System Specification

Cooperative Research Platform

# Goal of the document

This document summarizes the details of how the Cooperative Research Platform (CRP) is built up. It consists of the multiple layers:

* System Functionality,
* Architecture,
* Scenario Coverage and demonstration basis

# Supplements

## Corresponding terminology

* E2E: end-to-end, usually referred to as standalone operation of a function, without the need of e.g., pre-recorded data, and this function can be used by a non-technician user
* function: practical manifestation of technical implementation
* architecture: collection of components that are arranged into a pre-defined structure,
* ground architecture: white-paper definition of system components, without dependencies like Autoware,
* function architecture: real structure of the system components, that are directly usable in the vehicle.

### Testing concept

Implement function code in function layer, then integrate to application layer. At this stage, record raw data (mcaps) together with vehicle and controller integration layers. The resulting measurement file can be used for open-loop tests, that is satisfactory except for vehicle control components.

# Function Specification

This Section describes the high-level specifications of the covered functionality.

Autoware architecture: <https://app.diagrams.net/?lightbox=1#Uhttps%3A%2F%2Fautowarefoundation.github.io%2Fautoware-documentation%2Fmain%2Fdesign%2Fautoware-architecture%2Fnode-diagram%2Foverall-node-diagram-autoware-universe.drawio.svg>

## Intelligent Speed Adjustment

* Step 1 functionality: longitudinal speed control adjusted to static information, such as curve and local regulations (speed limit).
* Step 2 functionality: step 1 + speed adjustment on dynamic information, such as moving objects (e.g., followed vehicle).

For both: speed range is 0 <= vx <= 150 kph, which therefore includes automatic start/stop functionality. Function is illustrated in 1. Figure.



1. Figure Function illustration, both step 1 and step 2 functionality

## Longitudinal Emergency Function

Functionality: vehicle or delegated sensors provide information about static / dynamic objects. The function decides proper strategy to stop the vehicle (and where to stop it). Then, this strategy is accomplished by applying proper braking force. Function use cases are shown in 2. Figure.

Operation range: .



2. Figure Longitudinal emergency use cases from vehicle sensors and infrastructure sensors

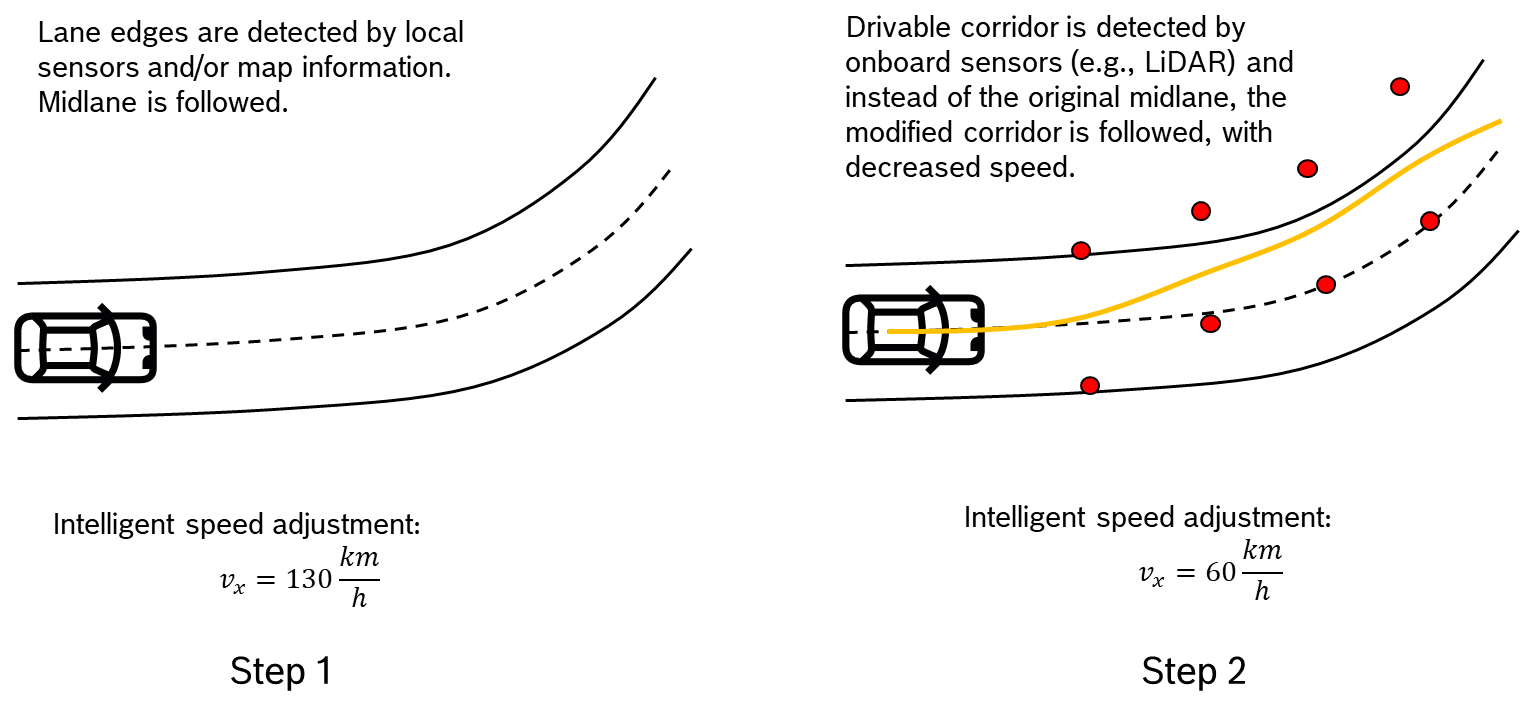
## Lane Follow

* Step 1 functionality: vehicle is running in a lane, which is bounded by lane edges (markers or only the edge of the drivable surface) and the vehicle follows the centerline of the lane (or externally defined local trajectory).

Operation range: , , .

* Step 2 functionality: Drivable corridor is shifted due to e.g., temporarily shifted road works, which is bounded by 3D obstacles like cones, walls...etc. Vehicle (with lower dynamics) can still navigate through this drivable corridors.

Operation range: , , .



3. Figure Lane follow functionality steps and covered operation.

# Architecture specification

## Comprehensive notes

The architecture is defined based on the E2E function specifications. There are two main concepts that must be considered at all time:

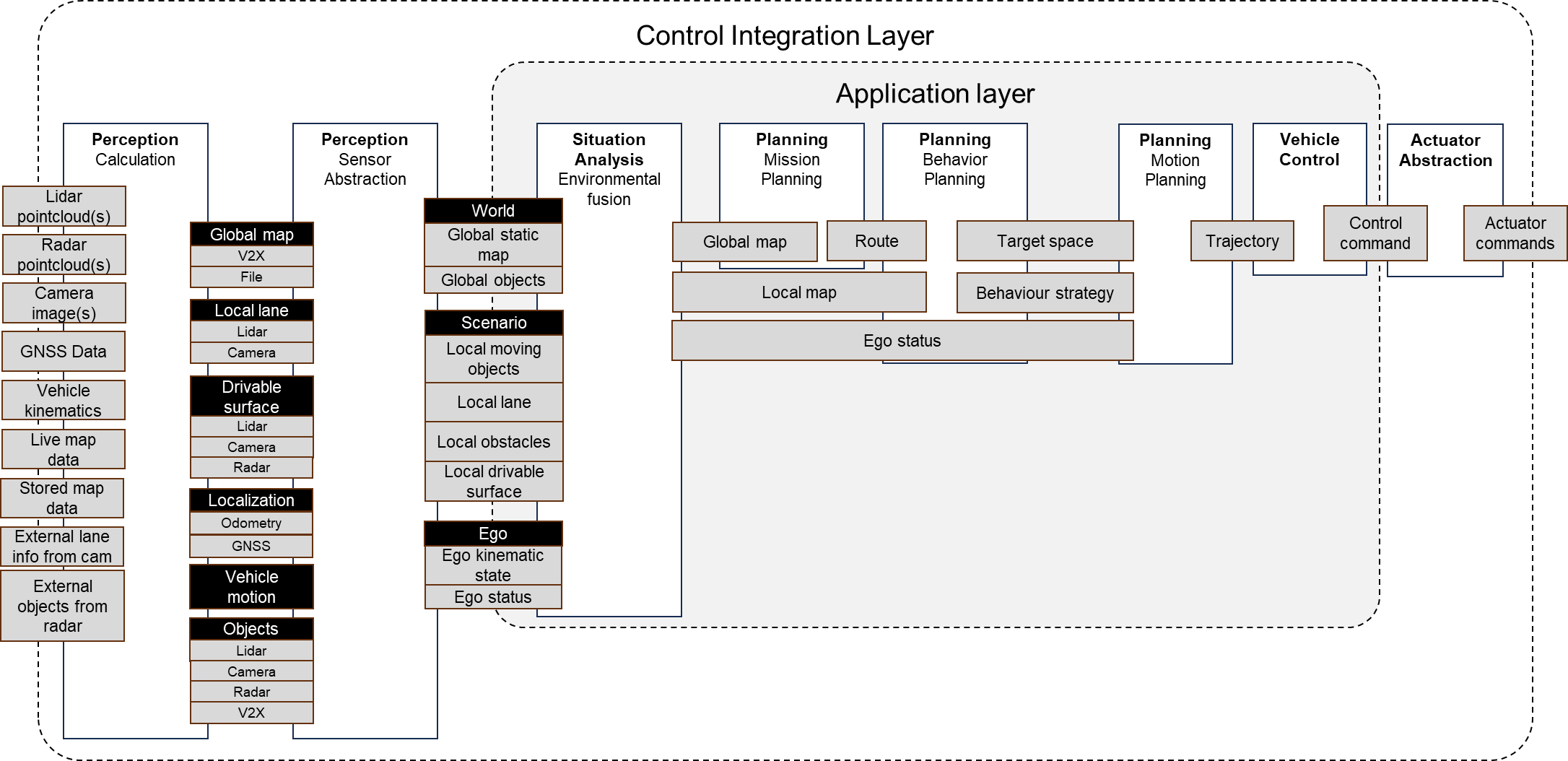
* fulfill E2E function requirements with the least architecture components,
* re-use Autoware components where possible, but keeping its number (or number/function) as low as possible.

Therefore, the following work model is strongly recommended:

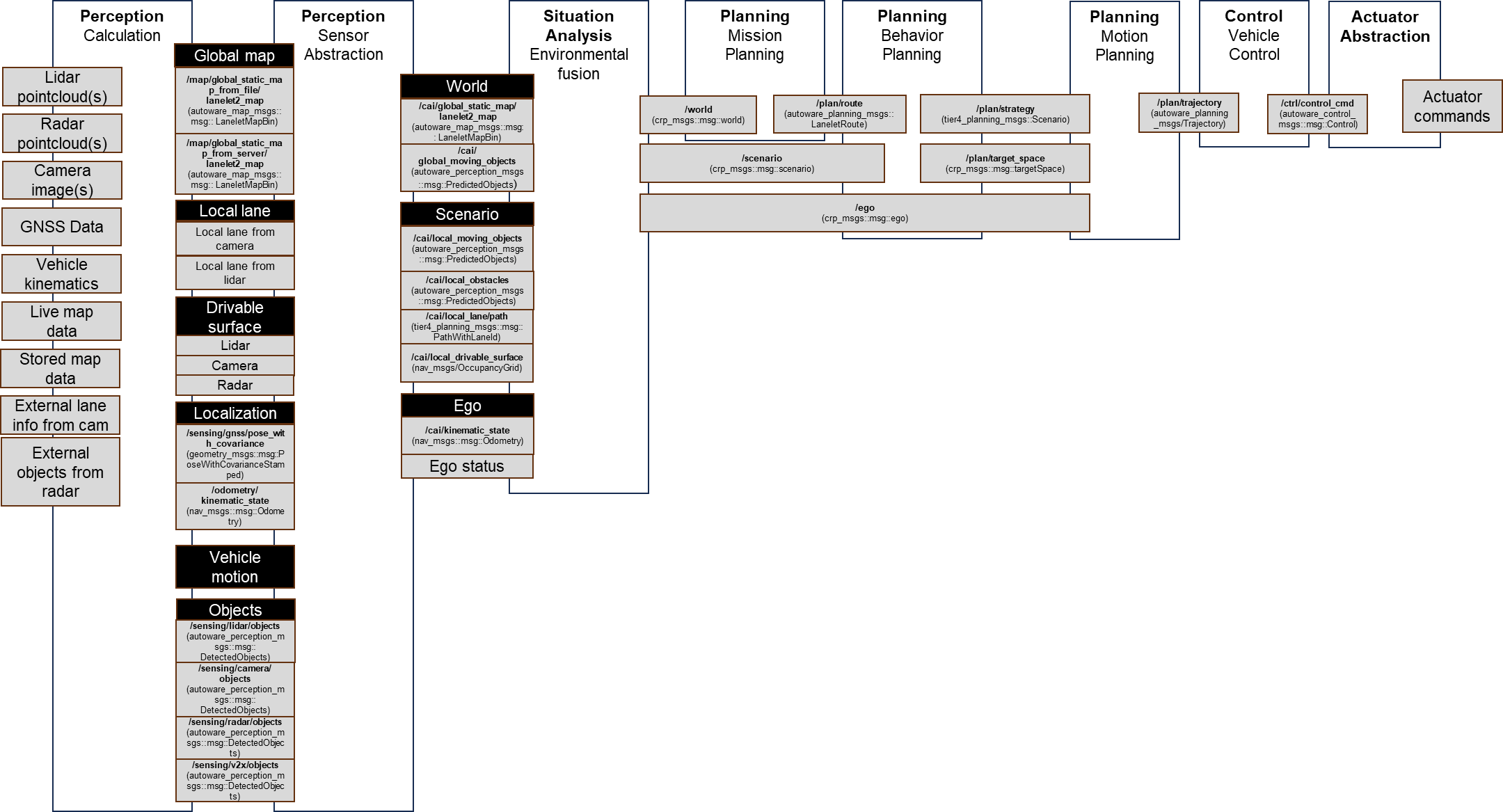
This concept may hinder the efficient/reliable planning of tasks regarding architecture definition, but ensures that the above two concepts are respected.

## General architecture

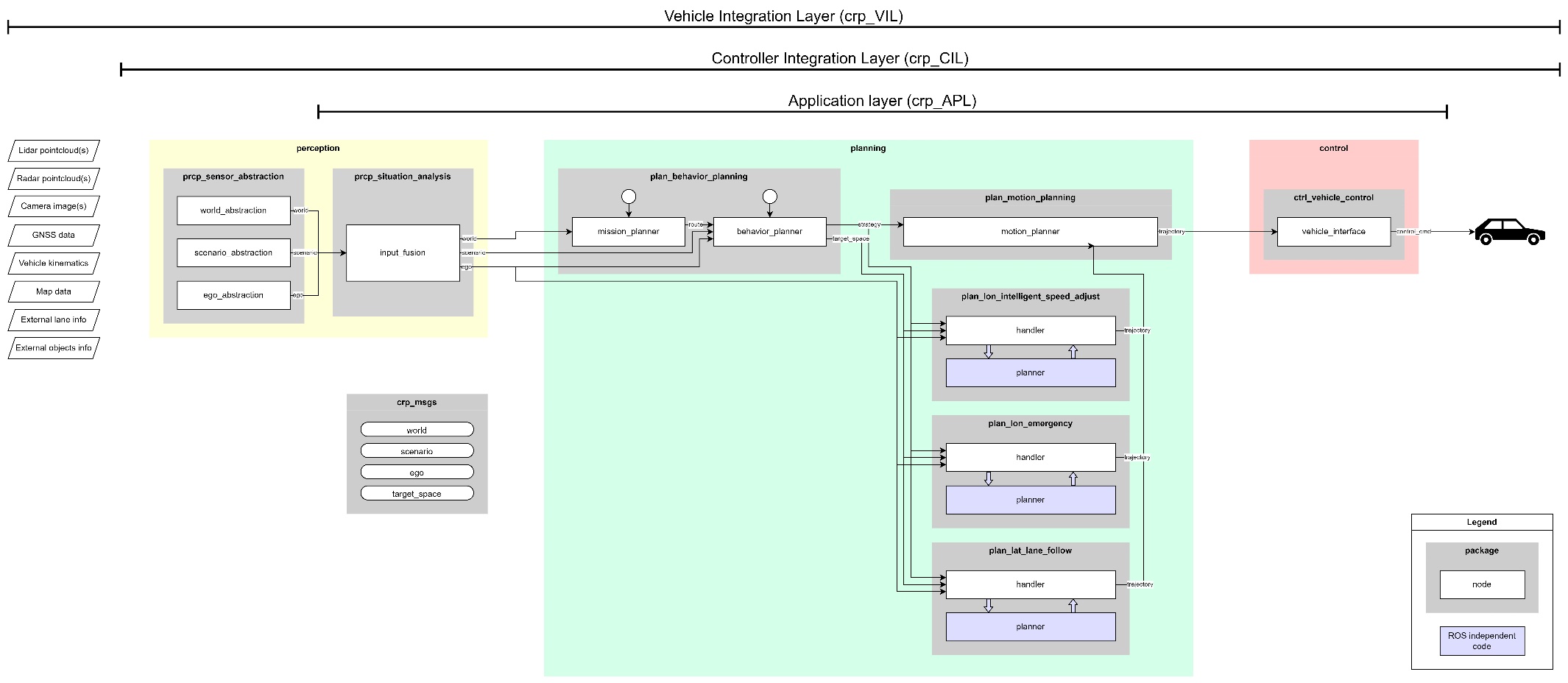
### High-level function architecture (black box)



### Component-detailed architecture (gray box)



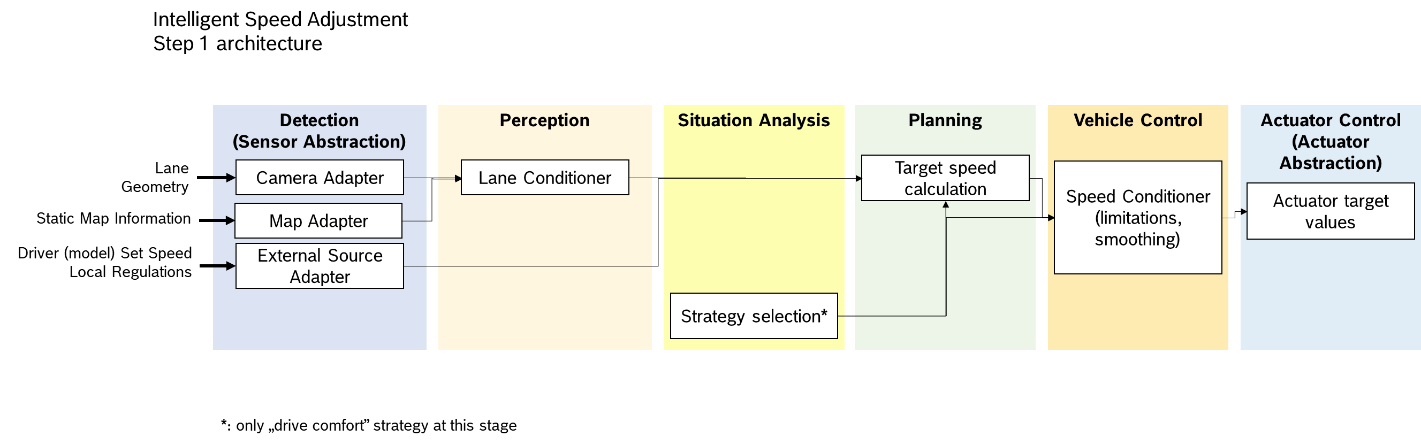
### ROS2 architecture



## Function Architecture Specification

### Intelligent Speed Adjustment (ISA)

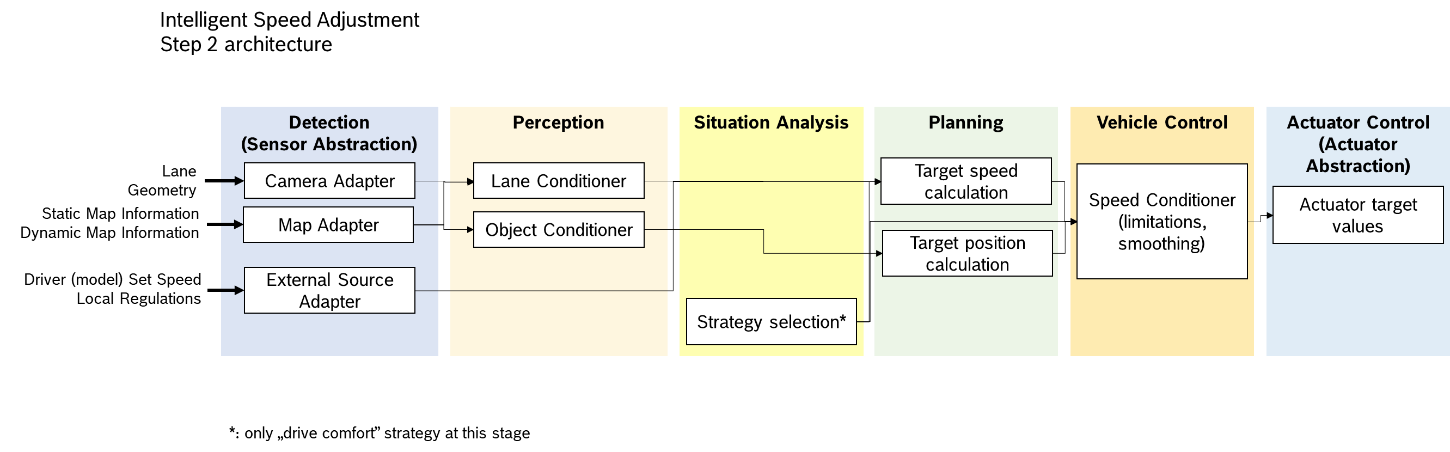
Step 1 architecture:



4. Figure. Architecture components of step 1 functionality of ISA

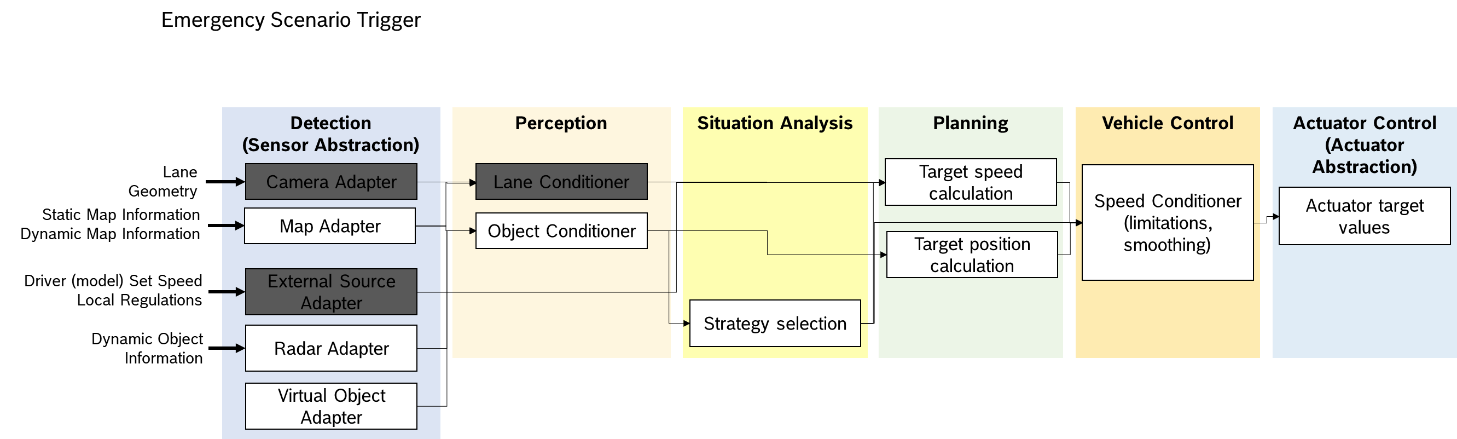
Note: even if we only control the vehicle longitudinally, the lateral path shall be filled with dummy values. Idea: add a straight line with no offset. Later, it must be solved that the vehicle is longitudinally controlled by the system, but laterally by the driver.

Step 2 architecture:



5. Figure Architecture components of step 2 functionality of ISA

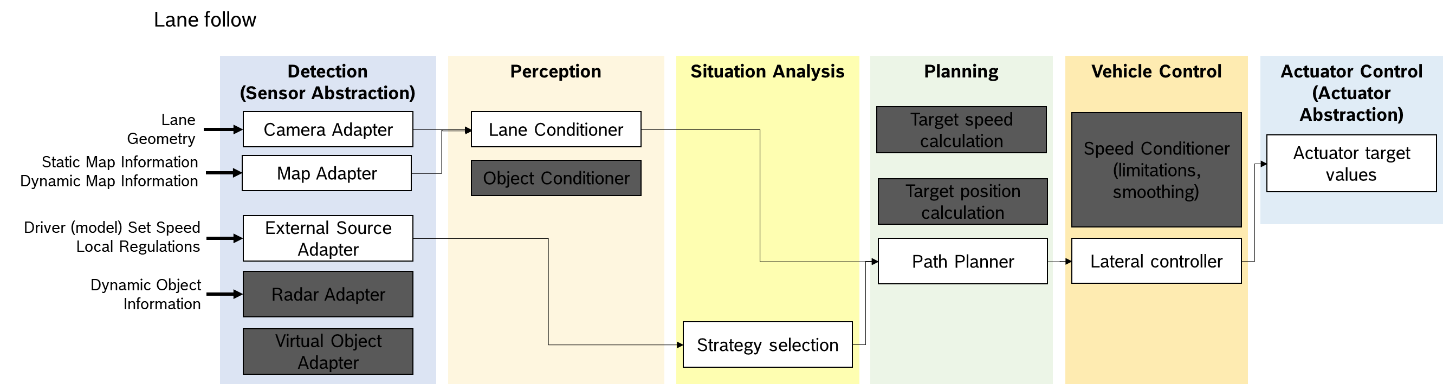
## Longitudinal emergency function



Based on distributed sensor data calculate the trigger of the emergency scenario.

## Lane follow

Step 1 architecture:

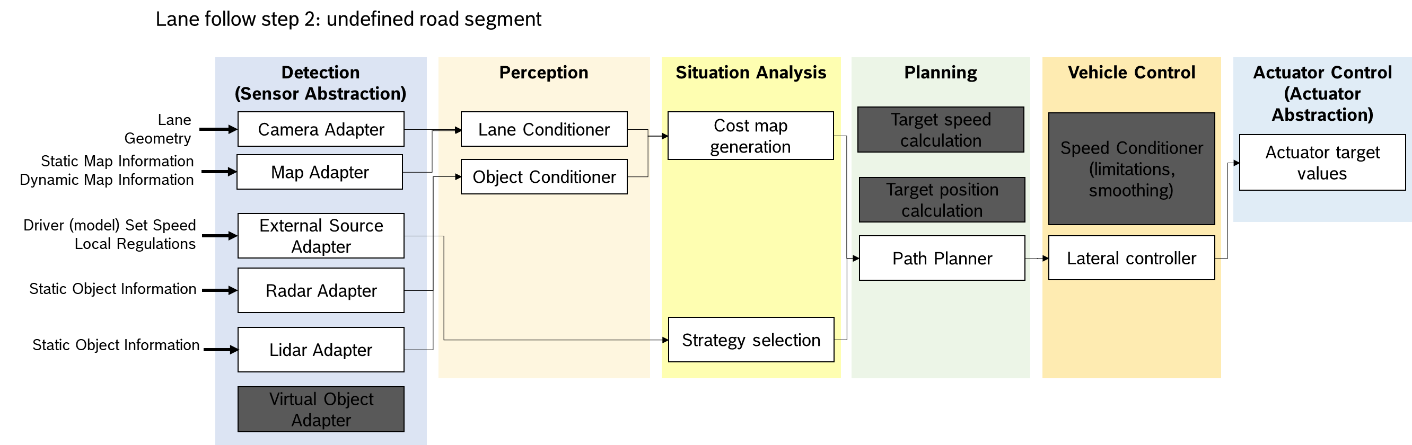


6. Figure. Architecture components of step 1 functionality of LF (lane follower) without behaviour layer

Lane follow architecture consists of new components of path planner, that takes geometry information of the road, and plans a smooth (drivable) path, which is then handed over to the lateral controller component. This controller controls only the lateral movement of the vehicle, producing output to the actuator control (i.e., steering angle).

Note: route input comes from mission planner, which is currently not part of the architecture. During integration process, it may be extended.

Step 2 architecture:



7. Figure. Architecture components of step 2 functionality of LF (lane follower) without behaviour layer

# Message definitions

## CRP messages – custom definitions

### crp\_msgs::msg::scenario

This interface holds information of four main types:

* local moving objects: highest layer which is associated with other objects that move around the ego vehicle, such as other vehicles, pedestrians, animals...etc.
* local obstacles: static items that are located around the ego vehicle.
* local lanes: the lanes that are mainly marked by painted markers and form the static driving corridors.
* local drivable surface: the most indefinite representation of the local environment, in the form of a generic occupancy grid.

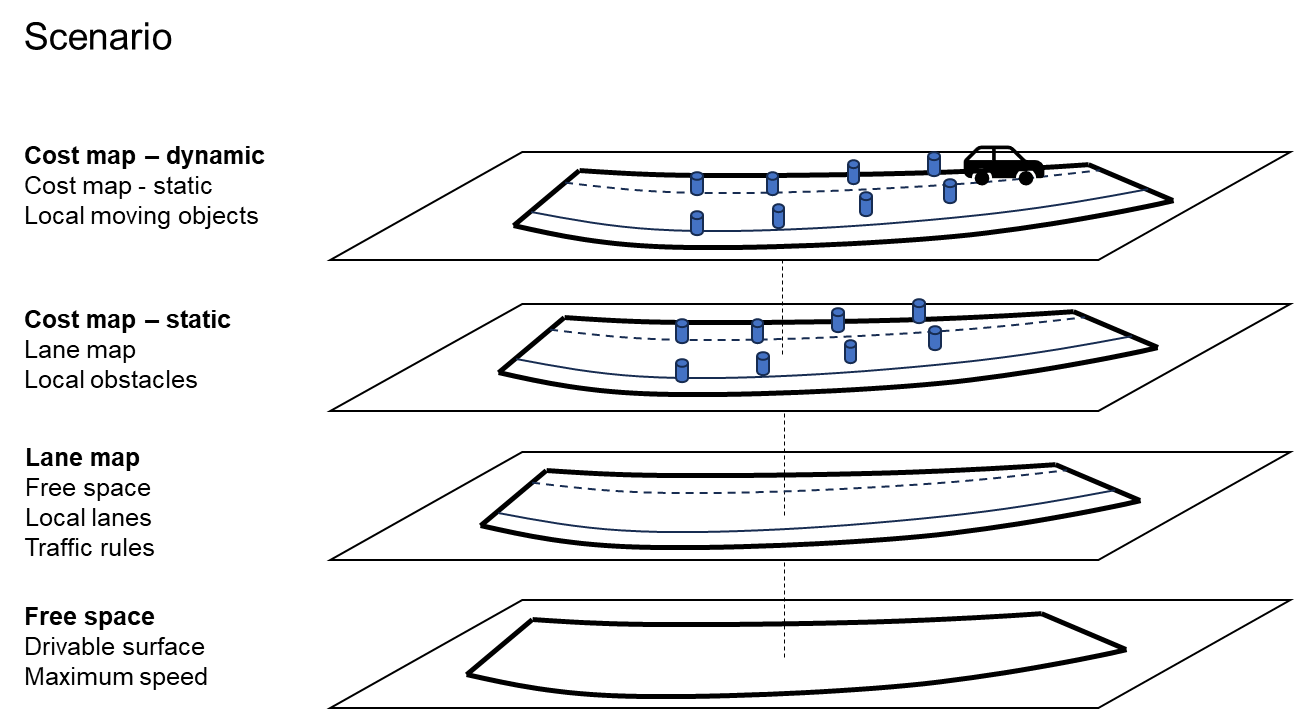
This interface type (in contrast to globally defined „world” interface) must contain data with as high accuracy as possible.

Message definition:

|  |
| --- |
| std\_msgs/Header header  autoware\_perception\_msgs/PredictedObject[] local\_moving\_objects  autoware\_perception\_msgs/PredictedObject[] local\_obstacles  tier4\_planning\_msgs/PathWithLaneId[] lanes  # traffic rules information to be added  nav\_msgs/OccupancyGrid free\_space  std\_msgs/Float32 maximum\_speed |

Note: the traffic rules collect all types of information that are coming from the static rules and can impact the selected bahviour. These are like:

* stop lines (stop signs and lines)
* speed limitation
* traffic light information (semi-static)
* etc.



8. Figure Illustration of the scenario layers.

### crp\_msgs/world

### crp\_msgs/ego

This interface contains every relevant information about the ego vehicle:

* pose: position, orientations and uncertainty
* velocity: linear and angular
* acceleration: linear and angular
* wheel angle

Message definition:

|  |
| --- |
| std\_msgs/Header header  geometry\_msgs/PoseWithCovariance pose  geometry\_msgs/TwistWithCovariance twist  geometry\_msgs/AccelWithCovariance accel  float32 wheel\_angle |

## Path

This is a tier4 autoware message extension, with the following definition:

|  |
| --- |
| std\_msgs/Header header  tier4\_planning\_msgs/PathPoint[] points  nav\_msgs/OccupancyGrid drivable\_area |

## Path point

|  |
| --- |
| uint8 REFERENCE=0  uint8 FIXED=1  geometry\_msgs/Pose pose  geometry\_msgs/Twist twist  uint8 type |

## Trajectory

|  |
| --- |
| std\_msgs/Header header  tier4\_planning\_msgs/TrajectoryPoint[] points |

## Trajectory point

|  |
| --- |
| geometry\_msgs/Pose pose  geometry\_msgs/Twist twist  geometry\_msgs/Accel accel |

# Coding rules

## Naming conventions

* Use camel case everywhere if not stated otherwise
* Classes and structs should start with upper case letter
* Methods should start with lower case
* Constant variables should be all upper case letters. Instead of camel case the words should be separated with ‘\_’
* In classes, private and protected attributes should start with the “m\_” prefix
* In methods, the parameters should start with the ‘???’ prefix
* Pointers should comply with the previous rules but should have a ‘\_’ suffix  
  (or p\_ prefix ???)
* Namespaces should be ??? (fully lower/upper case, like classes, ?)
* Maximum namespace depth should be ???
* File and folder names should be camel case where it is not restricted (e.g. ROS2 naming rules)

## C++

### General

* Header and source files should be separated (“include” folder for headers and “src” folder for source files)
* Header files should only contain declarations, the definitions should be in a source file that includes the header
* No recursion; No GOTOs
* Loops must have a fixed bound
* No dynamic memory allocation after initialization
* Restrict data to the smallest necessary scope
* Check function return values or explicit discard with (void)
* Avoid the preprocessor and especially complex uses of the preprocessor
* Prefer Fw and Os implementations. e.g. use FW\_ASSERT and Os::Mutex over cassert and std::mutex
* Compile without warnings, errors, and static analysis failures (e.g. pass continuous integration)
* Do not use Os::Task::delay to synchronize between threads
* Explicit enumeration values should be specified for all values or none at all .

### Header files

Every header file should…

* have the “.hpp” extension
* have header guards (#ifndef, #define, #endif)
* contain max. one class (???)

### Source files

Every source file should…

* have the “.cpp” extension

## ROS2

### Packages

* Non-driver package names should start with and abbreviation of the following categories:
  + prcp (perception)
  + plan (planning)
  + ctrl (control)

# Package documentations

## pacmod\_extender

### Purpose

The purpose of this package is to extend the default pacmod capabilities on the Lexus vehicle by decoding CAN messages or calculating new data from the inputs. The package is designed to work seamlessly with the already existing pacmod3 system.

### Usage

The package can be used by running the executable node. This way it uses the default namespace for subscriptions and publishers:

|  |
| --- |
| ros2 run pacmod\_extender pacmod\_extender\_node |

The other way is to use the launcher. This launcher is tailored for the Lexus vehicle by giving the executable the necessary namespace to match the other components:

|  |
| --- |
| ros2 launch pacmod\_extender pacmod\_extender.launch.py |

### I/O

Input topics:

|  |  |  |
| --- | --- | --- |
| **Data** | **Message name** | **Message type** |
| Raw can data | pacmod/can\_tx | can\_msgs/msg/Frame |
| Vehicle twist | vehicle\_status | geometry\_msgs/msg/TwistStamped |

Output topics:

|  |  |  |
| --- | --- | --- |
| **Data** | **Message name** | **Message type** |
| Linear acceleration | pacmod/linear\_accel\_rpt | pacmod3\_msgs/msg/LinearAccelRpt |
| Calculated yaw rate | pacmod/yaw\_rate\_calc\_rpt | Pacmod3\_msgs/msg/YawRateRpt |

### Inner workings

The main functionality is the decoding of previously not used CAN messages. The decodings are defined in the PacmodDefinitions class. Every message has a decode method that requires the CAN message as parameter. The message IDs are stored as constants in the class.

The PacmodExtender class is the main class that is executed as a node. It subscribes to the inputs, uses the PacmodDefinitions class to decode the CAN messages and outputs the new messages. The output rate of every message depends on the input frequencies. Every output value is in SI units.

Decodings:

* Linear acceleration
  + longitudinal, lateral, vertical acceleration in m/s

Calculations:

* Yaw rate

, where is the yaw rate, is the tire angle and is the wheelbase