X-CAR Reference Manual

Version 1.0.3

Authors: Goodarz Mehr

Project Contributors: Goodarz Mehr, Prasenjit Ghorai, Ce Zhang, Anshul Nayak, Darshit Patel, Shathushan Sivashangaran
Autonomous Systems and Intelligent Machines (ASIM) Laboratory
Professor Azim Eskandarian

January 4, 2022

Before You Begin

1. If you have made a contribution to the writing of this document, make sure to add your name to the end of the list of authors on the title page.

2. If you want to contribute to this document, make sure you are familiar with LaTeX. Overleaf has extensive LaTeX documentation and a guide to learn LaTeX in 30 minutes. You can find documentation for most packages on CTAN (Comprehensive TeX Archive Network) and forums such as TeX StackExchange are a great source of answers.

3. When working with the car, try to be as organized as possible. Remember that the car and related hardware are shared among all lab members and everyone should be considerate of others' works and research needs.

4. Try to keep the car processing unit as clutter-free as possible. Don't install or download **anything** on it unless absolutely necessary.

5. Google it! Any time you encounter a problem while using this software, the first thing you should do is Google the error message. It is very likely that someone has run into the same problem and posted about it online.

6. When writing new code for X-CAR or modifying existing code, make sure to follow the style of that specific piece of code to a T. This includes conventions for things like file, function, and variable names; spacing; indenting; comments; etc. This rule applies to the LaTeX source of this document as well and helps keep files readable, organized, and easy to work with for everyone. **Your access may be limited if you do not follow this rule.**

7. Bump the first version number from left after major revisions, bump the second number after minor edits, and bump the last one after tiny fixes. Add an entry with a summary of the changes made in proper format to the top of Changelog.

Changelog

- **V1.0.3 (02/22/2022) Goodarz Mehr**
  - Changed all instances of XCoRe to X-CAR.
- **V1.0.2 (02/19/2022) Goodarz Mehr**
  - Fixed typos in Section 2.4.
- **V1.0.1 (02/09/2022) Goodarz Mehr**
  - Added description of `bridge.yml` and `carma_docker.launch.py` configuration files.
- **V1.0.0 (02/06/2022) Goodarz Mehr**
  - Finished initial draft of the manual. Total of 53 pages.

Contents

List of Figures

List of Tables

List of Code Snippets

1 Introduction

This document serves as a reference manual for X-CAR (eXperimental Vehicle Platform for Connected Autonomy Research) that is developed by the Autonomous Systems and Intelligent Machines (ASIM) Laboratory at Virginia Tech. X-CAR is powered by the CARMA Platform, which is an open-source software initiated by the Federal Highway Administration (FHWA) to enable the testing and evaluation of cooperative automation concepts to improve safety and increase infrastructure efficiency. By implementing the CARMA Platform on more affordable, high quality hardware, X-CAR aims to increase the versatility of the CARMA Platform and facilitate its adoption for research and development (R&D) of cooperative driving automation (CDA).

Our 2017 Ford Fusion SE Hybrid serves as a basis for X-CAR, but in general any vehicle supported by Dataspeed's Drive-by-Wire Kit can be used[1]. Other elements of X-CAR include the Dataspeed Drive-by-Wire (DbW) kit and Intelligent Power Distribution System (iPDS), two Leopard Imaging (LI) cameras, one Ouster lidar, one Continental Radar, one Inertial Labs GNSS/INS module, and one Cohda dedicated short range communication (DSRC) module.

X-CAR uses the latest version of the CARMA Platform, or CARMA3. CARMA3 is a reusable, extensible platform for controlling SAE level 3+ connected autonomous vehicles (CAVs). It provides a rich, generic API for third-party plugins that implement vehicle guidance algorithms to plan vehicle trajectories. It is written in C++ and runs in a Robot Operating System (ROS) environment on Ubuntu.

The rest of this document is organized as follows: Section 2 provides some background on topis that are necessary for working with X-CAR and the CARMA Platform. Section 3 discusses the CARMA Platform, its installation process, and building the CARMA Platform from source. Section 4 provides information about each hardware module, while

[1] A list of the supported vehicles is available here.

Section 5 discusses the localization process performed by the CARMA Platform. Section 6 TBD. Section 7 TBD. Section 8 TBD. Section 9 discusses the calibration process of different sensors and our specific configuration of the CARMA Platform, while Section 10 discusses real-world operation of X-CAR. Finally, Section 11 provides any miscellanoeous information that did not suite any of the other sections.

2 Background

This section provides some background information that is needed for working with X-CAR and the CARMA Platform. In what follows, Section 2.1 to Section 2.7 briefly discuss C++, ROS, Ubuntu, Docker, Git, Autoware, and GStreamer.

2.1 C++

C++ is a general-purpose programming language created as an extension of the C programming language with support for Classes. Modern C++ has object-oriented, generic, and functional features in addition to facilitating low-level memory manipulation. C++ is almost always implemented as a compiled language. X-CAR hardware drivers and CARMA3 are mostly written in C++.

cplusplus.com is a good resource for learning C++. It has a tutorial for learning the language and a comprehensive reference for C++ standard libraries. Its forum as well as Stack Overflow are a great resource for asking questions and finding answers.

2.2 ROS

Robot Operating System (ROS) is an open-source robotics middleware suite. ROS is not an operating system, but rather a collection of software frameworks for robot software development. It provides services for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a graph architecture where processing takes place in nodes that may receive, publish, and manipulate sensor data, control, and other messages. ROS itself is not a real-time OS (RTOS). It is possible, however, to integrate ROS with real-time code.

The original ROS, or ROS 1, was started in 2007, and its last long-term support (LTS) distribution release was ROS Noetic Ninjemys, primarily targeting Ubuntu 20.04 (Focal Fossa) release and supported until May 2025. The new ROS, or ROS 2, is a major revision of the ROS application-program interface (API) which will take advantage of modern libraries and technologies for core ROS functionality and add support for real-time code and embedded hardware. ROS 2's latest LTS distribution release is ROS 2 Foxy Fitzroy which is supported for 3 years or until May 2023, and its latest interim distribution release is ROS 2 Galactic Geochelone which is supported for 1.5 years or until November 2022, both targeting Ubuntu 20.04 (Focal Fossa) release. CARMA3 is written using ROS 1, but it will be migrated to ROS 2 soon. For the time being, X-CAR hardware drivers are written using ROS 1.

Figure 1: ROS 1 Noetic Ninjemys.

ros.org is the primary resource for learning and working with ROS, with excellent documentation for ROS 1, ROS 2, and many ROS packages. Its forum is also a great resource for asking questions and finding answers.

2.3 Ubuntu

Ubuntu is a Linux distribution based on Debian and composed mostly of free and open-source software. Its most recent LTS release is 20.04 (Focal Fossa) which is supported until 2025, and its latest standard release is 21.10 (Impish Indri) which is supported for 9 months. CARMA3 targets Ubuntu 20.04 LTS (Focal Fossa) release.

Like other operating systems, Ubuntu has a shell, which is an interactive command interpreter environment within which commands may be typed at a prompt or entered into a file in the form of a script and executed. The default shell on Ubuntu is Bash (Bourne Again Shell), a Unix shell and command language written as a replacement for the Bourne Shell.

Ubuntu Tutorials and Bash Guide for Beginners are great resources for learning Ubuntu and Bash, respectively. Furthermore, Ubuntu Forums and Stack Overflow are great forums for asking questions and finding answers.

2.4 Docker

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries, and configuration files. Containers can

communicate with each other through well-defined channels. The software that hosts the containers is called the Docker Engine.

Containerization using Docker is different from virtualization. VirtualBox and VMWare are virtualization software that create virtual machines that are isolated at the hardware level and each have their own OS. In contrast, Docker isolates software at the OS level. With Docker, you can run multiple applications (containers) on the same host OS, sharing underlying hardware resources (CPU, RAM, etc.). Figure 3 illustrates the difference between these two approaches.

Figure 2: The Docker Engine.

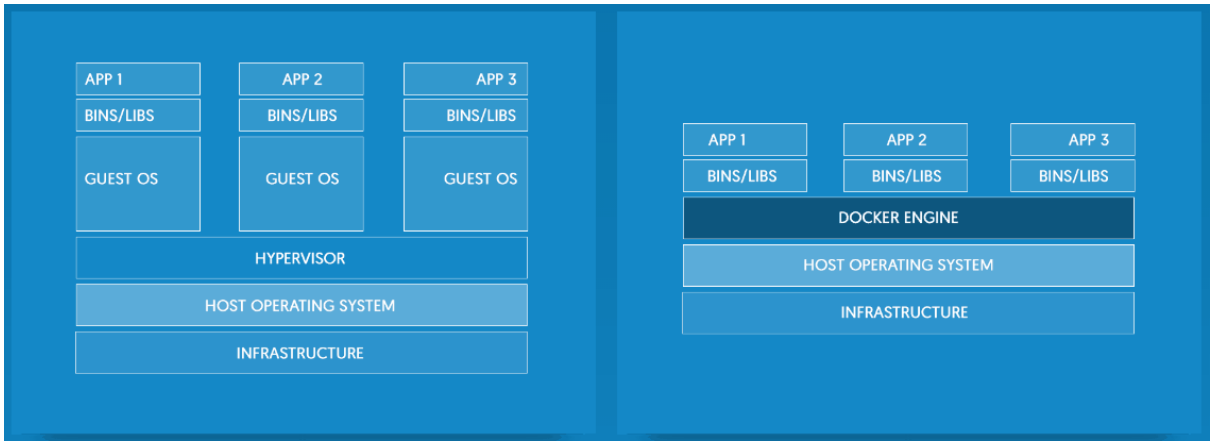


Figure 3: Comparison of virtualization (left) and containerization (right).

A Docker image can be thought of as a starting point for building containers, like a template. A Docker image can be built using a `Dockerfile`, which is a set of instructions that can, for example, define the parent image, install a series of packages using a package manager like `apt`, build some packages from source using `catkin`, and determine what commands are executed each time a container is spawned. Code Snippet 1 shows a sample `Dockerfile` used for building a Docker image of a ROS package. In it, `FROM` indicates the parent image, `RUN` indicates commands executed to build the image, and `ENTRYPOINT` indicates commands that are executed each time a container is spawned from this image. `ENTRYPOINT` commands cannot be replaced at runtime, but changing that entry to `CMD` makes this possible.

```
1  ARG ROS_DISTRO=melodic
2
3  FROM ros:${ROS_DISTRO}-ros-core AS build-env
4  ENV DEBIAN_FRONTEND=noninteractive \
5      BUILD_HOME=/var/lib/build \
6      OUSTER_SDK_PATH=/opt/ouster_example
7
8  RUN set -xue \
```

```
# Kinetic and melodic have python3 packages but they seem to conflict
&& [ $ROS_DISTRO = "noetic" ] && PY=python3 || PY=python \
# Turn off installing extra packages globally to slim down rosdep install
&& echo 'APT::Install-Recommends "0";' > /etc/apt/apt.conf.d/01norecommend
 ↪  \
&& apt-get update \
&& apt-get install -y \
 build-essential cmake \
 fakeroot dpkg-dev debhelper \
 $PY-rosdep $PY-rospkg $PY-bloom

# Set up non-root build user
ARG BUILD_UID=1000
ARG BUILD_GID=${BUILD_UID}

RUN set -xe \
&& groupadd -o -g ${BUILD_GID} build \
&& useradd -o -u ${BUILD_UID} -d ${BUILD_HOME} -rm -s /bin/bash -g build
 ↪  build

# Install build dependencies using rosdep
COPY --chown=build:build ouster_ros/package.xml
 ↪  ${OUSTER_SDK_PATH}/ouster_ros/package.xml

RUN set -xe \
&& apt-get update \
&& rosdep init \
&& rosdep update --rosdistro=${ROS_DISTRO} \
&& rosdep install -y --from-paths ${OUSTER_SDK_PATH}

# Set up build environment
COPY --chown=build:build . ${OUSTER_SDK_PATH}

USER build:build
WORKDIR ${BUILD_HOME}

RUN set -xe \
&& mkdir src \
&& ln -s ${OUSTER_SDK_PATH} ./src


FROM build-env

RUN /opt/ros/${ROS_DISTRO}/env.sh catkin_make -DCMAKE_BUILD_TYPE=Release

# Entrypoint for running Ouster ros:
#
# Usage: docker run --rm -it ouster-ros [ouster.launch parameters ..]
```

```
#
ENTRYPOINT ["bash", "-c", "set -e \
&& . ./devel/setup.bash \
&& roslaunch ouster_ros ouster.launch \"$@\" \
", "ros-entrypoint"]
```

Code Snippet 1: Sample `Dockerfile`.

A few useful Docker commands are listed below:

- `docker build [OPTIONS] PATH | URL | -`: builds Docker images from a Dockerfile and a context. A build's context is the set of files located in the specified `PATH` or `URL`. The build process can refer to any of the files in the context. The `URL` parameter can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts, and plain text files.
- `docker images [OPTIONS] [REPOSITORY[:TAG]]`: lists all top level images, their repository and tags, and their size. Docker images have intermediate layers that increase reusability, decrease disk usage, and speed up `docker build` by allowing each step to be cached. These intermediate layers are not shown by default.
- `docker ps [OPTIONS]`: lists all containers.
- `docker run [OPTIONS] IMAGE [COMMAND] [ARG...]`: creates a writeable container layer over the specified image, and then starts it using the specified command. In other words, spawns a container from a Docker image.

Another useful tool is Docker Compose. Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a `YAML` file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. Code Snippet 2 shows a sample `YAML` file used for creating two containers.

```
# Docker Compose Spec Version
version: '2'

services:
  platform:
    image: usdotfhwastoldev/carma-platform:develop
    network_mode: host
    container_name: platform
    volumes_from:
      - container:carma-config:ro
    volumes:
```

```
12        - /opt/carma/logs:/opt/carma/logs
13        - /opt/carma/.ros:/home/carma/.ros
14        - /opt/carma/vehicle/calibration:/opt/carma/vehicle/calibration
15        - /opt/carma/yolo:/opt/carma/yolo
16        - /opt/carma/maps:/opt/carma/maps
17        - /opt/carma/routes:/opt/carma/routes
18      environment:
19        - ROS_IP=127.0.0.1
20        - NVIDIA_VISIBLE_DEVICES=all
21      command: bash -c 'wait-for-it.sh localhost:11311 -- roslaunch
        ↪  /opt/carma/vehicle/config/carma_docker.launch'
22
23      inertiallabs-gnss-driver:
24      image: usdotfhwastoldev/carma-inertiallabs-gnss-driver:develop
25      container_name: inertiallabs-gnss-driver
26      network_mode: host
27      privileged: true
28      devices:
29        - /dev/ttyUSB0:/dev/ttyUSB0
30      volumes_from:
31        - container:carma-config:ro
32      environment:
33        - ROS_IP=127.0.0.1
34      volumes:
35        - /opt/carma/logs:/opt/carma/logs
36        - /opt/carma/.ros:/home/carma/.ros
37        - /opt/carma/vehicle/calibration:/opt/carma/vehicle/calibration
38      command: bash -c '. ./devel/setup.bash && export ROS_NAMESPACE=$${CARMA_INTR_NS}
        ↪  && wait-for-it.sh localhost:11311 -- roslaunch
        ↪  /opt/carma/vehicle/config/drivers.launch drivers:=inertiallabs_gnss'
```

Code Snippet 2: Sample `docker-compose.yml` file.

A Docker container is by design isolated from the rest of the OS, which means it has its own file system and by default cannot access any I/O devices (such as USB ports) or the network the OS is connected to. Because of this, you have to manually specify the volumes (folders) and I/O devices from the OS it has access to and how it handles networking. While these can be specified as options to the `docker run` command, it is often easier to specify them in a `docker-compose.yml` file. In Code Snippet 2, the `volumes` section of each container indicates which folders of the OS file system are mounted inside that container, so e.g. `/opt/carma/logs` folder of the OS file system is mounted at the same location in the container. Furthermore, `network_mode` determines how each container handles networking, which in our case indicates the containers have direct access to the OS network. It is also possible to map specific network addresses (IP and port) on the OS to an address inside the container. Furthermore, the `devices` section of the last container determines which I/O devices are mounted inside that container, which in this

case is `/dev/ttyUSB0` (a USB device), mounted at the same address inside the container. Finally, the `environment` section declares environment variables inside the container and the `command` section determines the commands that are executed when the container is created.

docs.docker.com is a great resource for learning and working with Docker and contains many references, manuals, guides, and tutorials.

2.5 Git

Git is a free and open-source distributed version control system (VCS). VCSs are tools used to track changes to the source code (or other collections of files and folders). These tools help maintain a history of changes and facilitate collaboration. They track changes to a folder and its contents in a series of snapshots, where each snapshot encapsulates the entire state of files/folders within a top-level directory. VCSs also maintain metadata like who created each snapshot, messages associated with each snapshot, and so on.

In Git, a history is a directed acyclic graph (DAG) of snapshots, i.e. each snapshot refers to a set of parents, or snapshots that preceded it. This history can be thought of as a tree with one or several branches. Each branch can be forked, i.e. you can clone the code on one branch and make changes to the cloned code, but not the original code. This can happen when for example you want to test some experimental feature but want to keep the original code intact. Branches can also be merged by creating pull requests. For instance, in the example above, after you are finished with the experimental feature and want to add it to the original code (the main branch), you can merge the two branches by creating a pull request, reviewing the changes, and resolving any potential conflicts. These concepts are illustrated in Figure 4.

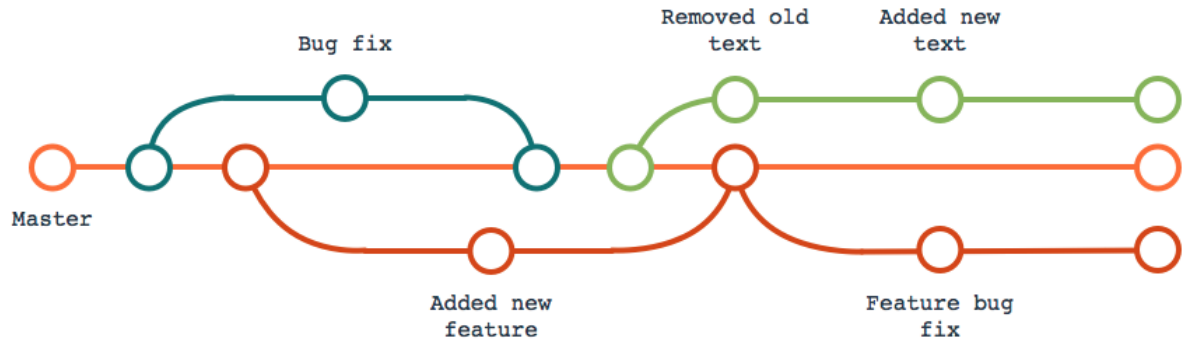


Figure 4: Git branching and merging.

CARMA3 and hardware driver codes located on the processing unit are clones of their respective repositories on GitHub and should be synced with those repositories periodically, especially before rebuilding any Docker images from source. For hardware drivers, any permanent changes made to the code installed on the processing unit (e.g. adding new features) has to be committed to the GitHub repository as well.

git-scm.com/docs is a great reference for Git commands and has a good tutorial for learning to work with Git.

2.6 Autoware

Autoware is a ROS-based open-source software for autonomous vehicles. It supports camera, lidar, and GNSS/INS as primary sensors, and its general structure is depicted in Figure 5.

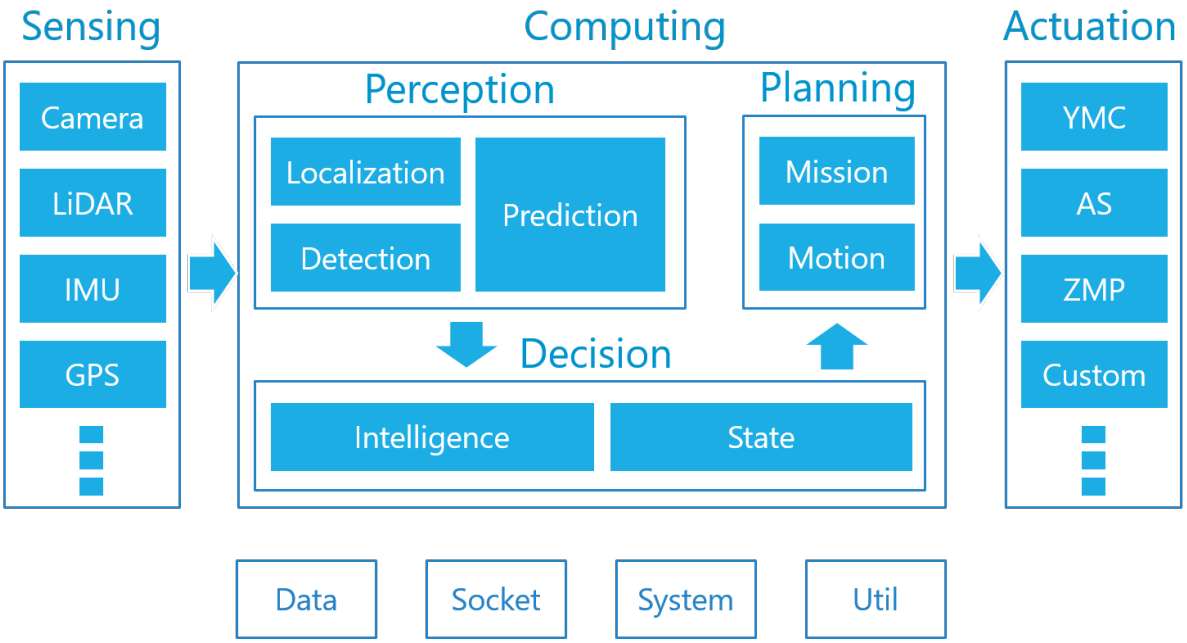


Figure 5: Overview of Autoware.

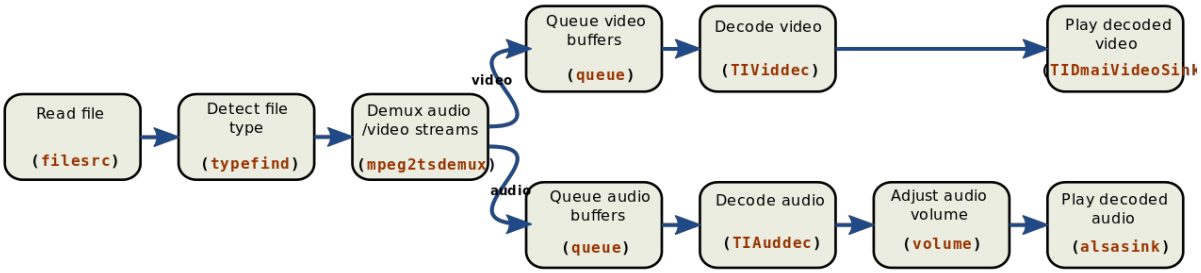
Autoware has modules for *Localization*, *Detection*, *Tracking* and *Prediction*, *Planning*, and *Control*. *Localization* depends on 3D high-definition (HD) map data and the NDT algorithm. The result of *Localization* is complemented by the Kalman Filter algorithm, using odometry information obtained from CAN messages and GNSS/INS sensors. *Detection* is empowered by camera and lidar devices in combination with 3D HD map data. The *Detection* module uses deep learning and sensor fusion approaches. *Tracking* and *prediction* are realized with the Kalman Filter algorithm and the lane network information provided by 3D HD map data. *Planning* is based on probabilistic robotics and rule-based systems, partly using deep learning approaches as well. Finally, *Control* defines the motion of the vehicle with a twist of velocity and angle (or curvature). The *Control* module falls into both the Autoware-side stack (MPC and Pure Pursuit) and the vehicle-side interface (PID variants).

Autoware unfortunately does not have great, up-to-date documentation, but this overview and Autoware Manuals are good reads.

2.7 GStreamer

GStreamer is a pipeline-based multimedia framework that leverages a graph-based structure to construct arbitrary pipelines and connect different media processing systems that enable complex workflows. GStreamer performs extremely lightweight data passing and has a compact core library, resulting in low latency and very high performance.

GStreamer handles media by linking multiple processing elements and creating a pipeline, where a plug-in provides each element. In order to create a hierarchical graph, different elements can be assorted into bins and further grouped together. Elements communicate with each other using pads, and to link two elements a source pad on one can be connected to a sink pad on the other. Data buffers flow from the source pad to the sink pad when the pipeline is active. Pre-defined capabilities of pads allow them to negotiate the type of data that is sent across the pipeline. An example GStreamer pipeline is shown in Figure 6.



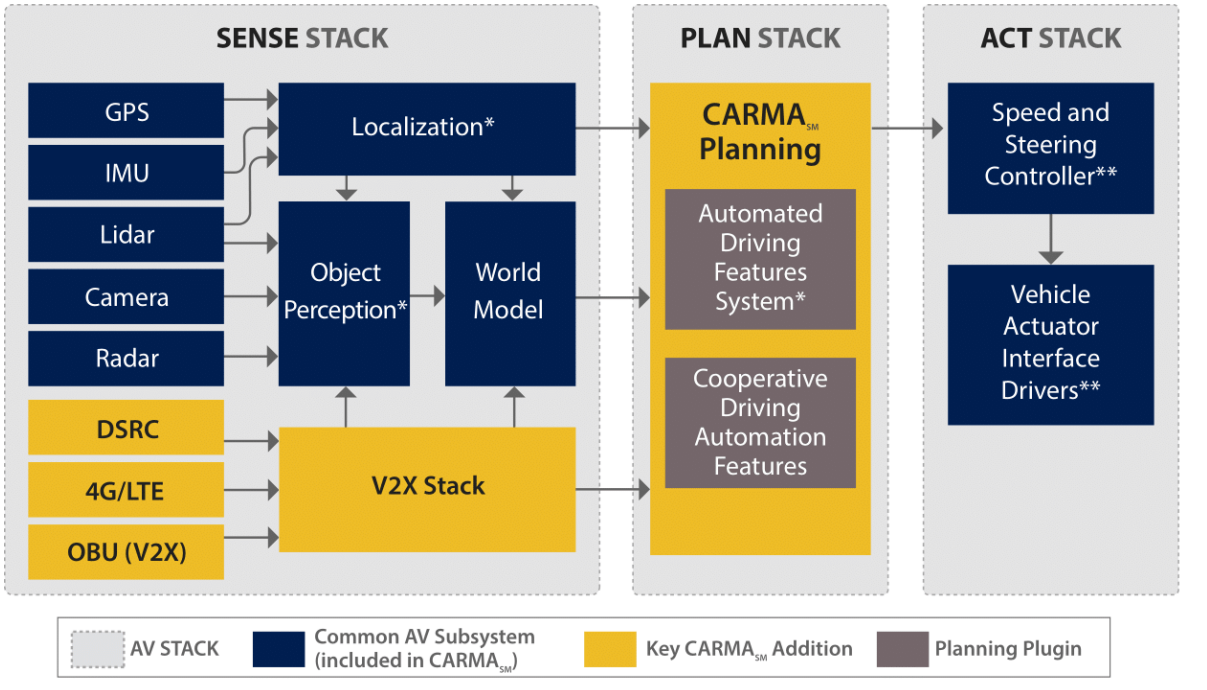

```
gst-launch filesrc location="video.ts" ! typefind ! mpeg2tsdemux name=demux \
       demux. ! 'video/x-h264' ! queue ! TIViddec ! TIDmaiVideoSink \
       demux. ! 'audio/mpeg'   ! queue ! TIAuddec ! volume volume=5 ! alsasink
```

Figure 6: An example GStreamer pipeline.

GStreamer Documentation is a great resource both as a reference and for learning GStreamer through tutorials. The GStreamer API is available in C, JavaScript, and Python.

3 The CARMA Platform

CARMA is a reusable, extensible platform for controlling connected autonomous vehicles (CAVs). It provides a rich, generic API for a variety of sensing and control hardware, as well as for third party plugins that implement vehicle guidance algorithms to plan vehicle trajectories. Several Autoware modules are used in CARMA either directly or with some modifications. An overview of the CARMA Platform can be seen in Figure 7.



AV - Automated Vehicle, **CARMA** - Cooperative Automation Research Mobility Applications, **GPS** - Global Positioning System, **IMU** - Inertial Measurement Unit, **OBU** - On-Board Units, **V2X** - Vehicle-to-Everything
* Supported by Autoware, **Supported vehicle controllers: Dataspeed, PACMOD, and New Eagle.

Figure 7: Overview of the CARMA Platform.

Managing autonomous vehicle (AV) motion involves three aspects. The first is navigation, also known as localization, which is the act of determining where the vehicle currently is with respect to the earth and with respect to the desired path of travel (its planned route). The second is guidance, also known as trajectory planning, which includes the processes of determining how the vehicle is to move from its current location to its destination. The destination and route will be handed to the guidance algorithms, and they then determine how the vehicle's motion needs to be adjusted at any time in order to follow the route. The third aspect of AV motion is control, which covers the actuation of the vehicle's physical devices to induce changes in motion (for land vehicles these typically include causing the wheels to rotate faster or slower and turning the steering wheel). Therefore, the navigation solution becomes an input to the guidance function, and the guidance solution becomes an input to the control function. As the vehicle moves, obviously its location changes so that the navigation function constantly needs to update its solution and the cycle iterates as quickly as necessary to produce a smooth and accurate vehicle motion. The rate of iteration is largely determined by the expected speed of the vehicle.

To provide more flexibility, each CARMA repository is distributable as one or more Docker images containing the compiled applications and their dependencies. These images will be built upon a base image called `carma-base` which contains (and abstracts away) the setup of the ROS system, the user configuration, and installation of any dependencies of the CARMA system. This base image can be shared among multiple images so it only needs to be downloaded onto the vehicle once per released version. Each hardware driver developed for X-CAR is distributable as a Docker image as well.

The CARMA Platform GitHub Page, the CARMA Platform System Architecture document, and CARMA Platform Detail Design documents are great references for learning about the CARMA Platform and working with it.

In what follows, Section 3.1 discusses the hardware and some software aspects of the processing unit, Section 3.2 describes installing the CARMA Platform for the first time, and Section 3.3 discusses building the CARMA Platform from source.

3.1 Processing Unit

The processing unit is the computer mounted inside the vehicle's trunk that handles most of the computation required for X-CAR (everything except camera image processing). For the processing unit, we initially considered pre-built options such as Crystal Rugged's AVC0161, AutonomouStuff Spectra, and Nvidia Drive AGX. However, due to their high quoted price estimates (~$30,000 for Crystal Rugged's AVC0161, ~$20,000 for Nvidia Drive AGX Xavier, and ~$50,000 for Nvidia Drive AGX Pegasus), as well as lack of support for newer components (e.g. support only for older 8th and 9th generation Intel Xeon processors and no support for AMD products), we decided to build our own processing unit using commonly available consumer hardware. Section 3.1.1 discusses the

components used and the assembly process, while Section 3.1.2 discusses setting up the software environment on the processing unit.

3.1.1 Components and Assembly Process

Table 1 shows a list of processing unit components, along with their price at the time of purchase.

Table 1: List of processing unit components.

Item	Component	Price
Case	SilverStone GD08	$207.53
CPU	AMD Ryzen 9 5900X	$569.99
Motherboard	Gigabyte X570 AORUS Master ATX	$359.99
CPU Cooler	Noctua NH-C14S	$84.95
RAM	G.Skill Trident Z RGB 64 GB (2×32 GB) DDR4-3600 CL18	$349.99
Graphics Card	EVGA Nvidia GeForce RTX 3060 XC Gaming	$799.99
Power Supply Unit	EVGA SuperNOVA P2 750W 80+ Platinum Fully Modular ATX	$156.00
Boot Drive	Samsung 980 Pro 1 TB M.2-2280 NVMe SSD	$199.99
High-Speed Storage Drives	2×Samsung 980 Pro 2 TB M.2-2280 NVMe SSD	2×$319.99
Case Fan	5×Arctic P12 PST 120mm	$37.99
Small Case Fan	2×Arctic F8 80mm	2×$6.99
Inverter	Samlex America PST-1000-12	$465.10
Inverter Cable Kit	Samlex America DC-2000-KIT	$137.49
Total		$4022.97

For the case, we were looking for something that could fit the box-shaped volume on the right side of the installed Dataspeed trunk rack. Since most consumer cases are tower (vertical) shaped, our choices were limited to home theatre cases and server chassis. The choice was eventually narrowed down to either SilverStone GD08 or SuperMicro SuperChassis 842XTQ-R606B. We ultimately selected the SilverStone GD08 because of its affordability, better cooling configuration, and larger space for installing storage drives.

For the CPU, we first compared both Intel and AMD's offerings. At the time of purchase, at most price points Intel's products had a better single-thread performance (higher clock frequency), but had fewer cores (and hence threads) and were less power efficient (worse performance per Watt) compared to AMD's products. Therefore, we chose to use an AMD CPU. AMD's CPU offerings are branded as either Ryzen (for general consumers), Ryzen Threadripper and Ryzen Threadripper Pro (for professionals and workstations), or EPYC (for servers). Ryzen Threadripper, Ryzen Threadripper Pro, and EPYC products offer certain advantages over Ryzen products, such as more cores (16, 24, 32, and 64) and hence threads (32, 48, 64, and 128), more PCIe lanes, support for Registered ECC memory, and (on some models) support for up to 2 TB of RAM. However, they have worse single-thread performance, output more heat (larger TDP), and are prohibitively expensive. Ultimately, we chose the AMD Ryzen 9 5900X 12-core, 24-thread CPU, as it provides the best balance between the number of cores,

single-thread performance, heat output, and affordability.

Given our choice of the CPU, we had to select a motherboard with an X570 chipset. We ultimately selected the Gigabyte X570 AORUS Master ATX motherboard for a few reasons. First, its number and arrangement of PCIe slots (three x16 slots and two x1 slots) leaves room for possible future additions such as a second graphics card or a network interface card (NIC). Second, it has three M.2 NVMe slots. This enables installation of one boot drive that is connected to the CPU PCIe lanes (for minimum latency) and two high-speed storage drives in a mirrored pattern that are connected to the chipset PCIe lanes (though installation of the third M.2 NVMe drive limits the number of available SATA ports to 4). Third, it has two Ethernet ports, one 1 Gb/s port and another 2.5 Gb/s port. Finally, it offers proper cooling for motherboard components and VRMs and is reasonably affordable.

We opted for air-cooling the CPU, but using an All-in-One (AIO) water-cooling solution remains an option (the radiator can be installed on top of the floor case fans). The consensus is that Noctua manufactures the world's best CPU air coolers, so we decided to go with the Noctua NH-C14S. This choice was made primarily because of the height limitation of the case. While the Noctua NH-C14S cannot match the cooling performance of Noctua's bigger models like the Noctua NH-D15 (having a Noctua Standardized Performance Rating (NSPR) of 129 compared to 183), it's cooling performance is more than enough for our CPU and even leaves some room for overclocking if there is ever a need for it.

Our choice for RAM configuration was the G.Skill Trident Z RGB 64 GB (2×32 GB) DDR4-3600 CL18 package. It offers great performance and fast response times and given that there are four RAM slots available on the motherboard, choosing a 2×32 GB configuration leaves room for increasing the RAM in the future if needed.

We chose the EVGA Nvidia GeForce RTX 3060 XC Gaming as our graphics card, since the workload is not particularly heavy and mostly comprises machine learning inference tasks such as running YOLOv3 or point cloud data processing. The RTX 3060 GPU satisfies our performance needs, has a relatively low TDP, and its 12 GB of VRAM (which is only surpassed by the RTX 3090) can be useful when loading large datasets. Finally, this particular graphics card has a small profile and was (relatively) affordable at the time of purchase.

The processing unit has to be connected to the vehicle's 12V battery, so we briefly considered using a power supply with a 12V DC input (rather than a 110V AC one). However, the only product we could find was the PowerStream 750W ATX power supply, which is rather inefficient (producing a lot of heat) and more expensive than the alternative solution of using a regular power supply with a DC-AC inverter (we needed an inverter to power the Nvidia Jetson AGX Xavier DevKit anyway). In the end, we selected the EVGA SuperNOVA P2 750W 80+ Platinum Fully Modular ATX power supply based on the combined peak power consumption of our system components. In addition

to satisfying the latter condition, this power supply is very efficient with an 80+ Platinum power rating, fully modular (i.e. all cables can be disconnected and different cable combinations can be used), and reasonably affordable.

For our boot drive and high-speed storage, we selected Samsung 980 Pro M.2-2280 NVMe SSDs, since they are generally considered to have the best performance and reliability of any NVMe SSD. They support Gen4 PCIe read/write speeds, ensuring fast data transfers, and were reasonably affordable at the time of purchase.

For cooling, we selected Arctic P12 PST 120mm and Arctic F8 80mm case fans because of their good performance and affordable price.

Finally, we chose the Samlex America PST-1000-12 pure sine wave inverter as it is the only device that can be remotely controlled using Dataspeed's iPDS.

The following are steps taken for assembling the processing unit. There are many guides online and on YouTube (as well as the manuals of each component) that you can consult if you need help at any stage. The Linus Tech Tips YouTube Channel has great guides for building PCs and the accompanying Linus Tech Tips Forum is a great resource for asking questions and finding answers.

1. Before you begin, it is recommended that you lay down an anti-static mat on the table and ground yourself (e.g. by wearing a wrist strap attached to a grounding cable) to prevent electrostatic damage to the components.

2. Take the motherboard out of its box and electrostatic bag. Grab the motherboard from a raised location (e.g. the I/O shield) and try not to touch the printed circuit board (PCB). Peel any plastic off the motherboard.

3. Take the CPU out of its box and carefully install it in the CPU socket on the motherboard. Make sure that the small corner triangle on the CPU matches the one on the socket.

4. Take the RAM sticks out of their box and install them in the RAM slots on the motherboard. To do so, open the side latches and push down until you hear a click. If you have only two sticks of RAM, install them in the second and fourth slots (counting from the CPU socket) and leave the first and third slots empty. Peel any plastic off the RAM sticks.

5. Unscrew the three M.2 slot covers (heat shields) and peel the plastic off their backs. Install a stand at the 80 mm length hole (one before last) of the first two slots.

6. Install the 1 TB SSD in the first slot (closest to the CPU) and the other two in the remaining slots. Screw the ends, put the slot covers back on (make sure there is contact for heat dispersion) and tighten the screws.

7. Install the CPU cooler according to its manual. First attach the correct mounting brackets (for AM4 socket), then apply a pea-sized amount of thermal compound

on the middle of the CPU, and then screw the cooler onto the mounting brackets. Finally, install the accompanying CPU fan under (or over, case height permitting) the radiator. Peel any plastic off the CPU cooler before installation.

8. Take the case out of its box and remove the storage bracket to reveal the floor case fans.

9. Replace all pre-installed case fans with the five purchased 120 mm case fans. Pay attention to their orientation. The three floor case fans should suck air inside and the two side case fans should push air out.

10. Install the two smaller (80 mm) case fans above the I/O shield cut-out on the back side of the case. They should suck air inside. Since in our arrangement the air volume sucked inside is greater than the volume pushed out, a positive pressure is created inside the case that will push dust out from the remaining holes in the case.

11. Install the motherboard inside the case. Make sure that the I/O shield perfectly aligns with the cut-out on the back of the case.

12. Install the power supply unit with the proper cables (a 24-pin motherboard cable, CPU cable(s), and VGA cable(s) for the graphics card) attached.

13. Connect the front I/O cables to the appropriate headers on the motherboard. These are cables for the power and restart buttons and their LEDs, as well as those for the front USBs.

14. Connect the CPU fan cable to the appropriate header on the motherboard.

15. Daisy-chain the three floor case fan cables to each other and connect the last one to a case fan header on the motherboard. Do the same for the two case fans on the side as well as the smaller ones on the back.

16. Open the latch of the first PCIe slot. Install the graphics card by placing it in the slot and pushing down on it until you hear a click. Peel any plastic off the graphics card.

17. Connect the power supply cables to their appropriate headers on the motherboard and the graphics card.

18. Before closing the case, turn the processing unit on by pressing the power button. It should enter the BIOS after a few minutes. If it does not, look at the two-digit status code on the motherboard and consult the motherboard manual for more information.

19. Turn the processing unit off and close the case. Make sure none of the cables interfere with fan operation.

The steps above are illustrated in Figure 8.

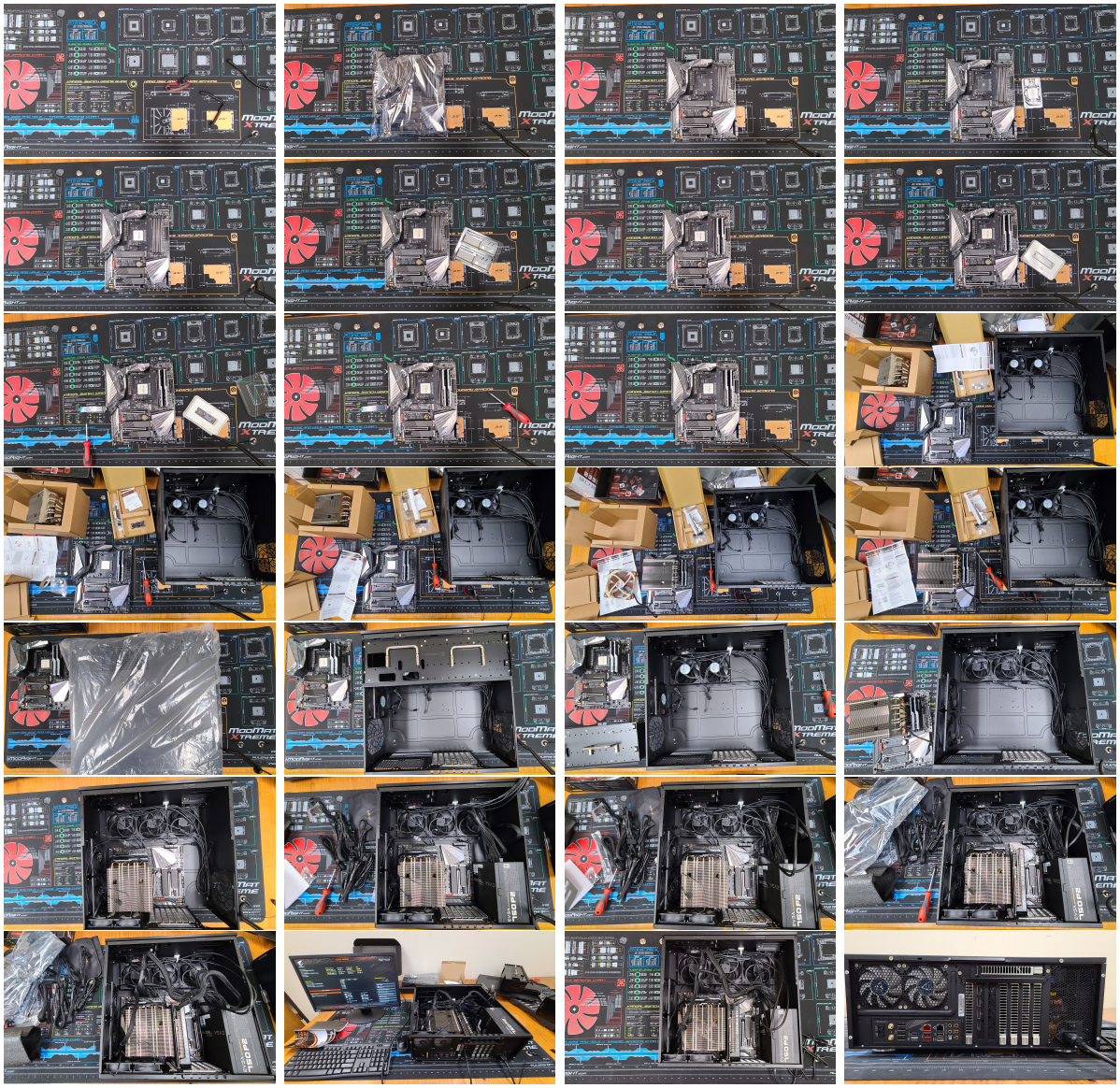


Figure 8: Processing unit assembly steps. Images are in order from left to right, rows are in order from top to bottom.

3.1.2 Software Environment Setup

Before installing the OS, you need to enter the motherboard's BIOS and change a few settings. To do so, press the `del` key repeatedly while the processing unit is booting up. Most BIOS settings are for advanced users and you should not change them unless you are absolutely sure you know what you are doing. Tampering with BIOS settings can result in system instability and/or damage to different components.

The BIOS will likely be in Easy Mode when you enter it. If it is not, press `F2`. Under **Information** check the motherboard, CPU, RAM, and BIOS version information to ensure they are accurate. Also check the CPU and RAM frequency, voltage, and

temperature information to the right to see if anything is out of the ordinary.

Under **DRAM Status**, enable the X.M.P. (eXtreme Memory Profile) Profile 1. This allows the RAM to operate at the advertised 3600 MT/s speed and CL18 latency rather than the default 2666 MT/s speed and CL22 latency, which should improve performance.

To the right of **DRAM Status**, check to ensure all SATA, PCIe, and M.2 devices are recognized. Additionally, make sure that **AMD RAIDXpert2 Tech** is set to Off.

In the **Smart Fan 5** panel, set all fan profiles to Full Speed. Moreover, enable warnings for when a measured temperature goes beyond 90 degrees Celsius or when a fan fails.

You can update the BIOS by downloading the latest BIOS, entering the **Q-Flash** panel and following these instructions. Note that the motherboard has two BIOSs, one primary and one backup, so only update the primary BIOS. This ensures that if the primary BIOS gets corrupted for any reason during this process, the motherboard does not get bricked and can use the backup BIOS instead.

Next, install the OS. Download Ubuntu 20.04.3 LTS; create a bootable Ubuntu USB stick on Windows, Ubuntu, or Mac OS; plug in the USB stick, and install Ubuntu. During the installation process, make sure to install the OS on the boot drive (nvme0). Also, set both the username and the computer's name to `carma`.

After the setup is complete and the OS boots up, go to **Software & Updates** and install the latest Nvidia driver that is tagged as both **proprietary** and **tested**. Then, open the **Terminal** and run the following shell commands.

```
sudo apt-get update
sudo apt-get upgrade -y
```

Finally, the two high-speed storage drives should be combined in a mirrored ZFS pool, which is equivalent to a RAID (Redundant Array of Independent Disks) 1 configuration. This pool shows up as a single volume and data written to this volume is written to both drives so that if one fails, the data is kept safe on the other one. To do so, run the Code Snippet 3 shell commands. The first line installs ZFS, the second line lists drive names, and the third line creates a mirrored ZFS pool named `nvmePool`, assuming that `/dev/nvme1n1` and `/dev/nvme2n1` are the drive names. `nvmePool` should be located under `/` and have a size of 2 TB (may show up as 1.8 TB due to different methods of counting storage size).

```
sudo apt install -y zfs
sudo fdisk -l
sudo zpool create nvmePool mirror /dev/nvme1n1 /dev/nvme2n1
```

Code Snippet 3: Installing and configuring `ZFS`.

3.2 Initial Installation of The CARMA Platform

For initial installation of the CARMA Platform, follow the steps under Development Environment Setup of the official CARMA documentation, starting from the Setup CARMA Platform Prerequisites section (since we are not installing the CARMA Platform on a Virtual Machine (VM)). During setup, do the following:

1. Install Chromium step: it is recommended that you install Google Chrome instead of Chromium.

2. Add Git Branch Display to Terminal step: this step is optional, but not recommended.

3. Cuda 11.2 step: since CUDA has already been installed when installing the Nvidia driver, skip the `# install cuda` portion of the code.

4. Cloning The Repository step: instead of line 3, run the following shell command.

   ```
   curl -L https://raw.githubusercontent.com/VT-ASIM-LAB/carma-platform/develop/carma-platform.repos |
   ↪ vcs import
   ```

   This ensures that our custom drivers and repositories are cloned instead of the standard CARMA ones.

5. Setup The `/opt/carma` Folder step: instead of line 2, run the following shell command.

   ```
   sudo bash ~/opt_carma_setup.bash
   ↪ ~/carma_ws/src/carma-config/example_calibration_folder/vehicle/
   ```

6. Stop after finishing the Setup The `/opt/carma` Folder step. If you proceed, the instructions will download the standard CARMA Docker images. Since we have made some modifications to the code, our Docker images have to be built from source.

7. Link the `/opt/carma/logs` folder to the `/nvmePool/carma/logs` folder so CARMA writes logs to the high-speed storage drives. Moreover, link the `/opt/carma/vehicle` folder to the `/home/carma/carma_ws/src/carma-config/ford_fusion_2017_calibration_folder/vehicle` folder so the proper configuration and calibration files are used.

8. Build the CARMA Platform from source (see Section 3.3).

3.3 Building The CARMA Platform from Source

The `carma-platform` Docker image is a child of the `autoware.ai` Docker image, which itself is a child of the `carma-base` Docker image. Therefore, to build the CARMA Platform from source you have to first build `carma-base`, then `autoware.ai`, and finally `carma-platform`. You also have to build the `carma-web-ui` Docker image, as well as that of any hardware driver you want to use.

To build the CARMA Platform from source make sure that all of the local repositories are synced with their respective GitHub ones. Then, run the Code Snippet 4 shell commands. Building a Docker image can take several minutes, so wait until each step is finished before moving onto the next step.

```
cd ~/carma_ws/src/carma-base/docker
sudo ./build-image.sh -d
cd ../../autoware.ai/docker
sudo ./build-image.sh -d
cd ../../carma-platform/docker
sudo ./build-image.sh -d
cd ../../carma-web-ui/docker
sudo ./build-image.sh -d
```

Code Snippet 4: Building the CARMA Platform from source.

4 Hardware Interface

4.1 Camera

4.1.1 Setup

4.1.2 Calibration

4.1.3 Operation

4.2 Lidar

A Lidar perceives its surrounding environment by sending out laser beams and measuring the time it takes for the reflected light to return to the receiver. Using this method, a lidar can create a detailed point cloud map of its surroundings that can be used to detect different objects and surfaces. The main obstacle to the use of lidars is their extremely high cost.

A wide variety of lidars are available on the market. Based on their scanning mechanism, they can be divided into mechanical (where a spindle spins to give a 360 degree view), solid state (where the field of view is fixed and there are no moving parts, and either MEMS or optical phase arrays are used to steer the beam), and flash (where a light is flashed over a large field of view) lidars. After careful consideration, we selected the Ouster OS1-64U lidar for X-CAR, with 64 beams arranged across a 45 degree vertical field of view in uniform spacing. It provides a detailed, 360 degree view of the surrounding environment with great accuracy, has an IP68 rating, and is extremely affordable compared to similar models from other manufacturers (~$10,000 as opposed to ~$50,000 for a Velodyne Ultra Puck). The lidar comes with a 2-year warranty.

In Section 4.2.1 to Section 4.2.3 we discuss the setup, calibration, and operation of this lidar. The Ouster Lidar Driver for CARMA GitHub page contains the X-CAR driver developed for Ouster lidars. Follow the instructions provided there to download

the driver code, build a Docker image of the driver, and setup that image to launch with the CARMA Platform.

4.2.1 Setup

The lidar setup consists of the components listed in Table 2.

Table 2: Lidar setup components.

Quantity	Item
1	OS1-64U sensor
1	Sensor to Interface Box cable/connector
1	Interface Box and 24V power supply
1	RJ45 cable

Mount the lidar sensor on a platform above the car such that no part of the car's body intersects any of the lidar beams. The platform should be level, the sensor should be laterally centered (be equidistant from the right and left sides of the car), and the connector port should directly face the back of the car. This is because in the sensor's internal coordinate frame the connector port is located in the -x direction (with the z direction being vertically upward).

Do not take the sensor's baseplate mount off. This can cause it to overheat during operation and eventually die.

Install the Interface Box below the iPDS (see Section 4.8) and connect it to the iPDS via a 12V/24V DC/DC converter. Connect the Interface Box to the sensor using the provided connector. Connect the Interface Box to the processing unit (or an intermediary network switch) using the RJ45 cable.

Using a web browser, go to the `http://os-[serial number].local/` address to access the sensor's home page. From there, you can install a firmware upgrade, see diagnostic information, and change configuration parameters.

For more information see the OS1 Datasheet, OS1 Hardware User Manual, OS1 Software User Manual, and the Ouster Sensor API Guide.

4.2.2 Calibration

The lidar sensor is pre-calibrated and does not need any calibration.

4.2.3 Operation

To connect to the lidar sensor, first create a network connection named `Ouster Lidar` on the processing unit. Under the **IPv4** tab, select `Manual` as the **IPv4 Method**, enter `10.5.5.1` as **Address**, and `255.255.255.0` as **Netmask**. Then run the following shell

command (as explained in this OS1 Example Code and wait until you see the sensor's serial number.

```
sudo dnsmasq -C /dev/null -kd -F 10.5.5.50,10.5.5.100 -i [eth name] --bind-dynamic
```

In the shell command above, `[eth name]` is the network interface you are connecting to and can be obtained by running the `ifconfig -a` shell command. If you have followed the driver installation instructions correctly, you should be able to launch the driver with the CARMA Platform.

In addition to installation instructions, The Ouster Lidar Driver for CARMA GitHub page contains information about the driver's ROS API. The driver is built upon Ouster's ROS driver with two modifications. First, it publishes a discovery message for the CARMA Platform indicating its current status (operational, degraded, fault, or off). Second, it uses ROS Wall Time as timestamp for published point cloud messages instead of the lidar sensor's internal clock, in line with the rest of the CARMA Platform.

4.3 GNSS/INS

A GNSS/INS module provides accurate position (latitude, longitude, and heading), velocity, and time information. It does so by receiving data from one or multiple global navigation satellite systems (GNSS) such as GPS, GLONASS, BDS, or Galileo, and fuses that information with data from its motion and rotation sensors that comprise an inertial navigation system (INS).

After careful evaluation of the products available on the market, we selected the Inertial Labs INS-D Dual Antenna GNSS/INS module. It provides position, velocity, and time information with great accuracy, is IP67 and MIL-STD-810G certified, and is reasonably affordable compared to similar models from other manufacturers (~$10,000 as opposed to ~$35,000 for a NovAtel PwrPak7 series GNSS/INS module).

In the following sections we discuss the setup, calibration, and operation of this GNSS/INS module. The Inertial Labs GNSS/INS Driver for CARMA GitHub page contains the X-CAR driver developed for Inertial Labs GNSS/INS modules. Follow the instructions provided there to download the driver code, build a Docker image of the driver, and setup that image to launch with the CARMA Platform.

4.3.1 Setup

The GNSS/INS setup consists of the components listed in Table 3.

Mount the INS-D module on a level surface below the iPDS and note the direction of coordinate frame axes displayed on the module. It is preferred that the x axis points to the right side of the car and the y axis points directly forward (i.e. the connectors point directly to the back of the car).

Table 3: GNSS/INS setup components.

Quantity	Item
1	INS-D module
1	Development board
3	RS232/RS422 to USB cable
1	Power adapter
2	GNSS antenna
2	Antenna to module TNC cable

Mount the primary and secondary antennas on top of the car with as much distance between them as possible. Note the location of the primary and secondary antennas relative to the INS-D module accelerometer mass-center.

Connect the two antennas to the INS-D module using the provided TNC cables, and connect the INS-D module to the development board. Connect the development board power connector to the iPDS. Finally, connect the development board to the processing unit using a RS232/RS422 to USB cable.

For more information see the INS Datasheet, as well as the INS Step-by-Step Quick Start Guide, the INS Interface Control Document, and the INS User Manual, which are available on the USB stick provided by Inertial Labs.

4.3.2 Calibration

In the INS GUI program provided by Inertial Labs, first go to **Test options**. Select the appropriate **Serial port** and click **Auto** for **Baud rate**. check **Allow auto start** and select **INS OPVT2A Output** as the desired **Output data format**.

Next, move on to **Device options**. Set **Data rate** to 20 Hz and provide default **Location** values for **Latitude**, **Longitude**, and **Altitude**. Based on the mounting locations and orientations of the INS-D module and the two antennas, fill in the appropriate values for **Alignment angles**, **Primary antenna position relative to the IMU**, and **Secondary antenna position relative to the IMU**. Leave **Antennas baseline orientation** unchanged (all zero). Finally, fill in the appropriate values for **PV measuring point relative to the IMU** with the position of `base_link` relative to the INS-D module. The CARMA Platform considers `base_link` to be the midpoint of the reflection of the rear axle on the ground.

INS sensors (accelerometer, gyroscope, and magnetometer) are pre-calibrated and do not need calibration.

4.3.3 Operation

To connect to the INS-D module, run the following shell command to ensure that `carma` is the owner of `/dev/ttyUSB0`.

```
sudo chown carma:carma /dev/ttyUSB0
```

In addition to installation instructions, the Inertial Labs GNSS/INS Driver for CARMA GitHub page contains information about the driver's ROS API. The driver is built upon Inertial Lab's ROS driver with two modifications. First, it publishes a discovery message for the CARMA Platform indicating its current status (operational, degraded, fault, or off). Second, instead of publishing the default Inertial Labs messages, it publishes fused GNSS/INS data as `gps_common/GPSFix` messages. It also publishes unfused INS data as `sensor_msgs/Imu` messages, which can be helpful when there is a need for a standalone inertial measurement unit (IMU) device.

4.4 Radar

4.4.1 Setup

4.4.2 Calibration

4.4.3 Operation

4.5 Ultrasound

4.5.1 Setup

4.5.2 Calibration

4.5.3 Operation

4.6 Dedicated Short Range Communication (DSRC)

4.6.1 Setup

4.6.2 Calibration

4.6.3 Operation

4.7 Drive-by-Wire (DbW) Kit

A Drive-by-Wire (DbW) kit acts an interface for communicating with the vehicle. Using a DbW kit, the CARMA Platform can receive vital information about the status of various vehicle functions (such as braking, steering, etc.) and send commands for controlling the vehicle. The DbW kit connects to the vehicle's CAN bus to read and publish messages. CAN bus message information (message IDs and their data format) are proprietary, so it takes an enormous effort to reverse-engineer the entire CAN bus, read CAN messages, and publish messages that fool the vehicle into thinking they are coming from a driver.

Therefore, most research outfits rely on a few companies that have done this reverse engineering and developed a DbW kit.

Dataspeed is the only company that provides a DbW kit for some Ford vehicles, including our 2017 Ford Fusion Hybrid SE. Therefore, we used the Dataspeed Drive-by-Wire Kit for X-CAR. The DbW kit is a complete hardware and software system that enables electronic control of a vehicle's brake, throttle, steering, and shifting, while maintaining safe operation by allowing a safety driver to override the system at any moment. The DbW kit also include's Dataspeed's Universal Lat/Lon Controller (ULC) that receives desired speed and angular velocity commands and publishes the appropriate steering, braking, and throttle commands. This function removes the need for additional (and costly) software such as AutonomouStuff's Speed and Steering Control (SSC) that are used in the standard CARMA configuration. The DbW kit is by far the most expensive part of X-CAR, costing north of $50,000.

In the following sections we discuss the setup, calibration, and operation of the DbW kit. The Dataspeed Drive-by-Wire CAN Driver for CARMA and Dataspeed Universal Lat/Lon Controller (ULC) Driver for CARMA GitHub pages contain the X-CAR drivers developed for the DbW kit and the ULC. Follow the instructions provided there to download driver codes, build a Docker image of the drivers, and setup those images to launch with the CARMA Platform.

4.7.1 Setup

The DbW kit consists of the components listed in Table 4.

Table 4: DbW kit components.

Quantity	Item
1	Brake Throttle Emulator Module
1	Shifting Interface Module
1	Steering Interface Module
1	CAN Gateway Module
1	CAN USB breakout board

DbW kit components are installed by Dataspeed. Connect the CAN Gateway Module to the processing unit using a USB cable.

For more information see the Dataspeed Drive-by-Wire Kit FAQ and download the appropriate Dataspeed Release Package.

4.7.2 Calibration

The DbW kit does not require any calibration, but internal parameters can be changed if needed. To do so, connect the respective module to a device operating Windows using the

module's USB cable. Then, use Dataspeed's software provided in their Release Package to connect to the module, change certain paramters, or install a new firmware.

4.7.3 Operation

To connect to the DbW kit, first run the `lsusb` shell command to see which bus and device it has been assigned to. Then, assuming for example that the DbW kit is assigned to device `007` on bus `001`, run the following shell command to ensure that `carma` is the owner of `/dev/bus/usb/001/007`.

```
1 sudo chown carma:carma /dev/bus/usb/001/007
```

In addition to installation instructions, the Dataspeed Drive-by-Wire CAN Driver for CARMA and Dataspeed Universal Lat/Lon Controller (ULC) Driver for CARMA GitHub pages contain information about the drivers' ROS API. The drivers are built upon Dataspeed's ROS drivers with two modifications. First, they each publish a discovery message for the CARMA Platform indicating their current status (operational, degraded, fault, or off). Second, the CAN driver publishes critical vehicle information reported by the CAN bus in the message formats required by CARMA, while the ULC driver receives an Autoware Vehicle Command message for controlling the vehicle.

4.8 Power Distribution System

A power distribution system connects to the car's 12V battery and provides power to various devices installed in the vehicle. It also allows monitoring the power consumption of different components.

Because Dataspeed provided the DbW kit, we also used their intelligent Power Distribution System (iPDS) along with a Samlex America PST-1000-12 pure sine wave inverter. The iPDS connects to the vehicle's 12V battery and provides 12 channels at 12V, 15A each, but can also run at 24V, 10A (if the vehicle has a 24V battery). It offers programmable startup and shutdown functionality; over-current feedback and diagnostics; and CAN, Ethernet, and USB communication. Moreover, it is relatively affordable at $4,500. The Samlex America PST-1000-12 inverter connects to the car's 12V battery and provides 120V AC outputs that can be used to power the processing unit and other devices such as the Nvidia Jetson AGX Xavier DevKit used in our camera setup. Moreover, it can be controlled through the iPDS interface via an Ethernet connection.

In the following sections we discuss the setup, calibration, and operation of the power distribution system.

4.8.1 Setup

The power distribution system consists of the components listed in Table 5.

Table 5: Power distribution system components.

Quantity	Item
1	iPDS module
1	iPDS touchscreen
1	iPDS switch panel
1	Samlex America PST-1000-12 inverter
1	Samlex America DC-2000-KIT cable kit

iPDS components are installed by Dataspeed. The iPDS module is installed in the trunk, the switch panel is installed in the storage bin between the front seats, and the touchscreen is installed in a cupholder.

To install the inverter, connect the black and red wires in the cable kit to the negative and positive terminals of the inverter. For your safety, disconnect the 300V battery of the car by taking out the orange fuse behind the back seat. Next, connect the red wire to the positive terminal of the 12V battery (accessible through the trunk, different from the 300V battery), then connect the black wire to the negative terminal of the battery (this may create some sparks). Connect one end of an AWG #8 cable to the ground terminal of the inverter and the other end to a location on the vehicle's body that is not painted (bare metal).

For more information see the Dataspeed intelligent Power Distribution System Datasheet and download the appropriate Dataspeed Release Package.

4.8.2 Calibration

The power distribution system does not require any calibration, but the iPDS can be accessed to change a configuration or for diagnostics if needed. To do so, connect the iPDS to a device operating Windows using a USB cable. Then, use Dataspeed's software provided in their Release Package to connect to the module, change a configuration, or install a new firmware.

4.8.3 Operation

The iPDS automatically turns on when the vehicle is turned on. To turn on all channels, press start on the touchscreen or the large button to the top right of the switch panel. To turn on individual channels, swipe left on the touchscreen and press on individual channel names. Alternatively, press on the small buttons on the switch panel. To turn off all channels, press the stop button on the touchscreen or the large button on the top left of the switch panel. The iPDS automatically turns off when the vehicle is turned off. Table 6 shows which channel each component is connected to.

To turn on the inverter, put the switch in the **I** (upper) position. To set the inverter to automatically turn on and off via an Ethernet connetion, put the switch in the **II**

Table 6: List of components connected to the iPDS.

Channel	Component
1	DbW kit
2	Radar
3	Lidar
4	DSRC
5	GNSS/INS
6	Camera

(lower) position. To turn the inverter off put the switch in the **O** (middle) position.

5 Localization

The localization subsystem of the CARMA Platform (from here on referred to as Carma) is responsible for determining and reporting the most accurate estimate of the vehicle's position, velocity, acceleration, and heading at each moment. This is done by using information from the GNSS/INS module, a point cloud map, and a vector map.

Before moving forward, it is essential to understand the coordinate frames used by Carma. The **earth (world)** coordinate frame is an Earth-centered, Earth-fixed (ECEF) coordinate system with its origin at the Earth's center of gravity where each location is determined by latitude and longitude coordinates. The **map** coordinate frame is an ENU (East North Up) coordinate frame with its origin at the starting point of the point cloud map, which is read from the `geoReference` tag of the vector map (more information about this is provided below). Finally, the **base_link** coordinate frame is a FLU (Forward Left Up) coordinate frame attached to the vehicle's body, with its origin at `base_link`. Carma considers the midpoint of the reflection of the rear axle on the ground as `base_link`. Each sensor also has its own coordinate frame, though the origin and axis orientation varies from sensor to sensor.

In what follows, Section 5.1 discusses point cloud map generation, Section 5.2 explains vector map generation, Section 5.3 discusses route file creation, and finally Section 5.4 examines the localization process in Carma.

5.1 Point Cloud Map Generation

A point cloud map is comprised of data points in space that represent the three-dimensional shape of the surrounding environment. To generate a point cloud map of an environment for autonomous driving, a car needs to drive around and scan that environment with its lidar sensor(s).

Individual lidar scans (each lidar scan is a point cloud generated at a specific time step) are usually merged to form a map using a process called normal distribution transform (NDT) mapping. In NDT mapping, each lidar scan is transformed into a grid-based representation, where each grid cell is approximated with a multivariate Gaussian distribution. Along with knowledge of the vehicle's motion (usually obtained from an IMU), this representation enables finding a transformation between consecutive scans, filtering out moving objects, and allowing stationary parts to be merged together to generate a map. For further information, see The Normal Distributions Transform: A New Approach to Laser Scan Matching and Autoware.Auto's NDT Knowledge Base.

Follow the steps below for point cloud map generation:

1. Determine the location of `base_link` (reflection of the midpoint of the rear axle) in the **earth** coordinate frame as accurately as possible. To do so, use the latitude and longitude coordinates reported by the GNSS/INS module. Confirm those values with the coordinates reported by Google Maps (or a similar mapping service) of the marked location as seen in satellite view. Also, note the heading (degrees from North) of the vehicle as accurately as possible.

2. Ensure that all of the values in the `~/carma_ws/src/carma-config/ford_fusion_2017_calibration_folder/vehicle/calibration/lidar_localizer/ndt_matching/params.yml` file are accurate and represent the transformation from `base_link` to the lidar sensor coordinate frame.

3. Ensure that all hardware drivers timestamp published information with ROS Wall Time. A mismatch between timestamps will result in a corrupted map.

4. Follow Step 2 of the CARMA User Guides' Point Cloud Map Generation instructions. In other words, run Carma and drive around. Try to drive relatively slowly, and drive each road in the opposing direction as well. Slow down for sharp turns and U-turns, and try to avoid going in reverse (if the IMU is not configured properly it may only report the absolute value of the car's linear and angular motion, which can confuse the NDT algorithm). When you are finished, stop Carma. The data is recorded as a rosbag file under `/opt/carma/logs`.

5. Using the value of heading recorded earlier, add the starting heading angle from East (in radians) to the `tf_yaw` field in the `params.yaml` file referenced in Step 2. For example, if the starting heading was towards North, add $\frac{\pi}{2}$ to the `tf_yaw` field. Alternatively, if the starting heading was towards South-West, add $-\frac{3\pi}{4}$ to the `tf_yaw` field. The Autoware NDT script uses these values to determine the origin of the point cloud map in relation to the recording device (lidar).

6. Follow Step 3 of the CARMA User Guides' Point Cloud Map Generation instructions. Monitor the point cloud map generation process in RViz to ensure its accuracy. This should result in a `*.pcd` file.

7. Inspect the `*.pcd` file for accuracy. If there are inaccuracies (e.g. if some trees have ended up in the middle of a road), change some NDT mapping parameters and try again. These parameters can be changed in either of the following launch files, `~/carma_ws/src/carma-platform/carma/launch/utility/carma_ndt_mapping.launch` and `~/carma_ws/src/autoware.ai/core_perception/lidar_localizer/launch/ndt_mapping.launch`, or the NDT mapping source code at `~/carma_ws/src/autoware.ai/core_perception/lidar_localizer/nodes/ndt_mapping/ndt_mapping.cpp` (you will need to rebuild the CARMA Platform from source, see Section 3.3). These paramters include `method_type` (point cloud processing method used, either `pcl_generic`, `pcl_anh`, `pcl_anh_gpu`, and `pcl_openmp`), `max_iter` (maximum number of iterations), `ndt_res` (NDT resolution), `step_size` (step size), `trans_eps` (transformation epsilon), `voxel_leaf_size` (leaf size of voxel grid filter), `min_scan_range` (minimum detection range of the lidar), `max_scan_range` (maximum detection range of the lidar), `use_imu` (whether the IMU was used when recording the data), `use_odom` (whether vehicle odometery was used when recording the data), and `imu_upside_down` (whether the IMU was upside down).

8. Import the `*.pcd` file into MATLAB (or a similar software) and downsample it to reduce loading time when Carma is initiated. Use the `gridAverage` method with a `gridStep` between 0.2 and 0.3 as a start. Inspect the resulting point cloud map and try a higher or lower `gridStep` if needed.

9. Save the final point cloud map under `/opt/carma/maps` as `pcd_map.pcd` (Carma looks for a point cloud map file named `pcd_map.pcd` in the directory above when it is initialized).

5.2 Vector Map Generation

A vector map is a vector representation of map data and traffic rules. Carma employs the Lanelet2 vector map framework. It is based on atomic lane sections, or *Lanelets*, which together form the road through their neighborhood relationship. Traffic rules are represented by so-called *Regulatory Elements*, which represent objects like traffic lights and stop lines.

A Lanelet2 map is divided into a *physical layer*, which contains the observable (usually real) elements and a *relational layer*, in which the elements of the physical layer are connected to lanes, areas and traffic rules. A third layer results implicitly from the contexts and neighbourhood relationships of the relational layer: the *topological layer*. In this layer, the elements of the relational layer are combined into a network of potentially passable regions that depend on the road user and driving scenario.

A Lanelet2 map consists of five elements: *points* and *line strings* belonging to the physical layer and *lanelets*, *areas*, and *regulatory elements* belonging to the relational

layer. Each element is identified by a unique ID (which is useful for an efficient construction of the topological layer) and can be assigned attributes in the form of key-value pairs. Some of these attributes are fixed, but additional attributes can be used to enhance the map. You can learn more about Lanelet2 maps in Lanelet2: A High-Definition Map Framework for the Future of automated Driving.

We use MathWorks' RoadRunner, which is a proprietary software, for vector map generation. However, free and open-source alternatives such as Tier 4's VectorMapBuilder are also available.

Follow the steps below for vector map generation:

1. Follow the instructions in Download GIS Data for Use in RoadRunner to download the geographic information system (GIS) data relevant to the area being considered for vector map generation. Download all three recommended data sets, though you will only need imagery data.

2. Follow the instructions in Create Roads Around Imported GIS Assets. In the **Download and Import GIS Assets into RoadRunner** step, add the point cloud map generated in Section 5.1 as an asset alongside other GIS assets. Then, follow the instructions in the **Set World Origin** step. Follow the instructions in the **Add GIS Assets** step, but only add the GIS imagery data. Instead of adding the GIS elevation and point cloud data, add the point cloud map generated in Section 5.1, using its origin latitude and longitude coordinates to set the **File Projection (WKT)** attribute. Verify that the point cloud map matches the GIS imagery data.

3. Select the proper road type from the **RoadStyles** folder under **Library Browser** and use RoadRunner's various tools to draw and adjust non-intersecting road segments. Create cross sections along each road segment and use the **2D Editor** to match the road profile to the profile of the point cloud data. Ensure that each segment matches the curvature, width, and lane arrangement of the underlying road in the point cloud map. It is vital that the point cloud map and the vector map align with each other as accurately as possible since Carma uses the former for localization and the latter for guidance (path planning and navigation).

4. Create intersections as needed. Adjust the road and center line curvatures to ensure smooth driving behavior. Try to make the curvatures as large as possible.

5. Add regulatory elements as needed, including but not limited to speed limits, stop signs, stop lines, and traffic lights.

6. Save the resulting scene. Then, export it as an OpenDRIVE (`*.xodr`) file. Inspect the `*.xodr` file and verify its accuracy.

7. Convert the OpenDRIVE vector map to a Lanelet2 vector map by running the following shell commands.

```
docker build -t opendrive2lanelet2convertor .
docker run --rm -it -v [path to the folder containing the OpenDRIVE maps]:/root/opendrive2lanelet/map opendrive2lanelet2convertor
```

8. Inspect the generated `*.osm` vector map and verify the accuracy of the map and the `geoReference` tag. Save the final vector map under `/opt/carma/maps` as `vector_map.osm` (Carma looks for a vector map file named `vector_map.osm` in the directory above when it is initialized).

5.3 Route Creation

A route in Carma is a set of one (or multiple) destination points that the vehicle has to reach consecutively. When creating a route file, you need to ensure that a viable path connecting consecutive destination points exists on the vector map.

A route file is a `*.csv` file that has four columns, representing latitude, longitude, ID, and description of each destination point, respectively. Follow the instructions in Carma Development Resources' Creating a New Route File to create a route file. Save your route file to the `/opt/carma/routes` folder.

5.4 Localization

Carma uses data from both the GNSS/INS module and the lidar sensor for localization. The GNSS/INS module provides the latitude and longitude coordinates of the vehicle's location which are then transformed into coordinates in the **map** frame. In contrast, real-time lidar scans are used for localization in a process called NDT matching, which is much more accurate. The uncertainty in the reported location of the former is around 0.5 to 1 meters, but only around 0.1 meters for the latter.

In NDT matching, much like NDT mapping, each real-time lidar scan is transformed into a grid-based representation, where each grid cell is approximated with a multivariate Gaussian distribution. The point cloud map is similarly transformed, and an optimization problem is solved to find the best fit between the two grid-based representations. This, in turn, determines the vehicle's location with high accuracy.

Carma has five overall localization modes: `NDT only`, `GNSS only`, `auto select between NDT and GNSS with GNSS timeout`, `auto select between NDT and GNSS without GNSS timeout`, and `GNSS only with NDT initialization`. The localization mode can be changed in the `~/carma_ws/src/carma-platform/localization_manager/config/parameters.yaml` file.

While Carma uses the last localization mode by default, we prefer the fourth localization mode (`auto select between NDT and GNSS without GNSS timeout`) since we found the reported location to constantly drift in the fifth localization mode. The fourth localization mode mostly relies on NDT matching, but can use the location reported by the GNSS/INS module in case of a fault in the lidar sensor or the absence of a point cloud map.

In the fourth localization mode, the vehicle's location is initially determined by the GNSS/INS module when Carma starts up. For each iteration of NDT matching, first the number of points in the point cloud is reduced using a voxel grid filter. Then, a number of (by default 700) points are chosen at random to form the point cloud used for NDT matching. The location calculated by the NDT matching process is then used in an extended Kalman filter (EKF) to further the accuracy of the calculated location. The first iteration of NDT matching is initialized by the location reported by the GNSS/INS module or a user input in RViz, while subsequent iterations use the EKF calculated location in the previous step as the initial guess for NDT matching.

6 World Model

7 V2X

8 Guidance

9 Configuration and Calibration

Core Carma subsystems are designed to be agnostic of the platform (vehicle and sensor suite) they are installed on, which allows Carma to be versatile and run on any vehicle and sensor suite that satisfies its API requirements. As a result, the ~/carma_ws/src/carma-config directory contains vehicle-specific and vehicle-class-specific configuration and calibration files that are needed for operation of Carma on different vehicles.

A configuration folder contains vehicle configuration data that is specific to that class of vehicle and sensor suite, such as the launch file for the sensors used on that vehicle, or the Docker compose files that launch Carma with the appropriate configuration. In contrast, a calibration folder contains information that is specific to individual vehicles, such as precise sensor orientations or controller tuning. In what follows, Section 9.1 discusses the X-CAR configuration files for a 2017 Ford Fusion Hybrid SE, while Section 9.2 lists the calibration files.

9.1 X-CAR Configuration

X-CAR configuration files are located in the ~/carma_ws/src/carma-config/ford_fusion_2017 directory and include the following files, copied from the ford_fusion_sehybrid_2019 folder of the official Carma installation with some changes.

- bridge.yml: contains explicit topic mapings between ROS 1 and ROS 2 portions of Carma while the ROS 2 migration is underway.
- build-image.sh: generic build script for Carma configuration Docker images.
- carma_docker.launch: the ROS launch file for the Carma ROS network. Launches all the needed ROS nodes and sets up the parameter server. Also sets up all static transforms used by TF2 within the system. Sets the path for vehicle configuration

and calibration directories, route files, and point cloud and vector maps. The `rosbag_exclusions` argument was added to this file for X-CAR use.

- `carma_docker.launch.py`: Python file for launching the Carma system.
- `carma_rosconsole.conf`: sets the default ROS output level as well as package specific log levels to one of `ALL`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, or `OFF`. It can be useful for debugging issues with specific packages.
- `carma.config.js`: handles Javascript namespaces and modules for running Carma. The `ip` variable was changed from `192.168.88.10` to `127.0.0.1` for X-CAR use.
- `carma.env`: sets up the environment variables needed to run Carma.
- `docker-compose-background.yml`: launches background Docker containers required for running Carma. `container_name` was changed from `web-ui` to `carma-web-ui` and the `ROS_IP` environment variable was changed from `192.168.88.10` to `127.0.0.1` for X-CAR use.
- `docker-compose.yml`: launches Docker containers required for running Carma. Alongside launching roscore and Carma, it allows the X-CAR user to launch any of the mock CAN, controller, lidar, GNSS/INS, and camera drivers, as well as Dataspeed CAN and ULC drivers, Ouster lidar driver, Inertial Labs GNSS/INS driver, and GStreamer camera driver.
- `Dockerfile`: generic template for CARMA configuration Dockerfiles.
- `drivers.launch`: launches the ROS nodes for mock or real hardware drivers. Default drivers were replaced with X-CAR hardware drivers and their launch arguments.
- `system_release.sh`: takes a system release name and version number as arguments, and updates version dependencies in `docker-compose.yml` and `docker-compose-background.yml` accordingly.
- `VehicleConfigParams.yaml`: provides the ROS parameters that define the characteristics of the host vehicle configuration. Values of the `vehicle_year`, `vehicle_length`, `vehicle_width`, `vehicle_height`, and `required_drivers` fields were changed for X-CAR use.

9.2 X-CAR Calibration

X-CAR calibration files are located in the `~/carma_ws/src/carma-config/ford_fusion_2017_calibration_folder/vehicle/calibration` directory and include the following files, copied from the `example_calibration_folder/vehicle/calibration` folder of the official Carma installation with some changes.

- `identifiers/UniqueVehicleParams.yaml`: provides the ROS parameters that define a unique vehicle's specific traits.
- `lidar_euclidean_cluster_detect/calibration.yaml`: defines lidar parameters for Autoware's lidar_euclidean_cluster_detect package that detects individual objects from point cloud data.
- `lidar_localizer/ndt_matching/params.yaml`: defines TF2 transformation parameters used for NDT matching.
- `mpc_follower/calibration.yaml`: defines model predictive control (MPC) follower weights and parameters.
- `naive_motion_predict/calibration.yaml`: defines lidar parameters for Autoware's motion prediction package.
- `ouster`: stores Ouster lidar's metadata.
- `point_preprocessor/ray_ground_filter/calibration.yaml`: defines lidar parameters for Autoware's ground detection package.
- `pure_pursuit/calibration.yaml`: pure pursuit lookahead calibration parameters.
- `range_vision_fusion`: contains lidar-camera calibration files. To perform the calibration see CARMA Development Resources' Vehicle Sensor Calibration. During the calibration process, ensure that the correct intrinsic camera calibration file is used. Following the calibration, inspect results and verify that the calculated transformation from the lidar to each camera matches their real-world physical relationship.
- `urdf/carma.urdf`: defines the static TF2 transformations required for the operation of Carma on a vehicle. All transformations are relative to `base_link`, which is defined as the midpoint of the rear axle's reflection on the ground. These transformations need to be defined accurately to ensure proper vehicle operation. Note that the Inertial Labs GNSS frame is located at `base_link` (with a 180-degree rotation) since the GNSS/INS module is configured to report the location of `base_link`.
- `vision_beyond_track/calibration.yaml`: defines camera parameters for Autoware's multi-object tracking.

10 Operation

Before operating X-CAR, ensure that all necessary Docker images for Carma and the desired hardware drivers have been built. You can verify this by running the `docker images` shell command. Note that Carma is an experimental software meant for research and education, so before operating X-CAR ensure that the vehicle is in an open area away from other vehicles and vulnerable road users. Have a safety driver behind the steering wheel and verify that the emergency stop button works.

Follow the steps below to operate X-CAR:

1. Run the following shell commands to build the latest Carma configuration Docker image and set it as the configuration used by Carma.

```
cd ~/carma_ws/src/carma-config/ford_fusion_2017
sudo ./build-image.sh -d
carma config set usdotfhwastoldev/carma-config:develop-ford-fusion-2017
```

2. Carma requires CAN, controller, lidar, GNSS/INS, and camera drivers to run (otherwise it shuts down), though some of these can be mock drivers. At a minimum, Carma needs the CAN, ULC, and lidar drivers to be functional. To start Carma, run the following shell command.

```
carma start all roscore platform [can driver] [controller driver] [lidar driver]
↪  [gnss driver] [camera driver]
```

   In the command above, each driver can be either a mock driver or a hardware driver, but not both. For example, if you want to run X-CAR without cameras, run the following shell command to start Carma.

```
carma start all roscore platform dataspeed-can-driver
↪ dataspeed-controller-driver ouster-lidar-driver inertiallabs-gnss-driver
↪ mock-camera-driver
```

3. Copy the `carma_default.rviz` file from the `~/carma_ws/src/carma-platform/carma/rviz` directory into the `/opt/carma/maps` directory. Open a new Terminal tab and run the following shell command to launch RViz.

```
rviz -d /opt/carma/maps/carma_default.rviz
```

4. Verify that the sensors are working properly. You can visualize lidar and camera outputs inside RViz. To check the output of the GNSS/INS module and the Dataspeed DbW kit, open a new Terminal tab and run the `carma exec` shell command to open a shell inside the `usdotfhwastoldev/carma-platform:develop` Docker image. Then use `rostopic echo` to view the output of the relevant ROS topics.

5. Wait for the vector and point cloud maps to load. When they are loaded they will be shown in RViz.

6. In Google Chrome (or Chromium), go to `localhost` or `127.0.0.1`. Click on the **Start Platform** button. You can turn on **Debug Mode** from the lower right corner of the screen.

7. In the new screen, click on **Logs** and verify that *All essential drivers are ready* message is displayed.

8. On the top right corner, verify that the localization status indicator is green. You can also verify accurate localization in RViz, where the real-time lidar scan (shown in red) should match the point cloud map (shown in white).

9. Select a route from the list of routes. If the route is valid, RViz will show the waypoints generated from the vehicle to the final destination.

10. In Google Chrome, click on **Status** and check the **Robot Active** and **Robot Enabled** fields. Both will likely report **false**.

11. Open a new Terminal tab and run the `carma exec -i usdotfhwastoldev/carma-dataspeed-controller-driver:develop` shell command to open a shell inside the `usdotfhwastoldev/carma-dataspeed-controller-driver:develop` Docker image. Then run the following shell commands.

```
. ./devel/setup.bash
rostopic pub /vehicle/dbw_enabled std_msgs/Bool '{data: True}'
```

Check Google Chrome and verify that **Robot Active** now reports **True**. Then terminate the `rostopic pub`, but do not exit the Docker image shell.

12. In the same Docker image shell, run the following command.

```
rostopic pub /ulc_cmd dataspeed_ulc_msgs/UlcCmd '{enable_pedals: True,
↪  enable_shifting: True, enable_steering: True, shift_from_park: True}'
```

This will tell the ULC to be ready to receive controlling commands. You will likely hear a siren from the car. Press on the brakes and then terminate the command. This will set the `override` flag of the ULC to 1, which we will clear in Step 14.

13. In Google Chrome, click on the big green button on the lower left. It should start glowing.

14. In the same Docker image shell as above, run the following command.

```
rostopic pub /ulc_cmd dataspeed_ulc_msgs/UlcCmd '{clear: True}'
```

Now the vehicle should start moving and the big glowing green button in Google Chrome should turn blue. The vehicle will drive towards designated destinations and stop after reaching the last destination.

15. Once you are finished, terminate Carma, RViz, and exit any shell environment created inside a Docker image. Then, run the following shell command.

```
carma stop all
```

11 Miscellanoues