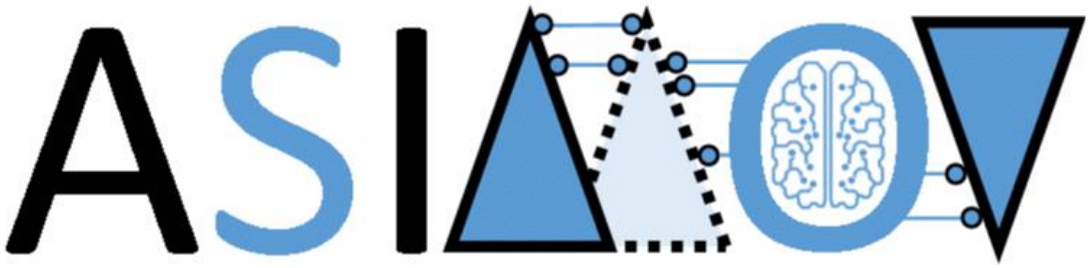
Proof of Concept Demonstration and Evaluation of Unmanned Utility Vehicle

[WP1; T1.3; Internal Report: D1.3 version 1]

public



AI training using Simulated Instruments for Machine Optimization and Verification

Document Information

Project	ASIMOV
Grant Agreement No.	20216 ASIMOV - ITEA
Deliverable No.	D1.3
Deliverable No. in WP	WP1; T1.3
Deliverable Title	Proof of Concept – Use case Unmanned Utility Vehicle
Dissemination Level	external
Document Version	Version 1
Date	2022.11.25
Contact	Niklas Braun
Organization	AVL Deutschland GmbH
E-Mail	Niklas.braun@avl.com

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
Niklas Braun	AVL	niklas.braun@avl.com
Elias Modrakowski	DLR	elias.modrakowski@dlr.de
Andreas Eich	LiangDao	andreas.eich@liangdao.de
Lukas Schmidt	Norcom	Lukas.schmidt@norcom.de
Ezra Tampubolon	Norcom	Ezra.tampubolon@norcom.de
Thomas Kotschenreuther	RA Consulting	thomas.kotschenreuther@rac.de
Sebastian Moritz	Triangraphics	Sebastian.moritz@triangraphics.de

Formal Reviewers

Version	Date	Reviewer
1.0	2022.12.05	Pieter Goosen (TNO/ESI); Jan van Doremalen (CQM)

Change History

Version	Date	Reason for Change
0.1	2022.11.22	Initial Version

Abstract

The testing and calibration of diverse Unmanned Utility Vehicles requires a specialized set of driving tests, that challenge the vehicle. Driven by this idea, this deliverable gives an overview of the Components required to create such a system. It uses the methods and structure that has been proposed by other ASIMOV deliverables. A more detailed Introduction on vehicle testing, as well as unmanned utility vehicles is given before explaining the reasoning behind choosing the ASIMOV approach, including Reinforcement Learning and Digital Twins. After that, the general way of working inside the German ASIMOV Consortium is explained and the working results are presented. This is structured in 4 Workgroups, that have been established to face the various challenges in the UUV proof of concept. Hereby, the workgroups “Environment Simulation”, “Feature Engineering”, “Reinforcement Learning” as well as “Storage and Compute” are presenting their tasks, decisions and implementation results. The document closes with an overall status update and plans for the future.

Contents

Table of Figures

Table of Tables

1. Introduction

1.1 Unmanned Utility Vehicle

Unmanned Utility Vehicles (UUVs) offer a great possibility to deliver goods and lead to a sustainable alternative in public transportation while improving safety. In order to deploy a system of UUVs, for flexible people or goods transport, multiple different UUVs need to be developed and calibrated. Calibration reaches from UUV individual parameters for control units of the drive train up to parameters for communication between the UUVs or a teleoperator, which can act as Incident Management. Considering the large variety of possible UUVs, each designed with a specific purpose in mind, individual manual calibration becomes unfeasible when deploying a fleet of vehicles. To improve scalability of mass fleet deployment, as well as leading to a significant reduction in application cost during development, Digital Twins (DTs) and AI-based system optimization can be used as an enabler for smart mobility solutions. In Figure 1 such a vehicle, specialized for public transport can be seen in an urban environment.

Figure 1 - Unmanned Utility Vehicle example [1]

DTs and AI-based system optimization for UUVs offers a tool to significantly lower the need for testing on proving grounds and public streets and therefore lead to a reduction in cost. ASIMOV not only provides companies the possibility to scale in production, but also serves as a tool for development of alternative vehicle concepts in a more research orientated organisation.

Digital Twinning and AI-based parameter optimization is also not limited to vehicle parameters but can also be applied to the testing itself. The testing process as such can benefit in two ways from the ASIMOV idea. A well calibrated test bed leads not only to more accurate data but also to a wider accessible range of possible tests, including highly dynamic ones, which offer insight into vehicles driving characteristics in safety relevant scenarios. Additionally, optimizing the parameters of the relevant test scenarios leads to a more effective way of gathering meaningful measurement data and therefore reduces the required testing time in the lab and on the road.

The Unmanned Utility Vehicle Use case will be divided into two sub use cases, with the first one focusing on improving the testing itself by automatically creating a test plan that is best suited to test the vehicle

and the second one using this generated test plan to optimize a Sensor setup on an Unmanned Utility Vehicle.

As the access to such a vehicle is rather limited, both sub use cases will rely on virtual validation of the developed toolchain and methods. The autonomous driving (AD) stack itself will not be part of the optimization.

1.2 Scenario-based testing

Testing of Vehicles will be done by using Scenario-based testing methods. Formally, Scenario-based testing can be defined as "the enabling tool for the homologation of automated driving (AD) systems […]. This is mainly due to the aspiration of scenario-based testing for gaining understanding about the tested AD system, which provides support for arguments that shall convince ourselves about its capabilities" [2]. For this, a specific traffic situation is parameterized and used to estimate the vehicles behaviour in this specific situation. This allows the creation of test scenarios, that are interpretable, as they resemble real world situations. Its main advantage is to conduct systematically the verification, validation and accreditation (VV&A) in a controlled environment and putting emphasis on the rarest of events instead of testing the system out on the road.

Each traffic scenario uses a road network, described via the ASAM OpenDRIVE [3] standard and a scenario description, which defines the actors and the movement of the actors in an ASAM OpenSCENARIO [4] file.

1.3 Digital Twin

We aim for the use of DTs in this project as it is unfeasible to train a Reinforcement Learning Agent (RLA) directly on a vehicle testbed. Such a DT serves as a virtual playground for the RLA, where it can test actions and receive rewards and states, without having the cost-intensive operation of a vehicle testbed. The DT therefore has to be a realistic representation of its Physical Twin (PT), so that its interaction with the RLA is similar enough for the RLA to gain meaningful training data out of it. The digital representation of vehicle and environment will be further examined in the "Environment Simulation" Workgroup in 2.1.

1.4 Reinforcement Learning

Reinforcement Learning (RL) [5] is a goal-oriented learning paradigm alternative to the popular supervised learning approach. It consists of iterative cycles of observing – taking action – being rewarded or penalized. An agent gradually optimizes its actions by interacting with a training environment at discrete time steps, typically to maximize its cumulated rewards (return) in order to fulfil some given goals.

More formally, the RL paradigm is designed for an underlying discrete-time Markov decision process (MDP) [6] $\mathcal{M} = (S, \mathcal{A}, P, R, \gamma, D)$, where: $s \in S$ is a state; $a \in A(s) \subset \mathcal{A}$ denotes an action allowed in s; $P(s, a, s') = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$ is the transition probability at any discrete time t (stationary w.r.t. time), which is only dependent on the current state and action (Markovian property); $r_t \sim R(s_t, a_t)$ is the probabilistic reward at time t; $\gamma \in [0,1]$ is a discount rate for the long-term return; $D(s_0)$ is an initial state distribution. Given this, the agent starts in an initial state $s_0 \in S$ and takes an action $a_t \in A(s_t)$ at each time step t. Then, the system makes a transition to $s_{t+1} \sim P(s_t, a_t)$ and the agent receives an immediate reward $r_t \sim R(s_t, a_t)$.

In order to maximize the return $\sum_{t=0}^{\infty} \gamma^t r_t$, RL aims at optimizing the agent's policy, modelled as a probability distribution $\pi(a \mid s)$, by use of state/state-action value functions, i.e., expected returns for following a given policy π. Common RL approaches include (Deep) Q-Learning [5] [7], SARSA [8], Temporal Difference Learning [5] and (direct) Policy Search methods [9].

2. Proposed workflow

The UUV.1 use case, will cover an automated adaptation of the traffic scenarios 3D environment, based on the vehicle's interaction with the environment. The resulting scenarios shall be critical and novel. This shall allow for the creation of an automated test plan generation, that tailors its test plan based on the individual weak spots of the tested vehicle. As the 3D environment is subject to be optimized for the test plan, the focus of the test plans clearly lies on the sensor and perception of the vehicle. The more conventional testing of vehicle driving scenarios with different dynamic scenario parameters, e.g., target

speed, distance offset, etc. Could also be optimized in a similar way, but will not be changed in this use case. This further shifts the focus of this test plan optimization to perception and sensors.
The UUV.2 use case will use the same architecture, but will optimize the vehicles sensor parameters instead of the test plan. This use case will not be described in this deliverable, but will instead be part of a future update.
To coordinate the work in the German UUV use case, four Workgroups were formed to cover all aspects of the envisioned solution. The workgroups are *Environment Simulation*, *Feature Engineering*, *Reinforcement Learning* and *Storage and Compute*. A visual representation of their connections and working fields can be found in Figure 2.

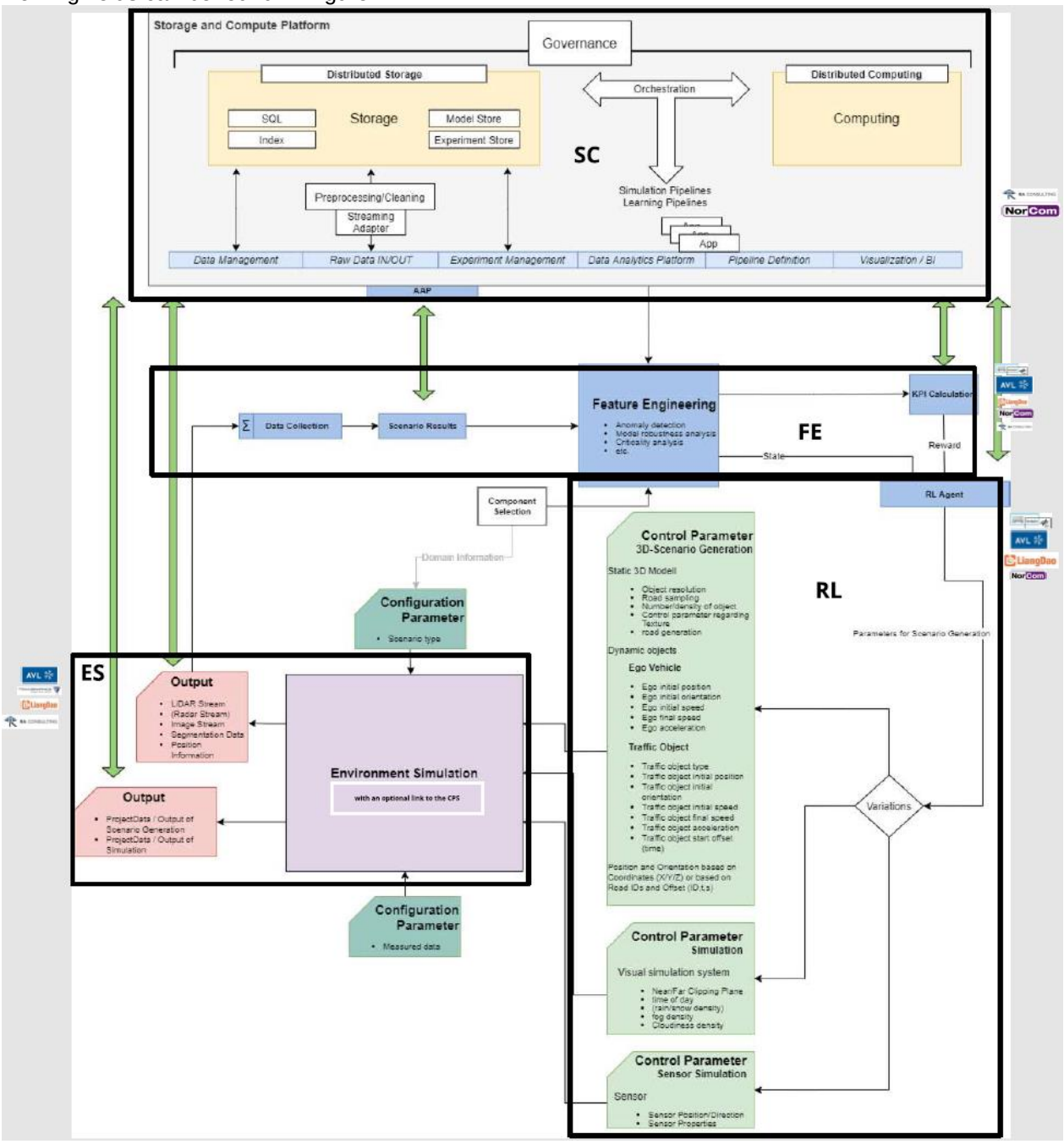


Figure 2 - Overview of the different Workgroups in the UUV Use Case

The next sections will describe the work of these groups in detail.

2.1 Workgroup: Environment Simulation

The environment simulation (ES) workgroup covers every aspect of the UUV Use Case, that is concerned with the simulation of the vehicle in its virtual environment using a co-simulation platform. It can be further split into two parts. One covers the execution and setup of the simulation toolchain, while the other focusses on the creation of a realistic 3D environment, that includes variable elements that can be configured by the RLA.

2.1.1 Environment Simulation using a Co-Simulation Platform

This section will describe every component, that is used to gather simulation data. This includes components that are used in the PT, as well as in the DT, such as the virtual 3D Environment and Scenario Playback, as well as components that are used for training purposes, such as the vehicle dynamics, driving function, perception and sensor. During operation, these components could however serve as models for the PT-connected DT and could be synchronized with their physical counterparts on the testbed. Further concepts on the synchronization of DT and PT can be found in the ASIMOV Deliverable D4.1.

2.1.1.1 Scenario Engine

Every traffic scenario relies on the definition of a certain road layout, as well as the description of how all traffic participants interact with each other. Both of these descriptions can be realized in standards provided by ASAM. The ASAM OpenX Standards rely on an xml-based description language. The definition of the road layout can be done using OpenDRIVE. Here, all the streets, lanes, junctions, etc. can be defined. The positions of the vehicles and other traffic participants can be assigned to the roads and lanes, that are defined in the OpenDRIVE file. Triggers can be defined inside an OpenSCENARIO file as a set of conditions, that need to apply, for certain e.g., speed or lane changes to happen. With multiple of these conditions in place, complex traffic situations can be created.

To playback the behavior of all these traffic participants, a program is needed that can interpret OpenSCENARIO and OpenDRIVE files and move the traffic participants accordingly. In this project, we are using esmini [10] to serve this purpose.

2.1.1.2 Sensor Models

The sensor models are directly integrated into the Unreal [11] environment. Currently, a virtual camera for lane detection is attached to the vehicle. Its properties in terms of Field-of-View, focal length, aperture, sensor dimension and resolution represent a camera used in an AVL Test Vehicle. The camera sensor and its resulting video stream is used for lane detection in this use case. More complex applications, that also rely on sensor fusion would be possible, but are – as the focus of this Use Case is to optimize the testing and not the vehicle and its driving function – not necessary at this point.

Besides the camera, a LiDAR model is implemented, to create virtual point cloud data. The LiDAR implementation consists of a raytracing function, that creates datapoints when hitting nearby objects. The rays are sent out angular equally distributed. The resolution of the sensor can be configured by setting the angular distance and vertical line count. The model does not (yet) take unequal point distribution, or the reflection physics of air and different surfaces or the rotation of the sensor into account.

After further discussions, some of these properties might be included. However, an all-encompassing representation of the sensor or of the environment is not needed and even counterproductive. E.g., already with a slight rotation of a LiDAR sensor, the resulting point cloud data is unusable due to calibration loss. The physical sensor needs to be deactivated or ignored until realignment; a digital representation of this case is not necessary. Another example is the sensitivity of sensors. An increase of sensitivity of actual LiDAR sensors might not be pushed to higher resolutions or to distinguish (slightly) different reflective surfaces or variations due to different air properties. This would increase the amount of resulting data, which is already in the ballpark of several TB per hour, causing problems for data processing and storage. Therefore, including the mentioned properties into the digital twin would increase its complexity and consume resources without further benefit.

Furthermore, a radar sensor is implemented. It is configured in a similar fashion to the LiDAR sensor, but directly outputs the distance and velocity of the object it hits, instead of raw point cloud data. It is used as a data source for the Adaptive Cruise Control (ACC) and Emergency Braking (AEB) function, included in the Driving Function.

2.1.1.3 Driving Function and Perception

The Driving Function and Perception can be seen as the two components that translate everything seen in the environment into concrete actions on how to move inside that environment. The Driving function provides basic Adaptive Cruise Control (ACC), as well as Autonomous Emergency Braking (AEB) and Lane Keep Assist (LKA) functionality. These driving function elements provide enough functionality to test the desired scenario variation effects, while still being easy enough to allow for debugging. It requires relevant information about perceived vehicles, as well as information about the detected lanes to function properly and output steering, as well as accelerator and brake pedal output.

The perception is divided into two parts. ACC and AEB both require information about the current target vehicles' relative position to the ego vehicle, as well as its relative speed. All this information is extracted from object lists, generated by the virtual sensor setup, or – for debugging purposes – from the ground truth data provided by the Scenario Engine. The Perception therefore looks at the provided object lists and identifies which of the vehicles are either currently directly in front of the ego vehicle, or are about to cross its current trajectory. Based on this approximate calculation, the most relevant vehicle is selected and its distance as well as speed are forwarded to the Driving Function.

The Lane Detection forms the second part of the Perception. It uses an Image stream, provided by a virtual camera sensor, to detect the lane markings and calculate the relative distance of the ego vehicle to these lanes.

2.1.1.4 Vehicle Dynamics

Especially when the ego vehicle enters highly dynamic driving situations, like an emergency braking, a detailed model of its physically correct driving behavior becomes very important. In the ASIMOV project, we use a dedicated vehicle dynamics simulation as Physics Engine for that. It provides the possibility to define the kinematics of the vehicle's suspension, weight distribution, tire properties among other relevant parameters of the vehicle simulation model. Its outputs not only provide necessary input to calculate the grip of every tire, but also the resulting position, orientation and movement of the vehicle chassis. This information is then used by the Sensor Engine, which moves the ego vehicle accordingly. As the sensor is mounted on that vehicle, it directly influences the perception.

2.1.1.5 Co-Simulation

The Co-Simulation environment is used to bring together all the different components that are necessary to simulate a vehicle together with its driving function, sensors, vehicle dynamics and environment. It has the purpose of managing interfaces between different tools and taking care of the timing, so that every part of the simulation can run with consistent data. Furthermore, the signals that are exchanged between different components can be logged and stored together, without having to depend on logging possibilities of individual tools.

An important aspect of the co-simulation environment is also the variety of tools that can be integrated in that co-simulation.

2.1.2 3D Environment Variation Pipeline

The goal of the 3D Environment Variation Pipeline is to create a 3D environment that can be used in the simulation environment CARLA. [12] Which variations are performed is controlled by a JSON file, which is provided by the RLA before the pipeline run begins. At the moment it is planned to vary the position of

parking cars, the distance of trees in an area and the number of lanterns in a street. The 3D Environment Variation Pipeline consists of the following four steps, which can also be seen in Figure 3:

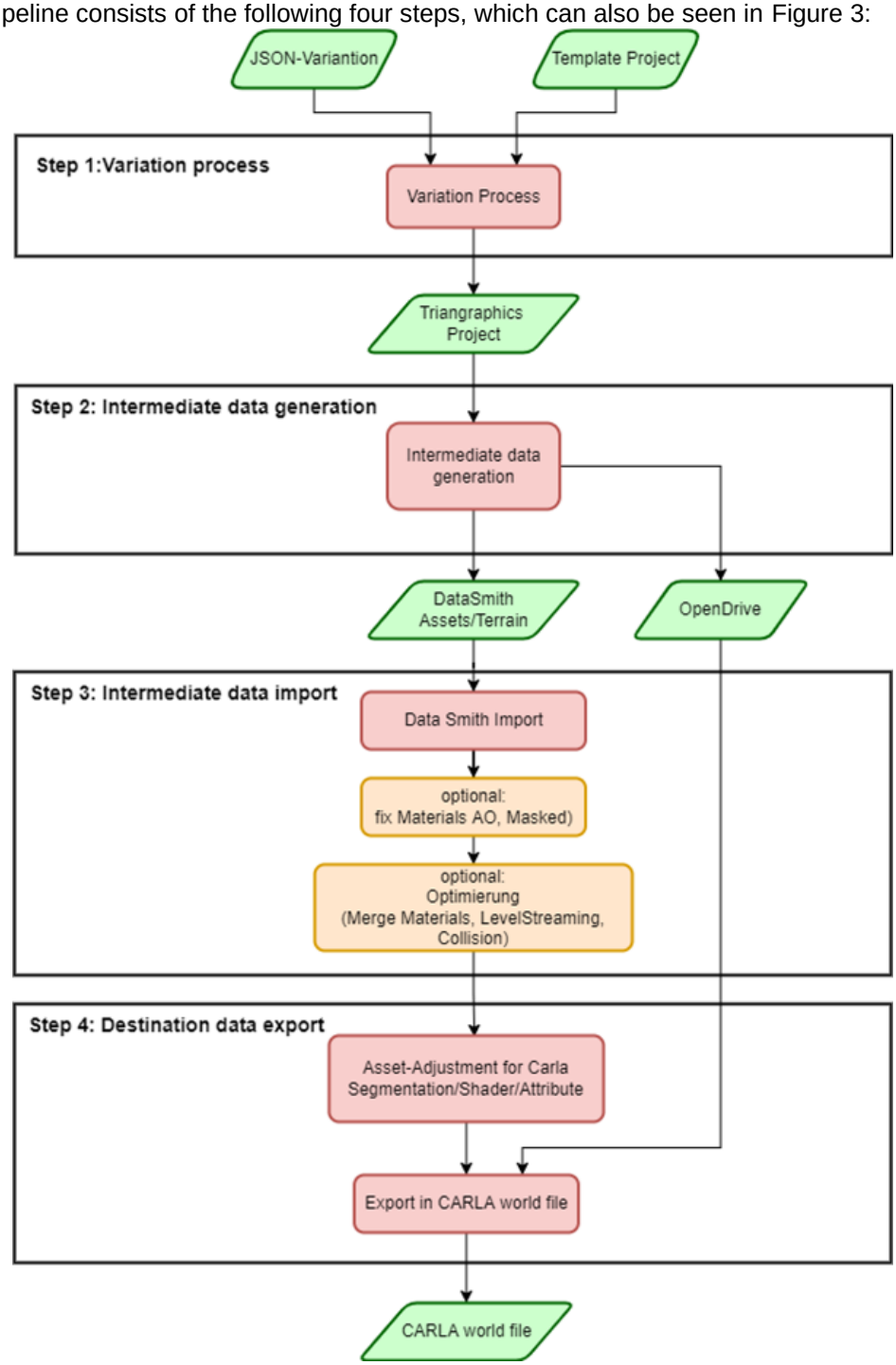


Figure 3 - 3D Environment variation pipeline

2.1.2.1 Step 1: Variations process

The aim of the variation process is to create a Triangraphics (TG) project file that contains all the information for a particular varied 3D environment that can be used to create intermediate files.

To create this TG project file, the variation process receives a JSON file from the RLA. The JSON file contains information about what changes are to be made. The values in the JSON file describe a specific 3D environment and allow the 3D environment to be reproduced at any time.
In addition to the JSON file, the variation process needs a TG project file that serves as a template. In the template file, objects are defined that are not to be changed (e.g. houses, road network). In addition, the template file specifies which elements can be changed and in which area this change may take place.

2.1.2.2 Step 2: Intermediate data generation

The aim of step 2 is to generate intermediate format files that are used for further processing in the following step.

For the generation of these intermediate formats, the TG project file created in step 1 serves as input for the generation process. The most important intermediate formats include the following formats: OpenDRIVE, Datasmith, gltf, OpenSCENARIO. Currently, only the first two formats are supported in the pipeline. It is planned to support the other formats in the future.

2.1.2.3 Step 3: intermediate data import

In step 3, the generated intermediate data from the previous step is imported by the unreal editor in order to process the data in the next step.

The import is done through the Unreal Editor using script files called Blueprints. During the import of the intermediate data, various optional adjustments can be made to improve the appearance of the 3D environment. In addition, optional optimizations can be made to improve performance. These include the merging of materials and settings for level streaming and collisions.

2.1.2.4 Step 4: Destination data export

In step 4, the data imported in the previous step with the Unreal Editor is exported so that it is available in the Carla simulation environment.

The export is also done through the Unreal Editor using Blueprints. The correct segmentation of the objects is achieved during export by sorting the assets into specific subfolders that correspond to a certain segmentation. Shaders and attributes are also adjusted to achieve a realistic look. At the end of the CARLA export, a CARLA world file is created.

2.2 Workgroup: Feature Engineering

The feature engineering (FE) workgroup represents the bridge between raw simulation data and useful features for the RLA to base its decisions about future environment variations on. FE is furthermore used to guide the variations in a specific direction, depending on the desired focus of the tests. The process of creating these features can include expert knowledge.

As the goal of the Use Case is to provide safety critical and diverse Environment Variations, we can divide the FE into two main parts. One covers the Quantification of Safety Criticality via appropriate KPIs, while the other quantifies the diversity of the resulting simulation data.

2.2.1 Criticality KPIs

The criticality KPIs provide the means to quantify if a traffic scenario was critical or dangerous. This helps in identifying scenarios, which are interesting in terms of validating a driving function from a safety or comfort point of view and help to improve the vehicle and its driving function. As there are different types of KPIs, we identified two major safety groups, which we wanted to quantify. The driving function used in this project, can be roughly divided into a lateral and a longitudinal controller and the KPI groups reflect that. An overview of possible KPIs can be found in [13].

Based on the driving function and the testing scenarios, the multiple KPIs have been selected.

For **KPI Group 1 (AEB/ACC)** the following metrics can be used:

- Time to React
  - Minimal Time to Maneuver (TTM): Time to Brake (TTB), Time to Steer (TTS), Time to Kickdown (TTK)
- Deceleration to Safety Time
  - Required Acceleration to maintain a set safety time distance
- Required Longitudinal Acceleration => Required Acceleration
  - Required Longitudinal Acceleration, to avoid an accident
- Brake Threat Number
  - Required Amount of Deceleration in comparison to the vehicle's maximum capabilities

For **KPI Group 2 (LKA)** the following metrics can be used:

- Required Latitudinal Acceleration => Required Acceleration
- Area out of Lane
- Steer Threat Number
  - similar to Brake Threat Number in lateral direction

For each of the KPIs, a respective reference value is also added to normalize the resulting value. Multiple of these KPIs can then be aggregated via a weighted sum, in which the weights offer the possibility to decide how important a specific KPI is in a scenario.

An overview of the proposed workflow can be seen in Figure 4. This overview also includes the anomaly detection, which is presented in the next section.

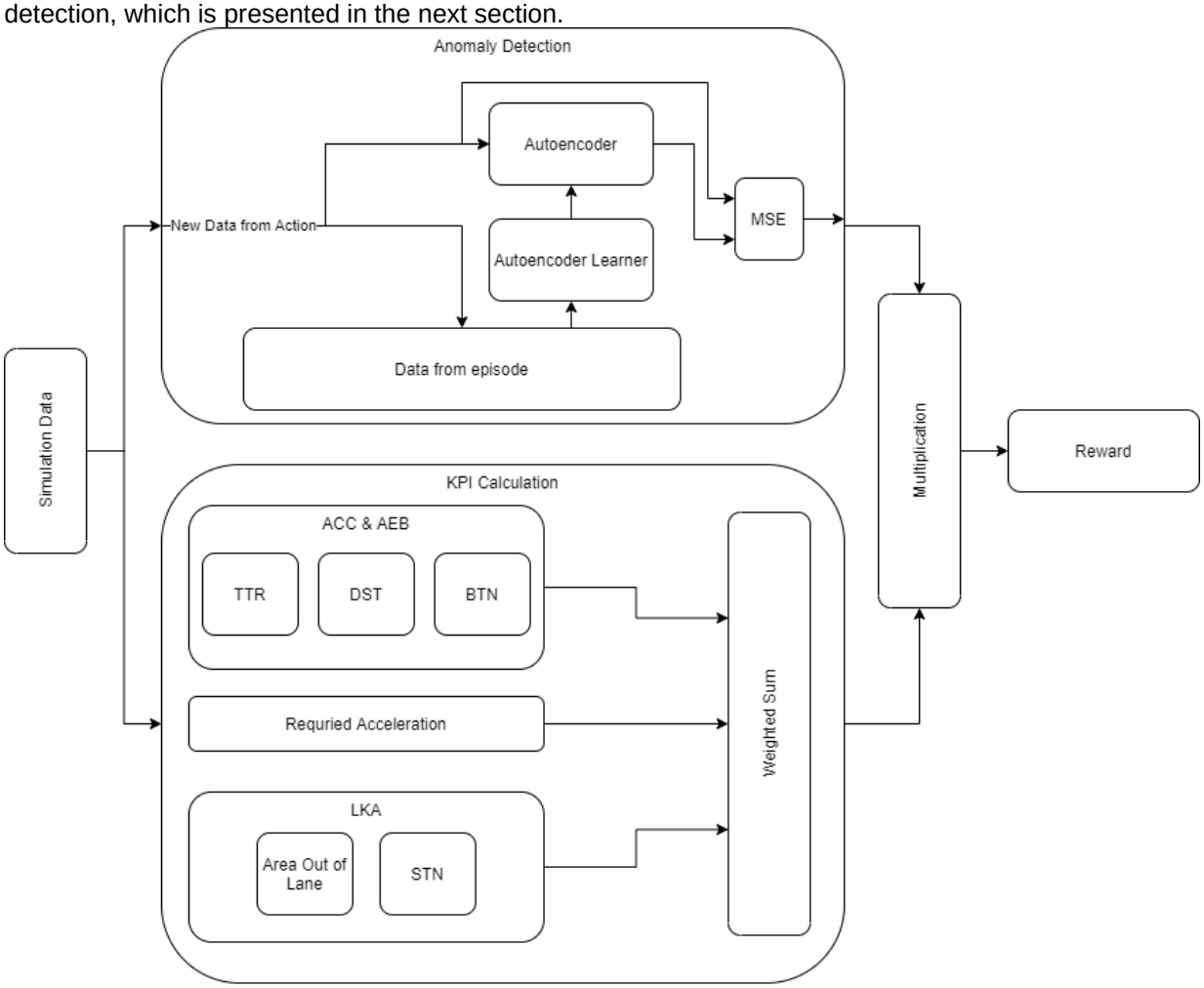


Figure 4 - Overview Feature Engineering in UUV Use Case

2.2.2 Diversity Quantification

If the RLA would only be rewarded for finding critical parameter variations, it would inevitably converge to a single parameter variation, that maximizes the criticality. But as we do not want to find the single best parameter set, but a useful test sequence, including multiple parameter variations, such an approach is not suitable. Instead, the RLA also has to take into account if certain parameter suggestions lead to similar or completely new simulation data. That requires part of the reward to quantify the novelty of a dataset compared to previously experienced datasets. A high level of dissimilarity is desired and shall provide a high reward.

To achieve such quantification, an auto encoder neural network shall be used. The autoencoder network comprises an encoder and decoder part. The encoder compresses high dimensional input data into lower dimensional compressed data, while the decoder uses this compressed data to try to reconstruct the original high dimensional data, with as little deviation as possible. Encoding and decoding happens through multiple layers of artificial neurons. Encoding starts with an input layer, where the number of neurons is defined by the input vector size. Following “deeper” layers contain successively less neurons to achieve the information compression effect. The decoder then reverses that technique and contains multiple layers with an increasing number of neurons. The output layer then has the same number of neurons as the input layer.

A typical use case for autoencoders is therefore data compression or feature extraction from high dimensional inputs such as images. To extract the most relevant features, a diverse, but representative set of training data is required. The more diverse the dataset is, the harder it becomes for the autoencoder to find a common compression technique, resulting in bigger losses, when reconstructing the input data in the decoder.

The sensitivity of the reconstruction loss to unseen and fundamentally different data compared to training data, makes the autoencoder suitable to quantify data similarity. In the case of the UUV use case, data from multiple simulation runs can be seen as a sequence of independent vectors, with the size equaling the number of recorded data channels. A novelty quantification can therefore not only be provided for an entire simulation run, but rather for the individual timesteps of such a run. The overall novelty value can then be calculated by using the sum, average or any other aggregational method across all timesteps of the simulation run.

After the quantification is done, the neural net needs to be adapted, so that it incorporates the new information, provided by the latest dataset. The new data points therefore are added to the common pool of training samples. A retraining of the neural network using samples from this pool then incorporates the new information into the autoencoders internal structure. The limited number of neurons and therefore limited capacity of the neural network itself, leads to generalization. The difference in reconstruction error between “expected” and “unexpected” samples is however still great enough to easily identify novel datapoints.

2.2.2.1 Novelty Quantification Demonstrator

For the UUV use case, a small demonstrator for quantification of these novelty values has been created. For demonstration purposes, demo data has been gathered in a vehicle dynamics simulation. It features the following simulation runs, each containing variable simulation duration and 56 data channels:

- 1 Lap around the AVL test track in Gratkorn (two left-hand corners connected via two straights)
- 1 Lap around the Hockenheimring racetrack
- Driving in a circle with increasing velocity
- Acceleration followed by travel at constant speed
- Acceleration followed by left-hand turn
- Acceleration followed by right-hand turn

The three acceleration simulations are similar during the acceleration phase of the simulation run. This offers the possibility to easily identify if similar phases of simulation runs are distinguishable from novel parts.

The order in which the novelty quantification evaluates the datasets has been varied in the different examples to show the effect on the quantification. The autoencoder was always trained until convergence on the Hockenheim racetrack dataset, to provide a foundation that covers a wide variety of dynamic driving situations.

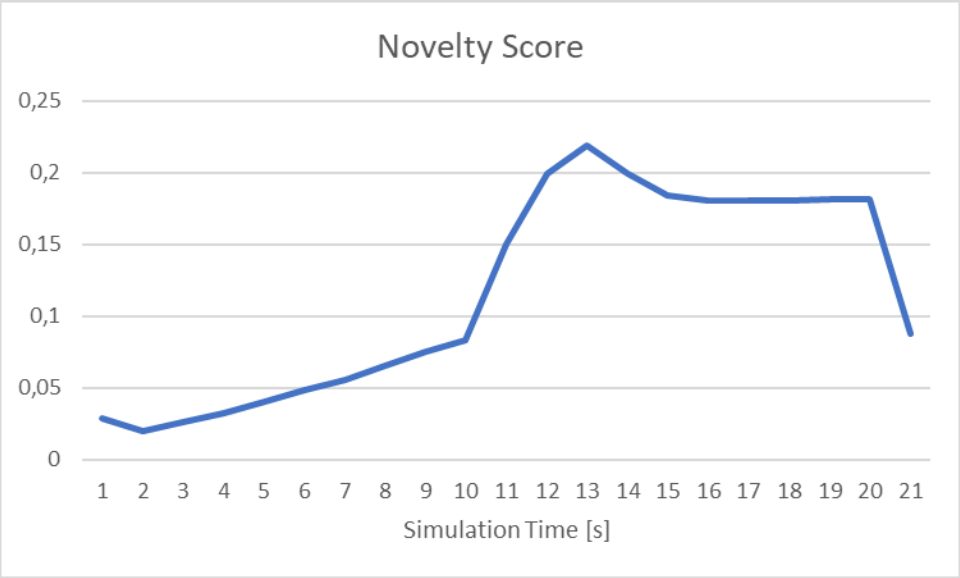


Figure 5 - Novelty Detection - Holding Speed Dataset

When evaluating the first dataset of the vehicle accelerating and holding a constant speed, it can clearly be seen, that the acceleration phase until the 10 second mark, features a lower novelty value than the phase of constant speed after that. This can be seen as a result of the racetrack basis dataset, where traveling at constant speed was not performed.

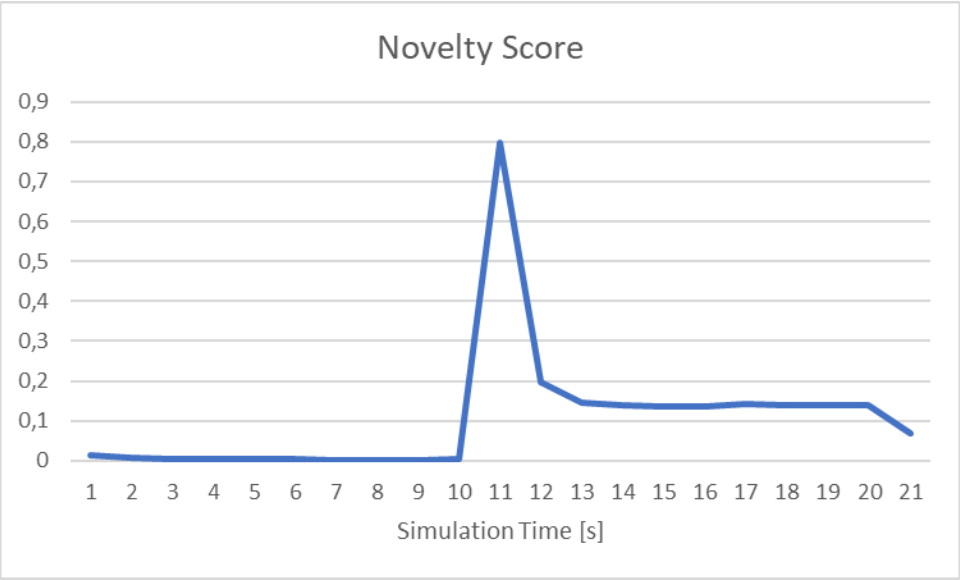


Figure 6 - Novelty Detection - Sharp Turn Dataset

The next simulation run contained an acceleration, followed by a sharp turn. It can be seen, that the acceleration phase has very low novelty values, as it is similar to the one seen in the sample before. As soon as the sudden turn in happens, the novelty value rises, but reduces again, during the constant steering phase. The novelty values correlate with the subjective opinion on which part of the simulation is interesting quite well.

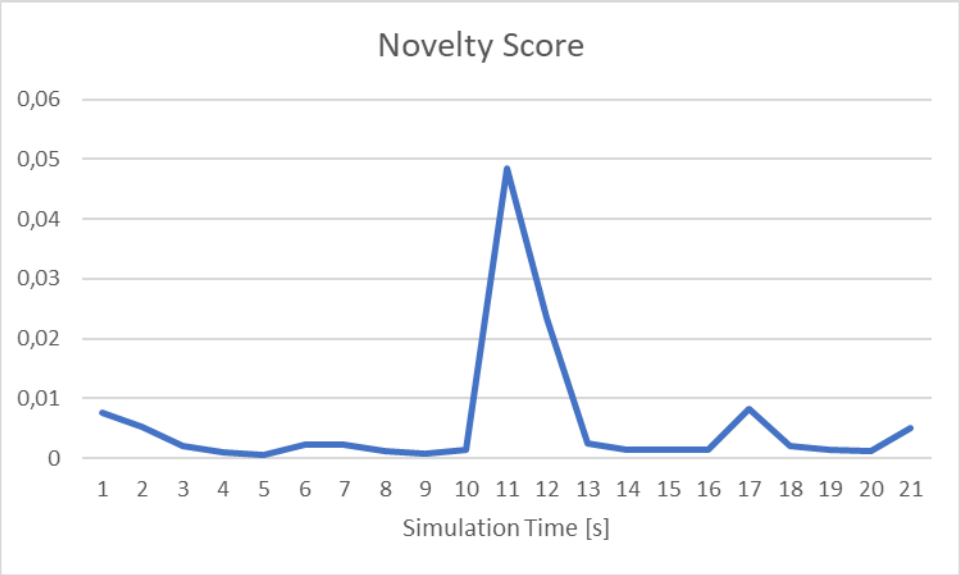


Figure 7 - Novelty Detection - Sharp Turn Dataset 2

Providing the autoencoder with the same sample again, shows again a spike at the 10 second mark, but overall provides much lower novelty values across the whole simulation time.

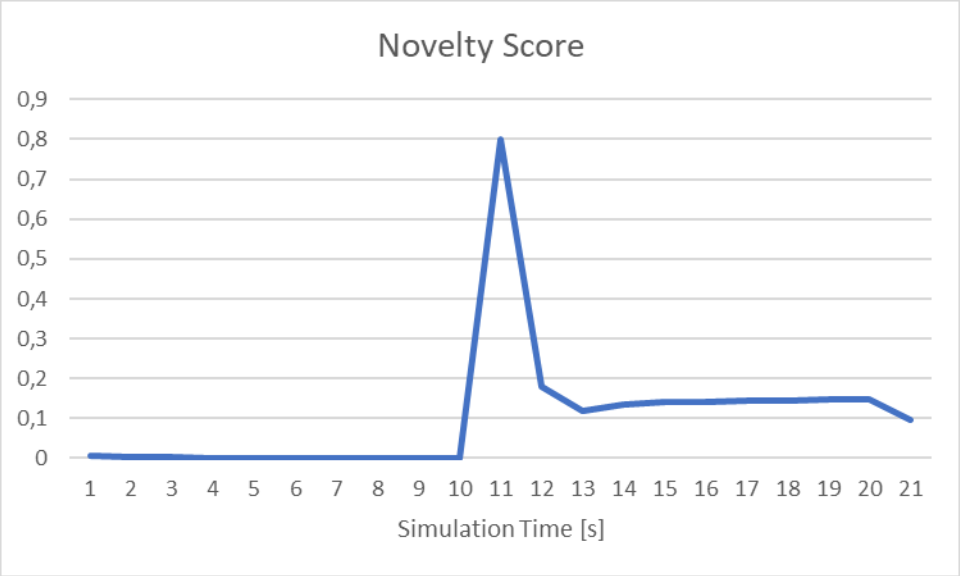


Figure 8 - Novelty Detection - Sharp Turn Different Direction

Changing the direction of Turn in from a left- to a right-hand turn, also quantifies in the expected way. The phase of constant acceleration has a novelty value of almost 0, while the values after turn in, are quite similar to the first turn in example.

The results are promising, so that the novelty quantification is likely to be used in the main pipeline of the UUV use case.

2.3 Workgroup: Reinforcement Learning

The workgroup “Reinforcement learning” is concerned with the development of the RLA which implies its training and application. From a RL standpoint of view and as already introduced in 1.4, the RLA interacts with an “environment” by state, rewards, and actions (see [5]). It received the two previous from the

Feature Engineering while sending the latter to the Environment Simulation (Scenario Generator). In the following the current status of RLA's progress is shown.
The development follows an iterative process. First, requirements are elated based on the needs and possibilities given by the Use Case and the general ASIMOV solution. In continuation an algorithm is selected which meets these requirements and a first prototype is implemented. Based on knowledge that is gained by testing the initial version, refinements and additions will be made.

2.3.1 Requirements elicitation

The following requirements were elated by continuous discussion between the partners involved. In several sessions, questions that had been raised over the course of the project were collected, addressed and answered. In the following the resulting requirements are categorized into (I) System/functional requirements, (II) external interface requirements, and (III) non-functional requirements. If all are met, the RLA should correctly interact with the adjacent system, produce good optimization suggestions, in a reasonable way.

2.3.1.1 System/functional requirements

Table 1 – Functional Requirements on the State

Req. number	Name	Description
G.1	Episodicity	The application of the RLA is structured in episodes. The characteristics of the vehicle change slightly, but we stay inside the ODD.
G.2	Complex environment	We need to assume that the system the RLA shall optimize is highly complex. Thus, the algorithm needs to be capable to deal with this level of complexity.
G.3	Distributed learning	Ideally, the algorithm is capable of being trained in parallel.

2.3.1.2 External interface requirements

Table 2 - External Interface Requirements on the State

Req. number	Name	Description
S.1	State definition	The state shall consist of anomaly values for each data stream (S.3), criticality metrics (S.4), and basic vehicle configurations (S.5).
S.2	State transition	In each iteration, the criticality values and anomaly values of the new scenario are summed to the old state values.
S.3	Anomaly values	Each anomaly value represents the data streams' contribution of the overall anomaly value of that scenario. There is one value per data channel (sensor value, braking, acceleration…) Which data channels are used is TBD.
S.4	Criticality values	Each criticality value represents one unique criticality metric. Which are used is to be determined in the FE workgroup.
S.5	Basic vehicle configurations	Basic vehicle configurations are used to give the RLA some basic knowledge about the vehicle. This information shall be as restricted as possible. Examples are vehicle weight, …

S.6	Number of states	The state space is continuous.
S.7	Initial state	The first state is blank (values = 0) except basic vehicle configurations.

Table 3 - Requirements on the Reward

Req. number	Name	Description
R.1	Input	The RLA shall be able to take one numerical variable as the reward.
(R.2)	Weighted sum	The reward shall be a weighted sum out of all KPIs.
(R.3)	KPIs	The KPIs is first a criticality metric. Later it is a combination of criticality metrics and anomaly value.

Table 4 - Requirements on the Action

Req. number	Name	Description
A.1	One action = one scenario	In the first step, one action of the RLA shall be the input for one scenario.
A.2	Systematic scenario errors	The possible actions shall be designed in such a way that any parameter combination produces a valid scenario.
A.3	Parameters	First: Static environment parameters (time of day, …) Later: More parts of the static environment + parts of the dynamic environment.
A.4	Output type	The actions shall support both discrete and continuous values.

Table 5 - Requirements on Diagnostics

Req. number	Name	Description
D.1	Basic diagnostics	The implementation should be capable of logging the main KPIs to track training progress.

2.3.1.3 Non-functional requirements

Table 6 - Non-functional Requirements

Req. number	Name	Description
N.1	Language	The RLA shall be written in python for easy connecting to Model.CONNECT in the future.
N.2	First step I/O	The RLA shall store a file with its actions and receive the state and reward as files in a folder as well.
N.3	Data efficiency	The algorithm shall be data efficient.

2.3.2 Introduction to selected algorithm

From all requirements, three stand out the most for being restrictive to the selection of algorithms: (I) Data efficiency, (II) Distributed Learning, and (III) support for discrete and continuous actions. While there is a multitude of RL algorithms and many would be capable of fulfilling the requirements, this section

introduces the selected algorithm. For a more thorough explanation of the decision process consult deliverable D3.3 of the project.

Degrave et al. [14] recently introduced a deep RL-designed magnetic controller for tokamak plasma (nuclear fusion technology) which is learned by interacting with a simulated environment and subsequently applied to the physical system of the tokamak in a zero-shot fashion. The approach comprises an actor-critic framework to learn appropriate voltage control commands, based on the current plasma state and control targets.

While the applications are not the same on first sight, it shows parallels to the UUV UC. On one hand a system needs to be controlled/optimized that is highly complex. On the other hand, due to the high costs of running a fusion reactor, a simulated environment is needed for training. The most relevant aspects of the approach are the following:

- Using an asymmetric actor-critic framework with large recurrent critic neural network (NN) to compensate for the non-Markovian properties of the environment and relatively small feedforward actor NN;
- Learning loops for episodic RL, using a distributed architecture with a single learner instance and several actors each running an independent instance of the simulator;
- Applying the Maximum a Posteriori Policy Optimization (MPO) algorithm by [15] as RL approach, and possibly combining this with relative entropy regularized policy iteration [16].

This indicates that MPO is a good candidate for fulfilling the requirements given by the UC except of providing native support for both discrete and actions. In order to fulfill this, the MPO algorithm is expanded by the work outlined in [17]. Without going much into detail, it is not based on a paradigm conversion of e.g., discrete to continuous actions but rather sees the policy as a combination of a policy with discrete and continuous actions.

$$\pi(a|s) = \pi^c(a^C|s)\pi^d(a^D|s)$$

The evaluation and training of both “sub”-policies happens simultaneously (see in following sections below).

2.3.3 Step-by-Step RLA training with MPO and implementation

In the following the MPO algorithm is explained by showing the implementation process step by step. For a more formal and in-depth explanation consider the aforementioned literature [16] [15] and [17]. In general, the training works as seen in Figure 9 in multiple iterations.

1. In each iteration step of “training iteration”, a number of trajectories are sampled and stored in a replay buffer.
2. Experiences are sampled from the replay buffer.
3. With these experiences, the critic is updated (considered step 1 according to [16]).
4. The E-step and M-step (both forming step 2 according to [16]) are performed.
5. Either the learning steps 2,3 and 4 are rerun for further improvement of the actor and critic or the improved actor and critic are set to be the target actor and critic. The target actor/critic is their respective version from the last training iteration. It is used as a benchmark for the improvement.

In the following, the steps are explained in more detail.

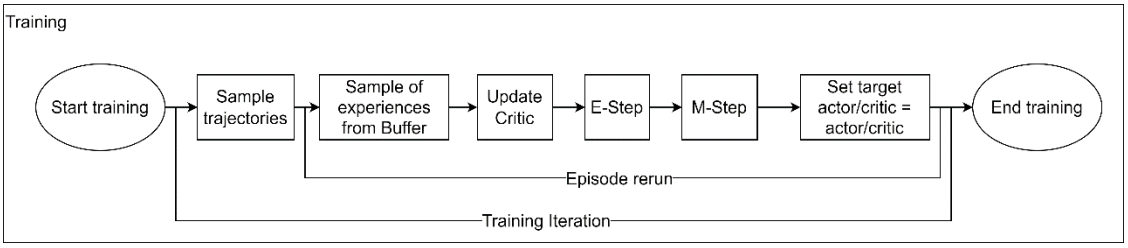


Figure 9 - Process diagram of the RLA's training

2.3.3.1 Sample trajectories

A trajectory is a sequence of experience, where the actor reiteratively suggests an action based on the state of the environment which consequently transitions to a new state and is evaluated with a reward. A trajectory is synonymous to an episode. The sampling of trajectories is straight forward and can be seen in Figure 10.

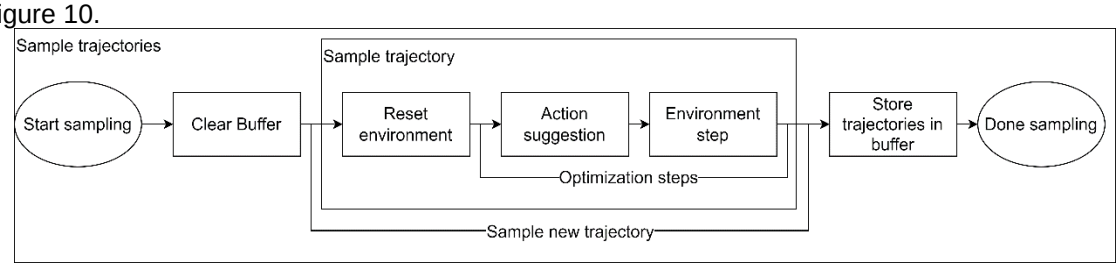


Figure 10 - Process diagram of the sampling of trajectories

The sampling of trajectories happens as follows:

1. At the beginning, the replay buffer is cleared in order to have only trajectories using the newest actor. This is actually not required by the algorithm since the off-policy is capable of dealing with samples from older RLA iterations but it is beneficial to the training time since not all trajectories need to be stored over all training iterations.
2. Sampling a number of trajectories is simply their execution in serial.
    a. The environment is reset to an initial state.
    b. Based on the given state, an action is sampled from the target actor which is the parameterized $\pi(a|s)$. Whether the actor or target actor is used, does not matter at this point since just previous the sampling of trajectories the actor and target actor have been set to be equal.
    c. In light of the action and previous state, the next state and reward of the last action is returned from the environment. If the goal of the optimization is not reached the next action is suggested.
3. All trajectories are stored in the episode buffer as experiences. In line with [18] we see the tuple s_t, a_t, r_t, s_{t+1} as an experience of timestep t.

2.3.3.2 Sample experiences from Buffer

In continuation to the sampling of trajectories, within each "episode rerun" B experiences are sampled from that buffer.

2.3.3.3 Update critic

The critic represents the parameterized approximation of the Q-function $Q(s,a)$. Its update is explained in more detail as the "Policy evaluation" (Step 1) in [16]. In essence, the critic's NN's parameters are optimized via gradient descent in such a way that the squared difference between the actual Q-value of the sampled actions and states and the estimated Q-value calculated by the critic is minimized. This implementation as shown in Figure 11.

1) Sample from the target actor I alternative actions based on the B next states s_{t+1} drawn from the buffer.
2) Get expected Q-value Q_{Enext}. of state-action pairs B I actions drawn in step 1. In addition, the mean over all Q-values from each B which represents the expected Q-value of the next states called
3) Calculate the "actual" Q-value Q_a of the performed actions a_t and states s_t by
$$Q_a = r + \gamma \cdot Q_{Enext}$$
with reward r and discount factor γ.
4) Use the critic to calculate the estimated Q-values Q_{est} of the state s_t and action a_t.
5) Calculate the loss of Q by the squared difference between Q_a and Q_{est} and apply gradient descent to minimize it.

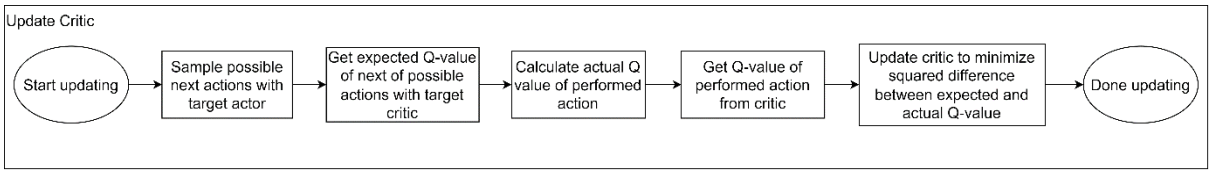


Figure 11 - Process Diagram of updating the critic

2.3.3.4 E-Step

The E-Step can be seen as the preparation for the following M-step. It is referred to by [16] as "finding action weights". Section 4.1 in [16] and section 3.2 in [15] explain this step in more detail, but essentially "we want to first re-adjust the probabilities for the given actions in each state such that better actions have higher probability. These updated probabilities are expressed via the weights q_{ij}…" [16]. As proven in [16], q_{ij} can be expressed by the soft-max over Q-values adjusted by the temperature η:

$$q_{ij} = q(a_i, s_j) = \frac{\exp\left(Q^{\pi^k}(s_j, a_i)/\eta\right)}{\sum_i \exp\left(Q^{\pi^k}(s_j, a_i)/\eta\right)}$$

With $i = 1 \dots N$, $j = 1 \dots K$, and Q^{π^k} being the Q-function (target critic). The temperature η which is a term introduced by solving the constraints on the weights which limits the change in policy (see Appendix D of [15]) can be found by solving the convex dual function:

$$\eta = \text{argmin}_\eta \, \eta\epsilon + \eta \sum_j^K \frac{1}{K} \log\left(\sum_i^N \frac{1}{N} exp\left(\frac{Q(s_j, a_i)}{\eta}\right)\right)$$

The heat factor η is a term introduced from solving the constraints on the weights which limits the change of the policy (see Appendix D of [15]). The equations are calculated as shown in Figure 12.

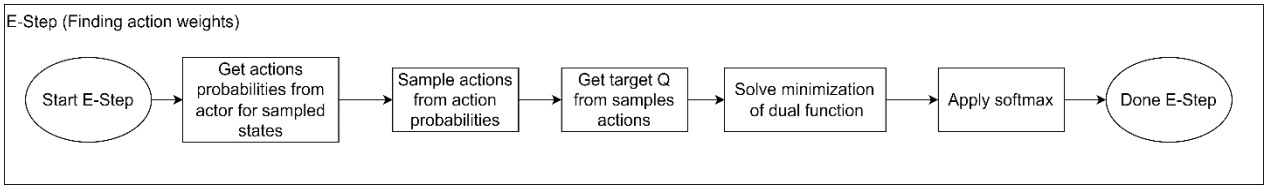


Figure 12 - Process diagram of the E-Step

1. K states s_j are sampled from the replay buffer and the distributions are obtained from the actor. Namely for continuous values, the vector of means μ of size equal to the number of continuous actions, the continuous actions covariance in form of A as the triangular matrix via its Cholesky decomposition, and the probability of options of the discrete actions p_{probs}.
2. Sampling of N actions a_i from the given probabilities.
3. Get Q-values for each pair of (a_i, s_j).
4. Find solution to η by minimizing dual convex function (see above).
5. Apply soft-max to by η adjusted Q-values (see step 3).

2.3.3.5 M-Step

The M-Step of the policy update is the iterative fitting of the policy represented by the actor to the re-weighted action probabilities. This is synonymous to the minimization of the Kullback-Leibler divergence (KL) between the two said probability distributions (sample-based distribution from the E-Step and the parametric policy represent by the actor).

$$\pi^{k+1} = \arg\max_{\pi_\theta} \sum_j \sum_i q_{ij} \log \pi_\theta(a_i|s_j)$$

As pointed out in [16]: "sample based maximum likelihood estimation can suffer from overfitting". Thus, the change of the parametric policy (the actor) needs to be limited.

$$\sum_j \frac{1}{K} KL\left(\pi^k(a|s_j)||\pi_\theta(a_i|s_j)\right) < \epsilon$$

As shown in [16] this optimization in combination with a constraint can be solved using Lagrangian Relaxation. However, as we deploy an actor capable to provide continuous and discrete actions, the Lagrangian equation needs to be adapted according to [17]. Here the change of the parametric policy is limited in three separate ways: once for the categorical distribution of the discrete actions and for the means and covariance of the multivariate distribution from which the continuous actions are sampled each.

$$L(\theta, \eta_{c,\mu}, \eta_{C,A}, \eta_d) = \sum_j \sum_i q_{ij} \log \pi_\theta(a_i|s_j) + \eta_{C,\mu}(\epsilon_{c,\mu} - \mathrm{T}_{c,\mu}) + \eta_{C,A}(\epsilon_{c,A} - \mathrm{T}_{c,A}) + \eta_d(\epsilon_d - T_d)$$

with $\eta_{C,\mu}, \eta_{C,A}, \eta_d$, $\epsilon_{c,\mu}, \epsilon_{c,A}, \epsilon_d$, and $\mathrm{T}_{c,\mu}, \mathrm{T}_{c,A}, T_d$ corresponding to the Lagrangian multipliers, the allowed expected change of the policy in KL divergence, and the sample-based KL divergence respectively for continuous mean and covariance $(c, \mu \; or \; c, A)$ and discrete (d) actions. It is then solved as

$$\max_\theta \min_{\eta_{C,\mu}>0, \eta_{C,A}>0, \eta_d>0} L(\theta, \eta_{C,\mu}, \eta_{C,A}, \eta_d)$$

Iteratively solving the inner and outer optimization independently. This process is implemented as shown in Figure 13 as follows:

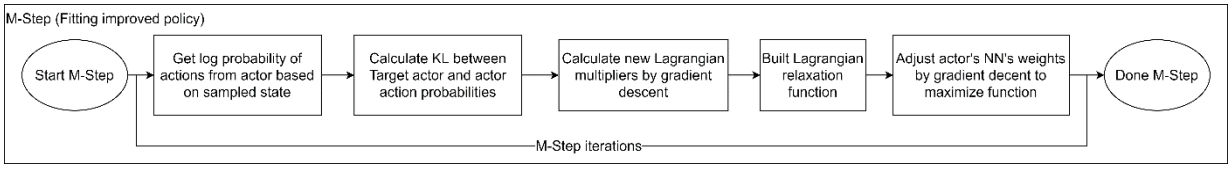


Figure 13 - Process Diagram of the M-Step

1. Get logarithmic probability (log prob) of each action from the current actor by getting μ, A and p_{probs}, convert them into torch distribution and obtain log prob.
2. Calculate average KL divergence of discrete actions (categorical KL) and continuous actions (Gaussian KL). The categorical KL is given by

$$\mathrm{T}_\mathrm{d} = KL_D(p_1||p_2) = \frac{1}{K}\sum_{k=1}^{K}\sum_n p_{1_{n,k}} \log\left(\frac{p_{1_{n,k}}}{p_{2_{n,k}}}\right)$$

   with K being the number of sampled states, n the number of options per discrete action, and p_1 and p_2 being the probabilities given by the actor and target actor respectively. The KL divergence of two multivariate Gaussian distribution, as proven in [19], can be expressed as follows:

$$KL_C(p1(\mu_1, A_1)|\,|p_2(\mu_2, A_2)) = \frac{1}{2}\left(\log\left(\frac{det\, A_2}{det\, A_1} - n + tr(A_2^{-1}A_1) + (\mu_2 - \mu_1)^T A_2^{-1}(\mu_2 - \mu_1)\right)\right)$$

   Note that in this case, A is the covariance matrix instead of the Cholesky decomposition of the covariance as given by the actor's implementation. In order to find the divergence specifically between either the means or covariances, the following is applied respectively.

$$\mathrm{T}_{c,\mu} = KL_C(p(x\,|\mu_1, A_1)||\,p(x|\mu_2, A_1)\,)$$
$$\mathrm{T}_{c,A} = KL_C(p(x\,|\mu_1, A_1)||\,p(x|\mu_1, A_2)\,)$$

3. Update Lagrangian multipliers by gradient descent according to

$$\eta_{C,\mu_{t+1}} = \eta_{C,\mu_t} - \alpha_{c,\mu} * \frac{\partial \mathrm{L}}{\partial \eta_{\mathrm{C,\mu}}} = \eta_{C,\mu_t} - \alpha_{c,\mu} * (\epsilon_{c,\mu} - \mathrm{T}_{c,\mu})$$
$$\eta_{C,A_{t+1}} = \eta_{C,A_t} - \alpha_{c,A} * \frac{\partial \mathrm{L}}{\partial \eta_{\mathrm{C,A}}} = \eta_{C,A_t} - \alpha_{c,A} * (\epsilon_{c,A} - \mathrm{T}_{c,A})$$
$$\eta_{d_{t+1}} = \eta_{d_t} - \alpha_d * \frac{\partial \mathrm{L}}{\partial \eta_\mathrm{d}} = \eta_{d_t} - \alpha_d * (\epsilon_d - \mathrm{T}_d)$$

   Note that η_{C,μ_t}, η_{C,A_t}, and η_{d_t} are called α_μ, α_Σ, and α respectively. The corresponding α is given as α_*_scale in the code.

4. Build Lagrangian equation according to the function above.
5. Apply gradient decent to the Lagrangian equation. The optimization done within the M-Step is repeated a fixed number of times.

2.3.4 Actor description

The actor is represented by two parts: On one hand, it consists of a neural net which takes the states given by the environment and outputs the weights of options for each discrete action and the mean μ and covariance in form of its Cholesky decomposition A. This is shown as illustrated in Figure 14. On the other, the actor consists of simple transformation functions which take the values describing distributions from the neural net, create such distributions (see right side equations of Figure 14) as functions and sample the action from it.

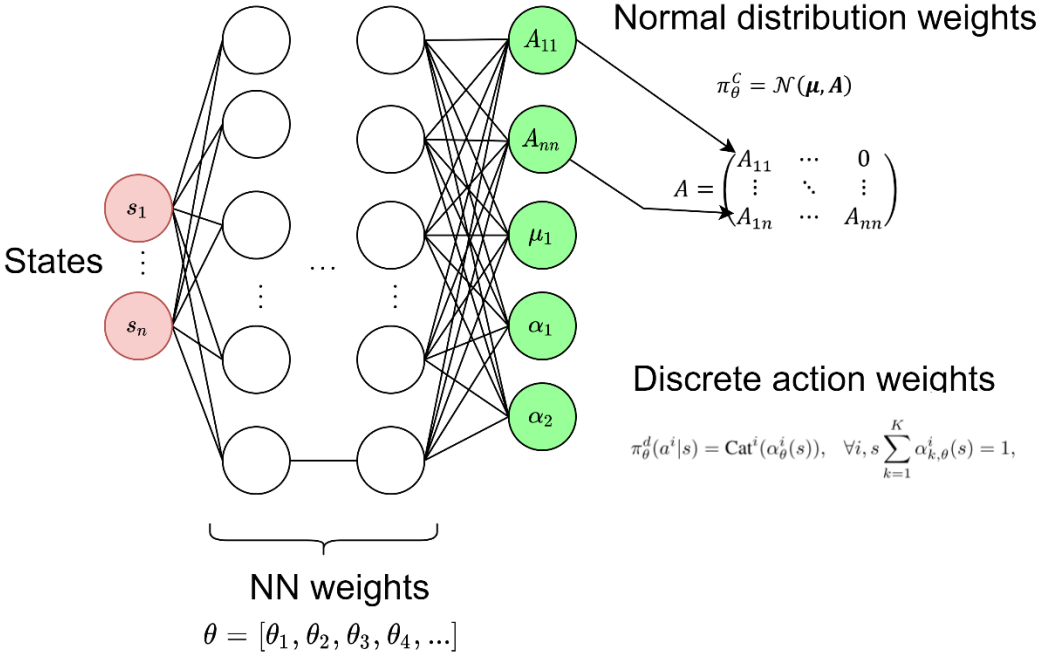


Figure 14 - Visualization of the actor's NN layout

2.3.5 Critic description

The critic is represented by a neural net similar to the actor. However, states and actions are used as inputs and result in a single output which represents the estimated Q-value (see Figure 15).

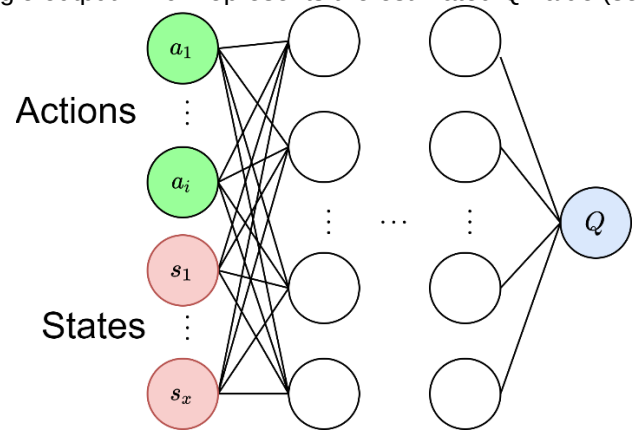


Figure 15 - Visualization of ciritc's NN layout

2.3.6 Performance verification

In order to test the correct functioning of the algorithm, a controlled and known environment needs to be used to isolate errors of the environment from a poor performance of the MPO algorithm. Therefore, a standard OpenAI gym library [20] is used to verify the correct training. Since it does not provide an environment which takes both discrete and continuous actions, a superposition of a discrete and continuous environment is used. For this, the "LunarLander" is used in its discrete and continuous action version. The observations are concatenated together while the resulting action are split again according to their nature.
As a base-line default values have been used (blue line in Figure 16) but also alterations in the training as well as of the actor's neural net were investigated. The tests have been run for 20000 training iterations but showed that already after 1000 runs (see figure Figure 16) the algorithm has converged to the maximum reward possible. This was repeated with different variation of training strategy as well as changes to the actor. The training results indicate that an increased re-use of the sampled data is highly efficient while an increased size of the actor's neural net is in this case unnecessary and has a negative impact on the training.

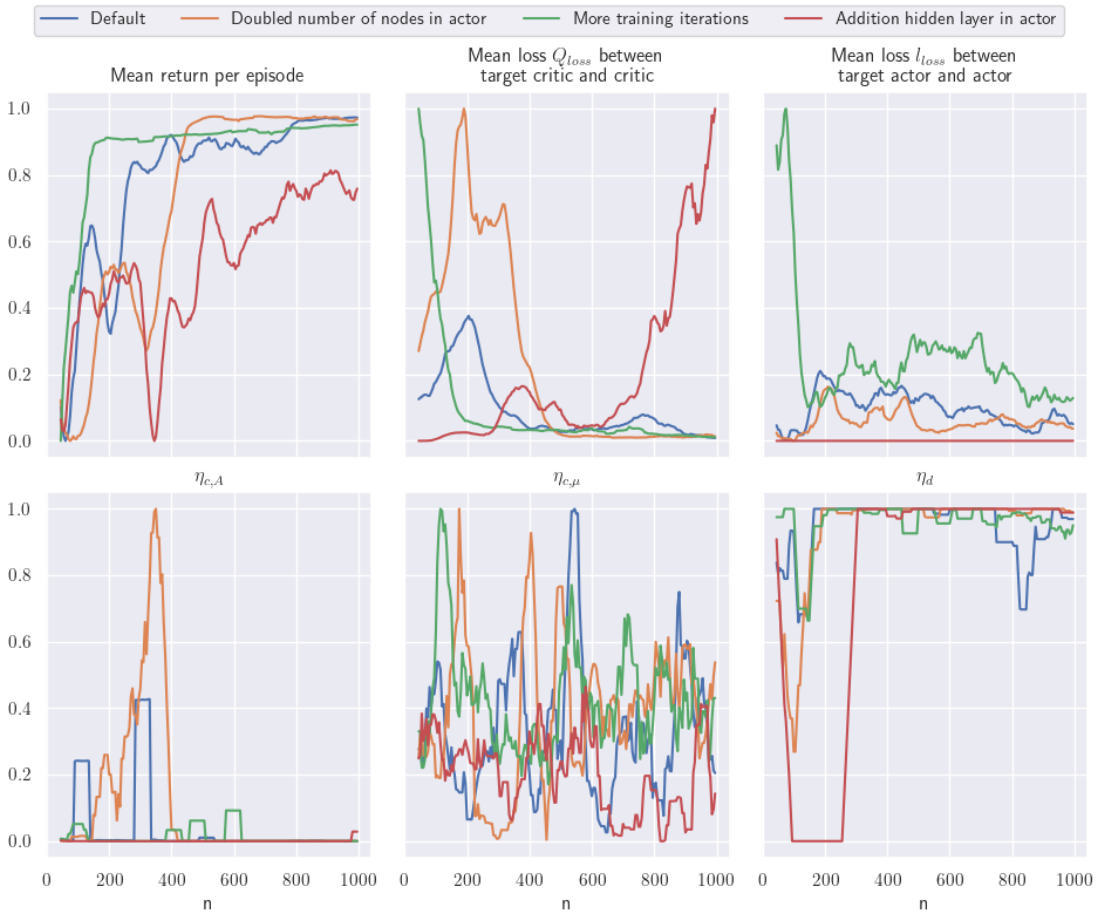


Figure 16 - Key performance indicators for training over the training iteration steps with different configurations of the training or actor

Mean return is the average of the sum of rewards over an episode, Q_loss is mean difference of Q-values from the critic and target critic (see 2.3.3.3). l_loss is the resulting average difference between the policy of the actor and the target actor (corresponding to the result of the Lagrangian equation of 2.3.3.5. The three Lagrangian multipliers for the covariance and average of the discrete actions and discrete actions (lower row, left to right) (see 2.3.3.5) indicate where points of improvement can still be found.

2.3.7 Action conversion

While the algorithm provides primarily continuous values between 0 and 1 or discrete options in terms of integers (0,1,2,3…), the scenario generator requires the actions to be in a specific format, range, and data types. Specifically, it requires the objects to be passed as a json5-file containing a hierarchy of information. Without going into much detail, the format and content of the file, which represents the action as the optimization of the scenario, is given by a template.
The hierarchy is built as follows. Recursively, every item given as a string (e.g. “StaticModel”) has content either as another dictionary containing items (curly brackets) or a list of items (squared brackets). An item can also have a value associated with it. Items with a discrete value (e.g. “ruleID”) have their options listed as a string. Item containing a continuous value (e.g. “s”) are represented by a list which indicated the lower and upper bound.
An algorithm recursively walks through the file, identifying the hierarchy and continuous and discrete actions (including the constraints). From there, the action space is determined. In order to provide the action and therefore the file, this algorithm is reversed to create the hierarchy and substitute the constraints with concrete values.

2.3.8 State and reward conversion

The state and reward are currently supplied via a file which contains a simple list of values. These are read by the algorithm after a new state is requested.

2.3.9 Future improvements

Since this is the first prototype, some aspect is simplified but can be improved in the future. After closing the training loop, integrating the Scenario generator and the Feature Engineering and testing the working of and successful training in a small case study, big improvements are expected for the following issues.

On one hand, with the goal of being able to have the majority of parameters optimized by the RLA during the optimization, it is easy to say that the dimensionality of the action space gets out of control fast resulting in inefficiencies. This is especially true when dealing with discrete actions with possibly hundreds of options. Therefore, new techniques need to implemented. An example is the work of [21] which deals with large discrete action spaces which motivates their work with the large systems like Youtube or Amazon where new items are added constantly resulting in a huge number of options for recommendation. The approach “leverages prior information about the actions to embed them in a continuous space upon which it can generalize. Additionally, approximate nearest-neighbor methods allow for logarithmic-time lookup complexity relative to the number of actions.” [21]

The second issue is the serialized sampling of trajectories. Due to the resources the simulation of scenario requires, parallelization is key. Thanks to the already implemented replay buffer, the sampling of trajectories with multiple copies of the actor can be offloaded to clusters with the results be fed into it. Then, the training can be done on demand resulting in a new version of the actor and critic.

2.4 Workgroup: Storage and Compute

2.4.1 Introduction

Storage and Compute workgroup is concerned with the development of an RL architecture utilizing DT. The emphasize of this work group is on specifying the interplay between all the components, i.e., reinforcement learners, feature engineering module, and also the scenario generator in the DT.
In the following subsections, we define the prototypes for the components and also the interfaces between them in the platform. In doing this, we address certain crucial soft requirements such the scalability and maintainability of the prototype. We also address in the following some possible extensions for the basic models such as queueing of the learning process, tracking and storing the RL experiments, and also management of data.

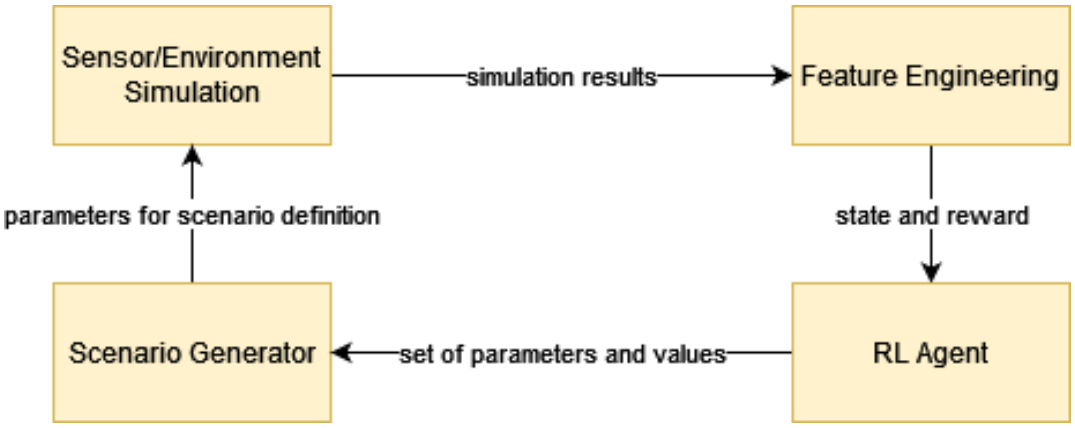


Figure 17 - UUV Use Case Process Diagram

2.4.2 Starting Structure for First Prototype

In our prototype of the RL architecture, we realize the RL components (Feature Engineering, RLA and Scenario Generator) as the so-called docker containers [22] having executable python files containing the designed algorithms. The applications within the containers are supported by the flask framework [23] so that they can mutually communicate via HTTP request.
The

Docker containers constitutes a state-of-the-art way for abstractions at the app layer and can be seen as a Linux container technology. As codes, dependencies, and environments are packed together in docker containers, there is no need for substantial preparation for deploying them. In this hindsight, it is easy to distribute the RL architecture realized with this technology to the project partners, e.g., for testing purposes. Moreover, multiple containers can run on the same machine as isolated processes, so that the separation between the RL components can be implemented by means of this technology.

Another reason for using the docker containers for realizing the RL architecture is that the possibility to additionally use the Kubernetes framework [24] for management purposes. Kubernetes constitutes a state-of-the-art technology for orchestration of containerized applications. It supports the automatic deployment, scaling, and management of Docker containers which suits perfectly with the ASIMOV's goal of creating a scalable and automated RL methodology with DT.

The specific first prototype of the RL architecture realized within this workgroup is a microservice architecture as depicted in the following figure:

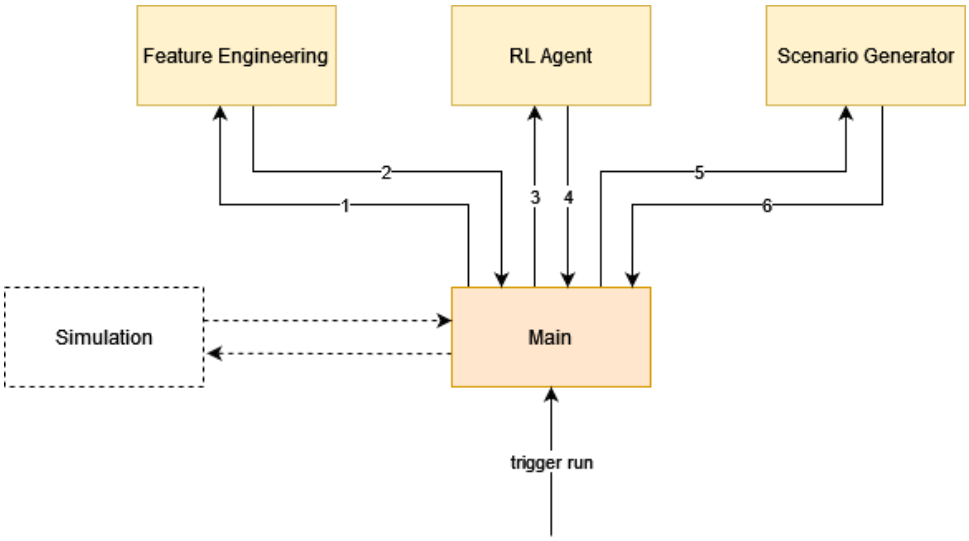


Figure 18 - Microservice Architecture

As already mentioned above, all the components, i.e., Feature Engineering, RL Agent, and Scenario Generator are isolated docker flask containers on the same host machine. Those containers are addressable via HTTP request. In our first prototype, the management of a complete training run is done by a main component in the host machine. User of the RL solution can only interact with this component. Triggering a RL run via the main component causes the main component to send a HTTP-request to the feature engineering component for processing the available training data. Immediately after completion of this task, the feature engineering component sends a corresponding response to the main components. In case the feature engineering task is successful, the main components can then send a request to the RL agent which in turn triggers the learning process defined by the python file given in the docker container of the RL Agent component. The completion of the learning process causes the RL agent container to send a response to the main component triggering then via HTTP request the generation of learning scenarios done by the scenario generator container. Again, the completion of this task yields in a response from scenario generator to the main component. Finally, the generated scenarios can then be used by a simulation instance to generate training data for the next learning round.

In our first prototype the (processed) data are available in the host machine and can be accessed by the RL components via a mount point. Moreover, the components have the ability to store the processed data in the host machine again via the mount point. This data exchange process can later, if further isolation of the components needed, be also implemented via HTTP payload and content negotiation. In this framework, the processed data is firstly only given in the corresponding processor container. For further processing, the processor container then sends the data as a payload to the next processor component.

Further extension of the above prototype which we aim to implement is the introduction of a queue infrastructure. For this purpose, we will utilize the RabbitMQ package [25]. The specific approach that will be done in the next step is illustrated in the following figure:

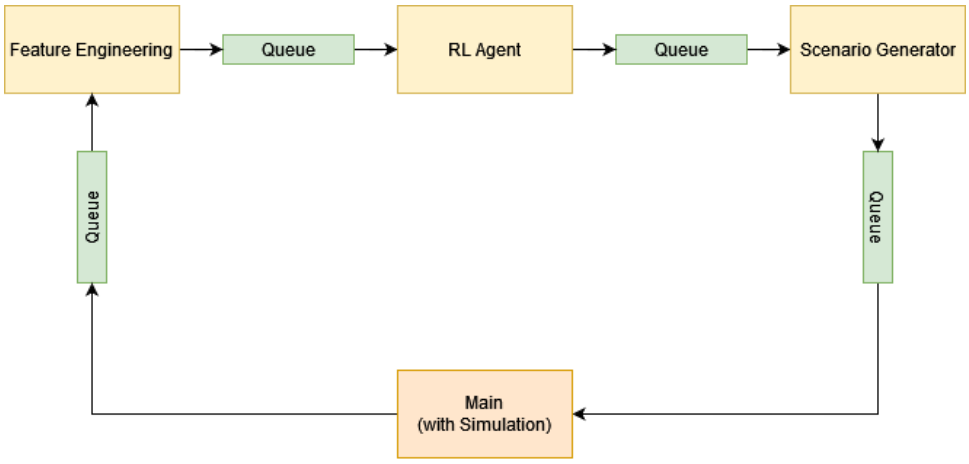


Figure 19 - Queue Infrastructure

The main contrast between this extension and our first prototype is that the tasks for each component in the RL architecture are stored in a separate queue buffer. The advantage of this approach is that each component does not need to wait for the request from the main component in order to execute the task. Merely, the components execute the tasks stacked in their buffer. This may open the possibilities to parallelize the RL components workflow, which in turn increases the performance of the ASIMOV's solution.

2.4.3 Components

The inner workings of the Feature Engineering, the RL Agent and the Scenario Generator component are described in the previous sections. Together with the Main component and the not yet integrated simulation, these components form the main components of the prototype.

2.4.4 Additional Components

One additional component we aim to embed in our RL architecture is a logging and tracking component for the machine learning process. With logging and tracking of the RL experiments, one has a better control of the learning process by being able to e.g.:

- observe the parameter used for the experiments,
- observe the model used in the previous experiment,
- find out the best hyperparameters in the overall experiment,
- and compare the result of the learning experiments over time.

This functionality opens up for instance the possibilities of transferring learning results between different times and different learning agents. Moreover, by utilizing this component one can involve experts to additionally analyze the learning outcome of the experiments and reproduce the learning experiments if further analysis is needed.
The framework we aim to use for this purpose is the ML-Flow [26] whose structure is depicted in the following figure:

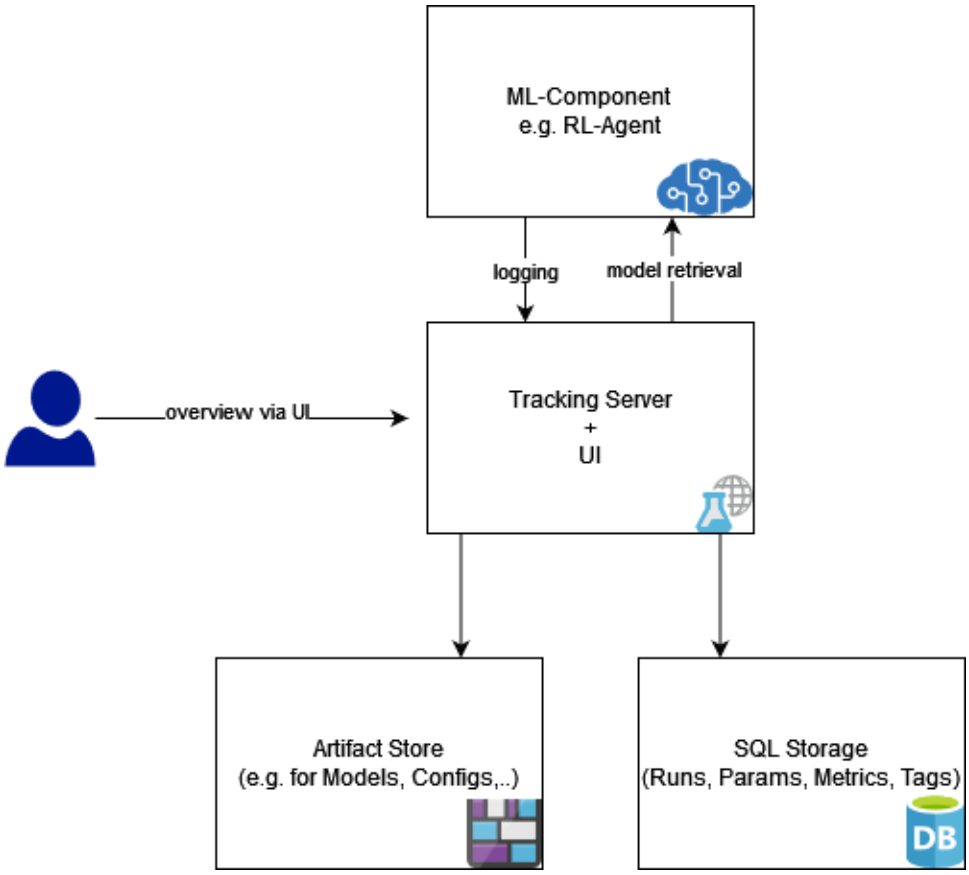


Figure 20 - MLflow structure

ML Flow constitutes a platform for both, logging and storing the learning experiments into the database, and also retrieving models given in the database for reproduction by e.g., an RL agent. The storage option is given by two instances:

- an artifact store for storing models and configuration,
- and a SQL storage for storing runs, parameters, metrics and also tags of the experiments

With ML Flow, one has the possibility to interact with the components via a given UI for e.g., analysis purposes.
Having this framework at hand, we aim also to integrate the DaSense components into this solution. DaSense can in this context be used as a performant frontend component and file management system to manage both the data generated by the ML Flow and also by the learning components.

3. Summary and future steps

This report described the UUV1 UC development. It showcases the four main groups in which the task of creating the ASIMOV solution for optimizing scenario-based testing is divided: Storage and compute, Reinforcement learning, Environment Simulation, and Feature engineering. Their current status of development is described in great detail. It shows that all components are currently at an implementation stage with the main concepts and even details already developed.
On a top level, the next step is finalizing the implementation work and thorough testing of the individual components. In continuation, the next primary milestone is to achieve the first full prototype by closing the loop and being able to optimize simple scenarios. This will give insights and issues might arise which will be handled in the continuous iterations of the demonstrator.

4. Terms, Abbreviations and Definitions

Table 7 - Terms, Abbreviations and Definitions

ABBREVIATION	EXPLANATION
ACC	Adaptive Cruise Control
AD	Autonomous Driving
AEB	Emergency Braking
DT	Digital Twin
ES	Environment Simulation
FE	Feature Engineering
HTTP	HyperText Transfer Protocol
KL	Kullback-Leibler Divergence
LKA	Lane Keep Assist
MDP	Markov Decision Process
ML	Machine Learning
MPO	Maximum a Posteriori Policy Optimization
NN	Neural Network
PT	Physical Twin
RL	Reinforcement Learning
RLA	Reinforcement Learning Agent
SC	Storage And Compute
SQL	Structured Query Language
TTB	Time To Brake
TTK	Time To Kickdown
TTM	Minimal Time To Maneuver
TTS	Time To Steer
UC	Use Case
UUV	Unmanned Utility Vehicle
VV&A	Verification, Validation and Accreditation

5. Bibliography

[1] “Continental,” [Online]. Available: https://www.continental.com/de/presse/pressemitteilungen/2019-12-12-hmi-cube-204622. [Accessed 22 11 2022].

[2] T. Brade and C. N. Birte Kramer, “Paradigms in Scenario-Based Testing for Automated Driving,” in *International Symposium on Electrical, Electronics and Information Engineering*, NewYork, 2021, pp. 108-114.

[3] “ASAM OpenDRIVE,” [Online]. Available: https://www.asam.net/standards/detail/opendrive/. [Accessed 25 11 2021].

[4] “ASAM OpenSCENARIO,” [Online]. Available: https://www.asam.net/standards/detail/openscenario/. [Accessed 25 11 2021].

[5] R. S. Sutton and B. Andrew, Reinforcement learning. An introduction, Cambridge, Massachusetts, London: The MIT Press (Adaptive computation and machine learning, 2018.

[6] R. Bellman, “A Markovian decision process,” in *Journal of mathematics and mechanics*, 1957, pp. 679-684.

[7] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare and J. Pineau, An introduction to deep reinforcement learning, arXiv preprint:1811.12560, 2018.

[8] R. G. A and N. Mahesan, On-line Q-learning using connectionist systems, Citeseer (37), 1994.

[9] M. P. Deisenroth, G. Neumann, J. Peters and others, “A survey on policy search for robotics,” in *Foundations and trends in Robotics 2*, 2013, pp. 388-403.

[10] esmini, “esmini - Github,” [Online]. Available: https://github.com/esmini/esmini. [Accessed 23 11 2022].

[11] “Unreal Engine,” [Online]. Available: https://www.unrealengine.com/en-US/. [Accessed 25 11 2021].

[12] “CARLA,” [Online]. Available: https://carla.org/. [Accessed 25 11 2021].

[13] L. N. C. K. T. e. a. Westhofen, “Criticality Metrics for Automated Driving: A Review and Suitability Analysis of the State of the Art,” *Arch Computat Methods Eng,* 2022.

[14] J. F. F. B. J. e. a. Degrave, “Magnetic control of tokamak plasmas through deep reinforcement learning,” *Nature,* vol. 602, pp. 414-419, 2022.

[15] A. Abdolmaleki, J. T. Springenberg, Y. Tassa, R. Munos, N. Heess and M. Riedmiller, Maximum a Posteriori Policy Optimisation, https://arxiv.org/pdf/1806.06920, 2018.

[16] A. Abdolmaleki, J. T. Springenberg, J. Degrave, S. Bohez, Y. Tassa and D. e. a. Belov, Relative Entropy Regularized Policy Iteration, http://arxiv.org/pdf/1812.02256v1., 2018.

[17] M. Neunert, A. Abdolmaleki, M. Wulfmeier, T. Lampe, T. Springenberger and R. e. a. Hafner, “Continuous-Discrete Reinforcement Learning for Hybrid Control in Robotics,” *Leslie Pack Kaelbling, Danica Kragic, Komei Suguira (Eds.): 3nd Conference on Robot Learning: PMLR (Proceedings of Machine Learning Research, 100),* pp. 735-751, 2020.

[18] N. P. Farazi, T. Ahamed, L. Barua and B. Zou, Deep Reinforcement Learning and Transportation Research: A Comprehensive Review, University of Illinois at Chicago, 2020.

[19] J. Duchi, Derivations of Linear Algebra and Optimization.

[20] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, OpenAI Gym, 2016.

[21] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap and J. e. a. Hunt, Deep Reinforcement Learning in Large Discrete Action Spaces, http://arxiv.org/pdf/1512.07679v2, 2015.

[22] Docker, “Docker Website,” [Online]. Available: https://docs.docker.com/get-started/#what-is-a-container. [Accessed 23 11 2022].

[23] Flask, “Flask Website,” [Online]. Available: https://palletsprojects.com/p/flask/. [Accessed 23 11 2022].

[24] Kubernetes, “Kubernetes Website,” [Online]. Available: https://kubernetes.io/. [Accessed 23 11 2022].

[25] RabbitMQ, “RabbitMQ Website,” [Online]. Available: https://www.rabbitmq.com/. [Accessed 23 11 2022].

[26] MLflow, “MLflow Website,” [Online]. Available: https://mlflow.org/. [Accessed 23 11 2022].
[27] ASIMOV-consortium, *ASIMOV - Full Project Proposal,* 2020.