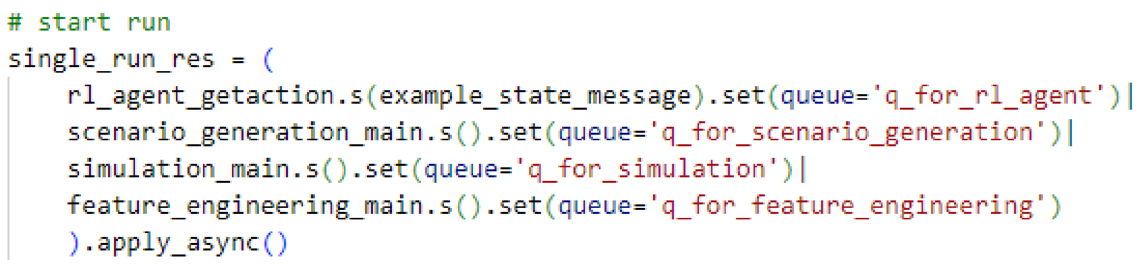
3.4.2 Communication

Communication between the components of the system is crucial for its functionality and efficiency. In the initial stages of testing and prototype development, HTTP is employed for calling the REST interfaces directly. This choice enables quick iteration and validation of individual components.

For future scalability, robustness and efficiency, we recommend adopting a queuing mechanism for messages, such as it is provided by RabbitMQ. Queuing systems offer asynchronous communication, decoupling producers and consumers, and providing fault tolerance and load balancing capabilities.

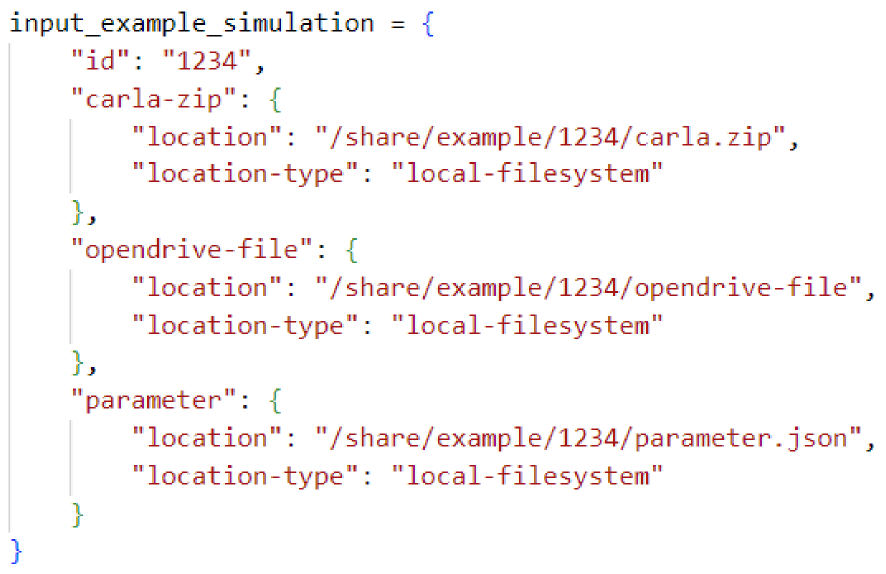
We propose JSON [39] messages for communication due to their low syntactical overhead, human readability, and support for schema validation. JSON schemas provide a means to validate message structures and helping to ensure integrity and consistency in an evolving system. However, for use-cases requiring advanced schema validation, consideration of XML [40] and XML Schema Definition (XSD) [41] files may be warranted, as XSD provides more extensive validation capabilities.

When larger data must be transferred, we propose to minimize the message size and transmit only the data location. This ensures that data access happens via the original data sources and keeps the load on the queuing software low (see example in Figure 11: Example of a communication payload with a run-id and a pointer to Figure 11). Larger messages may also make it difficult to log the communication for debug purposes. The choice of protocol depends on the specific use case, the already existing data sources and the nature of the data that has to be transferred. Examples of data sources may be a network filesystem like nfs, a blobstore like Amazon S3 or the Data Management Platform DaSense of NorCom [42].



```
# start run
single_run_res = (
    rl_agent_getaction.s(example_state_message).set(queue='q_for_rl_agent')|
    scenario_generation_main.s().set(queue='q_for_scenario_generation')|
    simulation_main.s().set(queue='q_for_simulation')|
    feature_engineering_main.s().set(queue='q_for_feature_engineering')
    ).apply_async()
```

Figure 10: Example of python-code for chaining functions from different components in the UUV use-case with the celery library and RabbitMQ. The pipe-symbol "|" indicates that the result of the respective function call is given as input for the next function. Note, that only the signatures of the functions are used for the code and execution of the respective functions happens on the workers that fetch messages from their queue.



```
input_example_simulation = {
    "id": "1234",
    "carla-zip": {
        "location": "/share/example/1234/carla.zip",
        "location-type": "local-filesystem"
    },
    "opendrive-file": {
        "location": "/share/example/1234/opendrive-file",
        "location-type": "local-filesystem"
    },
    "parameter": {
        "location": "/share/example/1234/parameter.json",
        "location-type": "local-filesystem"
    }
}
```

Figure 11: Example of a communication payload with a run-id and a pointer to additional files needed as inputs.

3.4.3 Performance aspects

The performance of the ASIMOV solution's architecture is crucial for its successful implementation, particularly in handling numerous iterations of the reinforcement learning (RL) agent within short timeframes. Achieving such performance requires a strategic approach, including the possibility of scaling the application both vertically and horizontally.

Vertical scaling entails optimizing the container application itself to enhance its performance. This can involve employing performance analyzers and profilers to identify areas within the codebase that offer the greatest potential for optimization. Additionally, upgrading hardware components to faster alternatives can help reduce compute and I/O times, thereby accelerating processing.

This includes in particular Graphics Processing Units (GPUs) as their utilization can significantly enhance processing speed. GPUs are specifically designed to handle parallel computations efficiently, making them well-suited for tasks involving matrix operations and neural network training.

For interpreted languages like Python, maximizing efficiency often involves leveraging optimized libraries and minimizing reliance on pure Python code. Utilizing libraries such as NumPy or Pytorch accelerates for example the execution of vector and matrix operations that can also make use of optimized system libraries like BLAS, enabling parallel computation across multiple CPUs [43].

Horizontal scaling, on the other hand, involves deploying multiple containers of the same type to distribute workload and increase throughput. However, this approach necessitates careful consideration during the development phase of the ASIMOV solution. The RL agent must either inherently support parallel execution or be structured in a manner that allows concurrent optimization processes.

In scenarios where experiments are independent, a simple parallel deployment of separate setups can be done to save time. However, for setups where RL algorithms themselves allow parallelism it is desired to scale only bottleneck containers for efficient resource usage.

In the UUV use case, parallel utilization is anticipated within the architecture. A queue worker pattern has been implemented, wherein one or more workers are instantiated for each logical component [44]. These components, besides being accessible via their REST interfaces, can also be interacted with through messaging queues. Upon receiving a message, an available worker processes it and forwards the result to the designated queue, ensuring high utilization rates and minimizing overhead associated with the HTTP protocol [45].

The technical implementation of this architecture was realized using Flask [46] and Celery [47]frameworks, along with RabbitMQ [48] for messaging capabilities. This setup allows for efficient parallelization of tasks within the ASIMOV solution, optimizing its performance and scalability.

3.4.4 Information and system management

The ASIMOV Solution can benefit from information and system management through logging systems and data management platforms. These instances enable quick identification of potential issues and facilitate historical analysis, crucial for pinpointing areas for improvement. Additionally, they streamline communication between teams and instances, while improving the auditability of results for the actual system by organizing data in a structured manner.

3.4.4.1 Experiment Logging

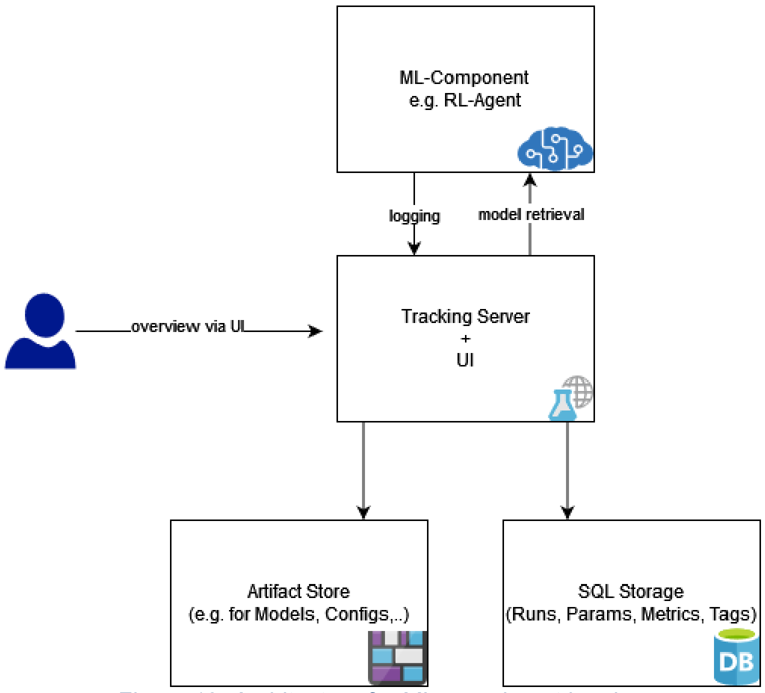


Figure 12: Architecture for ML experiment logging.

To log machine learning experiments, we can employ the MLFlow framework. A crucial prerequisite is having an instance hosting the corresponding tracking server. This server is responsible for receiving and managing data sent from the learning instance and provides an ergonomic MLFlow UI for users to visualize learning results. Setting up this server can be done within a Docker container, functioning as a service for storing tracked data.

Typically, this data is structured in tabular format for parameters and metrics, while other data formats like models and images are categorized as artifacts (see also Figure 12). Parameter and metric data are commonly managed in a SQL database, whereas artifacts are stored either in a local file system or a cloud storage solution. Implementing the interface for storing artifacts (Artifact Store) to support both local file systems and cloud storage systems (e.g., S3) can be achieved using MinIO [49] supported by MLFlow. To implement the MLFlow architecture, there are various approaches available, as illustrated in Figure 13. The third setup depicted in the figure aligns with the architecture described earlier. This architecture is in accordance with the containerization requirement outlined in the ASIMOV solution.

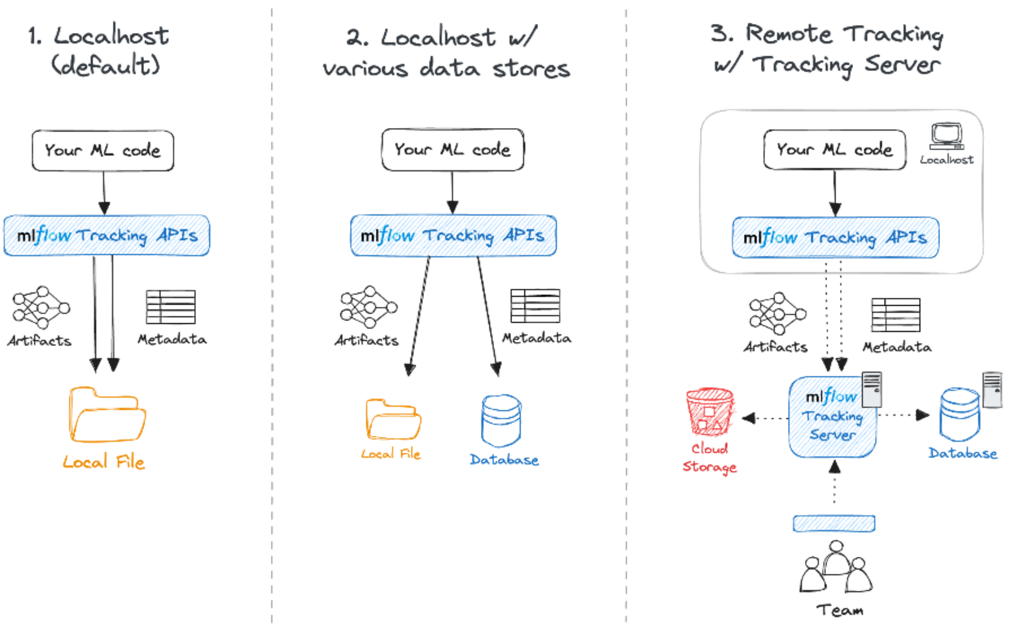


Figure 13: Common MLFlow Setups. Source: [50]

For logging the learning process, one can simply add the MLFlow-API to the code of the ML-Component. Clearly after connecting the MLFlow Component to the tracing server by inserting the URL of the tracking server into the API-Configuration. Information to be logged are e.g. Model, Model Hyperparameter, Specific Information about the environments, Performance Measure, Explainability. MLFlow provides support for standard ML libraries such as SKLearn, TensorFlow, and PyTorch. When utilizing these libraries, it's advisable to utilize the corresponding MLFlow class instance. For example, when using the SKLearn package, commands with the prefix "mlflow.sklearn" can be employed. Additionally, MLFlow offers automatic logging of parameters with predefined configurations through the "autolog" command. However, utilizing this functionality might lead to excessive overhead data, especially if the logged quantities are not utilized in the ASIMOV solution. In practical scenarios, custom quantities and plots may need to be logged into MLFlow. Therefore, transitioning from the autologging option to customized logging is recommended.
The organization of ML training tracking is structured around the concept of a "run," which represents the execution of ML training code with fixed hyperparameters. Each run stores metadata (various information about the run like metrics, parameters, start and end times) and artifacts (output files from the run such as model hyperparameters, plots, etc.). At a higher level in the MLFlow process is the "experiment," which groups runs together for a specific task, such as optimizing hyperparameters within a certain range.
When optimizing the ML model, one can utilize the user interface hosted on the tracking server to visualize the logged data. For example, users may compare the performance of hyperparameters across different runs to gauge the potential direction of improvement. Furthermore, within a specific run, users can use the UI to examine additional information about the trained model, such as its explainability, in order to gain insights into the optimal model or to audit a model, provided that the corresponding plots of explainability metrics are logged.
Finally, once a "good" model is achieved, it can be deployed within a Docker container to serve inference requests. MLFlow supports this functionality. During the operational phase of the ASIMOV solution, the corresponding inference instance can obtain model responses by making HTTP requests to this container.

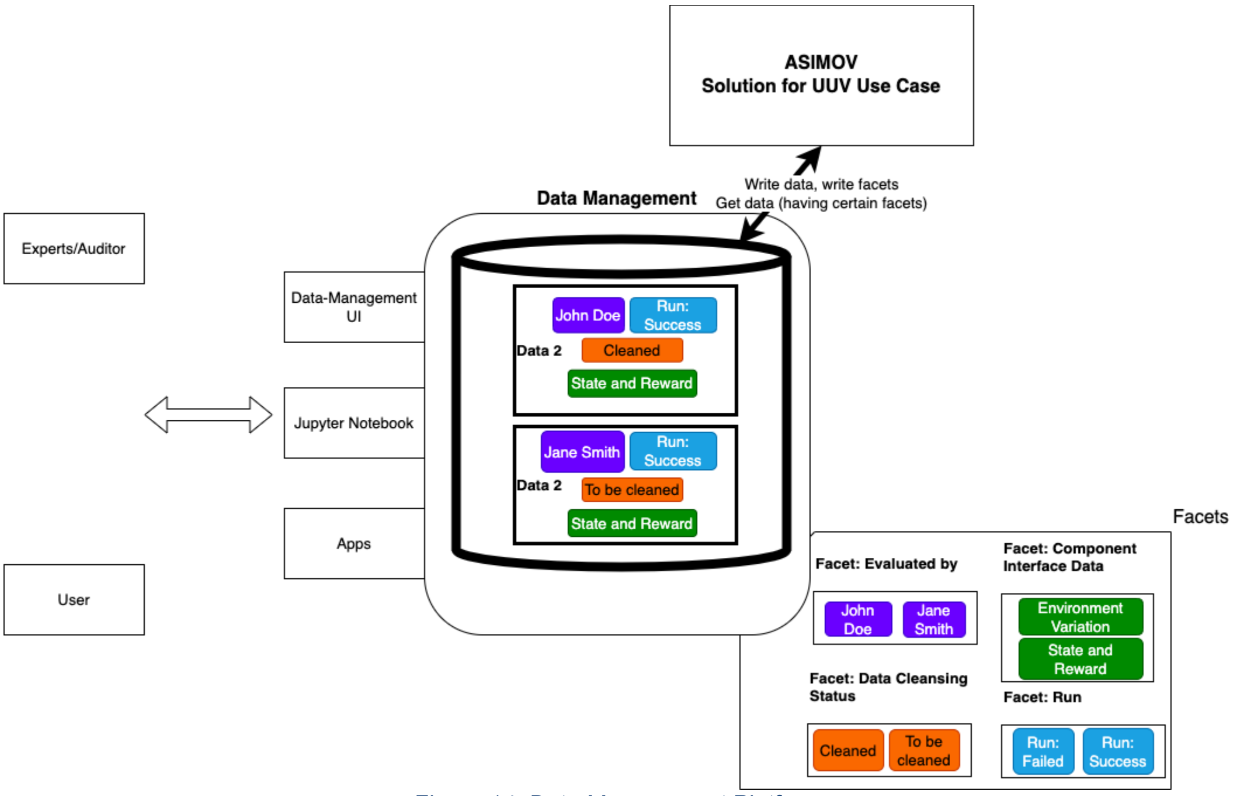


Figure 14: Data Management Platform

3.4.4.2 Data Management Platform

Furthermore, to optimize the utility of ASIMOV's solution, it proves advantageous to establish a comprehensive management and documentation framework that extends beyond the training phase and encompasses the data management aspect as well. This holistic approach ensures that the RL solution is seamlessly integrated into the entire data lifecycle, covering further critical stages such as data preprocessing, ingestion, curation, discovery, and subsequent utilization across a spectrum of analytical applications.
When the RL solution is intricately woven into the data management system, it stands to benefit from additional possibilities for data organization, access control and data validation applications. Moreover, as the foundational data platforms evolve to better support emerging initiatives, such as providing direct compatibility with tools like Jupyter notebooks, it becomes feasible to facilitate the development of AI-driven applications and the creation of further intricate data models.
For example, integrating the ASIMOV solution into NorCom's data management platform DaSense [42] offers the following advantages:

- **Rapid Application Development:** Often there's a need for expeditious application creation based on the available data, with the added capability to design user interfaces for the deployment of ASIMOV's solution. This streamlines the process of building and deploying applications for data analysis.
- **Hierarchical Data Labeling (Facets):** The ability to assign hierarchical labels to the data facilitates structured organization and categorization of data, making it easier to access and utilize effectively. Such labels can be seen as facets of the data. Furthermore, this label can be also used for realizing the component data interface replacing folders allowing flexibility in handling.
- **Integration with Applications:** Integrating with various applications, allows for a cohesive connection between data and applications, i.e., based on the assigned labels. This integration enhances the accessibility and utility of ASIMOV's solution within the broader ecosystem.

A schematic overview is depicted in Figure 14. By fulfilling these data management requirements, ASIMOV's solution can unlock its full potential, enabling efficient application development, structured data organization, and seamless integration with analytical tools and models.

3.4.5 Standards, data formats

Within the ASIMOV solution, the use of standards is considered to be of great importance. Standardized interfaces and common data interchange formats are two important aspects to standards in the ASIMOV solution. Making use of standards ensures applicability of the ASIMOV solution to coming projects.

The ASIMOV solution is realized with the UUV-demonstrator, a fully simulated environment to train an AI-based system. Therefore, the environment needs to be specified in a portable way, so that different software systems can participate in this multi-modal simulation. In ASIMOV this is achieved by utilizing the according standards to describe a traffic scenario. The scenario description is realized as an OpenSCENARIO (see section 2.4.1.1) specification. The OpenSCENARIO specifications describes all dynamic aspects of a scenario and is based on a Road-Network specified within a referenced OpenDRIVE specification (see section 2.4.1.2).

As parts of the UUV-System and also parts of other demonstrators are realized with ROS, also ROS-specific interfaces are used (section 2.4.2.1). This also includes the ROS-specific recording format ROSBAG (section 2.4.2.2).

A generic measurement data format, which could be used to initially record generic system information is the ASAM MDF format (section 2.4.1.4). This is utilized to handle the initial recordings to parametrize the first system instances, but also to support KPI-based validation measurements. Especially when recording information from real vehicles, MDF becomes important. Therefore, this format is also supported by the embedded components to allow comparison of real vehicle-data to simulated values.

3.4.6 Deployment, software aspects

When deploying the ASIMOV solution, there aren't any ASIMOV-specific considerations to bear in mind. Our recommendation is to adhere to the prevailing best practices for deploying multi-container solutions. This entails employing source code control systems such as Git, implementing continuous integration/continuous deployment (CI/CD) strategies, and establishing reproducible, well-documented build processes. Given the likelihood of distributed deployments in many scenarios, leveraging tools like docker-compose in combination with Docker Swarm, or Kubernetes can significantly streamline the deployment process.

4 Realizing an ASIMOV solution for other use-cases

4.1 Starting an ASIMOV implementation

In order to start with an implementation of the ASIMOV solution for another domain, the following steps may help with the process:

Identification of the system to optimize
Develop a clear picture of what parameters of the system should be optimized, both in the digital twin and the physical system. Think about the cost function that can be evaluated after running the system with a given set of parameters. The documents [] and [] may help to identify them.

If possible, create the interface for evaluating a set of parameters early. The interface doesn't have to be final yet – just enough to get the system running.

Grouping existing components to logical components
Identify all existing code, programs, databases, etc. that will be needed for running the optimization process.
Go through them, create containers for them and map them to the logical components as described in section 3.4. Note, that communication between logical components should only happen between their

interfaces to decouple them as much as possible. If, for example, a certain database is needed in two of the logical components think of including it in both components. Or implement an internal interface endpoint that returns the needed data. This ensures that only interface components have to be changed if external components change.

Writing mock components
For each logical component write a mock container, that is accessible via the defined interfaces but does not do any real work but accepts and produces valid in- and output. The resulting containers are very lightweight and should allow local deployment. Build up the architecture that was proposed in section 3.4.1 consisting of a microservice architecture with a worker / queue pattern. Having mock components allows to develop and test in parallel to the other components from very early on. This allows to detect problems early and facilitates the integration of all containers at a later point in time.

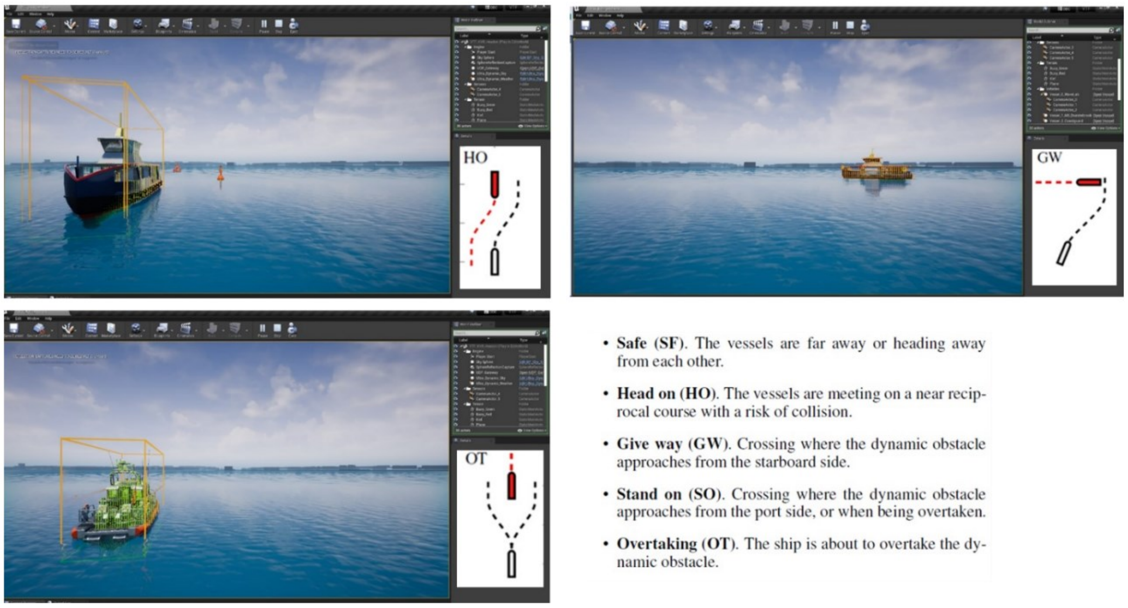
Integration and improving
Aim to get a solution without mock containers running as soon as possible. Having a first version running with a minimal control component facilitates the integration of more complex features step by step.
Apart from continuously improving the optimization process, possible next steps could be: Using existing standards where possible, integrating monitoring and logging, providing a user interface for the control component, moving to distributed storage for higher data volumes, running the setup in Kubernetes, implement user management and access control, etc.

4.2 Applicability in other domains - possible generalizations

The proposed reference architecture already proved its applicability on the two ASIMOV use cases in the form of the electron microscope and the unmanned utility vehicle testing. To further expand the view on its possible applications in different domains, we can also look at other domains such as maritime, rail and agriculture. Those domains can easily be adopted by the same architecture also used in the UUV use case. The modularity of the architecture itself, as well as the use of standard, wherever possible, reduces the complexity of such an adaptation.

- **Maritime:** Ships and vessels operate in a very different environment and with different timescales in their control functions. The step from human operated ships to autonomous ships requires a similar kind of testing, however. Even though ships are not directly tied to roads in the water, their allowed routes are typically defined by buoys, rivers and canals. This can be abstracted to effectively very wide roads, on which a set of rules apply. Combined with the fact that multiple, smaller and larger ships are using these roads combined, like vehicles, pedestrian and cyclists sharing the same road network, critical maneuvers in the maritime domain could be described using similar ASAM OpenX Standards. The usage of an ego ship, with more detailed dynamics can also be represented using a high-fidelity hydrodynamics model, instead of a vehicle dynamics model. The creation of a 3D environment has of course a different focus but can be achieved with the same 3D engines. This also allows for usage of the same sensor models.

  Overall, an adaptation of the ASIMOV approach to the maritime domain can be done with relatively low effort. Early implementations are also already available, as shown in Figure 15.



- **Safe (SF).** The vessels are far away or heading away from each other.
- **Head on (HO).** The vessels are meeting on a near reciprocal course with a risk of collision.
- **Give way (GW).** Crossing where the dynamic obstacle approaches from the starboard side.
- **Stand on (SO).** Crossing where the dynamic obstacle approaches from the port side, or when being overtaken.
- **Overtaking (OT).** The ship is about to overtake the dynamic obstacle.

Figure 15: Using the same Simulation structure in Förde5G [51]

- **Agriculture:** The agricultural domain is quite like the UUV domain. Main differentiation are different vehicle types, with trailers, more wheels, possibly chains and interaction of tools with the environment. These require higher fidelity ground models, as well as completely different vehicle dynamics models, that are less focused on suspension accuracy, but instead offer the possibility to realized complex vehicle architectures with extension arms and tools being operated by hydraulics. There also have been early demonstrators in this domain, using multi body simulation instead of conventional vehicle dynamics. An example can be seen in Figure 16.

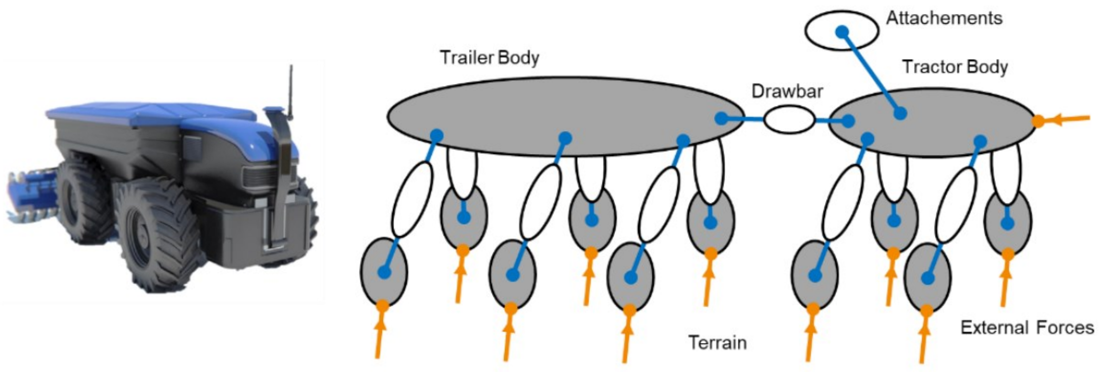


Figure 16: Multi Body Simulation for Agriculture [52]

- **Rail:** This domain limits the number of critical scenarios by eliminating the lateral control of the vehicle. The domain is however much more restricted in terms of real vehicle testing than the other domains, as the vehicle is very reliant on a track network as its infrastructure. But also here, the same abstractions would apply. A track network could be defined, with events happening in a scenario-based manner. It must be kept in mind that the variety of external scenarios that a train must and can react to is very limited. Trams could have more similarities in that regard.

5 Conclusion

This document provided an exploration of the technical aspects for implementing an ASIMOV solution and proposed strategies for establishing a scalable architecture. By addressing relevant technologies, standards, and methodologies, this deliverable aimed to facilitate the transferability of the solution across diverse industries and use-cases. As a desired effect, it accelerates the integration of digital twinning with artificial intelligence in various contexts and propels through collaborative efforts the advancements in digital twinning towards transformative impacts across sectors.

6 Terms, Abbreviations and Definitions

Table 1 - Terms, Abbreviations and Definitions

AI	Artificial Intelligence
ADAS	Advanced Driver Assistance Systems
ASAM	Association for Standardization of Automation and Measurement Systems
AWS	Amazon Web Services
CRG	Curved Regular Grid
DSL	Domain Specific Language
DT	Digital twin
TEM	Transmission Electron Microscopy
HDFS	Hadoop Distributed File System
K8	Kubernetes
KPI	Key Performance Indicator
MDF	Measurement Data Format
ML	Machine Learning
NFS	Network File System
OSI	Open Simulation Interface
PT	Physical twin
RL	Reinforcement Learning
RLA	Reinforcement learning agent
SMB	Server Message Block
SQL	Structured Query Language
UUV	Unmanned Utility Vehicle
VM	Virtual Machine
XML	Extensible Markup Language

7 Bibliography

[1] “ML-Flow,” [Online]. Available: https://mlflow.org/. [Accessed 18 03 2024].

[2] “Tensorflow,” [Online]. Available: https://www.tensorflow.org/. [Accessed 18 03 2024].

[3] “Pytorch,” [Online]. Available: https://pytorch.org/. [Accessed 18 03 2024].

[4] “MPO,” [Online]. Available: https://github.com/daisatojp/mpo. [Accessed 18 03 2024].

[5] “Scitkit-learn,” [Online]. Available: https://scikit-learn.org/stable/. [Accessed 18 03 2024].

[6] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams and N. de Freitas, “Taking the Human Out of the Loop: A Review of Bayesian Optimization,” *Proceedings of the IEEE,* vol. Vol. 104, no. No. 1, pp. 148-175, 2016.

[7] P. I. Frazier, “A Tutorial on Bayesian Optimization,” 10 07 2018. [Online]. Available: https://arxiv.org/pdf/1807.02811.

[8] “Hyperopt,” [Online]. Available: https://hyperopt.github.io/hyperopt/. [Accessed 18 03 2024].

[9] “Optuna,” [Online]. Available: https://hyperopt.github.io/hyperopt/. [Accessed 18 03 2024].

[10] ASIMOV-consortium, “D2.3 Architecture of optimized digital twins for AI-based training,” 2024.

[11] AVL, “Model.CONNECT,” [Online]. Available: https://www.avl.com/en/simulation-solutions/software-offering/simulation-tools-a-z/modelconnect. [Accessed 16 05 2024].

[12] “Association for Standardization and Measurement Systems,” [Online]. Available: https://www.asam.net.

[13] ASAM, “OpenSCENARIO DSL,” [Online]. Available: https://www.asam.net/standards/detail/openscenario-dsl/.

[14] ASAM, “OpenSCENARIO XML,” [Online]. Available: https://www.asam.net/standards/detail/openscenario-xml/.

[15] ASIMOV-consortium, “Proof of Concept Demonstration and Evaluation of Unmanned Utility Vehicle,” 2024.

[16] ASAM, “OpenDRIVE,” [Online]. Available: https://www.asam.net/project-detail/asam-opendrive-v180/.

[17] ASAM, “OpenCRG,” [Online]. Available: https://www.asam.net/standards/detail/opencrg/.

[18] ASAM, “MDF,” [Online]. Available: https://www.asam.net/standards/detail/mdf/.

[19] ASAM, “Mdf block structure,” [Online]. Available: https://www.asam.net/standards/detail/mdf/wiki/.

[20] ASAM, “Open Simulation Interface (OSI),” [Online]. Available: https://www.asam.net/standards/detail/osi/.

[21] ASAM, “Github - OpenSimulationInterface,” [Online]. Available: https://github.com/OpenSimulationInterface.

[22] Google LLC, “Protocol Buffers,” [Online]. Available: https://protobuf.dev/.

[23] ASAM, “OpenODD,” [Online]. Available: https://www.asam.net/standards/detail/openodd/.

[24] ISO/AWI, “ISO/AWI 34503 – Road vehicles – Taxonomy for Operational Design Domain for Automated Driving Systems,” International Organization for Standardization (ISO), Geneva, Switzerland, 2023.

[25] ASAM, “OpenDDD-v100,” [Online]. Available: https://www.asam.net/project-detail/asam-openodd-v100/.

[26] ASAM, “OpenLABEL,” [Online]. Available: https://www.asam.net/standards/detail/openlabel/].

[27] ASAM, “ODS,” [Online]. Available: https://www.asam.net/standards/detail/ods/.

[28] Open Robotics, “ROS - Robot Operating System,” [Online]. Available: www.ros.org.

[29] DDS Foundation, “Structure of the DDS System,” [Online]. Available: https://www.dds-foundation.org/what-is-dds-3/.

[30] ROS.org, “ROS bag format,” [Online]. Available: http://wiki.ros.org/Bags/Format.

[31] ISO/IEC/IEEE, “ISO/IEC/IEEE 42010. Systems and software engineering - Architecture description,” International Organization for Standardization (ISO)/International Electrotechnical

Commission (IEC)/Institute of Electrical and Electronics Engineers (IEEE), Geneva, Switzerland, 2011.

[32] ASIMOV-consortium, "D4.a ASIMOV Reference Architecture," 2022.

[33] ASIMOV-consortium, "D2.2 Methods and tools for Training AI with Digital Twin," 2024.

[34] ASIMOV-consortium, "D4.1 Application to Systems - Methods and Techniques," 2022.

[35] J. Lewis and M. Fowler, "Microservices," 25 03 2014. [Online]. Available: https://martinfowler.com/articles//microservices.html.

[36] S. Newman, Building Microservices, O'Reilly, 2015.

[37] C. R. Martin, Agile Software Development: Principles, Patterns, and Practices, Pearson Education, 2003.

[38] L. Richardson and S. Ruby, RESTful Web Services, O'Reilly & Associates, 2007.

[39] ECMA International, "ECMA-404 The JSON data interchange syntax," 12 2017. [Online]. Available: https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.

[40] World Wide Web Consortium (W3C), "Extensible Markup Language (XML)," 29 09 2006. [Online]. Available: https://www.w3.org/TR/xml11/.

[41] World Wide Web Consortium (W3C), "XML Schema Part 0: Primer," 28 10 2004. [Online]. Available: https://www.w3.org/TR/xmlschema-0/.

[42] NorCom, "KI-Plattform DaSense," [Online]. Available: https://www.norcom.de/ki-plattform-dasense.

[43] NumPy, "BLAS and LAPACK," [Online]. Available: https://numpy.org/devdocs/building/blas_lapack.html.

[44] M. S. Kirkpatrick, "Parallel Design Patterns," [Online]. Available: https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ParallelDesign.html.

[45] C. U. Smith and L. G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley Professional, 2001.

[46] "Flask," [Online]. Available: https://flask.palletsprojects.com.

[47] "Celery - Distributed Task Queue," [Online]. Available: https://docs.celeryq.dev/.

[48] "RabbitMQ," [Online]. Available: https://www.rabbitmq.com/.

[49] "MinIO," [Online]. Available: https://min.io.

[50] ML-Flow, "Common setups," [Online]. Available: https://mlflow.org/docs/latest/tracking.html#common-setups. [Accessed 18 03 2024].

[51] "Förde 5G," [Online]. Available: https://foerde5g.de/.

[52] C. Schyr, N. Braun and S. Oberpeilsteiner, "KI-basierte Optimierung digitaler Zwillinge von unbemannten Nutzfahrzeugen," in *International Commercial Vehicle Technology Symposium*, 2022.