Instituto Superior de Engenharia

Politécnico de Coimbra

DEPARTMENT OF SYSTEMS AND COMPUTER ENGINEERING

Simulating the effects of sensor failures on autonomous vehicles

Project Report to fulfill the Master's degree in Informatics Engineering

Specialization in Software Engineering

Author

Francisco dos Santos Matos

Supervisors

Professor João Carlos Costa Faria da Cunha

Professor João António Pereira Almeida Durães

INSTITUTO POLITÉCNICO DE COIMBRA

INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA

Coimbra, June, 2025

RESUMO

Os veículos autónomos (VA) estão a tornar-se cada vez mais prevalentes na nossa sociedade, impulsionados por rápidos avanços tecnológicos nos sensores e no software de perceção. Contudo, a fiabilidade e segurança dos VA dependem diretamente da capacidade dos seus sistemas em lidar com falhas nos sensores, que podem surgir devido a avarias ou ruído. Este trabalho avalia o impacto das falhas dos sensores no desempenho dos veículos autónomos através de uma metodologia de injeção de falhas aplicada no simulador CARLA, utilizando o Autoware como sistema de condução autónoma.

Inicialmente, foi realizado um *survey* para identificar os sensores críticos utilizados em veículos autónomos, as funcionalidades, limitações e falhas típicas, estabelecendo assim uma base para compreender os desafios de fiabilidade destes sensores. Posteriormente, implementou-se um mecanismo de injeção de falhas para introduzir sistematicamente avarias nos sensores, categorizadas em avaria silenciosa e ruído severo, em sensores como o LiDAR, IMU (giroscópio, acelerómetro, *quaternions*) e GNSS. Os resultados experimentais demonstraram degradações significativas no desempenho, particularmente com avarias no LiDAR e no giroscópio, levando a comportamentos de condução imprevisiveis, colisões e situações perigosas. No entanto, avarias em outros sensores foram mais toleradas devido à redundância interna e estratégias de fusão de sensores.

Estas contribuições apoiam a investigação em tolerância a falhas de veículos autónomos, demonstrando uma abordagem prática para a simulação e análise de falhas nos sensores.

Palavras-chave: Veículos Autónomos, CARLA, Autoware, Avarias de Sensores, Injeção de Falhas

ABSTRACT

Autonomous vehicles (AVs) are becoming increasingly prevalent in our society, driven by rapid technological advancements in sensors and perception software. However, the reliability and safety of AVs heavily depend on how well their systems handle sensor faults, which can arise from malfunctions, or noise. This work evaluates the impact of sensor faults on AV performance using a fault injection methodology applied within the CARLA simulator, using Autoware as the autonomous driving stack.

Initially, a comprehensive survey identified critical sensors utilized in AVs, their functional roles, common limitations, and typical failure modes, laying a foundation for understanding sensor reliability issues. Then, a fault injection mechanism was implemented to systematically introduce sensor faults, categorized into silent sensor failure, and severe noise in sensors such as LiDAR, IMU (gyroscope, accelerometer, quaternion), and GNSS. Experimental results demonstrated significant performance degradation, particularly with LiDAR and gyroscopic faults, which led to erratic driving behavior, collisions, and unsafe outcomes. However, other sensor failures were better tolerated due to built-in redundancy and sensor fusion strategies.

These contributions support ongoing research in autonomous vehicles safety by demonstrating an approach to simulating and observing sensor faults.

Keywords: Autonomous Vehicles, CARLA, Autoware, Sensor Failures, Fault Injection

AGRADECIMENTO

Gostaria de expressar o meu profundo agradecimento aos meus orientadores, o Professor João Carlos Costa Faria da Cunha, e o Professor João António Pereira Almeida Durães, pelo acompanhamento constante, pelas sugestões, e pelo incentivo ao longo de todo o desenvolvimento deste trabalho. O conhecimento e disponibilidade foram essenciais para a concretização deste projeto.

Agradeço igualmente à minha família pelo apoio incondicional ao longo destes anos, pela paciência nos momentos mais exigentes, e pela motivação que sempre me transmitiram.

Por fim, um agradecimento muito especial à minha companheira, pela compreensão, incentivo e carinho que foram fundamentais para alcançar esta etapa.

TABLE OF CONTENTS

Resumo ... i
Abstract ... iii
Agradecimento ... v
Table of contents ... vii
List of figures ... xi
List of Tables ... xiii
List of acronyms and abbreviations ... xv
1 Introduction ... 1
1.1 Vision and Motivation ... 1
1.2 Goals and Approach ... 2
1.3 Results and Contributions ... 2
1.4 Report Structure ... 3
2 Autonomous Driving Systems ... 5
2.1 Autonomous Vehicles ... 5
2.2 Automation Levels ... 6
2.3 Architecture of Autonomous Driving Systems ... 7
2.3.1 Perception ... 8
2.3.2 Planning and Decision ... 9
2.3.3 Motion and Vehicle Control ... 9
2.3.4 System Supervision and Coordination ... 10
2.3.5 Summary ... 10
3 Sensors used in Autonomous Driving Systems ... 13
3.1 Ultrasonic Sensors ... 14
3.2 RADAR: Radio Detection and Ranging ... 14
3.3 LiDAR: Light Detection and Ranging ... 15
3.4 Camera ... 16
3.5 GNSS ... 18
3.6 IMU ... 19
3.7 Summary of Problems and Weaknesses of Interceptive and Exteroceptive Sensors ... 19

3.8 Sensor Fusion 21
3.8.1 Sensor Fusion Methodologies 22
3.8.2 Sensor Fusion Techniques and Algorithms 22
4 Simulators and Autonomous Driving Systems 25
4.1 Simulators 25
4.1.1 Search criteria 25
4.1.2 CARLA 26
4.1.3 AirSim 27
4.1.4 LGSVL Simulator (Shutdown: January 2022) 28
4.1.5 DeepDrive 29
4.1.6 Summit 30
4.1.7 MetaDrive 31
4.1.8 Webots 32
4.1.9 Vista Simulator 33
4.1.10 Nvidia Drive Sim 33
4.1.11 Simulator Evaluation Summary 34
4.1.12 AiSim 35
4.2 Autonomous driving algorithms 39
4.2.1 Autoware 39
4.2.2 Apollo 40
4.2.3 OpenPilot 41
4.2.4 Algorithm choice 41
5 Fault Injection in Autonomous Vehicles 43
6 Framework Implementation 45
6.1 CARLA 45
6.1.1 Scenario Management and Ego Vehicle 47
6.1.2 CARLA – Autonomous Driving System Communication (ROS) 47
6.2 Autoware 48
6.2.1 Autoware Architecture 49
6.2.2 Sensing 50
6.2.3 Map 52
6.2.4 Localization 54
6.2.5 Perception 58

6.2.6 Planning ... 59
6.2.7 Control ... 61
6.2.8 Vehicle Interface ... 61
6.2.9 Autoware Challenges ... 61
6.3 Framework Integration ... 62
6.3.1 Framework Overview ... 62
6.3.2 Scenario Runner Modifications ... 63
6.3.3 Sensor Fault Injection System Implementation ... 63
6.3.4 Logging and Results Collection ... 66
7 Experimental Setup ... 67
7.1 Simulation Environment and Sensor Configuration ... 67
7.2 Fault Model ... 67
7.3 Severe Noise Values per Fault Location ... 68
7.3.1 LiDAR ... 68
7.3.2 GNSS ... 69
7.3.3 IMU ... 70
7.3.4 Summary of Noise Values per Location ... 70
7.4 Fault Trigger ... 71
7.4.1 Trigger 1: Starting Point ... 72
7.4.2 Trigger 2: Stop Sign ... 72
7.4.3 Trigger 3: Right Turn at Intersection ... 72
7.4.4 Trigger 4: Pedestrian Crosswalk Encounter ... 73
7.4.5 Trigger 5: Final Intersection ... 73
7.5 Failure Mode Classification ... 73
7.6 Experimental Procedure ... 74
7.7 Experimental Setup Validation Process ... 74
8 Experimental Execution and Results ... 77
8.1 Golden Runs ... 77
8.2 Experiment Execution and Analysis ... 78
9 Conclusions and future work ... 81
References ... 83
Annexes ... i
Annex A. Published Paper ... iii

Annex B. Experiment Results.. iv

LIST OF FIGURES

Figure 1 - SAE automation levels (from [2]).......... 6
Figure 2 - Functional architecture for an autonomous driving system (from [2]). ... 8
Figure 3 - Sensors in an autonomous vehicle 13
Figure 4 - Light behavior on different surfaces (from [2]). 16
Figure 5 - Simulated camera failures (from [49]). 17
Figure 6 - Number of collisions due to camera failures (from [49]). 18
Figure 7 - Simulators comparison chart.......... 35
Figure 8 - AiSim simulation worspace 36
Figure 9 - AiSim sensors workspace 37
Figure 10 - AiSim scenario workspace.......... 38
Figure 11 - AiSim environment workspace.......... 38
Figure 12 - ROS node example.......... 39
Figure 13 - Autoware system with CARLA 40
Figure 14 - High-level overview of the framework.......... 45
Figure 15 - CARLA-Client communication using its API 46
Figure 16 - CARLA ROS Bridge Diagram.......... 48
Figure 17 - Autoware Architecture (from [100]) 50
Figure 18 - Autoware - sensing component (from [100]).......... 51
Figure 19 - Autoware - Map Architecture (from [100]).......... 54
Figure 20 - Autoware - Localization architecture (from [100]).......... 56
Figure 21 - Autoware - TF example (from [100]).......... 57
Figure 22 - Autoware - Perception Architecture (from [100]).......... 58
Figure 23 - Autoware - Planning Architecture (from [100]) 60
Figure 24 - CARLA Autoware modified bridge (based of [94]). 63
Figure 25 - Fault injection class diagram 64
Figure 26 - Sensor Fault configuration example.......... 65
Figure 27 - Scenarios Route 72
Figure 28 - Fourth Trigger 73

LIST OF TABLES

Table 1 - Sensor's advantages, limitations, and weaknesses (based on [33]). 19
Table 2 - Sensor failures and their impact. 20
Table 3 - Comparing sensors' strengths in self-driving cars (based on [25]). 21
Table 4 - Sensor fusion algorithms and characteristics (based on [25]) 23
Table 5 - CARLA criteria score 27
Table 6 - Airsim criteria score 28
Table 7 - LGSVL criteria score 29
Table 8 - DeepDrive criteria score 29
Table 9 - Summit criteria score 31
Table 10 - MetaDrive criteria score 31
Table 11 - Webots criteria score 32
Table 12 - Summary of noise values used in fault model 71
Table 13 - Golden run results 77
Table 14 - Test run results 78
Table 15 - Additional gyroscope test results 80

LIST OF ACRONYMS AND ABBREVIATIONS

AD	Autonomous Driving
ADAS	Advanced Driving Autonomous System
ADS	Autonomous Driving System
AI	Artificial Intelligence
API	Application Programming Interface
AV	Autonomous Vehicle
AVFI	Autonomous Vehicle Fault Injection
CARLA	Car Learning to Act
CPU	Central Processing Unit
CVC	Computer Vision Center
DAG	Directed Acyclic Graph
DVS	Dynamic Vision Sensor
FMCW	Frequency-Modulated Continuous Wave
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HMI	Human-Machine Interface
IMU	Inertial Measurement Unit
INS	Inertial Navigation System
IP	Internet Protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LGSVL	LG Simulator for Autonomous Vehicles
MPU	Microprocessor Unit
NDT	Normal Distributions Transform
RADAR	Radio Detection and Ranging
RGB	Red Green Blue (color model)
ROS	Robot Operating System
SAE	Society of Automotive Engineers
SLAM	Simultaneous Localization and Mapping
TCP	Transmission Control Protocol
WCET	Worst-Case Execution Time
WGS	World Geodetic System
XML	eXtensible Markup Language

1 INTRODUCTION

This section introduces the vision and motivation behind the research conducted in this work. Initially, it outlines the growing importance and relevance of autonomous vehicle technologies, highlighting the need for reliable sensor systems and robust software to ensure safety. Then, the specific challenges and problems addressed in this research are presented, followed by the defined objectives and the main contributions achieved throughout the project. Lastly, an overview of the report structure is provided to guide the reader through subsequent chapters.

1.1 Vision and Motivation

Autonomous vehicles (AVs) are becoming more relevant, with ongoing advancements bringing the industry closer to fully autonomous systems. As their integration into public roads accelerates, ensuring their safe operation is a priority. AVs hold significant potential to improve road safety, mobility, and environmental sustainability [1]. However, realizing this potential depends heavily on the robustness of the sensor-based perception systems, which continuously monitor the environment and provide input to the vehicle's decision-making processes.

These sensors are susceptible to faults, which may lead to incorrect perception of the environment and result in unsafe behavior, posing risks to passengers, pedestrians, and other road users. Therefore, autonomous systems must be designed to operate safely not only under ideal conditions but also in the presence of sensor faults, becoming fail-safe systems.

To address these challenges, validation and fault-tolerance testing are required to ensure that AV systems can detect, handle, and mitigate sensor anomalies. Standards such as ISO 26262, which define functional safety requirements for automotive systems, underscore the importance of identifying and managing potential failure modes throughout the system lifecycle. This work investigates how autonomous driving systems respond to sensor faults using simulation-based methods. A survey was conducted [2] to identify the sensors commonly used in AVs, their failures, limitations, and existing mitigation strategies. Afterwards, a fault injection mechanism was developed to manipulate sensor data transmitted from the CARLA simulator to the Autoware stack, enabling the simulation of faults such as noise, and data dropout. While the experimental testing is limited in scope, it provides insight into how sensor failures can influence system behavior.

1.2 Goals and Approach

The primary goal of this project is to analyze the impacts of sensor failures on the safety of autonomous vehicles. To support this goal, an initial survey was conducted to gather background knowledge on autonomous vehicles and, more specifically, to examine the perception systems they rely on. The focus was on identifying the sensors commonly used in perception, analyzing their limitations, typical failure cases, and operational weaknesses. The survey also reviewed existing mitigation strategies employed to handle such failures, providing a foundation for designing fault-injection tests in realistic conditions.

Following the survey, it was necessary to create a controlled testing environment in which sensor faults could be safely introduced and analyzed. To achieve this, a simulation platform was required to replicate real-world driving scenarios, generate sensor data, and allow sensor fault injections without physical risk. An evaluation of available simulation tools was conducted, and CARLA was selected for its high configurability, support for detailed sensor modeling, and ability to define complex, multi-segment scenarios. In parallel, a autonomous driving software system was required to process perception data, plan trajectories, and control vehicle behavior, providing realistic responses to sensor disturbances. Autoware was chosen as an open-source ADAS/AV framework with modular support for perception, planning, and control, making it suitable for fault injection studies and consistent with real-world autonomous system architectures.

A multi-segment test scenario was then developed within CARLA, in which sensor faults could be introduced under controlled conditions. To support this, a fault injection mechanism was implemented within the communication bridge between CARLA and Autoware, enabling real-time modification of sensor data. A test model was also defined to specify the location, and type of each fault, including the relevant triggers and affected sensors. Finally, the system's responses to the injected faults were observed to assess its robustness and highlight the importance of fail-safe design.

1.3 Results and Contributions

Throughout the project, several results and contributions were achieved:

- A survey of sensors used in autonomous vehicles was carried out, analyzing their functionalities, limitations, common failures, and mitigation strategies. This work was published as a scientific article to a journal (Annex A), providing a reference for researchers investigating sensor reliability in AVs.
- A comprehensive evaluation of ADAS resulted in the choice of Autoware as the control and perception system to interact with the simulator.

- An analysis of existing simulation platforms was performed, leading to the selection of the CARLA Simulator as the most appropriate environment for conducting sensor fault injection tests.
- A fault injection mechanism was implemented to simulate sensor failures by manipulating the sensor data between CARLA and Autoware.
- Preliminary testing was conducted to observe system behavior in response to different fault types, offering initial insights into how sensor failures may impact AV performance and safety.
- Extensive testing was performed to observe system behavior in response to different fault types, offering insights into how LiDAR, GNSS, and IMU failures may impact AV performance and safety. The results of this work, together with the developed framework, were submitted to a journal as a second article and are currently under review.
- The developed framework has been structured to be reusable by third parties, available on GIT, serving as a foundation for future research on robustness testing of autonomous driving systems.

These contributions support ongoing research in autonomous vehicle safety by demonstrating an approach to simulating and observing sensor faults. While the testing was exploratory in nature, the tools and methodology established in this work can serve as a basis for more extensive validation and fault tolerance studies in future AV development efforts.

1.4 Report Structure

The remainder of this report is structured as follows:

- Section 2 presents Autonomous Driving Systems, including autonomous vehicles, SAE automation levels, and the overall software architecture with subsections on perception, localization, prediction, planning, control, and system supervision.
- Section 3 describes the sensors used in autonomous driving systems (ultrasonic, RADAR, LiDAR, camera, GNSS, IMU), summarizes typical issues and weaknesses, and introduces sensor fusion (methodologies, techniques, and algorithms).
- Section 4 reviews simulators and autonomous driving systems, outlining search/selection criteria and analyzing platforms such as CARLA, AirSim, LGSVL, DeepDrive, Summit, MetaDrive, Webots, Vista Simulator, Nvidia Drive Sim, and AiSim. It also reviews autonomous driving algorithms (Autoware, Apollo, OpenPilot) and the final algorithm choice.

- Section 5 introduces fault injection in autonomous vehicles, providing the background and concepts that motivate the use of fault injection for resilience assessment.
- Section 6 details the framework implementation, covering CARLA, Autoware, and the framework integration. It includes the scenario management and ROS communication, Autoware architecture and modules, and the integration pieces (framework overview, Scenario Runner modifications, sensor fault injection system, and logging/results collection).
- Section 7 describes the experimental setup, including the simulation environment and sensor configuration, the fault model, severe-noise definitions per sensor location, the fault triggers, failure-mode classification, and the experimental procedure.
- Section 8 reports the experimental execution and results, beginning with golden runs and followed by the execution and analysis of the main experiments.
- Section 9 concludes the report by summarizing the main findings, highlighting the contributions of this work, discussing limitations, and outlining directions for future research.

2 AUTONOMOUS DRIVING SYSTEMS

This section provides an overview of autonomous driving systems, including their fundamental principles, architecture, and the technologies that enable them. It begins by defining autonomous vehicles (AVs) and highlighting their growing role in modern transportation. As vehicle autonomy evolves, so do the technical challenges relate to perception, control, and decision-making.

The section describes the society of automotive engineers (SAE) levels of automation, which classify vehicles based on the degree of autonomy they offer, from complete human control to full automation. This classification provides a framework for understanding the capabilities and limitations of different AV systems.

This section also discusses the architecture of autonomous driving systems, breaking down the system into functional layers such as perception, planning, and control. Special emphasis is placed on the perception module, which plays a critical role in interpreting the environment and enabling safe and intelligent decision-making.

2.1 Autonomous Vehicles

Autonomous vehicles (AVs) are vehicles that perceive their surroundings and can move from point A to point B without human intervention. Through a combination of sensors, control and machine-learning algorithms, AVs can make real-time decisions. They are expected not only to reduce the number and severity of car accidents caused by human errors but also to reduce the carbon footprint through efficient driving [1]. However, despite these expected advantages, autonomous driving poses challenges such as a lack of governing legislation, a perceived increase in unemployment in the transportation sector, and cybersecurity threats [3], [4].

Notwithstanding the current disbelief and uncertainty about fully autonomous vehicles [5], [6], the number of vehicles using some degree of autonomous driving is increasing every year [7]. Many new vehicles now include driver assistance features that allow humans to perform multiple tasks daily, using sensors to perceive the vehicles' surroundings and assist the human drivers in actions such as blind spot detection, lane change assistant, rear cross-traffic alert, forward cross-traffic alert, and adaptive cruise control, among many other examples. These actions represent incremental steps toward full autonomy. Nevertheless, current sensor technology is not reliable in all environments in which vehicles may operate [8], [9]. Factors such as adverse weather conditions, complex urban settings, and sensor failures can significantly impair the performance of these systems. This unreliability poses challenges to the advancement of autonomous vehicles.

2.2 Automation Levels

According to the J3016 standard proposed by the Society of Automotive Engineers (SAE) [10], autonomous vehicles can be classified into 6 levels of increasing degrees of autonomy. Level 0 refers to the weakest automation, where the driver has full control of the vehicle; Level 5 is the highest, corresponding to full automation, where the vehicle controls all aspects of driving and requires no human intervention. There are other classification schemes (e.g., the German Federal Highway Research Institute (BASt) and NHTSA [11]), but the J3016 is the most used in the industry and academia. The levels are shown in Figure 1.

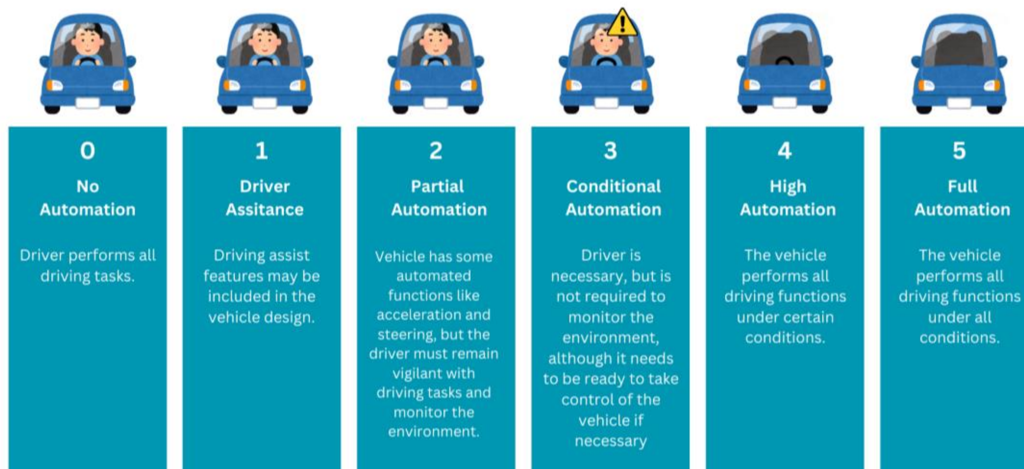

Figure 1 - SAE automation levels (from [2]).

Level 0 – No Automation

The driver performs all driving tasks, although the vehicle may provide simple warnings or emergency assistance without sustained automation.

Example: Typical older vehicles without advanced driver assistance features.

Level 1 – Driver Assistance

The vehicle includes single-function automation to assist with steering or acceleration/braking, but the driver remains fully engaged and responsible.

Examples: Adaptive cruise control or lane-keeping assistance systems.

Vehicles: 2018 Toyota Camry, 2019 Volkswagen Golf.

Level 2 – Partial Automation

The vehicle simultaneously automates steering and acceleration/braking under specific conditions (e.g., highways). However, the driver must continuously supervise and remain ready to take control immediately.

Examples: Tesla Autopilot, Mercedes-Benz Drive Pilot, Volvo Pilot Assist.

Vehicles: Tesla Model 3 (2020+), Volvo XC60 (2021+), Mercedes-Benz C-Class (2021+).

Level 3 – Conditional Automation

The vehicle handles most driving tasks and monitoring under certain conditions, allowing the driver to safely disengage attention from the driving task in specific scenarios (e.g., highway traffic jams). However, the driver must be available to take over when requested by the system.

Examples: Mercedes-Benz Drive Pilot (first certified Level 3 system available in select regions).

Vehicles: Mercedes-Benz S-Class (2022+) and EQS equipped with Drive Pilot in select markets.

Level 4 – High Automation

The vehicle operates autonomously without human intervention within limited operational design domains (e.g., specific geofenced urban areas, certain weather conditions). Outside these domains, human intervention may still be required.

Examples: Waymo One robotaxis, Cruise autonomous vehicles operating in San Francisco and Phoenix.

Vehicles: Waymo's autonomous Chrysler Pacifica and Jaguar I-PACE, Cruise's Chevy Bolt EV (modified for full autonomous use).

Level 5 – Full Automation

The vehicle operates autonomously under all roadway and environmental conditions without human intervention. No steering wheels or pedals are required, as the system is fully capable of handling all driving tasks. Currently no commercially available vehicles achieve full SAE Level 5 autonomy.

This classification serves as a guideline for a better understanding of what a given vehicle is capable of and its automation level.

2.3 Architecture of Autonomous Driving Systems

The Sense–Plan–Act (SPA) model has served as a conceptual foundation for autonomous systems, describing a sequential process of sensing, planning, and acting. While useful as a simplification, SPA is rarely used in its pure form for modern autonomous vehicles (AVs) because it struggles with real time uncertainty, continuous adaptation, and highly dynamic road scenes. SPA's reliance on static world models, absence of continuous feedback, and latency introduced by complete re-planning cycles make it less suited to real-time driving scenarios [12].

Modern AV software stacks adopt hybrid and reactive architectures, which blend fast feedback loops with higher-level deliberation. These designs often include multiple layers, reactive control, behavior sequencing, and planning, working in parallel and exchanging information continuously. Architectures such as Subsumption [13], Three-Tier (3T) [14], and Behavior Trees [15] are widely used in

robotics, with adaptations for AVs. Additional advancements integrate Model Predictive Control (MPC) [16] for short-term re-planning, and learning-based perception–action loops for adaptive decision-making.

A common structure in AVs consists of Perception, Planning and Decision, Motion and Vehicle Control, and System Supervision modules [17]. These components operate in a tightly integrated loop, with frequent bidirectional data exchange and feedback paths, allowing for continuous adaptation to changing traffic conditions, sensor updates, and safety events. Figure 2 illustrates this modern modular organization.

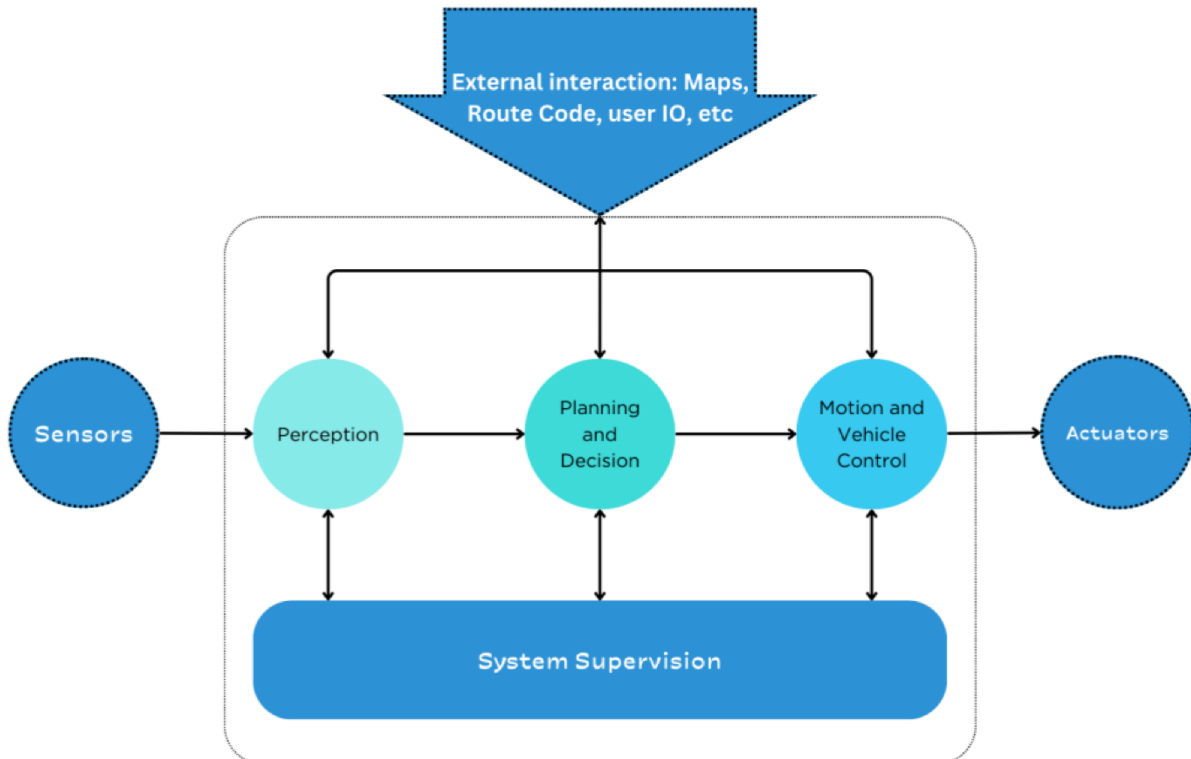

Figure 2 - Functional architecture for an autonomous driving system (from [2]).

2.3.1 Perception

Prediction is responsible for interpreting sensor data to build an understanding of the vehicle's surroundings. It receives data from sensors such as cameras, LiDARs, radars, ultrasonic sensors, IMUs, and GNSS, often combining these to improve resilience. Using this input, the perception software detects and classifies relevant environmental features: other vehicles, pedestrians, cyclists, road signs and traffic lights, lane markings, free space, obstacles, etc. It produces a structured world model or environmental representation, which may include the 3D positions and velocities of objects, their classifications (object types), traffic light states, and road geometry [18]. In short, perception "perceives" or senses the environment and answers the question: "What is around me and where?".

Within perception, localization determines the position and orientation (pose) of the vehicle in the world. While perception builds a map of what is around the vehicle, localization answers the question "Where am I?". The Localization module uses data from sensors such as GNSS for global position, inertial measurement units (IMU) for accelerations/rotational rates, and often uses features from perception (matching LiDAR scans or camera observations to a known map) to estimate the vehicle's location with high accuracy. Localization operates continuously in the background, compensating for GNSS/IMU drift over time by incorporating environmental cues [19], [20].

Perception also incorporates prediction, which predicts the future motion of dynamic objects in the environment. Given the current state of surrounding vehicles, pedestrians, or other moving agents (as perceived by the perception module), prediction algorithms estimate "What are they going to do next?". This typically involves computing short-term trajectory predictions for each relevant object, for example, predicting that a vehicle in front will continue straight at a certain speed, or that a pedestrian might begin crossing the street [21].

2.3.2 Planning and Decision

The Planning and Decision modules determine the vehicle's next actions based on the perceived environment, predicted object motion, and driving objectives. It processes data from perception, and maps to decide on a safe and efficient course of action. In essence, the planning module answers the question "What should I do now?" by generating a trajectory or path that leads the vehicle toward its destination. This plan is computed with the end goal in mind and is continuously updated to adapt to the current situation, ensuring safety and compliance with traffic rules along the way [22]. In this context, the plan refers to the high-level driving decision (for example, taking a lane change or stopping at an intersection), while the trajectory specifies the detailed spatiotemporal path the vehicle should follow, including position, speed, and heading over time. The trajectory is generated to respect safety margins, comfort, and traffic rules, and it is continuously updated to adapt to changes in the environment.

2.3.3 Motion and Vehicle Control

The Motion and Vehicle Control module takes the planned trajectory or target command from the planning and decision module and executes it by sending low-level control inputs to the vehicle's actuators (throttle, brake, and steering). Its role is to ensure the vehicle acts on the plan accurately and safely [23]. Control is commonly divided into lateral control (steering) and longitudinal control (acceleration and braking). Controllers rely on feedback from localization and onboard sensors to adjust commands in real time, closing the loop between planning and physical actuation [22], [23], [24].

2.3.4 System Supervision and Coordination

Overseeing all the above modules is the System Supervision (also referred to as the vehicle management or coordination layer). This module monitors the overall operation of the autonomous driving system to ensure it functions safely and reliably [18], [25]. The key responsibilities of system supervision are:

- Health Monitoring: The supervision module continuously checks the status of both hardware and software throughout the vehicle. If any module is not operating correctly, for example, a critical sensor drops out or the planner stops producing new trajectories, the supervision layer will detect this [18].
- Fault Management and Fail-Safe Mechanisms: Upon detecting an anomaly or failure, system supervision can initiate appropriate fail-safe mechanisms. This might mean alerting the driver or a remote operator or transitioning the vehicle into a safe state (such as gradually coming to a stop) if the autonomous system can no longer operate safely. This supervisory function is crucial for meeting functional safety requirements (ISO 26262 and similar standards) in a safety-critical system [18].
- Mission and Mode Management: It can manage transitions between autonomous driving and manual control, authorize engagement of self-driving when system checks pass, and abort or override the autonomy if necessary. It may also handle route initialization and high-level mission planning or interface with fleet management (in the context of robotaxis, for example), though these functions are sometimes considered separate. Additionally, system supervision can manage the Human-Machine Interface (HMI), for example, conveying system status to passengers or requesting driver takeover when needed [18].

This supervision module allows a complex AV software stack to maintain reliability, managing everything from fault diagnostics to system-level decision making.

2.3.5 Summary

The architecture of an autonomous driving system can be understood as a continuous feedback loop that cycles through sensing, planning, and acting many times per second. Sensors such as cameras, LiDAR, radar, GNSS, and IMUs feed data into perception and localization modules, which transform raw measurements into a structured representation of the environment and a precise estimate of the vehicle's position.

On this basis, prediction anticipates the short-term motion of surrounding agents, while planning and decision modules generate a safe and efficient trajectory toward the vehicle's goal. Motion and control modules then translate this trajectory into steering, throttle, and braking commands that guide the vehicle's movement.

System supervision oversees the entire loop, monitoring the health of modules, managing failures, and triggering fallback mechanisms if necessary. This ensures that the system not only executes the sense–plan–act cycle but also maintains safe and reliable operation under faults or unexpected events.

In short, modern AV software architecture integrates specialized modules into a tightly coupled, real-time control loop that enables vehicles to perceive their surroundings, decide on appropriate actions, and execute them safely in dynamic driving environments.

3 SENSORS USED IN AUTONOMOUS DRIVING SYSTEMS

Sensors can be categorized into internal state sensors (or proprioceptive sensors), and external state sensors (or exteroceptive sensors) [26]. Proprioceptive sensors are those used to measure the (internal) state of the vehicle. This category includes sensors such as global navigation satellite systems (GNSS), inertial measurement units (IMUs), inertial navigation systems (INS), and encoders, which are devices used to provide feedback on the position, speed, and rotation of moving parts within the vehicle such as the steering wheel, motor, brake, accelerator pedals, etc. These sensors are used to obtain information about the vehicle's position, motion, and odometry [27].

The autonomous vehicle (AV) can be positioned using relative or absolute methods. Relative positioning of an AV involves determining the vehicle's coordinates based on (relative to) its surrounding landmarks, while absolute positioning involves determining the vehicle's coordinates using a global reference frame (world)[25].

Exteroceptive sensors monitor the vehicle's surroundings to obtain data on the terrain, the environment, and external objects. These sensors include cameras, LiDARs (light detection and ranging), radars (radio detection and ranging), ultrasonic sensors.

In the typical AV use scenario, these sensors are used together to provide information about detection, lane occupancy, and more. Figure 3 shows an example of the type and location of sensors on a typical autonomous vehicle.

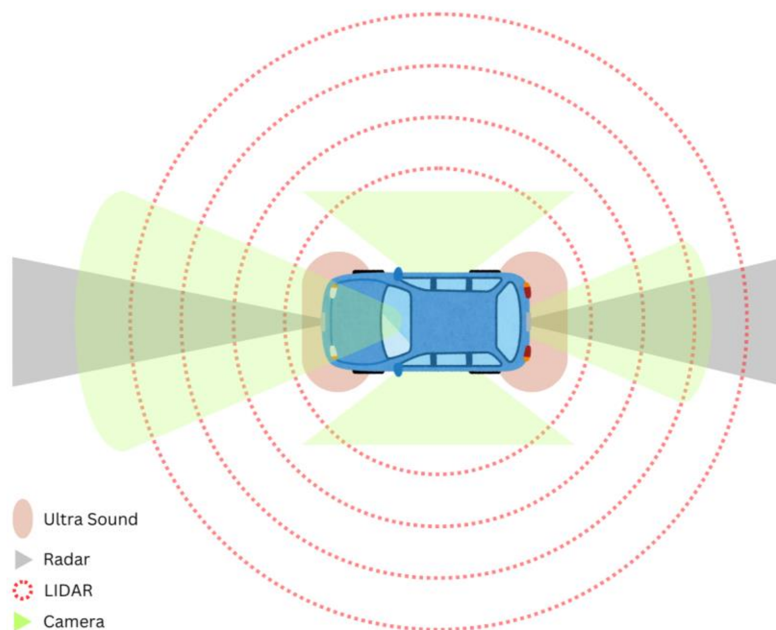

Figure 3 - Sensors in an autonomous vehicle

Sensors vary in technology and purpose, and each type of sensor has its weaknesses that are inherent in the technological capabilities that sensors use. To cover these weaknesses, some mitigation strategies are implemented, such as sensor fusion [25]. Sensor fusion is a crucial part of autonomous driving systems [27], [28] where input from multiple sensors is combined to reduce errors and overcome the limitations of individual sensors. Sensor fusion helps create a consistent and accurate representation of the environment in various harsh situations [29].

This section provides an overview of each sensor type, detailing its use and technical specifications. Next, a categorization of sensor failures and limitations/weaknesses is presented, such as radar interference and harsh environments, offering an overview of the various issues that can arise and their potential impact on AVs. This categorization is relevant to test teams, providing data that enhances the understanding of sensor issues in AVs to improve AV safety.

3.1 Ultrasonic Sensors

Ultrasonic sensors use sound waves to perceive distance and detect the presence of objects. Ultrasonic sensors use sound waves in the frequency range of 20 to 40 kHz [30], which falls outside the human hearing range. The distance is calculated by emitting a sound wave and measuring its time-of-flight (ToF) until an echo signal is received. These sensors are directional with a very narrow beam detection range [18]. They are most used as parking sensors [31] and perform well in adverse weather conditions and dusty situations [8], [32].

Given its narrow beam detection range, several ultrasonic sensors are needed to capture a full-field image. However, several sensors will interfere with each other and create interference, so a unique signature or identification code is required to reject echoes from other ultrasonic sensors in the vicinity. Ultrasonic sensors have a limited range, detecting obstacles up to 2 m [32], [33].

3.2 RADAR: Radio Detection and Ranging

Radar sensors use millimeter waves (mm waves) and work by sending electromagnetic waves and then receiving them after they bounce back, using the Doppler shift effect. Radars can measure not only the exact distance but also the relative speed [28]. They operate at frequencies of 24/77/79 GHz, but most of the newly developed sensors operate in the frequency band of 76–81 GHz, as the use of the 24 GHz frequency was prohibited by regulators due to lower bandwidth, accuracy, and resolution. They typically have a perception range of 5 m up to 200 m [33], [34], perform well in all weather conditions (rain, fog, and dark environments), and can accurately detect close-range targets on all sides of the vehicle [8], [32], [33], [34]. Radars are used in blind spot detection (BSD), lane change assistant (LCA), rear

cross traffic alert (RCTA), forward cross traffic alert (FCTA), adaptive cruise control, and radar video fusion.

One of the common problems of radars is that millimeter waves can give false positives because of possible bounced waves from the environment. Due to the increasing number of vehicles equipped with FMCW (frequency-modulated continuous-wave) radars, shared frequency interference is expected to become a problem [34], [35].

3.3 LiDAR: Light Detection and Ranging

LiDAR sensors work by emitting pulses of light (laser) and measuring the time it takes for the light to bounce off objects and return to the sensor. By scanning the reflected laser beams emitted in various directions, LiDAR sensors can produce highly accurate spatial data, creating point clouds and distance maps of the environment [35], [36]. LiDAR sensors are used to identify objects and pedestrians and avoid collisions. Due to their availability, 905 nm pulse LiDAR devices were used in the early AV systems. However, these 905 nm LiDAR systems have several important limitations, including high cost, inefficient mechanical scanning (in what concerns the movement necessary to direct the laser and sensor across its field of view), interference from other light sources, and eye-safety concerns leading to power restrictions that limited their detection range to approximately 100 m. This led to a shift to the retina-safe 1550 nm band, allowing higher pulse power with increased ranges of up to 300 m [35], [36].

The performance of LiDAR in optimal environmental conditions is much better than that of radar, but as soon as there is fog, snow, or rain, its performance suffers [32], [33], [36], [37]. In [37], [38], the authors analyze the effects of mirror-like objects in LiDAR, explaining that laser scans can be completely reflected by mirrors, resulting in no range or intensity data, leading to the creation of a faulty map. They also explain the different behaviors of light reflection on multiple surfaces (Figure 4). This shows that LiDAR performance is strongly influenced by the environment.

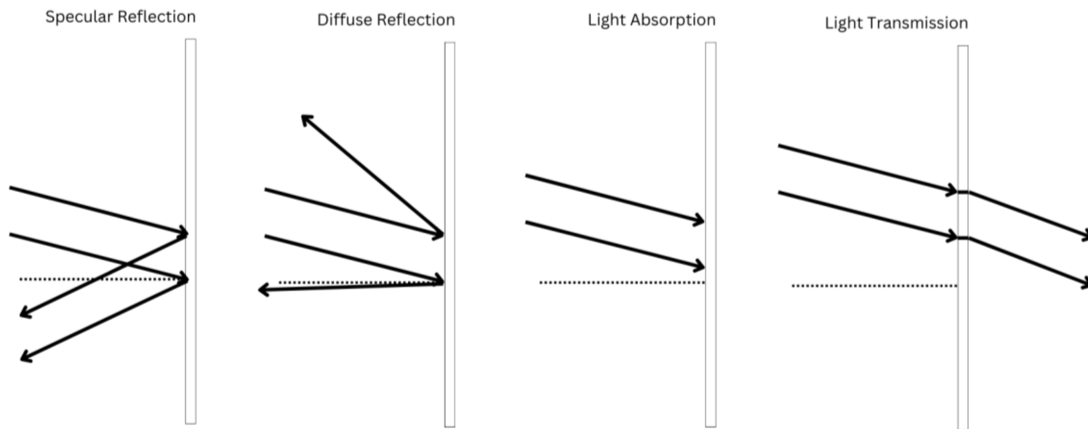

Figure 4 - Light behavior on different surfaces (from [2]).

3.4 Camera

Cameras can capture high-resolution details of objects up to 250 m away. Camer-as are categorized into types that use visible (VIS) or infrared (IR) light, range-gated technology, polarization technology, and event detection [38], [39]. They are based on one of two sensor technologies: charge-coupled device (CCD) or complementary met-al-oxide-semiconductor (CMOS). CCD image sensors are produced using an expensive manufacturing process delivering sensors with high quantification efficiency and low noise. CMOS sensor technology was developed to minimize the cost of CCD fabrication at the expense of performance [18], [25], [33], [34].

VIS cameras have a wide range of applications in AVs, including blind spot detection (BSD), lane change assistance (LCA), side view control, and accident recording. Deep learning algorithms can also be used with these cameras to detect and under-stand traffic signs and other objects [39], [40], [41], [42].

IR cameras are passive sensors that use light in infrared wavelengths (780 nm to 1 mm), resulting in less light interference. The common application for AVs is in scenarios with illumination peaks, and in the detection of hot objects such as pedestrians [42], [43], [44], animals [44], or other vehicles [45], [46].

Range-gated cameras are imaging systems that capture images based on the distance to the target by using a time-controlled gating mechanism. This gating mechanism synchronizes with a pulsed light source (such as a laser) to selectively allow light reflected from objects within a specific distance range to reach the camera sensor. By controlling the timing of the gate, the camera can effectively “slice” the scene at different distances, filtering out unwanted reflections and background noise to improve visibility in adverse conditions [46].

Polarization cameras are imaging devices that detect and measure the polarization state of light. Unlike conventional cameras that capture intensity and color,

polarization cameras provide additional information about the angle and degree of light polarization. This extra layer of information can be used to infer various properties of objects and scenes that are not visible in standard images, and one of its main advantages is that it can detect transparent objects [47].

Event cameras are a type of vision sensor that capture changes in a scene asynchronously, as opposed to traditional cameras that capture full frames at fixed intervals. Event cameras detect and record individual pixel-level changes in brightness, called "events," in real-time and with extremely low latency, and the main advantages are no motion blur and output over 1000 fps [48].

Cameras are strongly influenced by changes in lighting conditions, and by meteorological conditions such as rain, snow, and fog. Thus, cameras are typically paired with radar and/or LiDAR technologies to improve their resilience. In [49], the authors investigate the effects of degraded images on trained AI/ML agents, as seen in Figure 5. These camera failures were injected using the CARLA simulator [50], resulting in AV collisions, as shown in Figure 6.

Figure 5 - Simulated camera failures (from [49]).

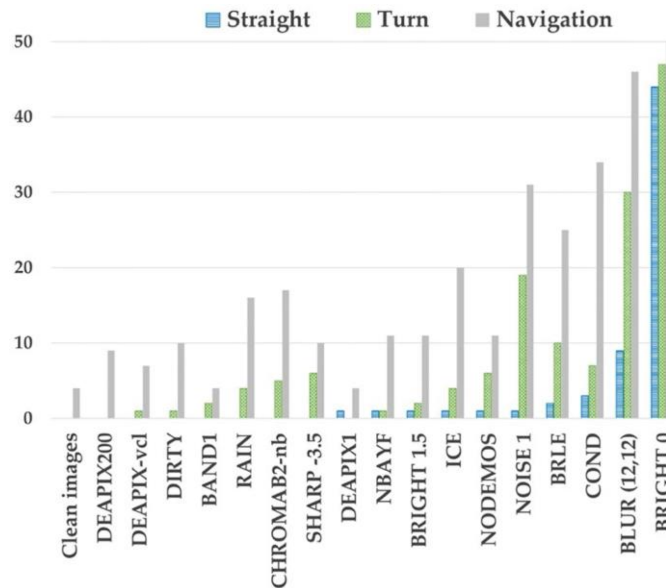

Figure 6 - Number of collisions due to camera failures (from [49]).

3.5 GNSS

The operating principle of the GNSS is based on the ability of the receiver to locate at least four satellites and then compute the relative distance to each of them. Since the location of the satellites is known, the receiver can extrapolate its global position using trilateration.

GNSS signals are prone to many errors that reduce the accuracy of the system, including the following:

- Clock bias and synchronization, due to residual satellite-clock error and receiver clock drift/jitter, clock/synchronization effects are present, but the receiver's absolute bias is estimated with (x,y,z), so stability, not absolute offset, governs positioning accuracy [51].
- Signal delays, caused by propagation through the ionosphere and troposphere.
- Multipath effect.
- Satellite orbit uncertainties.

Current vehicle positioning systems improve their accuracy by combining GNSS signals with data from other vehicle sensors (e.g., inertial measurement units (IMU), LiDARs, radars, and cameras) to produce trustworthy position information [52], [53], [54]. This mitigation strategy is called sensor fusion and is discussed in Section 3.8. GNSS is susceptible to jamming, such as when the receiver encounters interference from other radio transmission sources. GNSS receivers also suffer from spoofing, when fake GNSS signals are intentionally transmitted to feed false position information and divert the target from their intended trajectory.

3.6 IMU

Autonomous vehicles can detect slippage or lateral movement using inertial measurement units (IMU), which collect data from accelerometers, gyroscopes, and magnetometer sensors. Using this data, it is possible to detect the motion and orientation of the vehicle. The IMU, in conjunction with the other sensors, can correct errors and improve the sampling rate of the measurement system [33], [54]. This approach is called inertial guidance.

3.7 Summary of Problems and Weaknesses of Interceptive and Exteroceptive Sensors

In Table 1, it is presented a comparison between each sensor's advantages, as well as its limitations and weaknesses.

Table 1 - Sensor's advantages, limitations, and weaknesses (based on [33]).

Sensor	**Advantages**	**Limitations/Weaknesses**
Ultrasonic sensors	Affordable.	Limitations: The maximum range is 2 m. Very narrow beam detection range [25], [30]. Weaknesses: Interference.
Radar	Long range (up to 200 m). Performs well in various weather conditions.	Limitations: Lower resolution when compared to cameras and LiDARs. Weaknesses: Many false positives. Radar interference [34].
LiDAR	High resolution. Range up to 200 m. Accurate distance measurement.	Limitations: Expensive. Weaknesses: Suffers extremely from the weather [32], [36]. Reflective objects pose challenges [37].
Cameras	High resolution. Range up to 250 m. Capable of object recognition.	Limitations: Requires computational resources (Complex image processing). Weaknesses: Suffer from rough environmental conditions and several failures [49], [55].

Sensor	Advantages	Limitations/Weaknesses
GNSS	Provides global positioning.	Limitations: Limited accuracy in certain conditions, such as dense areas [56]. Dependency on satellite visibility [57]. Weaknesses: Latency [56]. Vulnerable to signal jamming and spoofing [58].
IMU	Measures acceleration and rotation. Provides orientation information.	Limitations: Drift over time without external reference [59]. Weaknesses: Requires frequent calibration to maintain accuracy.

Table 2 presents, for each sensor type, the impact of sensor failures on autonomous vehicles' performance and safety.

Table 2 - Sensor failures and their impact.

Sensor Type	Failure	Impact
Ultrasonic Sensors	Wrong perception due to interference between multiple sensors.	Extreme range errors due to overlapping ultrasonic signals. Requires unique identification to reject false echoes.
Radar	False positives due to bounced waves.	This can lead to incorrect object detection or classification due to reflected signals from the environment.
	Wrong perception due to frequency interference from multiple radars.	Shared frequency interference may cause inaccuracies in object detection and tracking.
LiDAR	Detection performance degradation due to adverse weather conditions.	Reduced effectiveness in fog, rain, or snow, leading to incomplete or inaccurate spatial data.
	Missing or wrong perception due to reflection from mirrors or highly reflective surfaces.	This can result in faulty maps or missing data due to the laser beams being completely reflected.
Camera	Poor object detection due to variability in lighting conditions.	Performance can be significantly impaired in varying light conditions, leading to poor object detection.
	Image degradation due to rain, snow, or fog.	This can result in blurred or obscured images, affecting the accuracy of perception tasks.
	Misinterpretation in ADAS due to degraded images.	Degraded images can lead to AV collisions if the AI/ML systems fail to properly interpret the information.

Sensor Type	Failure	Impact
GNSS	Timing errors due to clock differences.	This can affect the accuracy of location information, leading to incorrect positioning.
	Susceptibility to jamming and spoofing.	This can lead to loss of navigation accuracy or misdirection if the GNSS signals are blocked or falsified.
	Multipath effect and satellite orbit uncertainties.	This can lead to errors in location determination due to signal reflections and orbital inaccuracies.
IMU	Error accumulation and drift.	Errors in acceleration and rotational data can lead to inaccuracies in vehicle movement and orientation over time.

3.8 Sensor Fusion

Sensor fusion combines data originating from multiple types of sensors, taking advantage of their strengths while compensating for their weaknesses. For example, combining data from cameras and radar can provide high-resolution images and the relative speeds of obstacles in the area. Table 3 outlines the best sensor used in each factor [25].

Table 3 - Comparing sensors' strengths in self-driving cars (based on [25]).

Factors	Best Sensor
Range	Radar
Resolution	Camera
Distance Accuracy	LiDAR
Velocity	Radar
Color Perception (e.g., traffic lights)	Camera
Object Detection	LiDAR
Object Classification	Camera
Lane Detection	Camera
Obstacle Edge Detection	Camera and LiDAR

There is a considerable amount of research on multi-sensor fusion [18], [25], [28], [29], [41], [46], [52], [53], [54], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79]. The most common sensor combinations for obstacle detection are camera–LiDAR (CL), camera–radar (CR), and camera–LiDAR–radar (CLR). The CR sensor combination is the most used in multi-sensor fusion systems for environmental perception, followed by the CLR and CL [71], [73]. A best combination of sensors does not exist, as the definition of "best" depends on functionality and price. The CR sensor combination produces high-resolution images and provides distance and velocity data about the

surrounding obstacles [75], [80]. Similarly, the CLR sensor combination improves resolution over longer distances and provides accurate environmental information using LiDAR point clouds and depth map data [25].

3.8.1 Sensor Fusion Methodologies

Before fusing data, it is important to know which sensors to fuse and where to fuse, for instance, fuse only data corresponding to the front view of the vehicle or fuse everything around the car (bird's eye), and at which level the fusion should occur [76].

Multi-sensor data fusion (MSDF) frameworks have four levels: high-level fusion (HLF) also known as object-level; low-level fusion (LLF), also known as data-level fusion; mid-level fusion (MLF), also known as feature-level fusion; and hybrid-level fusion [70].

HLF (object-level fusion) approaches are often used due to their relatively low complexity when compared to LLF and MLF approaches. However, HLF provides insufficient information because classifications with lower confidence values are rejected, such as when there are multiple overlapping obstacles.

In contrast, the LLF (data-level fusion) technique integrates (or fuses) data from each sensor at the most basic level of abstraction (raw data). This preserves all information and can increase the accuracy of obstacle detection. This method requires precise external calibration, as it is heavily dependent on the good values of the sensors.

MLF (feature-level fusion) is an abstraction level between LLF and HLF, it combines multi-target features from the associated sensor data (raw measurements), such as color information from images or position features from radar and LiDAR, before performing recognition and classification on the merged multi-sensor features. However, MLF appears to be insufficient to achieve SAE automation levels 4 or 5 due to its limited sense of the environment and loss of contextual information [70].

Hybrid-level fusion can take the best of each level and merge the data. Although this gives good results, it greatly increases processing time [76].

3.8.2 Sensor Fusion Techniques and Algorithms

Although sensor fusion methodologies and algorithms have been extensively researched, a new study [63] suggests that it remains a difficult process due to the interdisciplinary variety of the suggested algorithms. Another study [65] classified these techniques and algorithms as classical sensor fusion algorithms and deep learning sensor fusion algorithms. Classical sensor fusion algorithms, such as knowledge-based methods, statistical methods, probabilistic methods, and so on, use theories of uncertainty from data imperfections, including inaccuracy and uncertainty, to fuse sensor data, whereas deep learning sensor fusion algorithms

generate various multi-layer neural networks that enable them to process raw data and extract features to perform challenging and intelligent tasks, such as object detection in an urban environment. In the field of AV, algorithms such as convolutional neural networks (CNN) and recurrent neural networks (RNN) are the most used perception systems, and other algorithms such as deep belief networks (DBN), and autoencoders (AE) [79] are also used.

Convolutional neural networks (CNNs) are specialized artificial neural networks designed to process and analyze visual data. They are particularly effective for tasks involving image recognition, classification, and computer vision.

Recurrent neural networks (RNNs) are designed to handle sequential data and temporal dependencies. They are commonly used for tasks involving time series data, natural language processing, and speech recognition.

Deep belief networks (DBNs) are a type of generative graphical model composed of multiple layers of stochastic, latent variables. They can learn to probabilistically re-construct their inputs by stacking restricted Boltzmann machines (RBMs).

Autoencoders (AEs) are neural networks designed to learn efficient encoding of input data by learning to encode the input into a compressed representation and then decode it back to the original input. They are used for data denoising and image retrieval.

In [25] the authors show some additional approaches using different algorithms, and Table 4 summarizes these techniques.

Table 4 - Sensor fusion algorithms and characteristics (based on [25]).

Sensor	Characteristics
YOLO	You Only Look Once (YOLO) is a single-stage detector that uses a single convolutional neural network (CNN) to predict bounding boxes and compute class probabilities and confidence scores for an image [69]. Advantages: • Real-time detection (single-pass detection). • Recognizes pedestrians and objects. Weaknesses: • Lower accuracy than SSD. The system struggles to recognize dense barriers, such as flocks of birds, due to its limited ability to propose more than two bounding boxes. It has poor detection of small objects.

Sensor	Characteristics
SSD	The Single-Shot Multibox Detector (SSD) is a single-stage CNN detector that converts bounding boxes into a collection of boxes with different sizes and aspect ratios to detect obstacles of various dimensions [81]. Advantages: • Real-time and accurate obstacle detection. • Single pass. • Detecting small objects can be challenging. However, it outperforms YOLO. Weaknesses: • Poor feature extraction in shallow layers. • Loses features in deep layers.
VoxelNet	VoxelNet is a generic 3D obstacle detection network that combines feature extraction and bounding box prediction into a single-stage, fully trainable deep network. It detects obstacles using a voxelized technique based on point cloud data [82]. Advantages: • No need for manual feature extraction. • Voxelization improves LIDAR data management by reducing sparsity. Weaknesses: • Training takes a large amount of data and memory.
PointNet	PointNet is a permutation-invariant deep neural network that learns global features from unordered point clouds using a two-stage detection approach [82]. Advantages: • Ability to maintain point clouds in any sequence, with permutation independence. Weaknesses: • Difficult to generalize unknown point configurations.

4 SIMULATORS AND AUTONOMOUS DRIVING SYSTEMS

Autonomous driving technology relies on simulation environments to validate and enhance safety and performance before real-world deployment. These environments replicate real-life urban scenarios, integrating entities like vehicles, pedestrians, and complex infrastructure elements. Central to these simulations is the Ego vehicle, which represents the autonomous vehicle under study, interacting dynamically within the simulated environment. The Ego vehicle's behavior is managed through decision-making algorithms, forming a closed-loop system where sensor data of the ego vehicle is sent to the autonomous driving algorithm, and subsequently the algorithm controls the ego vehicle. To do this, research on the current existing and used simulators, and open-source autonomous driving algorithms existing was made. This section describes these findings.

4.1 Simulators

This section describes the search criteria used to identify suitable simulators, gives a description of each one and then determines the simulator to be used in our case study based on a score system.

4.1.1 Search criteria

There are several simulation platforms available for testing ADAS systems. The selection of a simulator was guided by the following criteria:

- **Sensor Support**: Which sensors are supported?
- **Sensor Data Retrieval**: How can sensor data be accessed or exported during simulation runs?
- **Fault Injection Capabilities**: Is it feasible to simulate sensor failures, and if so, how?
- **Autonomous Driving Model Compatibility**: Which autonomous driving algorithms are compatible, and how do these algorithms integrate with sensor data within the simulator?
- **Documentation and Community Support:** How well documented the simulator is.
- **Scenario Creation Capabilities**: Does the simulator support scenario creation, and what is the method for creating them?
- **Ease of Use and Setup:** Is the simulator easy to install and use?
- **Performance and Efficiency:** Does the simulator require a lot of hardware resources?

To systematically evaluate and choose a simulator, a scoring system was established based on the defined criteria:

Criteria	**Points**
Sensor Support	10
Sensor Data Retrieval	10
Fault Injection Capabilities	10
Autonomous Driving Model Compatibility	10
Documentation and Community Support	10
Scenario Creation Capabilities	10
Ease of Use and Setup	10
Performance and Efficiency	10
Total	80

4.1.2 CARLA

CARLA is an open-source simulator designed for autonomous driving research and development [50].

Scenario Creation Capabilities: Supports scenario creation through RoadRunner, TrueVision Designer, and Scenario Runner, offering robust capabilities for designing both static and dynamic traffic scenarios.

Sensor Support: Provides LiDAR, semantic LiDAR, cameras (depth, segmentation, RGB, DVS), GNSS, radar, and IMU sensors.

Sensor Data Retrieval: Supports data logging in .log files and real-time data retrieval via ROS, allowing integration with autonomous stacks such as Autoware, Apollo, and OpenPilot.

Fault Injection Capabilities: Enables fault injection by manipulating sensor data within the ROS bridge connecting the simulator to external software.

Autonomous Driving Model Compatibility: Compatible with Autoware, Apollo, and OpenPilot through dedicated bridges.

Documentation and Community Support: Well-documented with extensive tutorials and an active community.

Ease of Use and Setup: Moderately complex setup due to Unreal Engine dependencies and required hardware resources.

Performance and Efficiency: GPU and CPU intensive, demanding significant computational resources for smooth performance.

The final score is in Table 5.

Table 5 - CARLA criteria score

Criteria	Score	Comments
Sensor Support	9	Good variety of sensors
Data Extraction	8	Flexible via log files and ROS
Fault Injection Capabilities	7	Limited built-in features, possible with addons
Autonomous Driving Model Compatibility	8	Compatible with Autoware, Apollo, OpenPilot (older version), among others
Documentation and Community Support	8	Highly active community
Scenario Creation Capabilities	8	Multiple scenario creation tools available
Ease of Use and Setup	4	Complex and resource-intensive installation
Performance and Efficiency	3	Demanding on GPU, CPU, and RAM
Total	55	

4.1.3 AirSim

AirSim was a simulator developed by Microsoft, originally designed for drone simulation, which later added support for ground vehicles [83].

Scenario Creation Capabilities: Scenarios can be created using Unreal Engine's built-in tools.

Sensor Support: Supports LiDAR, cameras, GPS, and IMU sensors. No official support for radar (workaround available).

Sensor Data Retrieval: Provides data via Python and C++ APIs; can integrate sensor streams with ROS.

Fault Injection Capabilities: Supports introducing sensor faults through JSON-based noise configuration.

Autonomous Driving Model Compatibility: Partial compatibility with Autoware; limited integration with other AV frameworks.

Documentation and Community Support: Reasonably comprehensive documentation but limited ongoing community activity.

Ease of Use and Setup: Moderately complex installation involving Unreal Engine dependencies and custom configurations.

Performance and Efficiency: Performs efficiently in low-complexity scenarios but resource-intensive in highly detailed simulations.

Unfortunately, AirSim was discontinued by Microsoft in July 2022, there are however some active projects based on AirSim.

Active projects based on AirSim:

- Colosseum: A current project using Unreal Engine 5.3.
- Cosys-Lab (Drone-focused simulator): Adds support for radar and sonar sensors. (Project page)

Table 6 presents a summary of the criteria and its score for AirSim.

Table 6 - Airsim criteria score

Criteria	Score	Comments
Sensor Support	7	Does not officially support Radar
Data Extraction	8	Good API-based data retrieval
Fault Injection Capabilities	8	Flexible fault and noise configuration
Autonomous Driving Model Compatibility	6	Limited support (e.g., Autoware)
Documentation and Community Support	6	Moderate activity and documentation
Scenario Creation Capabilities	5	Basic, relies on external tools (Unreal)
Ease of Use and Setup	3	Difficult setup process
Performance and Efficiency	5	Moderate resource requirements
Total	48	

4.1.4 LGSVL Simulator (Shutdown: January 2022)

The LGSVL Simulator, developed by LG, was officially discontinued in January 2022. Due to the shutdown of essential backend servers, the simulator is no longer functional. The official website has been taken down, and only the GitHub repository remains available [84].

Scenario Creation Capabilities: Built-in scenario editor featuring graphical interfaces and scripting tools.

Sensor Support: Supports LiDAR, cameras, radar, GNSS, IMU, and ultrasonic sensors.

Sensor Data Retrieval: Provides real-time data access through ROS/ROS2 integration.

Fault Injection Capabilities: Limited built-in support; possible via ROS-based data manipulation.

Autonomous Driving Model Compatibility: Strong integration with Autoware and Apollo frameworks.

Documentation and Community Support: Documentation is basic; community support has significantly declined since discontinuation.

Ease of Use and Setup: Complex setup process requiring Unity engine and ROS configurations.

Performance and Efficiency: Resource-intensive, demanding high-performance hardware.

Table 7 presents a summary of the criteria and its score for LGSVL.

Table 7 - LGSVL criteria score

Criteria	**Score**	**Comments**
Sensor Support	9	Broad support including Radar
Data Extraction	7	Via ROS and custom APIs
Fault Injection Capabilities	6	Limited out-of-the-box, extensible with effort
Autonomous Driving Model Compatibility	8	Supported Autoware and Apollo
Documentation and Community Support	4	Documentation remains, but the community is inactive
Scenario Creation Capabilities	7	Included a scenario editor
Ease of Use and Setup	3	Complex setup, now non-functional
Performance and Efficiency	4	Moderate performance, now outdated
Total	48	

4.1.5 DeepDrive

DeepDrive is a simulation platform focused on training and testing autonomous driving agents using vision-based input. The project appears to be inactive, with the last commit made over four years ago. It was not designed as a full-stack simulator for ADAS testing, but rather as a lightweight tool for reinforcement learning and perception research [85].

Scenario Creation Capabilities: Does not support scenario creation; provides fixed training environments.

Sensor Support: Only camera sensors supported.

Sensor Data Retrieval: Provides image-based data through a dedicated API.

Fault Injection Capabilities: No support for fault injection.

Autonomous Driving Model Compatibility: Limited compatibility, mainly vision-based agent training.

Documentation and Community Support: Basic documentation, minimal community support, and discontinued development.

Ease of Use and Setup: Straightforward and lightweight setup.

Performance and Efficiency: Highly efficient, optimized for image-based training.

Table 8 presents a summary of the criteria and its score for DeepDrive.

Table 8 - DeepDrive criteria score

Criteria	**Score**	**Comments**
Sensor Support	2	Only supports camera data
Data Extraction	3	Limited to image streams
Fault Injection Capabilities	1	No fault injection support
Autonomous Driving Model Compatibility	2	Not compatible with full stack driving systems

Documentation and Community Support	4	Some documentation is available, but the project is inactive
Scenario Creation Capabilities	1	No scenario creation functionality
Ease of Use and Setup	6	Lightweight and easy to run, but limited functionality
Performance and Efficiency	6	Efficient for image-based training tasks
Total	25	

4.1.6 Summit

SUMMIT is an open-source simulator designed to generate high-fidelity, interactive data for dense, unregulated urban traffic environments on complex real-world maps. It is built as an extension of the CARLA simulator, enhancing its capabilities with advanced procedural traffic generation and large-scale crowd simulation [86].

Key Features:

- **Procedural Simulation**: Scenarios are generated procedurally without the need to recompile the simulator for each new map. Map elements (roads, sidewalks, landmarks, satellite imagery) are dynamically sent from the client and instantiated on the server.
- **Crowd Simulation**: A client-side Python script enables the simulation of dense, interactive traffic. This includes both vehicles and pedestrians interacting in complex urban settings.
- **Advanced Behavior Modeling**: SUMMIT integrates GAMMA, a cutting-edge traffic motion prediction model, to simulate sophisticated agent behavior and realistic crowd dynamics.

Scenario Creation Capabilities: Supports advanced procedural scenario generation and dynamic crowd simulation.

Sensor Support: Inherits CARLA's full sensor suite, including LiDAR, cameras, radar, GNSS, and IMU.

Sensor Data Retrieval: Accessible via ROS integration, similar to CARLA.

Fault Injection Capabilities: Fault injection indirectly supported via ROS-based manipulation methods.

Autonomous Driving Model Compatibility: Likely compatible with Autoware/Apollo but requires further validation.

Documentation and Community Support: Limited documentation available, primarily research-oriented.

Ease of Use and Setup: Moderately complex setup relying on CARLA installation plus custom scripts.

Performance and Efficiency: Optimized for performance in large-scale urban simulations.

Table 9 presents a summary of the criteria and its score for Summit.

Table 9 - Summit criteria score

Criteria	**Score**	**Comments**
Sensor Support	9	Inherits rich sensor set from CARLA
Data Extraction	8	Based on CARLA's well-established data pipelines
Fault Injection Capabilities	6	Not directly documented, but likely possible through CARLA-based methods
Autonomous Driving Model Compatibility	5	Requires validation for integration with Autoware or Apollo
Documentation and Community Support	5	Limited documentation, niche research community
Scenario Creation Capabilities	9	Advanced procedural scenario generation
Ease of Use and Setup	5	Depending on CARLA setup, it may involve additional complexity
Performance and Efficiency	6	Potentially more demanding due to dense traffic and procedural logic
Total	53	

4.1.7 MetaDrive

MetaDrive is a high-performance, open-source driving simulator focused on procedural scenario generation. Its primary aim is to support reinforcement learning and autonomous driving research by providing diverse and scalable environments [87].

Scenario Creation Capabilities: Provides procedural scenario generation with diverse, configurable driving environments.

Sensor Support: Supports LiDAR and camera sensors.

Sensor Data Retrieval: Sensor data primarily accessed through log-based exports.

Fault Injection Capabilities: Fault injection via post-processing; real-time faults are not directly supported.

Autonomous Driving Model Compatibility: Compatible primarily with OpenPilot; limited support for other AV stacks.

Documentation and Community Support: Moderate documentation, actively used in academic circles.

Ease of Use and Setup: Lightweight Python-based installation and easy setup.

Performance and Efficiency: Highly efficient and performant on standard hardware.

Table 10 presents a summary of the criteria and its score for MetaDrive.

Table 10 - MetaDrive criteria score

Criteria	**Score**	**Comments**

Sensor Support	5	Limited to LiDAR and camera
Data Extraction	5	Based on log files
Fault Injection Capabilities	4	Only via post-simulation log modification
Autonomous Driving Model Compatibility	5	Confirmed support for OpenPilot
Documentation and Community Support	6	Growing community and active development
Scenario Creation Capabilities	8	Strong procedural generation tools
Ease of Use and Setup	6	Easy to install and run
Performance and Efficiency	7	Lightweight and efficient
Total	46	

4.1.8 Webots

Originally developed for general-purpose robotics simulation, Webots introduced support for autonomous vehicles in 2023. It is a flexible simulator with strong capabilities for robot control and environment modeling, but limited focus on autonomous driving research [88], [89].

Scenario Creation Capabilities: Supports map importation using OpenStreetMap (OSM), but limited advanced scenario capabilities.

Sensor Support: Offers basic sensor types such as cameras, GPS, gyro, LiDAR, position, and rotational sensors.

Sensor Data Retrieval: Provides sensor data access through dedicated API getter functions.

Fault Injection Capabilities: No built-in fault injection due to API limitations.

Autonomous Driving Model Compatibility: Limited compatibility with full AV stacks.

Documentation and Community Support: Extensive general robotics documentation, limited AV-specific guidance.

Ease of Use and Setup: Beginner-friendly and straightforward installation.

Performance and Efficiency: Moderate performance, suitable for educational and simple robotic simulations.

Table 11 presents a summary of the criteria and its score for Webots.

Table 11 - Webots criteria score

Criteria	**Score**	**Comments**
Sensor Support	5	Basic set of sensors provided
Data Extraction	5	Accessible via API (getters only)
Fault Injection Capabilities	0	No fault injection tools or documented API support
Autonomous Driving Model Compatibility	0	No known support or compatibility with AV stacks

Documentation and Community Support	3	Very limited for vehicle simulation
Scenario Creation Capabilities	4	OSM import available but limited high-level control
Ease of Use and Setup	3	Setup is basic but not tailored to AV use cases
Performance and Efficiency	4	Lightweight, but not designed for high-fidelity AV simulation
Total	24	

4.1.9 Vista Simulator

VISTA is a data-driven simulation engine developed by MIT CSAIL. Unlike traditional simulators that generate synthetic environments, VISTA reconstructs simulation scenes directly from real-world data, allowing for the generation of new viewpoints and trajectories within those captured environments. Its goal is to improve realism and eliminate the sim-to-real gap by using photorealistic, real-world datasets [90].

Scenario Creation Capabilities: No direct scenario creation; operates based on recorded real-world datasets.

Sensor Support: Supports camera and LiDAR sensors based on recorded data.

Sensor Data Retrieval: Static dataset-based approach; no real-time data retrieval.

Fault Injection Capabilities: No real-time fault injection due to fixed dataset architecture.

Autonomous Driving Model Compatibility: Limited to policy training and evaluation; minimal integration with AV stacks.

Documentation and Community Support: Limited documentation, mostly research oriented.

Ease of Use and Setup: Dataset-dependent setup; targeted toward academic research.

Performance and Efficiency: Efficient in operation, given its data-driven, fixed-scenario nature.

Conclusion:

Due to its narrow scope and data-driven design, VISTA is not suitable for the goals of this project, which require scenario creation, sensor fault injection, and real-time interaction capabilities.

4.1.10 Nvidia Drive Sim

NVIDIA Drive Sim, part of the NVIDIA Omniverse platform, is a physics-based simulation environment built specifically for autonomous vehicle development. It offers real-time sensor simulation and virtual scenario testing. However, the

platform is currently in closed early access, limiting its availability for general research or academic use.

Scenario Creation: NVIDIA provides custom scenario creation tools integrated into the Omniverse platform, enabling detailed and realistic scene generation.

Sensor Support: Supports a wide range of sensors including radar, LiDAR, and multiple camera types.

Sensor Data Extraction: Not publicly documented — data extraction methods are not available for evaluation.

Fault Injection Capability: Not documented — there is no public information on whether Drive Sim supports real-time or scripted fault injection.

Autonomous Driving Model Compatibility: Drive Sim is designed to integrate with NVIDIA's DRIVE AGX platform and ecosystem, with limited details on third-party AV stack compatibility.

Conclusion:

Due to its closed access, lack of public technical documentation, and limited transparency on sensor data handling and fault injection, NVIDIA Drive Sim is not considered for our spider chart.

4.1.11 Simulator Evaluation Summary

While all simulators offer valuable features, not all are equally suited for research involving fault injection and AV stack integration. Some platforms were included in the chart despite limitations, in order to provide a broader perspective:

- AirSim and DeepDrive were included for comparison purposes, even though both are no longer actively maintained. Their limitations in terms of sensor support and autonomous driving stack integration significantly reduce their suitability for fault injection testing.
- MetaDrive offers procedural scenario generation and efficient performance but lacks support for major AV frameworks like Autoware or Apollo.
- Webots, although versatile for robotics, lacks mature support for autonomous vehicle stacks and documentation related to advanced driving scenarios.

Among the compared simulators, CARLA stood out due to its comprehensive feature set, active development community, and native support for AV frameworks, making it the most suitable platform for conducting the fault injection experiments in this work, as it is also possible to see in Figure 7.

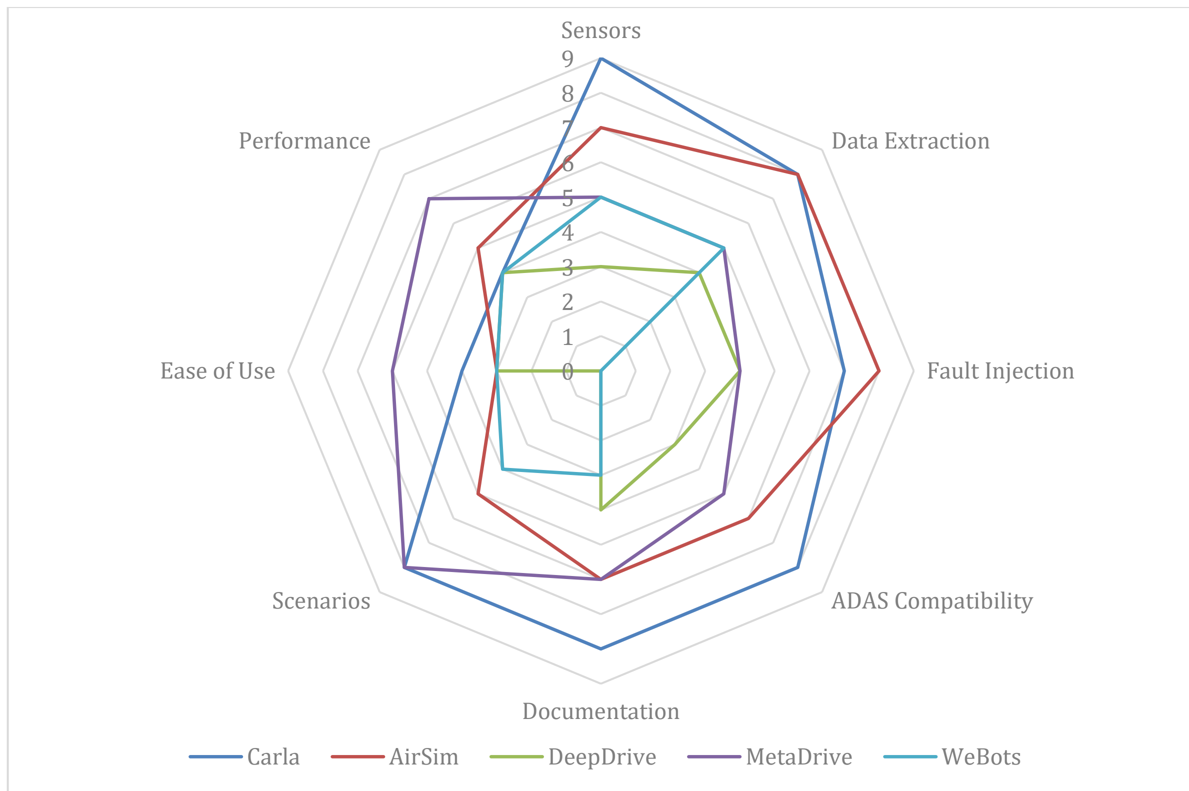

Figure 7 - Simulators comparison chart

Based on the score, community support, maturity, and autonomous driving algorithms CARLA was chosen as the simulator for this project.

4.1.12 AiSim

AiSim was discovered during the CARLA scenario tools creation research, although it is not publicly available, contact was successfully established with aiMotive, a company owned by Stellantis. Following a series of meetings, an academic license was granted, allowing access to the AiSim platform. However, due to limited technical support, time constraints, and the unfulfilled expectation that Autoware would work seamlessly with AiSim, the decision was made to transition to CARLA as the primary simulation platform.

AiSim is a simulator that was found after the simulator's comparison and easily became the primary choice. AiSim allows to test and validate AD systems performance in multiple and adverse driving scenarios, focusing on virtual, high-fidelity sensor data. The major features of aiSim are:

- **Environment & weather** – Simulate challenging weather conditions, including snowstorms, heavy rain, fog, and sunshine, it is also possible to configure the road properties such as painting, degradation and deterioration.
- **Sensors support** – it supports multiple sensors like LiDAR, RADAR, IMU, Camera (Stereo, ...)

- **Scenario Creation** – aiSim supports OPEN DRIVE and OPEN SCENARIO standards, also they have their own scenario creation tool implemented in their software.
- **Maps** – Although it already has a suitable number of maps it also supports the import of map formats like osm, openDRIVE and RoadRunner.
- **Toolchains** – aiMotive provides an SDK and code examples of some toolchains. Toolchains allows us to extend the functionality of aiSim.

AiSim GUI workspaces

AiSim GUI is divided into multiple workspaces.

The different tools of aiSim (Simulator, Sensor editor, Environment editor, large scale testing, etc.) are arranged into specific workspaces, in which all the functions, settings, and parameters are grouped into panels, aligned on a responsive layout grid.

Here are the workspaces, that were possible to access with an academic license.

Simulation

The aiSim GUI application's Simulation workspace lets us run simulations by selecting maps, scenarios, sensor configurations, and an environment preset (Sunny, Rain, Snow, Foggy, etc.). The camera viewports enable us to see the simulation running. As it is possible to observe in Figure 8.

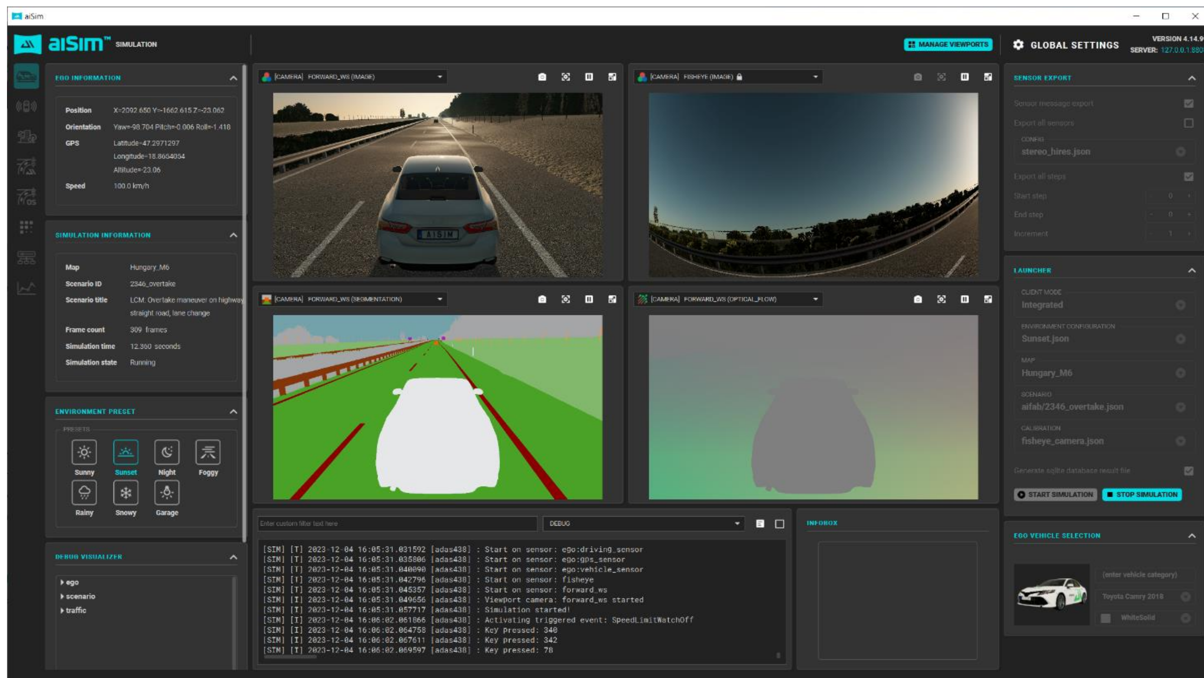

Figure 8 - AiSim simulation worspace

Sensors

AiSim allows users to create multiple sensor configurations and use them in different scenarios. Currently they advertise the support of 20+ cameras, 10+ radars, 10+ lidars in the same environment, by using multiple GPUs. The configuration was very intuitive and easy as it is possible to see in Figure 9.

Sensors supported by aiSim:

- **CPU-processed sensors**: Clock sensor; Fusion sensor; GPS sensor; IMU sensor; Occupancy grid sensor; Parking slot sensor; Scenario state sensor; Vehicle sensor; Ultrasonic sensor
- **Raytrace-capable sensors**: LiDAR sensor; Radar sensor; Camera sensors.

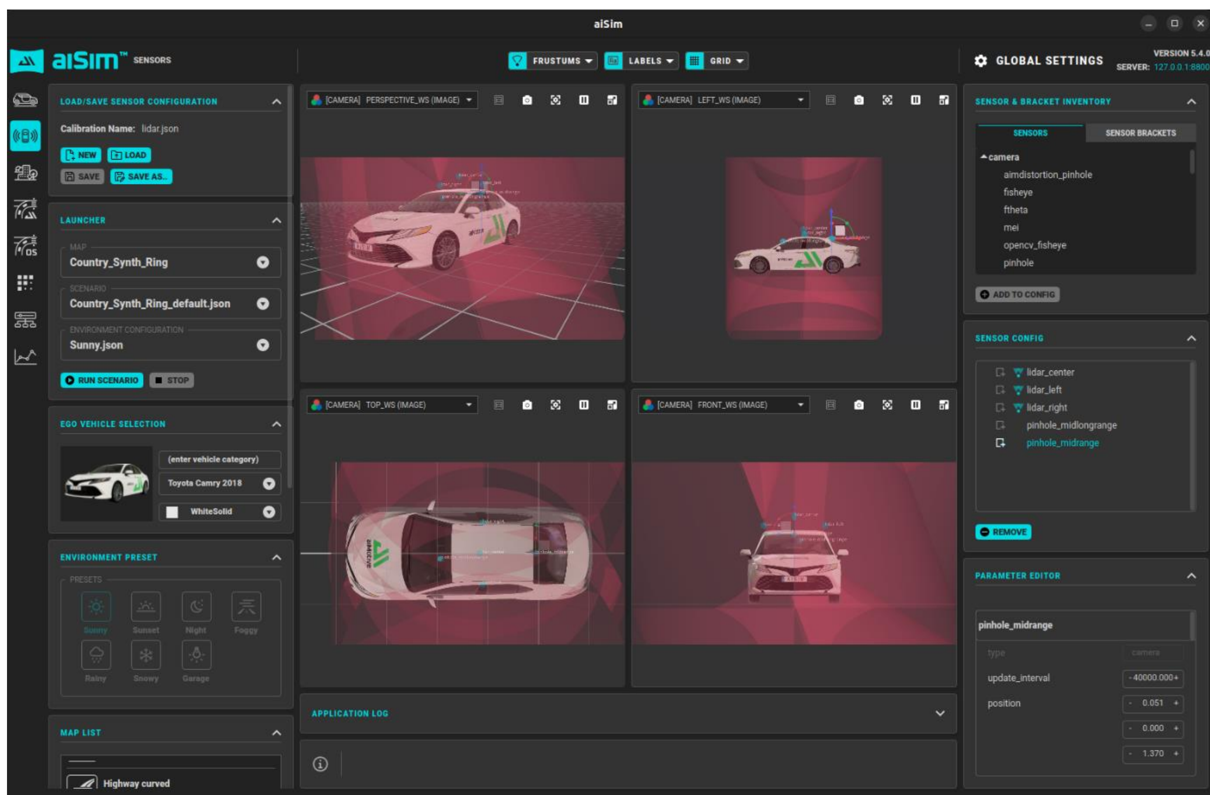

Figure 9 - AiSim sensors workspace

Scenarios

A Scenario is composed by a map, events, entities (ego vehicle, pedestrians, other vehicles), entities trajectory, and the Pass/Fail criteria (Figure 10).

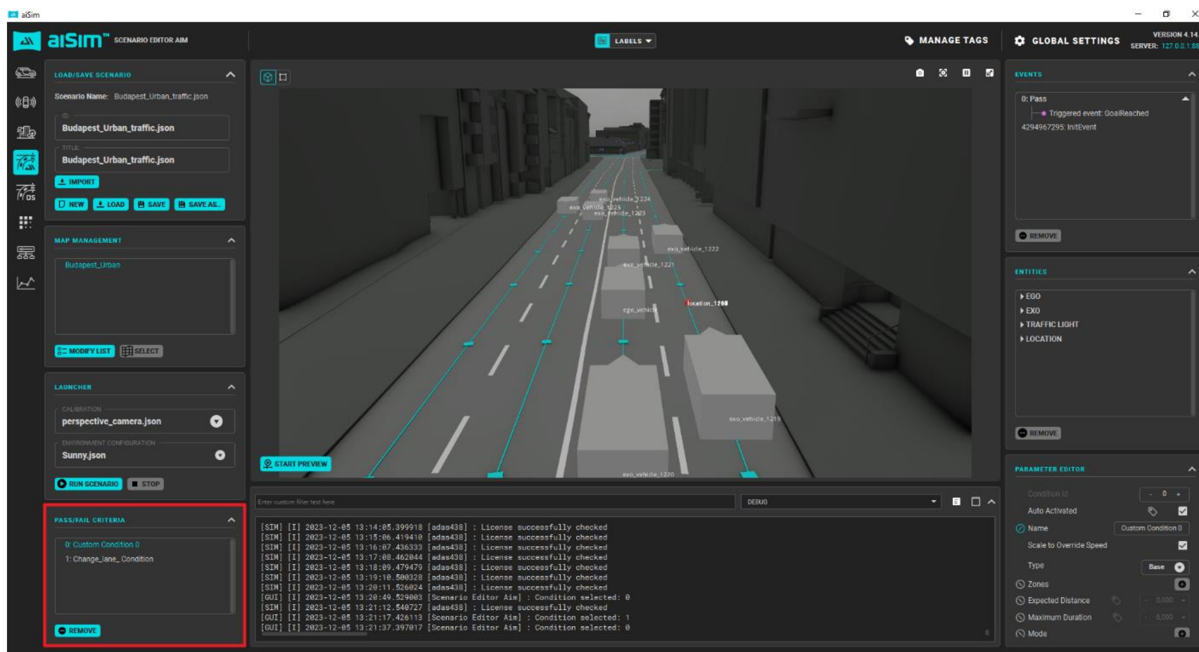

Figure 10 - AiSim scenario workspace

Environment

The environment, as it can be see in Figure 11, is where it is possible to define the weather and road conditions, such as roadmark quality (water, holes, degradation, paint degradation, snow).

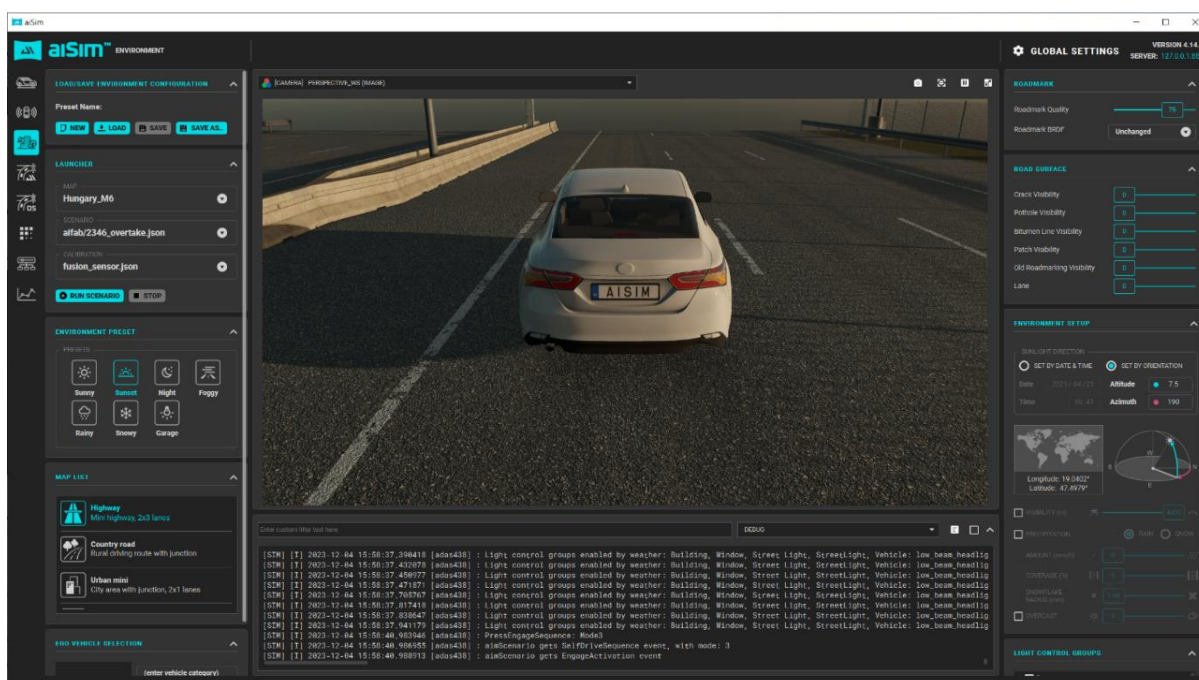

Figure 11 - AiSim environment workspace

Toolchains

Toolchains are source code systems that can add clients and plugins to aiSim.

AiSim already has a toolchain developed that provides its own ROS integration to be able to send sensor data to ROS Topics.

ROS Toolchain

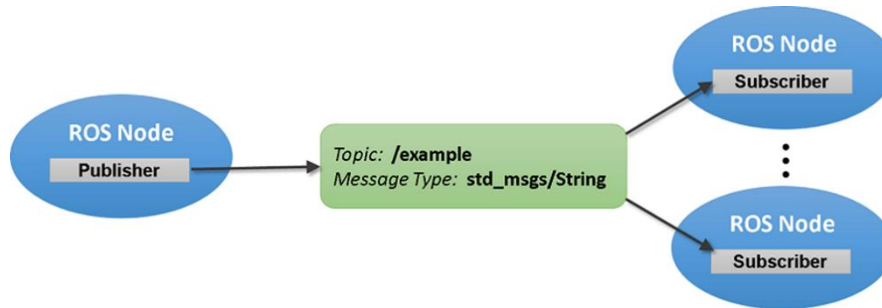

Figure 12 - ROS node example

Node is the ROS term for an executable that is connected to the ROS network. ROS uses a simplified messages description language for describing the data values (that is, messages) that ROS nodes publish. Topics are named buses over which nodes exchange messages. Topics are intended for unidirectional, streaming communication, this is represented in Figure 12.

ROS toolchain is loaded by aiSim then the ROS Bridge creates the aiSim node that publishes the given topics, the sensors publish data to their topics, and the driving sensor subscribes to driving commands

4.2 Autonomous driving algorithms

When it comes to open-source autonomous driving control algorithms, the main available options are Autoware, Apollo, and OpenPilot. While the selection is limited, all three frameworks are used in real-world applications. Autoware is primarily used in Japan, Apollo is used in China, and OpenPilot is available for consumer use in various compatible vehicles worldwide.

4.2.1 Autoware

Autoware is an open-source autonomous driving software stack built on the Robot Operating System (ROS). Originally developed in Japan by Tier IV and the Autoware Foundation, it is designed for urban autonomous vehicles, offering modules for perception, localization, planning, and control. Autoware is particularly suited for academic research and real-world deployments and is actively used in government-funded pilot programs and smart city projects in Japan and Europe [91]. It supports both ROS 1 and ROS 2, with the newer Autoware Universe and

Autoware.auto variants focusing on modularity and safety. Figure 13 shows Autoware running with CARLA.

Supported Sensors:

LiDAR, cameras (mono/stereo), GPS, IMU, Radar, and ultrasonic sensors

Compatible Simulators:

- CARLA
- AirSim
- LGSVL (Discontinued)

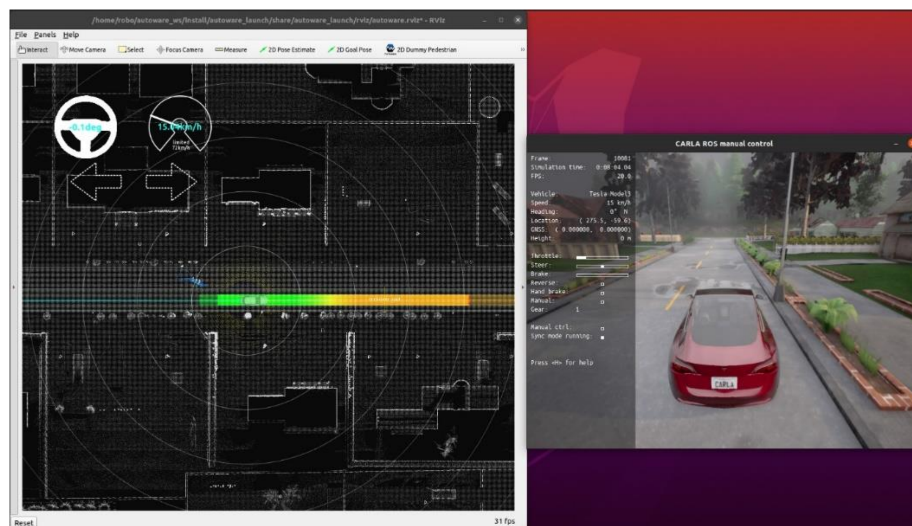

Figure 13 - Autoware system with CARLA

4.2.2 Apollo

Apollo, developed by Baidu, is a comprehensive open-source platform for autonomous driving, aimed at industrial and commercial use cases. It includes a full stack of tools for perception, prediction, planning, and control, and offers robust support for high-definition maps and vehicle interface protocols. Apollo is used in several autonomous vehicle fleets in China, including robotaxi and autonomous bus deployments. Although it is more complex to set up than other platforms, it is highly scalable and feature rich [92].

Supported Sensors:

LiDAR, cameras (mono/stereo), GPS, IMU, Radar, and ultrasonic sensors

Compatible Simulators:

- CARLA
- AirSim

- LGSVL (Discontinued)
- Dreamview Simulation Mode (built-in Apollo simulator)

4.2.3 OpenPilot

OpenPilot, developed by Comma.ai, is an open-source ADAS (Advanced Driver Assistance System) designed primarily for highway driving. Unlike Autoware or Apollo, OpenPilot is not a full autonomous driving stack but rather an aftermarket autopilot that can be installed on compatible vehicles. It uses inputs from cameras, GPS, IMU, and radar to perform lane-keeping, adaptive cruise control, and forward collision warnings. OpenPilot is popular due to its ease of installation and real-world deployment in hundreds of vehicles [93].

Supported Sensors:

Cameras (front-facing), GPS, IMU, and Radar.

Compatible Simulators:

- MetaDrive
- CARLA (experimental, research-only use) is not publicly available and no longer supported by OpenPilot since they changed to MetaDrive for their internal tests.

4.2.4 Algorithm choice

After reviewing all this algorithms OpenPilot was immediately excluded due to its lack of support with CARLA, and so the remaining algorithms were Autoware and Apollo which both had bridges developed by the community to communicate with CARLA, unfortunately due to numerous open issues and hardware incompatibility Apollo was also excluded leaving us with CARLA and Autoware which the integration is well documented [94].

5 Fault Injection in Autonomous Vehicles

The existing literature has focused on developing frameworks and simulation tools to systematically assess AV resilience under sensor faults, as well as evaluating AV performance in controlled virtual environments. This section reviews prior work on fault injection frameworks for AV safety validation.

Ensuring that autonomous vehicles can handle faults has led to a body of research on AV fault injection and dependability assessment. One of the early efforts is AVFI (Autonomous Vehicle Fault Injector) [95], which pioneered end-to-end fault injection in AV systems. AVFI introduced faults at various points: sensor inputs (e.g. simulating camera or LiDAR failures), internal processing (e.g. flipping bits in a neural network), and even actuator commands, all within a simulated driving environment. The tool measured domain-specific safety metrics such as mission success rate and number of traffic violations under each fault scenario. Preliminary results from AVFI showed that injecting faults could indeed cause an AV to commit traffic violations that would not occur otherwise, underlining the importance of testing AVs beyond nominal conditions. AVFI used the CARLA simulator as a basis, demonstrating CARLA's suitability for such fault experimentation [95].

Building on this, researchers have looked for smarter ways to inject faults. DriveFI (proposed by researchers at NVIDIA in 2019) [96] is a machine learning-based fault injection engine. Instead of manually specifying faults, DriveFI employs a Bayesian optimization approach to automatically discover the most "dangerous" faults and scenarios. By testing two industry-grade AV stacks (including NVIDIA's own and Baidu's Apollo), DriveFI was able to find hundreds of safety-critical vulnerabilities within hours, whereas random fault injection over weeks found few or none. The types of faults considered included sensor distortions and even logic bugs, and the impact was measured in terms of accidents or near-misses in simulation. This work highlights how broad the space of possible faults is, and the need for intelligent search methods to focus on impactful cases [96].

The CarFASE tool [97] is specifically designed to integrate with open-source driving stacks (like OpenPilot by Comma.ai) and evaluate their behavior under both accidental faults and malicious attacks. For example, CarFASE can simulate a sudden change in camera brightness to mimic sensor attacks and then observe how the driving policy reacts. The authors used CarFASE to test OpenPilot's resilience, and one use-case showed that increased brightness (simulating a camera blinding scenario) led to degraded lane-keeping performance. The platform provides a library of fault models and a campaign configurator to automate scenario runs. The emergence of CarFASE underscores the community's interest in accessible tools for fault injection using popular simulators like CARLA.

Aside from these, other works have explored particular sensor fault scenarios. Another work [49] focused on camera failures in an AV context, defining failure modes such as blurred images, blackout, and occlusions, and testing their effect on

object detection and a self-driving agent in a simulator. The results showed that certain camera failures significantly increase detection errors and can lead to collisions in the simulation, which reinforces the notion that redundancy (like having multiple cameras or additional sensors) is needed. There are also studies on injecting faults in ADS controllers or code (for instance using fuzzing or software mutation), but those are beyond the scope of sensor-level fault injection.

In contrast to prior AV fault injection studies that often focus on isolated algorithms, single sensors, or open-loop data perturbations, this work delivers a closed-loop, scenario-triggered fault injection framework operating at the middleware level of a full autonomous driving software stack. The framework supports multiple sensor types (LiDAR, IMU gyroscope/accelerometer/quaternion, GNSS), uses a structured and reproducible fault model with location, type, trigger, and duration dimensions. By running injections during complete simulated driving scenarios, it is possible to capture system-level behavior and safety outcomes, enabling consistent classification of whether a fault was tolerated or led to unsafe operation. This combination of closed-loop execution, multi-sensor scope, and reproducibility distinguishes this work from previous tools and studies. Furthermore, this framework is implemented on ROS 2 which is mostly used in university research, but it can be ported to AUTOSAR Adaptive by bridging ROS 2 topics to SOME/IP (ara::com) services. Recent work [98] proposes an integrated ROS 2 - AUTOSAR architecture (ASIRA) with a ROS 2–SOME/IP bridge and demonstrates data exchange in autonomous-driving scenarios, including Autoware interoperating with an Adaptive AUTOSAR simulator, providing a path to use this framework on production-oriented AV software stacks. In practice, porting requires mapping message schemas and QoS/timing semantics and implementing an AUTOSAR-native injector.

In summary, the related work shows a progression from general fault injection frameworks (AVFI) to targeted or intelligent frameworks (DriveFI, CarFASE), as well as specific investigations of individual sensor failures. This work differentiates itself by focusing on an integrated sensor fault injection in a complete open-source AV stack (Autoware). Many prior works used either proprietary stacks or simplified models of an AV. By using Autoware, it is possible to exercise a full production-grade autonomy stack. Additionally, while tools like DriveFI aim to find worst-case faults automatically, our approach emphasizes the analysis of representative sensor faults to understand their effects. By manually designing fault conditions, it is possible to observe how specific sensor anomalies lead to unsafe behavior in the autonomous vehicle. This complements the broader search approach by providing insight into failure mechanisms. The fault injection code is available for the community, similar in spirit to CarFASE but focused on ROS/Autoware users.

6 FRAMEWORK IMPLEMENTATION

This section presents the framework developed to evaluate the impact of sensor faults on autonomous vehicle systems in a controlled and reproducible way. The framework combines existing open-source tools with newly developed extensions, creating an environment where autonomous driving software can be tested under scenario-based fault conditions.

The experimental platform of four components. The CARLA simulator (version 0.9.15), which provides the virtual environment and sensor data, Autoware.Universe (version 2019.1), which serves as the autonomous driving stack, the ROS bridge, which enables communication between the simulator and Autoware, and Scenario Runner, which orchestrates simulation scenarios. On top of this, the framework introduces multiple contributions, including a sensor fault injection system integrated into the ROS bridge, sensor and event logging, and modifications to Scenario Runner for automated test execution and coordination with Autoware.

Figure 14 provides a high-level overview of the experimental framework, showing how CARLA, Autoware, the ROS Bridge, and Scenario Runner interact in a closed-loop setup. The following subsections describe each component in detail before presenting the complete integration and extensions introduced in this work.

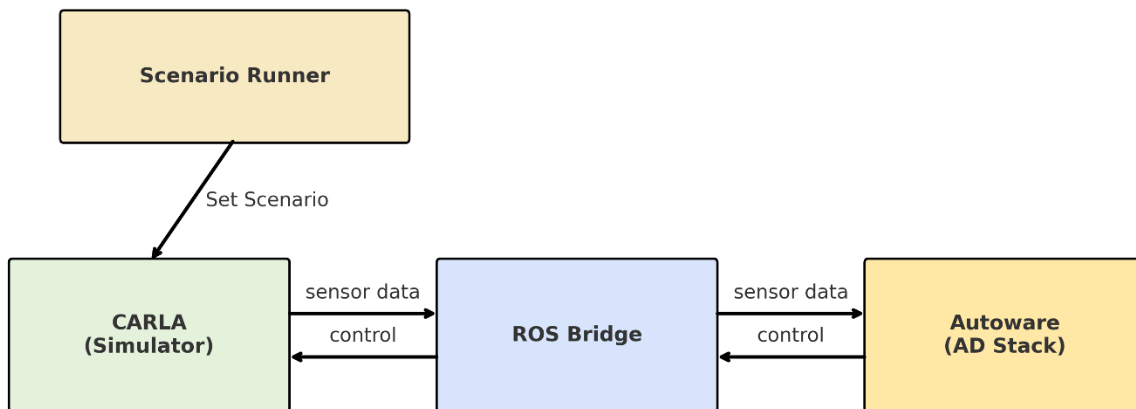

Figure 14 - High-level overview of the framework

6.1 CARLA

CARLA, one of the baseline components of the proposed framework, is an open-source simulator designed for autonomous driving research and development. Developed by the Computer Vision Center (CVC) at the Universitat Autònoma de Barcelona, CARLA provides a realistic 3D environment for testing and validating autonomous driving systems under a wide range of conditions. It supports high-fidelity urban scenes, configurable weather, traffic actors, and a broad suite of sensors, making it a widely adopted tool in both academic and industrial settings.

CARLA is an open platform. Uniquely, the content of urban environments provided with CARLA is also free. The simulation platform supports flexible setup of sensor suites and provides signals that can be used to train driving strategies, such as GPS coordinates, speed, acceleration, and detailed data on collisions and other infractions. A wide range of environmental conditions can be specified, including weather and time of day.

CARLA is built on Unreal Engine 4, which allows for advanced rendering and physics simulation, there is however a new version being built in Unreal Engine 5. The simulator operates using a client–server architecture (Figure 15), where:

- The CARLA server handles the simulation environment, including physics, rendering, traffic behavior, and all actors in the scene.
- A client, which may use a TCP/IP API or operate through a ROS interface, communicates with the server and it can control vehicles, place sensors, modify environment settings, and access simulation data. Each ego vehicle is controlled by a single client, for the sake of clarity this type of client is named "ego client", and then another client responsible for setting the environment settings and retrieve data, if needed.

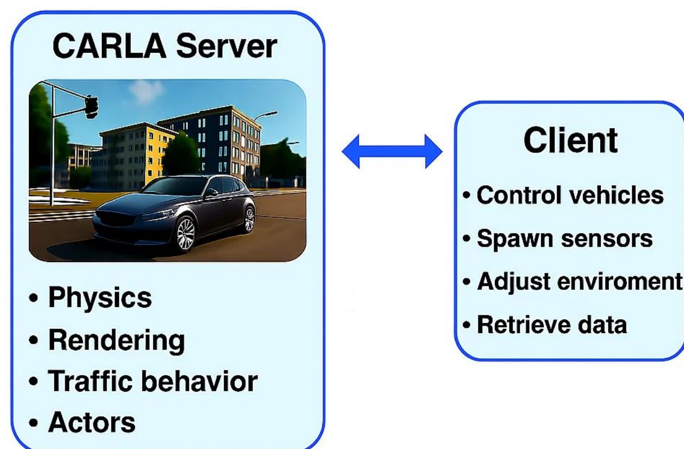

Figure 15 - CARLA-Client communication using its API

CARLA can operate in two simulation modes: asynchronous and synchronous [99]. In the default asynchronous mode, the simulator advances at its own pace without waiting for the connected algorithm or stack (ego-client) to complete its cycle. This means that if the computing resources are not fast enough, the simulator and the algorithm may fall out of sync. As a result, events can be missed and one side (the simulator or the algorithm) may continue operating based on outdated information, leading to inconsistencies and loss of determinism between runs.

The synchronous mode ensures determinism and repeatability. In this mode, the simulation advances strictly in lockstep with the client's commands, each simulation step only progresses when explicitly triggered by the ego client. Thus, simulation events occur based on virtual time, independent of how long the hardware takes to compute each step. Consequently, a slower machine will take more real-world time

to complete the same number of simulation steps compared to a faster machine, however, the internal timing, precision, and order of events within the simulation remain consistent. The concept of virtual time is crucial in this context, as it ensures simulation integrity and repeatability, making synchronous mode particularly suitable for experiments where deterministic and precise timing control is required.

6.1.1 Scenario Management and Ego Vehicle

The creation of scenarios is aided by the use of tools such as the Scenario Runner, which allows for the definition and control of events including intersections, pedestrian crossings, emergency stops, and vehicle overtakes. These scenarios define not only the behaviors and trajectories of the multiple actors in the simulation, but also the environmental conditions (such as weather or time of day).

At the core of each scenario is the **ego vehicle**, representing the autonomous vehicle under test. It can be configured with a wide range of sensors including LiDAR (standard and semantic), RGB and depth cameras, IMU, GNSS, radar, and dynamic vision sensors (DVS). These sensors are placed at configurable locations on the vehicle and stream data in real-time, enabling both online and offline analysis. The ego vehicle can be controlled manually via the client interface or connected to an external autonomous driving system such as Autoware.

Scenario Runner is a Python-based framework developed specifically for CARLA that enables the execution of these driving scenarios. It provides an interface for defining scenario behavior, controlling the timing and logic of actor actions, and monitoring simulation outcomes. Scenario descriptions can be written in Python scripts or in XML format, allowing for a modular and reusable setup.

Once a scenario is launched, Scenario Runner connects to the running CARLA server using the Python API. It then spawns the defined actors into the simulation world, orchestrates their behavior over time, and monitors key criteria such as collisions, route completion, traffic light compliance, and other successful conditions. This enables repeatable and automated testing of AV behavior under consistent environmental and situational parameters, which is essential for evaluating performance and safety under diverse and controlled conditions.

6.1.2 CARLA – Autonomous Driving System Communication (ROS)

The CARLA ROS Bridge functions as a middleware interface that facilitates the integration of the CARLA simulation environment with ROS, enabling communication between the ego vehicle simulated in CARLA and external autonomous driving software frameworks, a simple diagram is showcased in Figure 16.

ROS is an open-source middleware framework widely adopted in robotics research and development. It offers a modular architecture based on a publish–subscribe communication model, wherein distributed software components, referred to as

nodes, exchange information through topics. ROS additionally provides comprehensive tools for sensor integration, data visualization, debugging, and simulation control.

Figure 16 - CARLA ROS Bridge Diagram

Within the domain of autonomous driving, ROS links core modules such as perception, localization, planning, and control subsystems. The CARLA ROS bridge translates CARLA's internal simulation data into standard ROS message formats and vice versa. This bidirectional interface enables the deployment of advanced driving stacks, such as Autoware, within the simulated environment. CARLA ROS bridge core functionalities include:

- Providing Sensor Data (Lidar, Semantic lidar, Cameras, GNSS, Radar, IMU)
- Providing Object Data (Traffic light status, Visualization markers, Collision, Lane invasion)
- Controlling AD Agents (Steer/Throttle/Brake)
- Controlling CARLA (Play/pause simulation, Set simulation parameters)

This integration is essential for testing and validating autonomous driving algorithms, particularly under controlled fault conditions.

6.2 Autoware

Autoware, the autonomous driving stack integrated as another baseline component of the framework, is an open-source software stack for autonomous driving systems, built on top of ROS, supporting both ROS 1 (Autoware.AI) and ROS 2 (Autoware.Auto / Autoware.Universe), with newer ROS 2 based versions focusing on real-time safety, modularity, and compliance with automotive-grade standards. Autoware builds on ROS by integrating a curated set of perception, localization, planning and control modules tailored for autonomous driving. Compared with using ROS alone, Autoware provides a standardized interface to vehicle hardware, built-in sensor fusion pipelines (e.g., LiDAR/IMU/GNSS fusion), high-definition map support, object detection and tracking, behavior and trajectory planning, and vehicle control interfaces.

Autoware is designed to serve as a complete platform for research, prototyping, and deployment of self-driving vehicles, particularly in urban and suburban environments [100], and when integrated with CARLA, Autoware becomes a

powerful tool for testing, validation, and development in a safe and reproducible virtual environment.

Autoware has been successfully deployed in various real-world autonomous driving projects, especially in Japan and Europe, including autonomous shuttles, delivery robots, and smart city infrastructure tests [101].

6.2.1 Autoware Architecture

Autoware builds on ROS's communication infrastructure, which is based on a publisher–subscriber model. In this model, each module publishes data as a topic (for example, LiDAR scans, localization results, or planned trajectories), while other modules that need this information subscribe to the corresponding topic. The system ensures that data is exchanged asynchronously and in real time, without modules depending directly on each other. This loose coupling improves modularity and flexibility, as components can be added, removed, or replaced as long as they follow the same topic definitions. For instance, the perception module may publish detected obstacles to a topic that both planning and control modules subscribe to, ensuring consistent and up-to-date situational awareness across the stack.

Autoware provides a modular architecture, Figure 17, that includes all the essential components required for autonomous vehicle operation, such as:

- **Perception**: Object detection, segmentation, and tracking using data from LiDAR, and cameras.
- **Localization**: Using GNSS, IMU, and LiDAR-based SLAM or map-matching for accurate vehicle positioning.
- **Planning**: Path planning, behavior planning, and route generation based on real-time map and object data.
- **Control**: Low-level vehicle actuation commands for throttle, brake, and steering.
- **Interface**: Tools for vehicle-to-platform communication and human-machine interaction.

Autoware is distributed in two main variants: Core and Universe. These modules are built to be extensible and reusable, allowing developers to adapt and expand the system according to their needs [100].

Core module

The Core module provides the essential runtime and foundational components necessary for the core functions of autonomous driving including sensing, computation, and actuation. It is developed and maintained by the Autoware Foundation (AWF), with contributions from its architects and key members through dedicated working groups. While contributions from the wider community are

welcome, the criteria for accepting pull requests are stricter compared to the Universe module, ensuring high standards of reliability and maintainability.

Universe Module

The Universe module extends the Core by offering additional features that enhance sensing, computation, and actuation capabilities. AWF maintains a base version of the Universe module, which serves as a starting point for further extension. One of the strengths of Autoware's microautonomy architecture is that any individual or organization can contribute, making it a collaborative and flexible ecosystem. AWF oversees the development process and performs quality control, but not all Universe modules are officially verified or validated. As a result, users are free to choose and integrate the modules that best suit their application needs, whether fully certified or experimental.

Figure 17 presents the modular architecture of Autoware, the following subsections explain the role of each component.

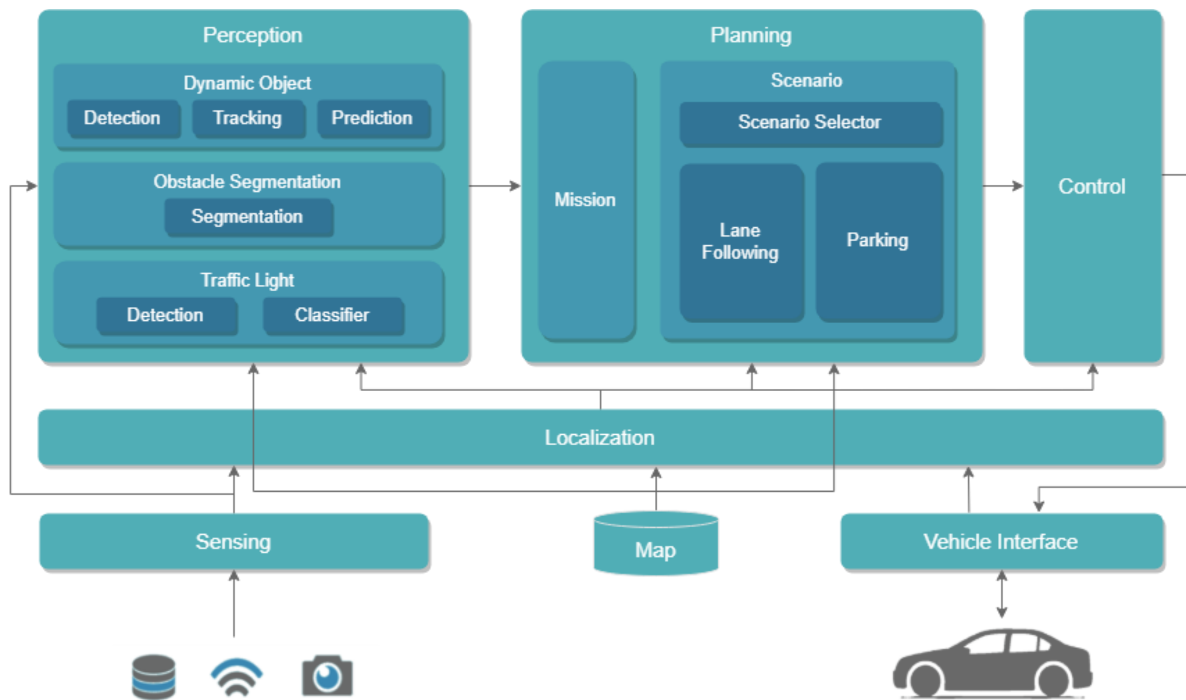

Figure 17 - Autoware Architecture (from [100])

6.2.2 Sensing

The sensing component in Autoware (Figure 18) is responsible for pre-processing raw sensor data and standardizing data formats. This ensures compatibility with various sensor brands and prepares the data for use in other modules such as localization, perception, and control.

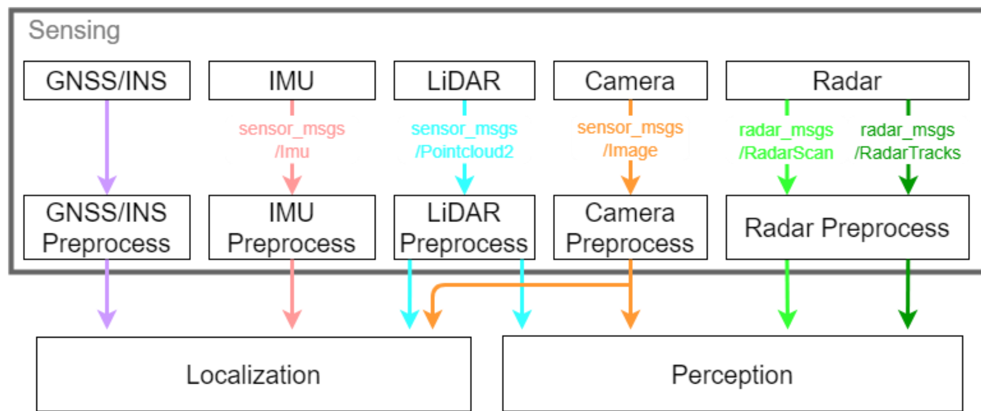

Figure 18 - Autoware - sensing component (from [100])

Supported Sensor Types and Data Outputs

- **GNSS/INS:** Autoware supports multiple data formats to extract position, orientation, and velocity from GNSS/INS devices:
    - **sensor_msgs/msg/NavSatFix:** Provides the vehicle's global position (latitude, longitude, altitude), along with status flags indicating the type of fix (e.g., RTK, float, single).
    - **autoware_sensing_msgs/msg/GnssInsOrientationStamped:** Provides orientation data as a quaternion (x, y, z, w), timestamped and aligned with the GNSS pose.
    - **geometry_msgs/msg/TwistWithCovarianceStamped:** Describes both linear and angular velocity, including uncertainty (covariance), typically used to convey GNSS/INS velocity information.
- **IMU:** The IMU provides motion-related data.
    - **sensor_msgs/msg/Imu:** Provides orientation, angular velocity, and linear acceleration. Covariance values are included for sensor fusion processes like Kalman filtering.
- **LIDAR:** LiDAR provides 3D point cloud data describing the vehicle's surroundings. The input consists of raw point clouds containing the spatial position of detected points along with intensity information. The data undergoes several preprocessing tasks such as downsampling through voxel grid filtering, optional ground removal, calibration and pose correction, and transformation into the vehicle's coordinate frame. This processed data is then used for localization, for example through scan-matching methods like NDT, as well as for perception tasks such as object detection and segmentation.
    - **sensor_msgs/msg/PointCloud2:** Represents 3D point clouds including XYZ positions, intensity values, and optionally time or reflectivity per point.

- **Camera:** Cameras provide image data for visual perception tasks.
  - **sensor_msgs/msg/Image:** Delivers RGB, monochrome, or depth images, along with metadata such as resolution and encoding format (e.g., rgb8, mono8).
- **RADAR:** Radar is used for detecting objects and measuring their motion.
  - **radar_msgs/msg/RadarScan**: Delivers raw detections including range, azimuth angle, velocity, and signal strength.
  - **radar_msgs/msg/RadarTracks**: Provides processed radar outputs with tracking data such as object ID, position, velocity, and confidence scores.

6.2.3 Map

In Autoware, the Map Component is the module responsible for managing map data and distributing it to other parts of the stack. It does not generate maps itself but consumes externally created maps and ensures they are provided in a consistent coordinate system to the relevant modules.

Autoware relies on point-cloud maps, which are three-dimensional representations of the environment composed of a dense set of points obtained from LiDAR scans, and vector maps, which describe road geometry, lanes, and traffic rules in a structured format. In practice, Autoware uses the Lanelet2 format for vector maps, where the road network is represented as a set of interconnected lane segments enriched with semantic information such as speed limits, traffic light locations, and priority rules. Together, these maps support tasks such as localization, route planning, traffic light detection, and trajectory prediction.

In summary, the Map Component acts as a data manager, ensuring that externally generated maps are consistent, accessible, and properly distributed to downstream Autoware modules.

Requirements

Map component takes two types of maps as input, a vector map and a point cloud map. The vector map contains information about the road network, lane geometry, traffic lights, and traffic signals. This type of map is used for route planning, traffic light detections, and trajectory predictions.

The point-cloud map is primarily used for LiDAR-based localization, this is later explained in the localization component, in short it compares what is "seen" by LiDAR with the map information to determine its current position.

In addition to these two maps, Autoware also requires a projection information file which specifies the coordinate system of the map in geodetic system (WGS).

Point Cloud Map

The point cloud map must be supplied as a file with the following requirements:

1. The point cloud map must be projected onto the same coordinate system defined in map_projection_loader to remain consistent with the Lanelet2 map and other packages that convert between local and geodetic coordinates. In this context, projection means transforming the raw point-cloud data from its original local coordinate frame into a global map reference system (for example, UTM), ensuring that all spatial data sources share the same frame of reference.
2. It must be in the PCD (Point Cloud Data) file format but can be a single PCD file or divided into multiple PCD files.
3. Each point in the map must have X, Y, and Z coordinates.
4. An intensity or RGB value for each point may be optionally included.
5. It must cover the entire operational area of the vehicle. It is also recommended to include an additional buffer zone according to the detection range of sensors attached to the vehicle.
6. Its resolution should be at least 0.2 m to yield reliable localization results.
7. It can be in either local or global coordinates but must be in global coordinates (georeferenced) to use GNSS data for localization.

Vector Map

The vector cloud map must be supplied as a file with the following requirements:

- It must be in Lanelet2 format, with additional modifications required by Autoware.
- It must contain the shape and position information of lanes, traffic lights, stop lines, crosswalks, parking spaces, and parking lots.
- Except at the beginning or end of a road, each lanelet in the map must be correctly connected to its predecessor, successors, left neighbor, and right neighbor.
- Each lanelet in the map must contain traffic rule information including its speed limit, right of way, traffic direction, associated traffic lights, stop lines, and traffic signs.
- It must cover the entire operational area of the vehicle.

Projection Information

The projection information must be supplied as a file with the following requirements:

- It must be in YAML format, provided into map_projection_loader in current Autoware Universe implementation.

- The file must contain the following information:
- The name of the projection method used to convert between local and global coordinates
- The parameters of the projection method (depending on the projection method)

Map Component Architecture

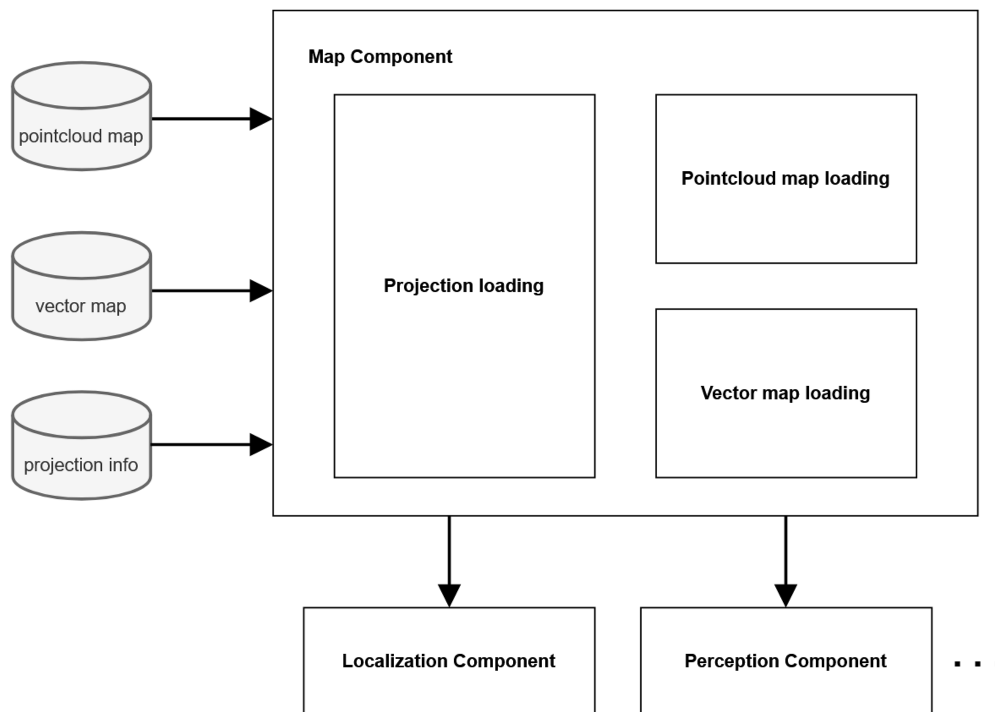

Figure 19 - Autoware - Map Architecture (from [100])

The input into the map component (Figure 19) is the maps already previously mentioned and as a result this component outputs:

- To **sensing** component, the projection information, that is used to convert gnss data to the local coordinate system.
- To **localization** component, the point cloud (used for LiDAR-based localization) and vector maps (used for localization methods based on road markings).
- To **perception** component, the point cloud map (used for obstacle segmentation) and vector map (used for vehicle trajectory prediction).
- To **Planning** component, the vector map for behavior planning.

6.2.4 Localization

The Localization component in Autoware is responsible for estimating the vehicle's position, velocity, and acceleration in real-time. It ensures that these estimations are

reliable, and if not, it generates errors or warning messages to inform the er-ror-monitoring system. The method used for localization depends on the available sensors and the characteristics of the environment.

One commonly used method involves combining 3D LiDAR with a point cloud map. This approach is particularly effective in urban settings where numerous buildings and structures provide distinctive features for alignment. It works by matching incoming LiDAR data with a pre-existing point cloud map using techniques such as scan matching or Normal Distributions Transform [102]. However, this method struggles in environments where the map lacks structural features, such as rural landscapes, open highways, or tunnels. It is also sensitive to environmental changes not represented on the map, including snow, construction work, or structural alterations. Signal issues such as reflections, glass surfaces, or interference from other laser sources may further degrade accuracy.

Another supported method involves using either 3D LiDAR or a camera in combination with a vector map. This configuration performs well in environments with clearly marked lanes, such as highways. Its accuracy can be affected by degraded or obstructed lane markings and variations in road surface reflections.

GNSS-based localization is suitable for open environments with minimal obstructions, such as rural areas. While it provides absolute global positioning, it is vulnerable to signal degradation, blockage, and spoofing. IMU-based localization relies on measurement of acceleration and angular velocity to estimate pose changes through dead reckoning. It works well in smooth and flat road conditions, but it is subject to drift over time due to inherent sensor biases which can be affected by environmental conditions such as temperature.

The Localization module integrates data from all these sources to improve accuracy and resilience. It requires that all incoming sensor data be correctly timestamped, valid, and consistent with the vehicle's configuration. Additionally, maps used for localization, such as point cloud or lanelet2 formats, must closely match the real environment and be aligned if multiple sources are used. Large discrepancies between the map and the actual scene may lead to localization errors.

Localization Architecture

The architecture of this module is presented in Figure 20.

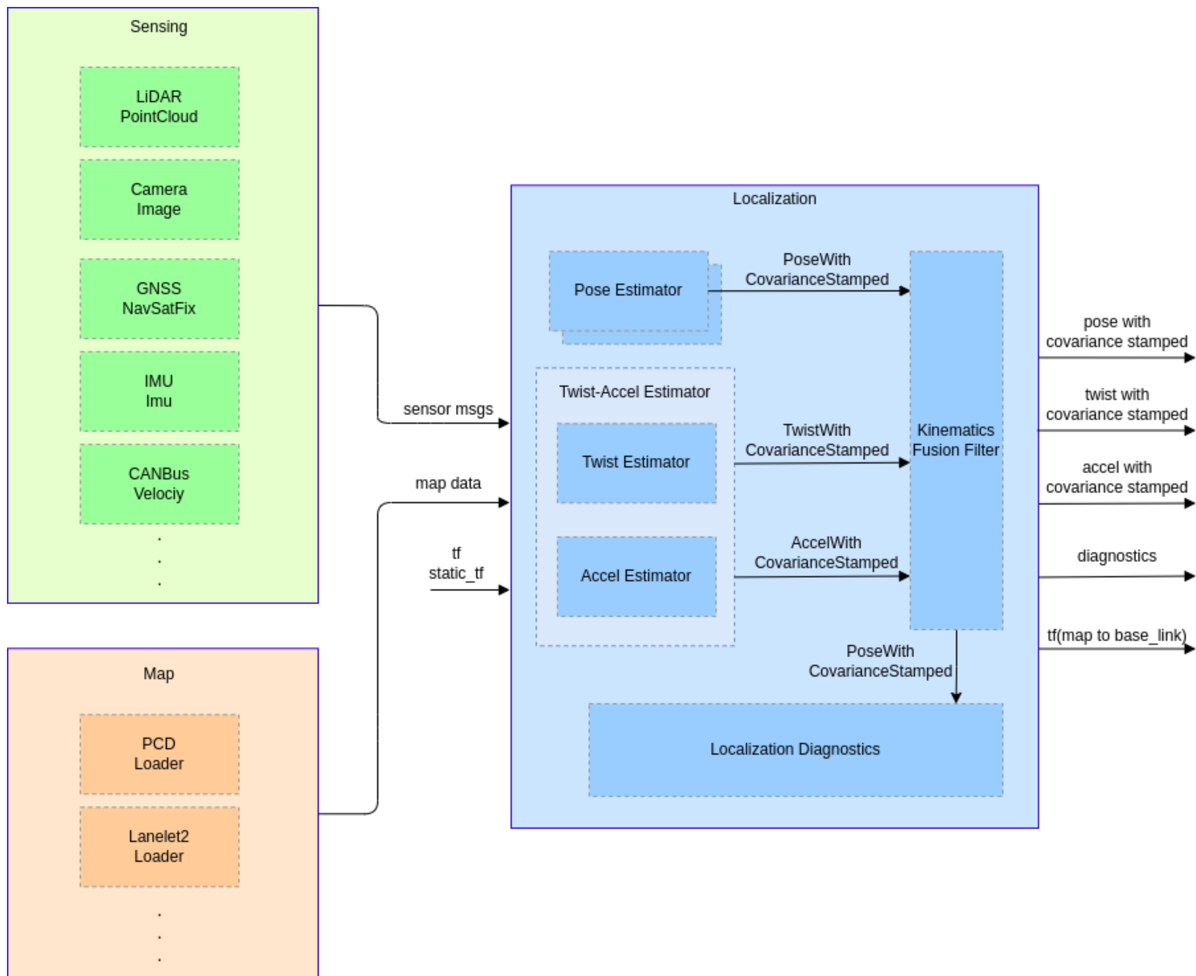

Figure 20 - Autoware - Localization architecture (from [100])

Localization Module Input

The localization module relies on several types of input data. First, it ingests sensor data, already preprocessed by the Sensing module. In addition, the system requires map inputs, such as high-definition point cloud maps and Lanelet2-based vector maps. These maps must closely reflect the actual environment. Discrepancies between the map and the real world, such as construction, road changes, or seasonal variations, can reduce localization accuracy. Furthermore, if multiple maps are used, they must be properly aligned in a common coordinate system. While Transform (TF) and static TF inputs are optional, they are recommended for accurate frame alignment. Incorrect sensor mounting positions or missing transform data may result in degraded localization performance. Figure 21 is an example of a TF given by Tier IV.

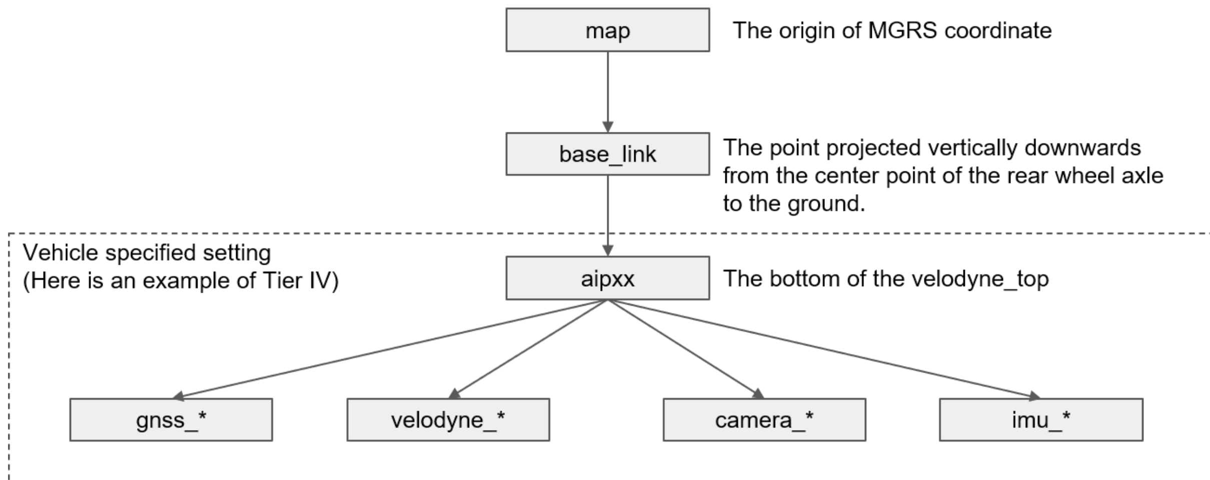

Figure 21 - Autoware - TF example (from [100])

Localization Module Output

The output of the localization module consists of several structured data messages, each essential for downstream modules such as planning, control, and system monitoring. The primary output is the vehicle's estimated pose, which is published as a PoseWithCovarianceStamped message. This includes the vehicle's position and orientation in the map frame, along with a covariance matrix indicating estimation uncertainty and a timestamp to maintain temporal consistency.

In addition to pose, the module provides a TwistWithCovarianceStamped message, which includes the vehicle's linear and angular velocity, expressed in the base_link coordinate frame. Likewise, acceleration data is published using the AccelWithCovarianceStamped format, containing both linear and angular acceleration along with their respective covariances. These dynamic quantities are crucial for understanding the vehicle's motion state and are used in both control and prediction layers of the autonomy stack.

The module also outputs diagnostic messages, which inform whether the localization module is operating reliably. These diagnostics can indicate degraded performance or failure conditions, which may trigger safety mechanisms in higher-level logic. Finally, if enabled, the module produces a TF transform from the map frame to the base_link frame, allowing other modules to align sensor data and planning operations spatially.

Internal Submodules

To produce these outputs, the localization module is structured around four key submodules. The first is the Pose Estimator, which aligns incoming sensor data, such as LiDAR or camera observations with the known map. This alignment process estimates the vehicle's position and orientation in global coordinates using methods like scan-matching or visual feature tracking.

The Twist-Accel Estimator calculates the vehicle's velocity and acceleration, including angular components. It provides not only the values themselves but also

associated covariances, reflecting the uncertainty in motion estimation. This data is particularly important in dynamic environments where precise motion tracking is needed for decision-making.

The Kinematics Fusion Filter serves as the integration module. It combines the static pose from the Pose Estimator with the motion data from the Twist-Accel Estimator to generate a consistent and refined estimate of the vehicle's pose, velocity, and acceleration, along with their uncertainties. This module is also responsible for generating the map-to-base_link transform (TF) used throughout the Autoware system.

Finally, the Localization Diagnostics module oversees the stability and reliability of the entire localization stack. It aggregates data from the other modules to evaluate system confidence. If it detects inconsistencies, signal degradation, or increasing uncertainty, it reports the issue through diagnostic outputs, allowing the broader system to react accordingly.

6.2.5 Perception

The role of the Perception component is to perceive the surrounding environment based on the data obtained through Sensing, Localization, and Map components, and adds semantic information (e.g., Object Recognition, Obstacle Segmentation, Traffic Light Recognition, Occupancy Grid Map), which is then passed on to Planning Component.

Perception Component Architecture

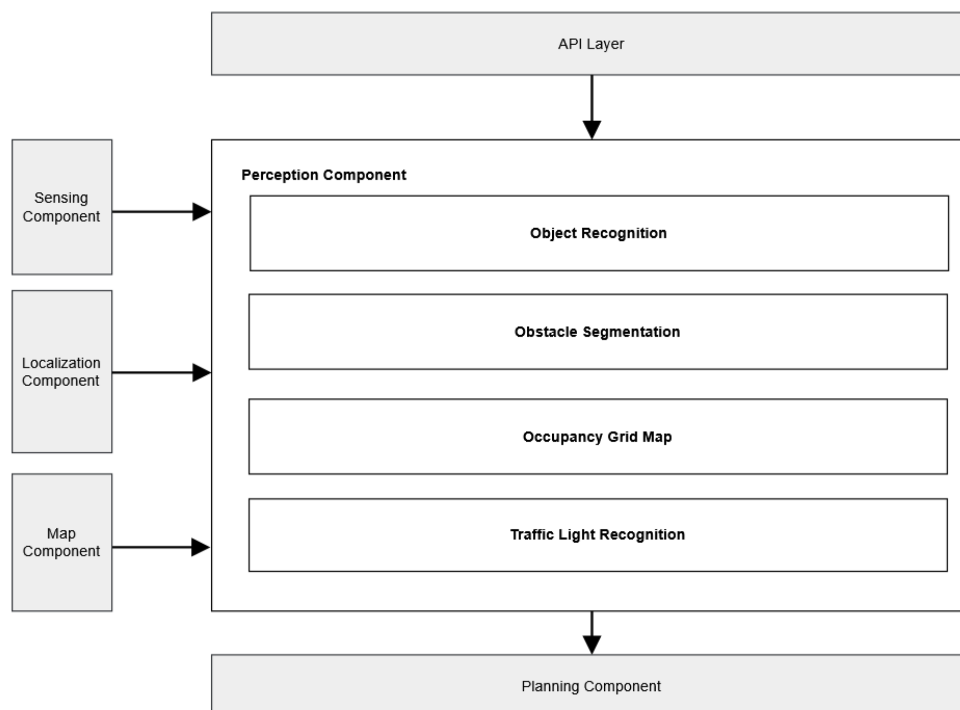

Figure 22 - Autoware - Perception Architecture (from [100])

In Figure 22 it is possible to observe that Autoware's perception component takes as input data from:

- Sensing component, camera image, point cloud and radar object data.
- Localization component, vehicle motion information.
- Map component, vector map and point cloud map.
- API, V2X, receives information like traffic signals state.

Then after processing this information, it is sent to the planning component:

- **Dynamic objects**: real-time information about pedestrians and other vehicles.
- **Obstacle segmentation**: location of detected obstacles.
- **Occupancy Grid Map**: information about occluded areas.
- **Traffic light recognition result:** provides the state of each traffic light in real time.

6.2.6 Planning

The Planning component decides where and how the vehicle should move. To do this, the component is divided into three sub-modules, Mission planning, Planning Modules, and Validation. This module (Figure 23) is also modular and it allows developers to add and develop their own modules.

Planning Component Architecture

The default sub-modules' responsibilities are as follows:

Mission Planning: It calculates routes from the current location to the destination, using the map.

Planning Modules: These modules plan the vehicle's behavior for the assigned mission, including target trajectory, blinker signaling, etc. They are divided into Behavior and Motion:

- **Behavior**: Calculates safe and rule-compliant routes, managing decisions for lane changes, intersection entries, and stoppings at a stop line.
- **Motion**: Works alongside Behavior modules to determine the vehicle's trajectory, considering its motion and ride comfort. It includes lateral and longitudinal planning for route shaping and speed calculation.

Validation: Ensures the safety of the planned trajectories.

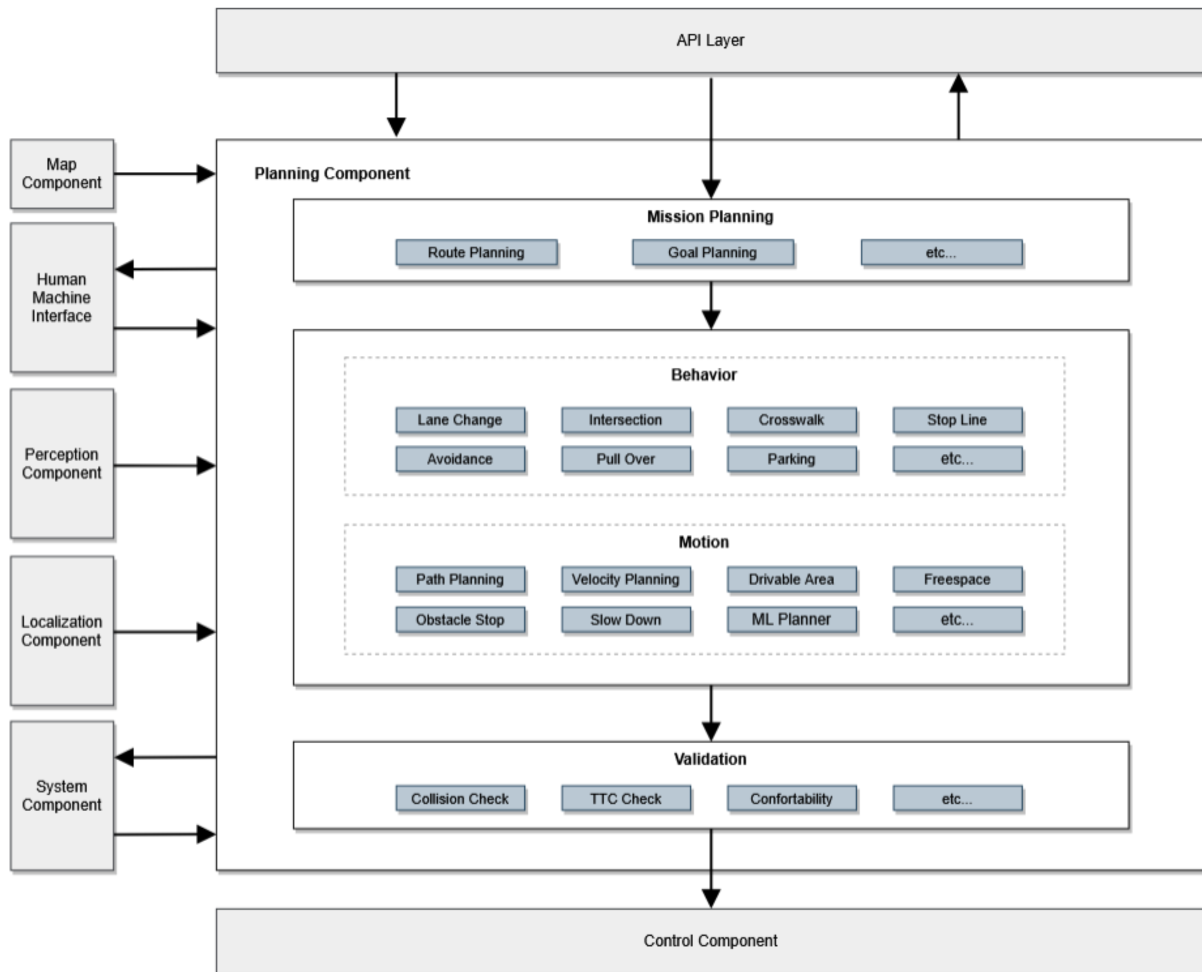

Figure 23 - Autoware - Planning Architecture (from [100])

This component takes as **input** information from:

- **Map component**: vector map containing all the necessary information for route planning.
- **Perception component:**
  - **Detected objects and obstacles**, so that the planning component plans maneuvers to avoid collisions.
  - **Occupancy map information**, information about occluded obstacles, so that the planning component can take them into account when planning.
  - **Traffic light recognition result**, current state of traffic lights, it allows the planning component to determine whether a stop at an intersection is necessary or not.
- **Localization component:** vehicle motion information.
- **System:** it indicates if the vehicle is in Autonomous mode.
- **Human Machine Interface (HMI):** Gets authorization to execute the plan.

- **API Layer:** It gets information like the destination, checkpoints and velocity limit.

With this input the component then produces results and sends them to the following components:

- **Control component:** trajectory and turn signals.
- **System component:** diagnostics, reports the current state of the planning component.
- **Human Machine Interface:** trajectory candidate and the status of operations that can be executed.
- **API:** it sends information about the reasoning behind a behavior.

6.2.7 Control

The Control component in Autoware is responsible for translating planned trajectories into low-level commands that can be executed by the vehicle's actuators (steering, throttle, and brake).

6.2.8 Vehicle Interface

The Vehicle Interface is the bridge between Autoware's software control commands and the vehicle's hardware or simulation backend. It ensures that the steering, throttle, brake, and gear commands generated by the Control module are translated correctly into actuator-level signals that the vehicle understands.

6.2.9 Autoware Challenges

A big challenge of the micro autonomy architecture is to achieve real-time capability, which guarantees all the technological components activated in the system to predictably meet timing constraints (given deadlines). In general, it is difficult, if not impossible, to tightly estimate the worst-case execution times (WCETs) of components.

In addition, it is also difficult, if not impossible, to tightly estimate the end-to-end latency of components connected by a DAG. Autonomous driving systems based on the microautonomy architecture, therefore, must be designed to be fail-safe but not never-fail. Timing constraints violation is accepted and attended (the given deadlines may be missed) as far as the overrun is considered. The overrun handlers are two-fold: (I) platform-defined and (ii) user-defined. The platform-defined handler is implemented as part of the platform by default, while the user-defined handler can overwrite it or add a new handler to the system. This is called "fail-safe" on a timely basis.

6.3 Framework Integration

With the baseline components of the framework established in Sections 6.1 and 6.2, this section focuses on how they were integrated and extended to support sensor fault injection and automated evaluation. The CARLA–Autoware connection through the ROS bridge forms the foundation, enabling closed-loop operation between the simulator and the autonomous driving stack. Building on this integration, several contributions were introduced in this work: modifications to Scenario Runner for orchestrating experiments, a sensor fault injection system integrated into the ROS bridge, and sensor and events logging. Together, these modifications transform the CARLA–Autoware integration into a experimental framework for assessing fault tolerance in autonomous driving.

6.3.1 Framework Overview

The integration of CARLA and Autoware forms the foundation of the experimental framework developed in this work. Beyond simply connecting the simulator to the autonomous driving stack, the framework introduces extensions that enable automated scenario execution, sensor fault injection, and logging.

At the core of this setup, the ROS bridge establishes communication between CARLA and Autoware. It converts and forwards sensor data provided by CARLA (via TCP/IP) into ROS 2 message structures, while also translating Autoware commands into CARLA's API using a controller developed by TUMFTM [94]. Additionally, the original CARLA Autoware Bridge manages initial server settings (e.g., map, and weather), sensor configurations of the AV, and spawning of the AV in the CARLA server.

To orchestrate scenario execution and synchronize fault events, a modified version of Scenario Runner was incorporated. In addition to managing scenarios, this extended version coordinates with the ROS bridge to handle sensor fault injection, start/stop logging, and automate evaluation of test outcomes.

The ROS bridge was extended to support scenario-based fault injection testing. Initially, it was enhanced to log collision and lane invasion events directly. Then introduced full sensor data logging, followed by a fault injection mechanism that intercepts and modifies sensor data before it reaches Autoware. These modifications also include a dedicated ROS node capable of receiving instructions from Scenario Runner. These instructions are (i) to start/stop sensor logging, and (ii) the fault injection configuration file, which is a JSON file defining the affected sensor, failure type, triggering location (map-based), and duration. Scenario Runner also enables automated evaluation of scenario outcomes. Criteria such as collision detection, timeouts, and lane invasions are used to determine whether a test run was successful or failed, eliminating the need for manual inspection of raw logs.

Figure 24 illustrates the final bridge design, showing the modified ROS bridge and its integration with Scenario Runner within the overall framework.

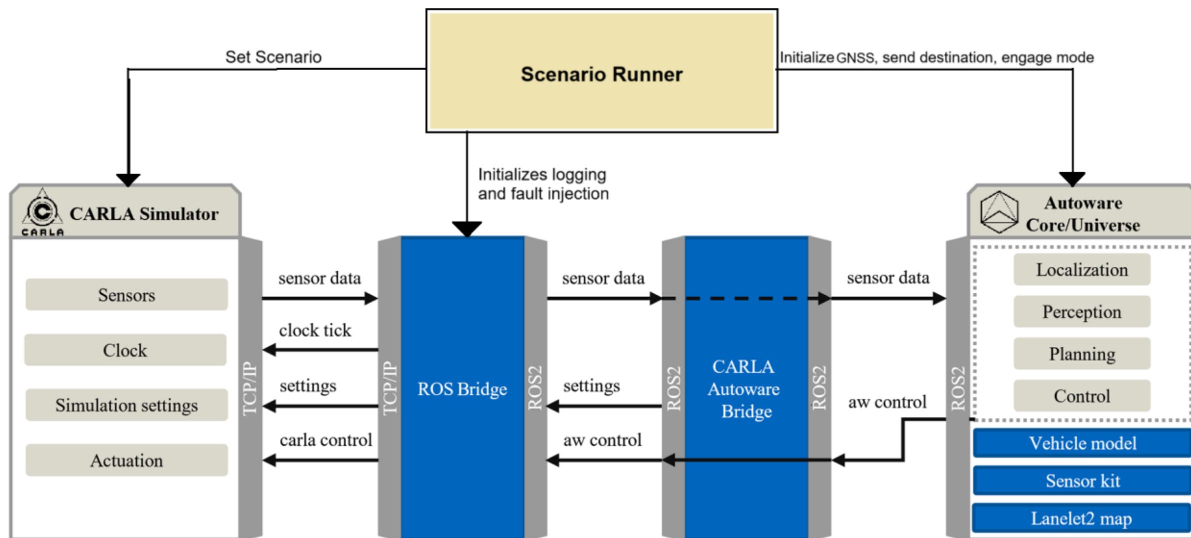

Figure 24 - CARLA Autoware modified bridge (based of [94]).

6.3.2 Scenario Runner Modifications

To evaluate system resilience under different fault conditions, each scenario must be executed multiple times with varying configurations. Running these manually is inefficient and prone to errors. For this reason, Scenario Runner was extended to support automated batch execution, where each run combines a scenario from the workload with a fault configuration from the faultload.

Additionally, since the ego vehicle is managed externally through the Autoware integration rather than by the CARLA Scenario Runner itself, Scenario Runner was modified to detect and attach to a pre-existing ego vehicle within the simulation. This avoids redundant instantiation and enables compatibility with externally controlled systems.

During each test run, sensor data and safety-related events (e.g., collisions, lane invasions) are logged, enabling post-analysis of how different fault types affect the vehicle's behavior. This automation improves reproducibility and expands test coverage while minimizing manual intervention.

6.3.3 Sensor Fault Injection System Implementation

The sensor fault injection framework is designed to be modular and extensible and is centered around a base class named FaultInjector. This module integrates directly into the ROS bridge, enabling the emulation of sensor faults by altering data streams before they are published on ROS topics. As a result, downstream modules within Autoware, such as perception and localization, process the modified data as if it originated from malfunctioning sensors. The class diagram is presented in Figure 25.

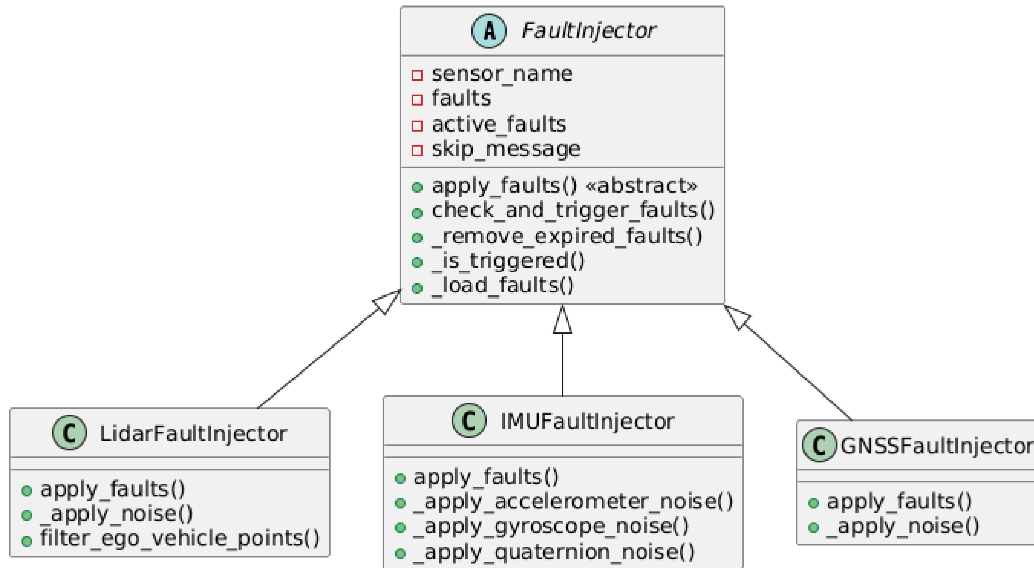

Figure 25 - Fault injection class diagram

The framework was implemented in Python and integrated into the CARLA ROS bridge, which is built on ROS 2 Humble. It uses JSON configuration files, depends on CARLA's Python API, and is compatible with Autoware.Universe.

Each sensor type, LiDAR, IMU, and GNSS, is handled by a dedicated subclass (LidarFaultInjector, IMUFaultInjector, and GNSSFaultInjector, respectively), all inheriting from the FaultInjector base class. This base class provides shared functionality, including fault management, activation logic, and the apply_faults() abstract method, which is implemented by each subclass. Although a camera sensor is included in our sensor configuration, no faults were injected into it, because it was not functional in the version of Autoware.Universe used in this work.

The FaultInjector class is tightly coupled with the Sensor class from the ROS bridge, which handles the conversion of CARLA sensor data into ROS 2 message formats. Upon initialization, FaultInjector loads a fault injection configuration file in JSON format. This configuration defines the location, type, triggering, and duration. Each time a new sensor message is received, the FaultInjector checks whether any faults should be applied, updates the list of active faults, and deactivates those whose duration has expired.

Two main fault types are supported, silent sensor failure and noise.

Silent sensor failures correspond to the sensor becoming mute. In this case, the sensor continues to exist in the system but stops providing valid data. Practically, this is realized by intercepting the sensor's data stream in the ROS bridge and replacing it with empty or constant values (such as zeros), which downstream Autoware modules interpret as missing data.

Noise faults simulate degraded sensor output caused by environmental or hardware-induced disturbances (e.g., electromagnetic interference, vibrations, or aging

components). The way noise is applied depends on the sensor type and is defined in detail for LiDAR, IMU, and GNSS.

The LidarFaultInjector modifies the point cloud data by adding configurable gaussian noise or stopping data publication to simulate silent failure. Noise parameters (e.g., min_noise, max_noise) are defined in the JSON configuration file.

The IMUFaultInjector injects gaussian noise independently into the gyroscope, accelerometer, and orientation quaternion data. It can also simulate silent failures for the entire IMU stream.

The GNSSFaultInjector applies noise to latitude, longitude, and altitude values and can simulate complete GNSS silent failures.

The fault injection configuration is sent by Scenario Runner as a JSON file and includes all necessary data, sensor ("LiDAR", "IMU", "GNSS"), faults ("noise" or "silent"), trigger (map-based location of the ego vehicle), duration (time in seconds, 0 if permanent), and other parameters (fault-specific configuration, such as noise amplitude range). Figure 26 showcases an example of a sensor configuration file.

```
[
    {
        "sensor": "GNSSSensor",
        "faults": [
            {
                "name": "noise",
                "trigger": {
                    "location": {
                        "x": -21.0,
                        "y": 66.0,
                        "z": 2.0}},
                "parameters": {
                    "noise_stddev": {
                        "latitude": 0.00002,
                        "longitude": 0.00002,
                        "altitude": 0.2}},
                "duration": 0
            }
        ]
    }
]
```

Figure 26 - Sensor Fault configuration example

This design ensures that faults are applied in a consistent and reproducible manner relative to the scenario configuration and trigger conditions, making the system suitable for controlled experiments across multiple test runs.

The current implementation supports silent failure and severe noise for LiDAR, GNSS, and IMU. The same injection point in the ROS bridge allows additional fault models (constant bias/offset, drift as a random walk, stuck-at outputs, latency/jitter, and intermittent dropouts, among others) to be added with minimal changes, and to be applied to other sensors as the system under test and simulator interfaces permit. In this first iteration we model "severe noise" as zero-mean Gaussian, giving us a basic way to perturb the sensors and offering a reproducible way to stress the AV stack without targeting a particular sensor's physics [103]. While Gaussian noise is a useful baseline, real measurement errors are often non-Gaussian or time-correlated.

For example, inertial sensors exhibit white noise plus bias instability/random walks [104], GNSS errors in urban scenes show heavy-tailed residuals from multipath and blockage [105]. LiDAR in rain, fog, or snow [106] produces outliers and missing returns rather than purely Gaussian jitter.

6.3.4 Logging and Results Collection

Scenario Runner provides mechanisms for collecting and exporting results of each test run. Each scenario defines a set of relevant criteria by adding them to a behavior tree. This set determines what is monitored and what constitutes success or failure for that scenario, then the result for each criterion is stored in a file (txt, xml, or json). The criteria can be, collision (if the ego vehicle collided), route completion (how much of the planned route was completed), running redlight (detects if the ego vehicle ran a red light), Off road (if the vehicle left the drivable area), side walk (of the ego vehicle drove on the side walk), maximum route speed (if the vehicle did not surpass a certain velocity), and many more. The outcome is then calculated based in each criterion, and if all the criteria "Passed" then the result is a "Success", otherwise a "Failure" occurred.

7 EXPERIMENTAL SETUP

This section describes the simulation environment, sensor configuration, fault model, and injection methodology used to assess the impact of sensor faults on autonomous vehicle behavior.

7.1 Simulation Environment and Sensor Configuration

We distinguish between the simulation framework and the system under test. The framework comprises CARLA, the Scenario Runner orchestration, and the modified ROS bridge where faults are injected and logs are recorded. The system under test is the Autoware based autonomous driving stack used as the demonstration case. The framework delivers scenarios and perturbed sensor streams but does not implement perception, planning, or control logic. All observed behaviors therefore reflect the system under test and its configuration.

Town10 map from CARLA's map library is used as the simulation environment. This urban scenario offers a complex road network with multiple intersections, varied lane curvature, and diverse traffic conditions, making it suitable for testing sensor performance and autonomous decision-making under realistic conditions. To better match real-world traffic regulations and ensure consistency with map-based planning in Autoware, we manually added stop signs to the corresponding Lanelet2 map. This adjustment was necessary to reflect road rules not originally encoded in the base map and to trigger more meaningful behavior in the planning module.

The system under test follows the default TUMFTM sensor suite, which includes LiDAR, GNSS, and IMU. The camera was excluded because it was not functional in the Autoware version used, the traffic-light perception pipeline did not operate reliably as it was constantly failing to recognize traffic-light, similar difficulties are reported in the Autoware issue tracker and CARLA–Autoware discussions. To avoid perception instability, we restricted the sensor configuration to LiDAR/GNSS/IMU for this study and left camera re-integration to future work [107], [108]. No internal Autoware modules were modified and all parameters relevant to perception, localization, planning, and control were kept constant across runs. This configuration enables closed loop interaction between CARLA and Autoware so that sensor level faults can be injected at the framework layer and their effects on the behavior of the system under test can be observed.

7.2 Fault Model

The fault model is characterized by the following four dimensions:

- **Location** – the location of the fault refers to the specific sensor affected. In this case, 5 different locations were considered: LiDAR, GNSS, and the three components of the IMU (gyroscope, accelerometer, and orientation expressed as quaternion).
- **Type** – In this work, we considered two types of sensor faults:
  - Silent Sensor Failure: the sensor stops transmitting data entirely, producing no output. This condition can occur due to hardware disconnection, power loss, or software crashes.
  - Severe Noise: the sensor continues to produce data, but the values are corrupted by large random deviations beyond the normal operating range, making the output unreliable.
- **Trigger** – the trigger defines the moment when a fault is injected. In this work, we defined five fault triggers based on the moments the ego vehicle reaches specific locations within the scenario map: start line, first intersection, mid-route, last intersection, and near goal position. These are described in more detail in Section 7.4.
- **Duration** – the framework supports both transient faults, defined to last for a specific number of seconds, and permanent faults, which remain active until the end of the test run. In this work, we focused exclusively on permanent faults to simplify the test space and to evaluate the system under the most disruptive conditions.

7.3 Severe Noise Values per Fault Location

This section describes how the Severe Noise fault type values were defined for each fault location, meaning each specific sensor affected. Since sensors have different characteristics, the way noise is introduced must be adapted accordingly. To define appropriate values for severe noise, first we identified the expected nominal noise levels for each sensor. These nominal values represent the typical measurement variability under normal operating conditions, as specified by manufacturers or reported in the literature. While such noise is always present in real-world deployments, it does not represent a fault. Instead, it serves as a reference baseline, helping us determine what constitutes abnormal or faulty behavior. Based on these nominal values, we established the corresponding severe noise levels used in our fault model.

7.3.1 LiDAR

To emulate noise in LiDAR data, we apply Gaussian noise to each point of the LiDAR point cloud. This fault represents small, random disturbances that real LiDAR sensors experience due to electronic interference, surface reflectivity,

environmental conditions (e.g., dust, rain), or inherent sensor limitations [36], [109], [110].

The fault is implemented by adding a random noise to each coordinate (x, y, z) of every point in the cloud. The noise follows a Gaussian distribution centered around zero, meaning it introduces unbiased jitter around the true value. However, rather than focusing on absolute noise values (e.g., ±0.006 meters), the significance of the fault is evaluated based on its relative deviation from the true point location.

Coordinate deviations up to 2% of the point's range (distance from sensor) are considered within expected tolerance for high-quality LiDARs. This value is consistent with findings in [111], which report measurement errors within 1–2% for commercial terrestrial laser scanners operating under controlled conditions. Deviations between 2% and 10% are interpreted as severe deviations and translate into significant measurement corruption, likely leading to misperception, missed obstacles, or map inconsistencies. This percentage-based approach ensures that the added noise is significant, no matter the distance of the objects. For example, a 1 cm deviation is negligible for an object located 50 meters away, but the same deviation at only 0.2 meters corresponds to a large relative error, heavily altering the perceived geometry at close range. Furthermore, LiDAR returns are more sensitive at close range due to angular resolution and time-of-flight constraints, making small absolute changes more impactful.

By using this percentage-based approach, the simulation can capture both realistic sensor imperfections and fault-level disturbances, depending on the severity of the injected noise.

7.3.2 GNSS

To emulate positioning errors in a simulated autonomous system, we applied Gaussian noise into the GNSS readings, specifically affecting latitude, longitude, and altitude values. This type of fault replicates common disturbances encountered in real-world GNSS signals caused by atmospheric interference, satellite geometry (e.g., dilution of precision), multipath reflections, and receiver limitations [54], [68], [112].

A deviation of 0.00002 degrees both in latitude and longitude corresponds to approximately 2 meters of horizontal error. The altitude noise was set to 0.2 meters. These values align with the performance of single-frequency, civilian-grade GNSS receivers under open-sky conditions, where horizontal errors are typically under 3 meters [113]. To emulate noise that exceeds nominal values, we adopted a standard deviation of 0.0002 degrees for latitude and longitude, corresponding to around 10 meters of horizontal error. For altitude, a standard deviation of 2.0 meters was defined. These values simulate degraded conditions due to poor satellite visibility, urban canyons, or intentional interference, which can degrade accuracy to over 10 meters.

7.3.3 IMU

IMU faults are emulated by interfering separately with the following IMU measurements: angular velocity (gyroscope), linear acceleration (accelerometer), and orientation (quaternion). The purpose is to inject faults that are more severe than the natural variability and imperfections found in real-world IMUs due to mechanical, electronic, and environmental factors [112].

In commercial gyroscopes, bias instability and rate noise typically produce errors in the range of 0.01–0.05 °/s, depending on environmental conditions and sampling rate. For example, devices such as the Bosch BMI160 [114] or InvenSense MPU-6000 [115] show noise densities around 0.005-0.01 $°/s/\sqrt{Hz}$. A 5% deviation threshold corresponds well to the upper bound of expected operating noise for angular velocity measurements under normal conditions. A deviation between 5 and 50% in gyroscopic output could indicate that the system is significantly misestimating rotational speed, leading to severe orientation estimation errors when fused with accelerometer and GNSS data.

Accelerometer nominal deviations are commonly expressed in $\mu g/\sqrt{Hz}$. For instance, the ADXL345 [116] shows typical deviation densities of ~150 $\mu g/\sqrt{Hz}$. For acceleration signals in the range of 1-2 m/s^2 (e.g., steady cruising), a 5% deviation equates to $\pm$0.05-0.1 m/s^2, matching real-world fluctuations from road vibration or minor sensor inaccuracies. Noise values between 5 and 50% can seriously distort vehicle state estimation, especially when the signal-to-noise ratio is low, such as during smooth braking or slow turns.

Quaternions, derived from fused gyroscope and accelerometer data, are sensitive to noise in both sensors [117]. Minor noise (<5%) can be expected due to integration drift or transient instability in attitude estimation filters. In simulation, this would translate to slight deviations in yaw, pitch, or roll, typical of nominal system behavior. Quaternion errors can simulate the effects of drift caused by improper sensor fusion or misalignment in orientation. This can lead to cascading failures in localization and control if not properly corrected. A distortion between 5 and 50% in quaternion values is equivalent to significant misestimation of heading or tilt.

7.3.4 Summary of Noise Values per Location

This section summarizes the fault injection parameters used to emulate Severe Noise conditions in the different sensors, as well as presenting the nominal values, i.e., the noise that naturally occurs in every sensor, according to its specifications. Table 12 presents a structured overview of the Nominal values and the Severe Noise values per sensor.

Table 12 - Summary of noise values used in fault model

Location	Nominal Values	Severe Noise Values
IG - IMU – gyroscope	Deviation up to 5% of angular velocity	Deviation between 5% and 50% of angular velocity
IA - IMU – accelerometer	Deviation up to 5% of linear acceleration	Deviation between 5% and 50% of linear acceleration
IQ – IMU – Quaternion	Rotational disturbance up to 0.01 rad (~0.57°)	Disturbance of 0.2 rad (~11.5°)
LiDAR	Point deviation up to 2% of point range (distance from sensor)	Point deviation between 2% and 10% of point range
GNSS	±2 meters positional jitter (≈ ±0.00002°),	±20 meters jitter (≈ ±0.0002°)

7.4 Fault Trigger

To evaluate the impact of sensor faults under conditions that realistically challenge autonomous driving systems, we focused on scenarios that involve critical urban driving events. These include interactions with traffic control elements such as stop signs, the presence of vulnerable road users like pedestrians, and maneuvers requiring precise localization and control, such as turns, or lane keeping on straight segments. Such situations are representative of safety-critical cases where perception, planning, and control modules must operate reliably despite degraded sensor inputs.

Based on these requirements, the experimental campaign was structured using the Town10 map from the CARLA simulator. Town10 provides a rich urban environment with multiple intersections, crosswalks, and varied road geometries, making it suitable for systematically injecting sensor faults and observing their effects. Within this map, we defined five specific fault triggers, each corresponding to a distinct driving event along the ego vehicle's route. The ego vehicle used in all scenarios is a green van running Autoware.Universe version 2019.1.

Traffic lights present in the simulation environment were not considered, as the specific version of Autoware.Universe traffic light detection was not working. Therefore, vehicle behavior at intersections was governed solely by map-based rules, stop signs, and dynamic interactions with other road users.

As seen in Figure 27, the yellow line represents the route that the vehicle takes, and each blue point is a fault injection trigger. By organizing the experiments into these five fault triggers, the testing framework captures a diverse set of real-world driving events and enables a detailed analysis of sensor fault impacts across multiple autonomous vehicle functions.

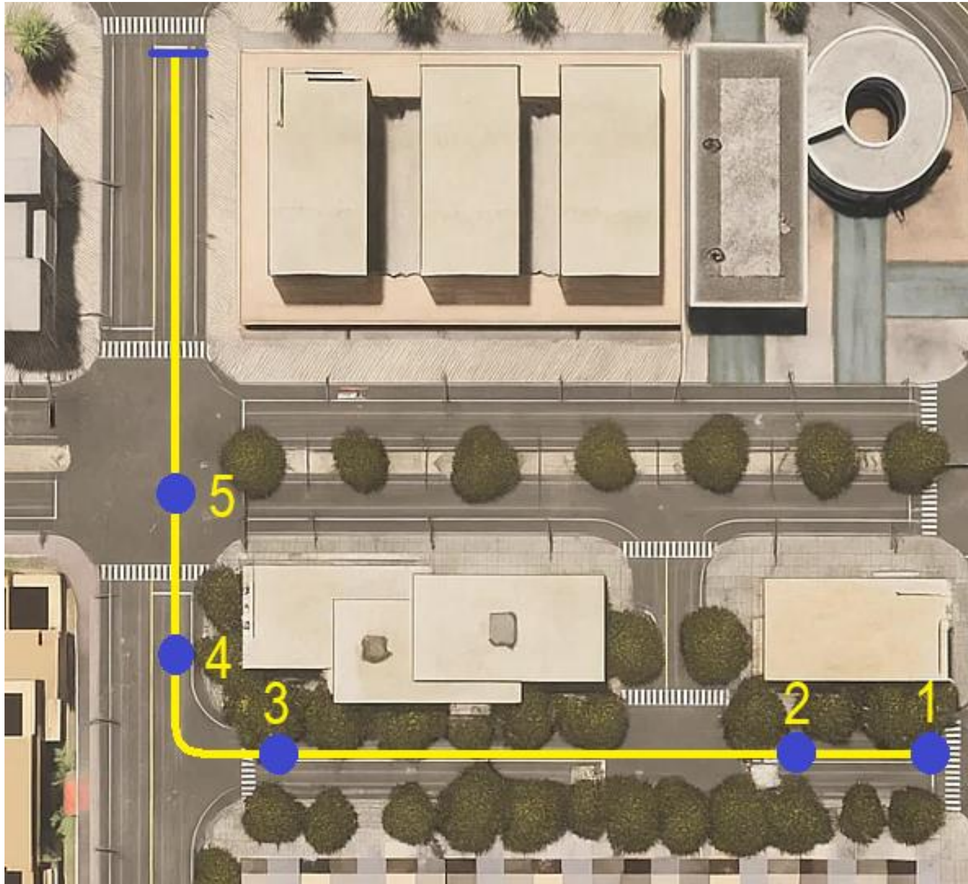

Figure 27 - Scenarios Route

7.4.1 Trigger 1: Starting Point

The vehicle begins its route in a straight road, requiring accurate lane keeping and localization. This initial phase occurs before reaching any traffic control elements and serves to validate basic navigation, localization, and sensor fusion during nominal driving. Sensor faults introduced here test the system's ability to maintain a stable trajectory in the absence of external constraints or decision points.

7.4.2 Trigger 2: Stop Sign

The vehicle approaches a stop sign and is required to halt and yield to a passing vehicle, and then it should wait for cross-traffic before proceeding. This scenario focuses on evaluating stop accuracy, map-based rule adherence, and cross-traffic awareness during controlled deceleration [118], [119].

7.4.3 Trigger 3: Right Turn at Intersection

The vehicle must execute a right turn at a T-junction. This maneuver tests the system's lateral control and path-following capabilities while navigating curved trajectories near road edges and potential obstacles [118], [119], [120].

7.4.4 Trigger 4: Pedestrian Crosswalk Encounter

At this point (see also Figure 28), the ego vehicle approaches a marked pedestrian crosswalk. A pedestrian actor is programmed to cross the street from the right side of the road as the ego vehicle nears the crosswalk. The objective is to evaluate the AV's perception and decision-making capabilities in response to dynamic pedestrian activity. The vehicle is expected to detect the pedestrian in advance, decelerate appropriately, and yield before the crossing zone. Sensor faults in this segment primarily challenge the detection of small and moving objects. Notably, faults affecting the LiDAR or camera may reduce detection confidence, while IMU or GNSS faults can affect precise positioning near the stop line.

Figure 28 - Fourth Trigger

7.4.5 Trigger 5: Final Intersection

In the final scenario, the vehicle reaches an intersection with no traffic signals, requiring it to yield to vehicles coming from the right. This situation stresses the vehicle's ability to assess right-of-way and respond correctly under partial perception failures.

7.5 Failure Mode Classification

To assess overall experiment results, we classify the vehicle's behavior on each fault injection run into four categories:

- Collision: The ego vehicle collides with another object, pedestrian, or road infrastructure.
- Out: The vehicle deviates from its designated lane or route but does not collide.

- Timeout: The vehicle fails to reach its destination within the time limit (3 minutes). It probably stopped, but did not collide or deviate from the designated lane or route
- OK: The vehicle completes the route without any incident.

The fault is tolerated only if the outcome of the run is OK, meaning that the mission was successfully accomplished, despite the injected fault. A Timeout means the fault was not tolerated, but the vehicle failed safely (no harm was done, i.e., no collision, nor lane/route departure). Out and Collision outcomes are both severe failures, since they can potentially result in harm to people, damage to the environment, or major loss of property.

7.6 Experimental Procedure

The mechanisms enabling fault injection were described in Section 6.3. This section explains instead the procedure followed to execute the experimental campaign in a structured and reproducible way.

A fault injection experiment consists of a series of test runs, each injecting a single fault, according to its location, type, trigger and duration.

A test runner iterates through the list of fault configurations, executing a separate test run for each one. After the test run, the outcomes, including criteria such as collisions, route completion, or traffic rule violations, are logged into a file for later analysis and results classification. The simulation environment (CARLA and Autoware) is then reset, the ego vehicle is positioned at the start position, the scenario is loaded, and the ego vehicle is started, launching the next test run. This ensures that every test begins in a consistent initial state, supporting structured and reproducible tests under the same initial conditions.

7.7 Experimental Setup Validation Process

All experiments run in synchronous simulation. Simulation time is logged, and all the messages across repeated runs and confirm that these traces are consistent. With injection disabled, ROS topics are captured immediately before and after the injection interface in the bridge and verify that payloads and message rates are identical.

Then test runs are executed with only nominal sensor noise. In these golden runs the system under test completes the scenario without incidents, which indicates that the framework does not introduce unintended disturbances. Repeating the golden runs yields the same outcome, which provides a reproducibility baseline for the experiments.

For the faults used in this study the expected signal effects are verified at the injection point. For silent failure, the post-injection topic rate drops to zero at activation and remains at zero until the end of the run. For severe noise, the post-injection stream shows the variance increase configured for each location. Activation occurs when the ego vehicle reaches the route positions and logs the trigger identifier and timestamps. The fault duration is permanent, until the end of each run.

These procedures demonstrate operational validity of the experimental infrastructure in nominal mode and correct application of the faults for the scenarios. They are not an exhaustive validation across all sensors, fault types, or environments.

8 EXPERIMENTAL EXECUTION AND RESULTS

This section presents the fault injection experimental results and analyzes the observed outcomes.

8.1 Golden Runs

Golden Runs are baseline tests executed to validate both the experimental setup and the fault injection mechanism. Their purpose is to confirm that the autonomous vehicle behaves correctly under nominal conditions, and to ensure that the fault injection mechanism does not introduce unintended side effects when no actual faults are activated.

In this context, nominal noise, meaning the expected variability naturally present in real sensor measurements, was explicitly introduced. This noise is not considered a fault, rather, it provides baseline which deviations can be measured. Applying nominal noise in the Golden Runs ensures that the framework is evaluated under conditions closer to real-world sensor operation, while maintaining the absence of fault-level disturbances.

An experimental campaign of 25 Golden Runs was conducted, combining nominal noise with the five defined fault locations and five fault triggers (see Table 12 and Table 13). As expected, Autoware successfully completed all runs without safety violations, route deviations, or abnormal behavior. These results confirm that the framework functions correctly in the absence of faults and that the injection mechanism itself does not interfere with normal system performance.

Table 13 - Golden run results

Location	Trigger 1	Trigger 2	Trigger 3	Trigger 4	Trigger 5
GNSS	OK	OK	OK	OK	OK
IMU – Accelerometer	OK	OK	OK	OK	OK
IMU – Gyroscope	OK	OK	OK	OK	OK
IMU – Quaternion	OK	OK	OK	OK	OK
LiDAR	OK	OK	OK	OK	OK

8.2 Experiment Execution and Analysis

Following the Golden Runs, the fault injection experiment was executed in a fully automated and repeatable manner. A total of 50 fault injection runs were executed, consisting of a combination of the 5 different Fault Locations (GNSS, LiDAR, IMU gyroscope, IMU accelerometer, and IMU quaternion), two fault types (Silent Failure and Severe Noise), and five injection triggers (Trigger 1 to Trigger 5). The use of these structured dimensions ensured consistency and coverage across relevant sensor failure modes and situations. Table 14 summarizes the results and provides an overall view of the results obtained by fault location, type, and trigger.

Table 14 - Test run results

Location	Type	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5
GNSS	Severe	OK	OK	OK	OK	OK
GNSS	Silent	OK	OK	OK	OK	OK
IMU - Accelerometer	Severe	OK	OK	OK	OK	OK
IMU - Accelerometer	Silent	OK	OK	OK	OK	OK
IMU - Gyroscope	Severe	Collision	Collision	Timeout	Collision	Out
IMU - Gyroscope	Silent	OK	OK	OK	OK	OK
IMU - Quaternion	Severe	OK	OK	OK	OK	OK
IMU - Quaternion	Silent	OK	OK	OK	OK	OK
LiDAR	Severe	Timeout	Timeout	Timeout	Timeout	Timeout
LiDAR	Silent	Collision	Collision	Collision	Collision	Collision

At a glance, the analysis of these 50 tests runs revealed patterns in how the applied faults affected the AV's behavior.

Regarding GNSS, Autoware showed resilience across all fault scenarios. Whether subjected to sensor silent failures or severe noise, Autoware reliably maintained accurate localization and safe navigation. This is expected for the tested stack, in which pose is produced by NDT LiDAR–map registration and an EKF that fuses

IMU and GNSS, which effectively compensate for GNSS inaccuracies or signal loss. In our urban route, LiDAR–map updates dominated the pose estimate, while GNSS acted as an initializer/low-authority absolute aid. Perturbing GNSS therefore did not change behavior in these scenarios. This should not be read as a general claim that GNSS is unimportant, since on open highways or in feature-poor areas, GNSS can have greater influence, however the scenario used in this study does not cover this case.

The tested configuration maintained stable control and accurate localization in all **IMU accelerometer** failures, including severe noise. With LiDAR–map pose updates anchoring the filter, accelerometer disturbances were down-weighted by the EKF and did not produce observable effects in these scenarios.

IMU Quaternion perturbations were absorbed without incident. With yaw dynamics driven primarily by the gyroscope and pose regularly corrected by LiDAR–map matching, moderate orientation noise in the fused quaternion did not destabilize planning or control in our runs.

The **LiDAR** sensor failure results revealed critical weaknesses in tested Autoware configuration design. Silent sensor failures (complete loss of LiDAR data) consistently led to immediate and severe failures, since the vehicle kept driving until colliding with other vehicles or pedestrians. In the other hand, Severe Noise LiDAR disturbances systematically stopped the car, leading to test timeouts, indicating Autoware's reliance on LiDAR-based perception and its limited capacity to effectively handle significant sensor degradation. However, no collision occurred, revealing fail-safe behavior from the vehicle.

Analyzing the **IMU gyroscope** results, the autonomous vehicle demonstrated strong resilience against silent sensor failures, consistently completing all scenarios successfully. However, under severe noise conditions of the IMU gyroscope, Autoware revealed significant vulnerabilities, producing erratic vehicle movements characterized by lateral instability. This reflects the tested stack's reliance on LiDAR for both perception and LiDAR–map localization, when the LiDAR signal is removed the system loses its primary pose/perception source, whereas strong degradation triggers fail-safe behavior. These severe disturbances frequently resulted in unsafe out-comes, such as collisions with pedestrians or obstacles, lane deviations, and timeouts due to the vehicle's inability to effectively plan or execute stable trajectories. To ensure data validity, additional tests were conducted in cases involving severe noise on the IMU gyroscope, with five extra runs performed for each trigger to account for the variability in vehicle behavior.

The repeated tests, in Table 15, further confirmed this instability, emphasizing the critical role of angular velocity information provided by the gyroscope for safe navigation and stable control of the vehicle. These results highlight the need for enhancing Autoware's fault detection mechanisms and redundancy strategies, specifically targeting the gyroscope's angular velocity data.

Table 15 - Additional gyroscope test results

Location	Type	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5
IMU - Gyroscope	Severe	Collision Collision Collision Out Timeout	Collision Collision Collision Out Timeout	Collision Collision Collision Out Out	Collision Collision Collision Collision Out	Collision Collision Out Timeout Timeout

In summary, the results of this experimental campaign provide clear guidance regarding the criticality and resilience of each sensor within Autoware's autonomous driving stack. The gyroscope and LiDAR sensors are particularly vulnerable to severe fault conditions, necessitating targeted improvements in fault detection, redundancy mechanisms, and enhanced fallback strategies. However, the accelerometer, quaternion orientation, and GNSS sensors demonstrated commendable resilience, effectively managed through Autoware's existing sensor fusion methodologies. These insights lay a foundation for future research directions, specifically aiming to reinforce the autonomous system's fault tolerance and overall safety performance under realistic sensor failure conditions.

The full list of tests, including the golden runs, are listed in Table B 1.

9 CONCLUSIONS AND FUTURE WORK

This work aim was to analyze how sensor faults affect the behavior of autonomous vehicles (AVs), using a simulation framework that integrates CARLA and Autoware. By combining controlled scenario execution with a structured fault model, the study demonstrated how fault injection can be used to systematically evaluate AV resilience under degraded sensing conditions. This study contributes in three main ways. First, a comprehensive survey of sensors used in AV perception systems was carried out and published as a scientific article (Annex A). This survey mapped sensor functionalities, common failure modes, and limitations, consolidating knowledge on the critical roles and vulnerabilities of LiDAR, GNSS, IMU, RADAR, and camera systems, and providing the foundation for the fault model used in this study. Second, the proposed fault injection framework enabled repeatable, scenario-based testing and revealed situations where specific faults consistently compromised vehicle safety in Autoware. Severe LiDAR noise and IMU gyroscope failures emerged as critical fault cases, while GNSS and other IMU components showed comparatively higher tolerance. Third, the research led to additional scientific outputs, including a second article currently under review, which further disseminates the findings and strengthens the contribution of this work to the community. Despite these contributions, the study has important limitations. The experiments were limited to a single map (Town10), which restricted environmental variability. Only permanent sensor faults were considered, leaving aside intermittent or progressive degradations that frequently occur in practice. Moreover, no fault detection or mitigation strategies were evaluated, even though such mechanisms are essential in production-grade AV systems. These constraints mean that the results should be interpreted as an initial exploration rather than a complete characterization of sensor resilience.

Future work should extend the fault model to include transient, progressive, or delayed failures, as well as systematic biases and latency effects. The evaluation of fault-tolerance mechanisms, such as runtime fault detection, multi-sensor redundancy, and adaptive control strategies, will be crucial to move from diagnosis to mitigation. Simulation runs should also incorporate adverse weather conditions, such as fog or heavy rain, to stress-test perception modules. Finally, complementing qualitative observations with quantitative safety metrics (deviation from path, breaking distance, failure recovery time) would allow for a more precise comparison of system performance under different fault scenarios.

This study shows that simulation-based fault injection is an effective method for identifying weak points in AV perception and control systems. The combination of a carefully defined fault model and a real-time autonomous stack (Autoware) revealed both vulnerabilities and resilience in a variety of scenarios. These findings highlight the importance of designing AV systems with robust error handling, multi-sensor redundancy, and scenario-based validation.

REFERENCES

[1] Ó. Silva, R. Cordera, E. González-González, and S. Nogués, “Environmental impacts of autonomous vehicles: A review of the scientific literature,” *Science of The Total Environment*, vol. 830, p. 154615, Jul. 2022, doi: 10.1016/j.scitotenv.2022.154615.

[2] F. Matos, J. Bernardino, J. Durães, and J. Cunha, “A Survey on Sensor Failures in Autonomous Vehicles: Challenges and Solutions,” *Sensors 2024, Vol. 24, Page 5108*, vol. 24, no. 16, p. 5108, Aug. 2024, doi: 10.3390/S24165108.

[3] G. Xie, Y. Li, Y. Han, Y. Xie, G. Zeng, and R. Li, “Recent Advances and Future Trends for Automotive Functional Safety Design Methodologies,” *IEEE Trans Industr Inform*, vol. 16, no. 9, pp. 5629–5642, Sep. 2020, doi: 10.1109/TII.2020.2978889.

[4] D. Rojas-Rueda, M. J. Nieuwenhuijsen, H. Khreis, and H. Frumkin, “Autonomous Vehicles and Public Health,” *Annu Rev Public Health*, vol. 41, no. 1, pp. 329–345, Apr. 2020, doi: 10.1146/annurev-publhealth-040119-094035.

[5] E. VINKHUYZEN and M. CEFKIN, “Developing Socially Acceptable Autonomous Vehicles,” *Ethnographic Praxis in Industry Conference Proceedings*, vol. 2016, no. 1, pp. 522–534, Nov. 2016, doi: 10.1111/1559-8918.2016.01108.

[6] T. Morita and S. Managi, “Autonomous vehicles: Willingness to pay and the social dilemma,” *Transp Res Part C Emerg Technol*, vol. 119, p. 102748, Oct. 2020, doi: 10.1016/j.trc.2020.102748.

[7] J. Wang, L. Zhang, Y. Huang, J. Zhao, and F. Bella, “Safety of Autonomous Vehicles,” *J Adv Transp*, vol. 2020, pp. 1–13, Sep. 2020, doi: 10.1155/2020/8867757.

[8] M. N. Ahangar, Q. Z. Ahmed, F. A. Khan, and M. Hafeez, “A Survey of Autonomous Vehicles: Enabling Communication Technologies and Challenges,” *Sensors*, vol. 21, no. 3, p. 706, Jan. 2021, doi: 10.3390/s21030706.

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

[10] M. A. Ramos, C. Correa Jullian, J. McCullough, J. Ma, and A. Mosleh, “Automated Driving Systems Operating as Mobility as a Service: Operational Risks and SAE J3016 Standard,” in *2023 Annual Reliability and Maintainability Symposium (RAMS)*, IEEE, Jan. 2023, pp. 1–6. doi: 10.1109/RAMS51473.2023.10088244.

[11] F. B. Scurt, T. Vesselenyi, R. C. Tarca, H. Beles, and G. Dragomir, “Autonomous vehicles: classification, technology and evolution,” *IOP Conf Ser Mater Sci Eng*, vol. 1169, no. 1, p. 012032, Aug. 2021, doi: 10.1088/1757-899X/1169/1/012032.

[12] E. Gat, “On Three-Layer Architectures”.

[13] “(PDF) Subsumption Control of a Mobile Robot.” Accessed: Aug. 14, 2025. [Online]. Available:

https://www.researchgate.net/publication/2875073_Subsumption_Control_of_a_Mobile_Robot

[14] "The 3T Intelligent Control Architecture | Download Scientific Diagram." Accessed: Aug. 14, 2025. [Online]. Available: https://www.researchgate.net/figure/The-3T-Intelligent-Control-Architecture_fig1_2851637

[15] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of Behavior Trees in robotics and AI," *Rob Auton Syst*, vol. 154, p. 104096, Aug. 2022, doi: 10.1016/J.ROBOT.2022.104096.

[16] C. E. García, D. M. Prett, and M. Morari, "Model predictive control: Theory and practice—A survey," *Automatica*, vol. 25, no. 3, pp. 335–348, May 1989, doi: 10.1016/0005-1098(89)90002-2.

[17] S. D. Pendleton *et al.*, "Perception, Planning, Control, and Coordination for Autonomous Vehicles," *Machines 2017, Vol. 5, Page 6*, vol. 5, no. 1, p. 6, Feb. 2017, doi: 10.3390/MACHINES5010006.

[18] G. Velasco-Hernandez, D. J. Yeong, J. Barry, and J. Walsh, "Autonomous Driving Architectures, Perception and Data Fusion: A Review," in *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*, IEEE, Sep. 2020, pp. 315–321. doi: 10.1109/ICCP51029.2020.9266268.

[19] D. Kumar and N. Muhammad, "A Survey on Localization for Autonomous Vehicles," *IEEE Access*, vol. 11, pp. 115865–115883, 2023, doi: 10.1109/ACCESS.2023.3326069.

[20] A. Chalvatzaras, I. Pratikakis, and A. A. Amanatiadis, "A Survey on Map-Based Localization Techniques for Autonomous Vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 8, no. 2, pp. 1574–1596, Feb. 2023, doi: 10.1109/TIV.2022.3192102.

[21] P. Karle, M. Geisslinger, J. Betz, and M. Lienkamp, "Scenario Understanding and Motion Prediction for Autonomous Vehicles - Review and Comparison," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 10, pp. 16962–16982, Oct. 2022, doi: 10.1109/TITS.2022.3156011.

[22] X. Wang, M. A. Maleki, M. W. Azhar, and P. Trancoso, "Moving Forward: A Review of Autonomous Driving Software and Hardware Systems," Nov. 2024, Accessed: Jun. 15, 2025. [Online]. Available: http://arxiv.org/abs/2411.10291

[23] Z. He, L. Nie, Z. Yin, and S. Huang, "A Two-Layer Controller for Lateral Path Tracking Control of Autonomous Vehicles," *Sensors 2020, Vol. 20, Page 3689*, vol. 20, no. 13, p. 3689, Jul. 2020, doi: 10.3390/S20133689.

[24] G. Chen, X. Zhao, Z. Gao, and M. Hua, "Dynamic Drifting Control for General Path Tracking of Autonomous Vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 8, no. 3, pp. 2527–2537, Mar. 2023, doi: 10.1109/TIV.2023.3235007.

[25] D. J. Yeong, G. Velasco-hernandez, J. Barry, and J. Walsh, "Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review," *Sensors 2021, Vol. 21, Page 2140*, vol. 21, no. 6, p. 2140, Mar. 2021, doi: 10.3390/S21062140.

[26] H. A. Ignatious, H.-E.- Sayed, and M. Khan, "An overview of sensors in Autonomous Vehicles," *Procedia Comput Sci*, vol. 198, pp. 736–741, 2022, doi: 10.1016/j.procs.2021.12.315.

[27] F. M. Ortiz, M. Sammarco, L. H. M. K. Costa, and M. Detyniecki, "Applications and Services Using Vehicular Exteroceptive Sensors: A Survey," *IEEE Transactions on Intelligent Vehicles*, vol. 8, no. 1, pp. 949–969, Jan. 2023, doi: 10.1109/TIV.2022.3182218.

[28] Z. Wang, Y. Wu, and Q. Niu, "Multi-Sensor Fusion in Automated Driving: A Survey," *IEEE Access*, vol. 8, pp. 2847–2868, 2020, doi: 10.1109/ACCESS.2019.2962554.

[29] S. AlZu'bi and Y. Jararweh, "Data Fusion in Autonomous Vehicles Research, Literature Tracing from Imaginary Idea to Smart Surrounding Community," in *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, IEEE, Apr. 2020, pp. 306–311. doi: 10.1109/FMEC49853.2020.9144916.

[30] E. N. Budisusila, M. Khosyi'in, S. A. D. Prasetyowati, B. Y. Suprapto, and Z. Nawawi, "Ultrasonic Multi-Sensor Detection Patterns On Autonomous Vehicles Using Data Stream Method," in *2021 8th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, IEEE, Oct. 2021, pp. 144–150. doi: 10.23919/EECSI53397.2021.9624313.

[31] V. Paidi, H. Fleyeh, J. Håkansson, and R. G. Nyberg, "Smart parking sensors, technologies and applications for open parking lots: a review," *IET Intelligent Transport Systems*, vol. 12, no. 8, pp. 735–741, Oct. 2018, doi: 10.1049/iet-its.2017.0406.

[32] J. Vargas, S. Alsweiss, O. Toker, R. Razdan, and J. Santos, "An Overview of Autonomous Vehicles Sensors and Their Vulnerability to Weather Conditions," *Sensors*, vol. 21, no. 16, p. 5397, Aug. 2021, doi: 10.3390/s21165397.

[33] F. Rosique, P. J. Navarro, C. Fernández, and A. Padilla, "A Systematic Review of Perception System and Simulators for Autonomous Vehicles Research," *Sensors*, vol. 19, no. 3, p. 648, Feb. 2019, doi: 10.3390/s19030648.

[34] R. Komissarov, V. Kozlov, D. Filonov, and P. Ginzburg, "Partially coherent radar unties range resolution from bandwidth limitations," *Nat Commun*, vol. 10, no. 1, p. 1423, Mar. 2019, doi: 10.1038/s41467-019-09380-x.

[35] Y. Li and J. Ibanez-Guzman, "Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems," *IEEE Signal Process Mag*, vol. 37, no. 4, pp. 50–61, Jul. 2020, doi: 10.1109/MSP.2020.2973615.

[36] M. Dreissig, D. Scheuble, F. Piewak, and J. Boedecker, "Survey on LiDAR Perception in Adverse Weather Conditions," in *2023 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, Jun. 2023, pp. 1–8. doi: 10.1109/IV55152.2023.10186539.

[37] D. Damodaran, S. Mozaffari, S. Alirezaee, and M. J. Ahamed, “Experimental Analysis of the Behavior of Mirror-like Objects in LiDAR-Based Robot Navigation,” *Applied Sciences*, vol. 13, no. 5, p. 2908, Feb. 2023, doi: 10.3390/app13052908.

[38] Y. Li, J. Moreau, and J. Ibanez-Guzman, “Emergent Visual Sensors for Autonomous Vehicles,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 5, pp. 4716–4737, May 2023, doi: 10.1109/TITS.2023.3248483.

[39] K. Roszyk, M. R. Nowicki, and P. Skrzypczyński, “Adopting the YOLOv4 Architecture for Low-Latency Multispectral Pedestrian Detection in Autonomous Driving,” *Sensors*, vol. 22, no. 3, p. 1082, Jan. 2022, doi: 10.3390/s22031082.

[40] C. Sun, Y. Chen, X. Qiu, R. Li, and L. You, “MRD-YOLO: A Multispectral Object Detection Algorithm for Complex Road Scenes,” *Sensors*, vol. 24, no. 10, p. 3222, May 2024, doi: 10.3390/s24103222.

[41] Y. Xie, L. Zhang, X. Yu, and W. Xie, “YOLO-MS: Multispectral Object Detection via Feature Interaction and Self-Attention Guided Fusion,” *IEEE Trans Cogn Dev Syst*, vol. 15, no. 4, pp. 2132–2143, Dec. 2023, doi: 10.1109/TCDS.2023.3238181.

[42] F. Altay and S. Velipasalar, “The Use of Thermal Cameras for Pedestrian Detection,” *IEEE Sens J*, vol. 22, no. 12, pp. 11489–11498, Jun. 2022, doi: 10.1109/JSEN.2022.3172386.

[43] Y. Chen and H. Shin, “Pedestrian Detection at Night in Infrared Images Using an Attention-Guided Encoder-Decoder Convolutional Neural Network,” *Applied Sciences*, vol. 10, no. 3, p. 809, Jan. 2020, doi: 10.3390/app10030809.

[44] M. Tan *et al.*, “Animal Detection and Classification from Camera Trap Images Using Different Mainstream Object Detection Architectures,” *Animals*, vol. 12, no. 15, p. 1976, Aug. 2022, doi: 10.3390/ani12151976.

[45] Y. Iwasaki, M. Misumi, and T. Nakamiya, “Robust Vehicle Detection under Various Environmental Conditions Using an Infrared Thermal Camera and Its Application to Road Traffic Flow Monitoring,” *Sensors*, vol. 13, no. 6, pp. 7756–7773, Jun. 2013, doi: 10.3390/s130607756.

[46] M. Bijelic *et al.*, “Seeing Through Fog Without Seeing Fog: Deep Multimodal Sensor Fusion in Unseen Adverse Weather,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2020, pp. 11679–11689. doi: 10.1109/CVPR42600.2020.01170.

[47] E. Yamaguchi, H. Higuchi, A. Yamashita, and H. Asama, “Glass Detection Using Polarization Camera and LRF for SLAM in Environment with Glass,” in *2020 21st International Conference on Research and Education in Mechatronics (REM)*, IEEE, Dec. 2020, pp. 1–6. doi: 10.1109/REM49740.2020.9313933.

[48] W. Shariff, M. S. Dilmaghani, P. Kielty, M. Moustafa, J. Lemley, and P. Corcoran, “Event Cameras in Automotive Sensing: A Review,” *IEEE Access*, vol. 12, pp. 51275–51306, 2024, doi: 10.1109/ACCESS.2024.3386032.

[49] A. Ceccarelli and F. Secci, "RGB Cameras Failures and Their Effects in Autonomous Driving Applications," *IEEE Trans Dependable Secure Comput*, vol. 20, no. 4, pp. 2731–2745, Jul. 2023, doi: 10.1109/TDSC.2022.3156941.

[50] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," Nov. 2017.

[51] K. Maciuk, "Determination of GNSS receiver elevation-dependent clock bias accuracy," *Measurement*, vol. 168, p. 108336, Jan. 2021, doi: 10.1016/J.MEASUREMENT.2020.108336.

[52] Ch. S. Raveena, R. S. Sravya, R. V. Kumar, and A. Chavan, "Sensor Fusion Module Using IMU and GPS Sensors For Autonomous Car," in *2020 IEEE International Conference for Innovation in Technology (INOCON)*, IEEE, Nov. 2020, pp. 1–6. doi: 10.1109/INOCON50539.2020.9298316.

[53] A. Yusefi, A. Durdu, F. Bozkaya, Ş. Tığlıoğlu, A. Yılmaz, and C. Sungur, "A Generalizable D-VIO and Its Fusion With GNSS/IMU for Improved Autonomous Vehicle Localization," *IEEE Transactions on Intelligent Vehicles*, vol. 9, no. 1, pp. 2893–2907, Jan. 2024, doi: 10.1109/TIV.2023.3316361.

[54] X. Xia, E. Hashemi, L. Xiong, A. Khajepour, and N. Xu, "Autonomous Vehicles Sideslip Angle Estimation: Single Antenna GNSS/IMU Fusion With Observability Analysis," *IEEE Internet Things J*, vol. 8, no. 19, pp. 14845–14859, Oct. 2021, doi: 10.1109/JIOT.2021.3072354.

[55] N. Goberville *et al.*, "Analysis of LiDAR and Camera Data in Real-World Weather Conditions for Autonomous Vehicle Operations," *SAE Int J Adv Curr Pract Mobil*, vol. 2, no. 5, pp. 2020-01–0093, Apr. 2020, doi: 10.4271/2020-01-0093.

[56] J. Raiyn, "Performance Metrics for Positioning Terminals Based on a GNSS in Autonomous Vehicle Networks," *Wirel Pers Commun*, vol. 114, no. 2, pp. 1519–1532, Sep. 2020, doi: 10.1007/s11277-020-07436-6.

[57] H. Jing, Y. Gao, S. Shahbeigi, and M. Dianati, "Integrity Monitoring of GNSS/INS Based Positioning Systems for Autonomous Vehicles: State-of-the-Art and Open Challenges," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 9, pp. 14166–14187, Sep. 2022, doi: 10.1109/TITS.2022.3149373.

[58] M. Kamal, A. Barua, C. Vitale, C. Laoudias, and G. Ellinas, "GPS Location Spoofing Attack Detection for Enhancing the Security of Autonomous Vehicles," in *2021 IEEE 94th Vehicular Technology Conference (VTC2021-Fall)*, IEEE, Sep. 2021, pp. 1–7. doi: 10.1109/VTC2021-Fall52928.2021.9625567.

[59] Z. Liu, L. Wang, F. Wen, and H. Zhang, "IMU/Vehicle Calibration and Integrated Localization for Autonomous Driving," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, May 2021, pp. 4013–4019. doi: 10.1109/ICRA48506.2021.9560767.

[60] B. Shahian Jahromi, T. Tulabandhula, and S. Cetin, "Real-Time Hybrid Multi-Sensor Fusion Framework for Perception in Autonomous Vehicles," *Sensors*, vol. 19, no. 20, p. 4357, Oct. 2019, doi: 10.3390/s19204357.

[61] F. Nobis, M. Geisslinger, M. Weber, J. Betz, and M. Lienkamp, "A Deep Learning-based Radar and Camera Sensor Fusion Architecture for Object Detection," in *2019 Sensor Data Fusion: Trends, Solutions, Applications (SDF)*, IEEE, Oct. 2019, pp. 1–7. doi: 10.1109/SDF.2019.8916629.

[62] Oleksandr Odukha, "How Sensor Fusion for Autonomous Cars Helps Avoid Deaths on the Road," https://intellias.com/sensor-fusion-autonomous-cars-helps-avoid-deaths-road/.

[63] R. F. Brena, A. A. Aguileta, L. A. Trejo, E. Molino-Minero-Re, and O. Mayora, "Choosing the Best Sensor Fusion Method: A Machine-Learning Approach," *Sensors*, vol. 20, no. 8, p. 2350, Apr. 2020, doi: 10.3390/s20082350.

[64] C. Xiang *et al.*, "Multi-Sensor Fusion and Cooperative Perception for Autonomous Driving: A Review," *IEEE Intelligent Transportation Systems Magazine*, vol. 15, no. 5, pp. 36–58, Sep. 2023, doi: 10.1109/MITS.2023.3283864.

[65] J. Fayyad, M. A. Jaradat, D. Gruyer, and H. Najjaran, "Deep Learning Sensor Fusion for Autonomous Vehicle Perception and Localization: A Review," *Sensors*, vol. 20, no. 15, p. 4220, Jul. 2020, doi: 10.3390/s20154220.

[66] S. Gu, Y. Zhang, J. Yang, J. M. Alvarez, and H. Kong, "Two-View Fusion based Convolutional Neural Network for Urban Road Detection," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, Nov. 2019, pp. 6144–6149. doi: 10.1109/IROS40897.2019.8968054.

[67] J. Alfred Daniel, C. Chandru Vignesh, B. A. Muthu, R. Senthil Kumar, C. Sivaparthipan, and C. E. M. Marin, "Fully convolutional neural networks for LIDAR–camera fusion for pedestrian detection in autonomous vehicle," *Multimed Tools Appl*, vol. 82, no. 16, pp. 25107–25130, Jul. 2023, doi: 10.1007/s11042-023-14417-x.

[68] L. Gao, X. Xia, Z. Zheng, and J. Ma, "GNSS/IMU/LiDAR fusion for vehicle localization in urban driving environments within a consensus framework," *Mech Syst Signal Process*, vol. 205, p. 110862, Dec. 2023, doi: 10.1016/j.ymssp.2023.110862.

[69] J. Kim, J. Kim, and J. Cho, "An advanced object classification strategy using YOLO through camera and LiDAR sensor fusion," in *2019 13th International Conference on Signal Processing and Communication Systems (ICSPCS)*, IEEE, Dec. 2019, pp. 1–5. doi: 10.1109/ICSPCS47537.2019.9008742.

[70] K. Banerjee, D. Notz, J. Windelen, S. Gavarraju, and M. He, "Online Camera LiDAR Fusion and Object Detection on Hybrid Data for Autonomous Driving," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, Jun. 2018, pp. 1632–1638. doi: 10.1109/IVS.2018.8500699.

[71] M. Pollach, F. Schiegg, and A. Knoll, "Low Latency And Low-Level Sensor Fusion For Automotive Use-Cases," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, May 2020, pp. 6780–6786. doi: 10.1109/ICRA40945.2020.9196717.

[72] X. Wang, K. Li, and A. Chehri, “Multi-Sensor Fusion Technology for 3D Object Detection in Autonomous Driving: A Review,” *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–18, 2023, doi: 10.1109/TITS.2023.3317372.

[73] X. Zhao, P. Sun, Z. Xu, H. Min, and H. Yu, “Fusion of 3D LIDAR and Camera Data for Object Detection in Autonomous Vehicle Applications,” *IEEE Sens J*, vol. 20, no. 9, pp. 4901–4913, May 2020, doi: 10.1109/JSEN.2020.2966034.

[74] M. Hasanujjaman, M. Z. Chowdhury, and Y. M. Jang, “Sensor Fusion in Autonomous Vehicle with Traffic Surveillance Camera System: Detection, Localization, and AI Networking,” *Sensors*, vol. 23, no. 6, p. 3335, Mar. 2023, doi: 10.3390/s23063335.

[75] I. Ogunrinde and S. Bernadin, “Deep Camera–Radar Fusion with an Attention Framework for Autonomous Vehicle Vision in Foggy Weather Conditions,” *Sensors*, vol. 23, no. 14, p. 6255, Jul. 2023, doi: 10.3390/s23146255.

[76] S. Yao *et al.*, “Radar-Camera Fusion for Object Detection and Semantic Segmentation in Autonomous Driving: A Comprehensive Review,” *IEEE Transactions on Intelligent Vehicles*, vol. 9, no. 1, pp. 2094–2128, Jan. 2024, doi: 10.1109/TIV.2023.3307157.

[77] J. D. Choi and M. Y. Kim, “A Sensor Fusion System with Thermal Infrared Camera and LiDAR for Autonomous Vehicles: Its Calibration and Application,” in *2021 Twelfth International Conference on Ubiquitous and Future Networks (ICUFN)*, IEEE, Aug. 2021, pp. 361–365. doi: 10.1109/ICUFN49451.2021.9528609.

[78] S. Wang *et al.*, “End-to-End Target Liveness Detection via mmWave Radar and Vision Fusion for Autonomous Vehicles,” *ACM Trans Sens Netw*, vol. 20, no. 4, pp. 1–26, Jul. 2024, doi: 10.1145/3628453.

[79] J. Shi, Y. Tang, J. Gao, C. Piao, and Z. Wang, “Multitarget-Tracking Method Based on the Fusion of Millimeter-Wave Radar and LiDAR Sensor Information for Autonomous Vehicles,” *Sensors*, vol. 23, no. 15, p. 6920, Aug. 2023, doi: 10.3390/s23156920.

[80] I. T. Kurniawan and B. R. Trilaksono, “ClusterFusion: Leveraging Radar Spatial Features for Radar-Camera 3D Object Detection in Autonomous Vehicles,” *IEEE Access*, vol. 11, pp. 121511–121528, 2023, doi: 10.1109/ACCESS.2023.3328953.

[81] W. Liu *et al.*, “SSD: Single Shot MultiBox Detector,” 2016, pp. 21–37. doi: 10.1007/978-3-319-46448-0_2.

[82] M. Roth, D. Jargot, and D. M. Gavrila, “Deep End-to-end 3D Person Detection from Camera and Lidar,” in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, IEEE, Oct. 2019, pp. 521–527. doi: 10.1109/ITSC.2019.8917366.

[83] S. Shah, D. Dey, C. Lovett, and A. Kapoor, “AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles,” *Springer Proceedings in Advanced Robotics*, vol. 5, pp. 621–635, 2018, doi: 10.1007/978-3-319-67361-5_40.

[84] G. Rong *et al.*, “LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving,” *2020 IEEE 23rd International Conference on Intelligent Transportation Systems, ITSC 2020*, Sep. 2020, doi: 10.1109/ITSC45102.2020.9294422.

[85] "GitHub - deepdrive/deepdrive: Deepdrive is a simulator that allows anyone with a PC to push the state-of-the-art in self-driving." Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/deepdrive/deepdrive?utm_source=chatgpt.com

[86] P. Cai, Y. Lee, Y. Luo, and D. Hsu, "SUMMIT: A Simulator for Urban Driving in Massive Mixed Traffic," *Proc IEEE Int Conf Robot Autom*, pp. 4023–4029, Nov. 2019, doi: 10.1109/ICRA40945.2020.9197228.

[87] Q. Li, Z. Peng, L. Feng, Q. Zhang, Z. Xue, and B. Zhou, "MetaDrive: Composing Diverse Driving Scenarios for Generalizable Reinforcement Learning," *IEEE Trans Pattern Anal Mach Intell*, vol. 45, no. 3, pp. 3461–3475, Sep. 2021, doi: 10.1109/TPAMI.2022.3190471.

[88] "Webots documentation: Webots for Automobiles." Accessed: Jun. 16, 2025. [Online]. Available: https://cyberbotics.com/doc/automobile/index

[89] E. Mumba, A. Sulaiman Gezawa, and C. Liu, "Enhanced autonomous driving within Webots simulation for student experiments," *Intelligent Service Robotics 2025*, pp. 1–21, Jun. 2025, doi: 10.1007/S11370-025-00608-Y.

[90] A. Amini *et al.*, "VISTA 2.0: An Open, Data-driven Simulator for Multimodal Sensing and Policy Learning for Autonomous Vehicles," *Proc IEEE Int Conf Robot Autom*, pp. 2419–2426, 2022, doi: 10.1109/ICRA46639.2022.9812276.

[91] S. Kato *et al.*, "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems," *Proceedings - 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018*, pp. 287–296, Aug. 2018, doi: 10.1109/ICCPS.2018.00035.

[92] H.-Y. Jung, D.-H. Paek, and S.-H. Kong, "Open-Source Autonomous Driving Software Platforms: Comparison of Autoware and Apollo," Jan. 2025, Accessed: Jun. 16, 2025. [Online]. Available: http://arxiv.org/abs/2501.18942

[93] L. Chen *et al.*, "Level 2 Autonomous Driving on a Single Device: Diving into the Devils of Openpilot," Jun. 2022, Accessed: Jun. 16, 2025. [Online]. Available: https://arxiv.org/abs/2206.08176v1

[94] G. Kaljavesi, T. Kerbl, T. Betz, K. Mitkovskii, and F. Diermeyer, "CARLA-Autoware-Bridge: Facilitating Autonomous Driving Research with a Unified Framework for Simulation and Module Development," *IEEE Intelligent Vehicles Symposium, Proceedings*, pp. 224–229, Feb. 2024, doi: 10.1109/IV55156.2024.10588623.

[95] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "AVFI: Fault Injection for Autonomous Vehicles," *Proceedings - 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W 2018*, pp. 55–56, Jul. 2018, doi: 10.1109/DSN-W.2018.00027.

[96] S. Jha *et al.*, "ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection," *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 112–124, Jun. 2019, doi: 10.1109/DSN.2019.00025.

[97] M. Maleki, A. Farooqui, and B. Sangchoolie, "CarFASE: A Carla-based Tool for Evaluating the Effects of Faults and Attacks on Autonomous Driving Stacks," *Proceedings - 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops Volume, DSN-W 2023*, pp. 92–99, 2023, doi: 10.1109/DSN-W58399.2023.00036.

[98] D. Hong and C. Moon, "Autonomous Driving System Architecture with Integrated ROS2 and Adaptive AUTOSAR," *Electronics 2024, Vol. 13, Page 1303*, vol. 13, no. 7, p. 1303, Mar. 2024, doi: 10.3390/ELECTRONICS13071303.

[99] Z. Lai, L. Le, V. Silva, and T. Bräunl, "A Comprehensive Comparative Analysis of Carla and Awsim: Open-Source Autonomous Driving Simulators," 2025, doi: 10.2139/SSRN.5096777.

[100] "Autoware Universe Documentation." Accessed: Jun. 16, 2025. [Online]. Available: https://autowarefoundation.github.io/autoware_universe/main/

[101] "Autoware Overview - Autoware." Accessed: Jul. 12, 2025. [Online]. Available: https://autoware.org/autoware-overview/

[102] K. W. Chiang, S. Srinara, S. Tsai, C. X. Lin, and M. L. Tsai, "High-Definition-Map-Based LiDAR Localization Through Dynamic Time-Synchronized Normal Distribution Transform Scan Matching," *IEEE Trans Veh Technol*, vol. 72, no. 6, pp. 7011–7023, Jun. 2023, doi: 10.1109/TVT.2023.3237275.

[103] A. Elmquist and D. Negrut, "Methods and Models for Simulating Autonomous Vehicle Sensors," *IEEE Transactions on Intelligent Vehicles*, vol. 5, no. 4, pp. 684–692, Dec. 2020, doi: 10.1109/TIV.2020.3003524.

[104] K. Lethander and C. Taylor, "Conservative estimation of inertial sensor errors using allan variance data," *Proceedings of the 34th International Technical Meeting of the Satellite Division of the Institute of Navigation, ION GNSS+ 2021*, pp. 2556–2564, 2021, doi: 10.33012/2021.17921.

[105] X. Fang, D. Song, C. Shi, L. Fan, and Z. Hu, "Multipath Error Modeling Methodology for GNSS Integrity Monitoring Using a Global Optimization Strategy," *Remote Sensing 2022, Vol. 14, Page 2130*, vol. 14, no. 9, p. 2130, Apr. 2022, doi: 10.3390/RS14092130.

[106] J. Kim, B. J. Park, and J. Kim, "Empirical Analysis of Autonomous Vehicle's LiDAR Detection Performance Degradation for Actual Road Driving in Rain and Fog," *Sensors (Basel)*, vol. 23, no. 6, p. 2972, Mar. 2023, doi: 10.3390/S23062972.

[107] "Cannot get the traffic_light_rois and the image raw in rviz2 when executing Autoware+AWSIM simulation · Issue #5567 · autowarefoundation/autoware_universe." Accessed: Aug. 13, 2025. [Online]. Available: https://github.com/autowarefoundation/autoware_universe/issues/5567?utm_source=chatgpt.com%3Futm_source%3Dchatgpt.com

[108] "Integrate Trafficlight Detection as an exemplary case for Town10. · Issue #8 · TUMFTM/Carla-Autoware-Bridge." Accessed: Aug. 13, 2025. [Online]. Available:

https://github.com/TUMFTM/Carla-Autoware-Bridge/issues/8?utm_source=chatgpt.com

[109] K. Burnett, A. P. Schoellig, and T. D. Barfoot, "Continuous-Time Radar-Inertial and Lidar-Inertial Odometry using a Gaussian Process Motion Prior", Accessed: Jun. 16, 2025. [Online]. Available: https://github.com/utiasASRL/steam

[110] C. Sun, P. Sun, J. Wang, Y. Guo, and X. Zhao, "Understanding LiDAR Performance for Autonomous Vehicles Under Snowfall Conditions," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–11, 2024, doi: 10.1109/TITS.2024.3409907.

[111] A. F. Habib, M. Al-Durgham, A. P. Kersting, and P. Quackenbush, "ERROR BUDGET OF LIDAR SYSTEMS AND QUALITY CONTROL OF THE DERIVED POINT CLOUD".

[112] X. Meng, H. Wang, and B. Liu, "A Robust Vehicle Localization Approach Based on GNSS/IMU/DMI/LiDAR Sensor Fusion for Autonomous Vehicles," *Sensors 2017, Vol. 17, Page 2140*, vol. 17, no. 9, p. 2140, Sep. 2017, doi: 10.3390/S17092140.

[113] P. K. Enge, "The Global Positioning System: Signals, measurements, and performance," *Int J Wirel Inf Netw*, vol. 1, no. 2, pp. 83–105, Apr. 1994, doi: 10.1007/BF02106512/METRICS.

[114] "BMI160 Datasheet | Enhanced Reader."

[115] "MPU-6000 and MPU-6050 Product Specification Revision 3.4 MPU-6000/MPU-6050 Product Specification," 2013.

[116] A. Devices, "ADXL345 (Rev. G)".

[117] A. M. Sabatini, "Quaternion-based extended Kalman filter for determining orientation by inertial and magnetic sensing," *IEEE Trans Biomed Eng*, vol. 53, no. 7, pp. 1346–1356, Jul. 2006, doi: 10.1109/TBME.2006.875664.

[118] X. Xiao, Y. Zhang, H. Li, H. Wang, and B. Li, "Camera-IMU Extrinsic Calibration Quality Monitoring for Autonomous Ground Vehicles," *IEEE Robot Autom Lett*, vol. 7, no. 2, pp. 4614–4621, Apr. 2022, doi: 10.1109/LRA.2022.3151970.

[119] J. P. Espineira, J. Robinson, J. Groenewald, P. H. Chan, and V. Donzella, "Realistic LiDAR with Noise Model for Real-Time Testing of Automated Vehicles in a Virtual Environment," *IEEE Sens J*, vol. 21, no. 8, pp. 9919–9926, Apr. 2021, doi: 10.1109/JSEN.2021.3059310.

[120] "IEEE Xplore Full-Text PDF:" Accessed: Jun. 16, 2025. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9925708

ANNEXES

ANNEX A. PUBLISHED PAPER

Matos, F., Bernardino, J., Durães, J., & Cunha, J. (2024). A Survey on Sensor Failures in Autonomous Vehicles: Challenges and Solutions. *Sensors*, *24*(16), 5108. https://doi.org/10.3390/s24165108

ANNEX B. EXPERIMENT RESULTS

Table B 1 - Experiment results

Fault ID	Location	Type	Trigger	Outcome
1	IMU - Gyroscope	Silent	Trigger 1	OK
2	IMU - Gyroscope	Silent	Trigger 2	OK
3	IMU - Gyroscope	Silent	Trigger 3	OK
4	IMU - Gyroscope	Silent	Trigger 4	OK
5	IMU - Gyroscope	Silent	Trigger 5	OK
6	IMU - Gyroscope	Noise	Trigger 1	OK
7	IMU - Gyroscope	Noise	Trigger 2	OK
8	IMU - Gyroscope	Noise	Trigger 3	OK
9	IMU - Gyroscope	Noise	Trigger 4	OK
10	IMU - Gyroscope	Noise	Trigger 5	OK
11	IMU - Gyroscope	Severe	Trigger 1	Collision
12	IMU - Gyroscope	Severe	Trigger 2	Collision
13	IMU - Gyroscope	Severe	Trigger 3	Timeout
14	IMU - Gyroscope	Severe	Trigger 4	Collision
15	IMU - Gyroscope	Severe	Trigger 5	Out
16	IMU - Accelerometer	Silent	Trigger 1	OK
17	IMU - Accelerometer	Silent	Trigger 2	OK
18	IMU - Accelerometer	Silent	Trigger 3	OK
19	IMU - Accelerometer	Silent	Trigger 4	OK
20	IMU - Accelerometer	Silent	Trigger 5	OK
21	IMU - Accelerometer	Noise	Trigger 1	OK
22	IMU - Accelerometer	Noise	Trigger 2	OK
23	IMU - Accelerometer	Noise	Trigger 3	OK
24	IMU - Accelerometer	Noise	Trigger 4	OK

Fault ID	**Location**	**Type**	**Trigger**	**Outcome**
25	IMU - Accelerometer	Noise	Trigger 5	OK
26	IMU - Accelerometer	Severe	Trigger 1	OK
27	IMU - Accelerometer	Severe	Trigger 2	OK
28	IMU - Accelerometer	Severe	Trigger 3	OK
29	IMU - Accelerometer	Severe	Trigger 4	OK
30	IMU - Accelerometer	Severe	Trigger 5	OK
31	IMU - Quaternion	Silent	Trigger 1	OK
32	IMU - Quaternion	Silent	Trigger 2	OK
33	IMU - Quaternion	Silent	Trigger 3	OK
34	IMU - Quaternion	Silent	Trigger 4	OK
35	IMU - Quaternion	Silent	Trigger 5	OK
36	IMU - Quaternion	Noise	Trigger 1	OK
37	IMU - Quaternion	Noise	Trigger 2	OK
38	IMU - Quaternion	Noise	Trigger 3	OK
39	IMU - Quaternion	Noise	Trigger 4	OK
40	IMU - Quaternion	Noise	Trigger 5	OK
41	IMU - Quaternion	Severe	Trigger 1	OK
42	IMU - Quaternion	Severe	Trigger 2	OK
43	IMU - Quaternion	Severe	Trigger 3	OK
44	IMU - Quaternion	Severe	Trigger 4	OK
45	IMU - Quaternion	Severe	Trigger 5	OK
46	LiDAR	Silent	Trigger 1	Collision
47	LiDAR	Silent	Trigger 2	Collision
48	LiDAR	Silent	Trigger 3	Collision
49	LiDAR	Silent	Trigger 4	Collision
50	LiDAR	Silent	Trigger 5	Collision
51	LiDAR	Noise	Trigger 1	OK

Fault ID	Location	Type	Trigger	Outcome
52	LiDAR	Noise	Trigger 2	OK
53	LiDAR	Noise	Trigger 3	OK
54	LiDAR	Noise	Trigger 4	OK
55	LiDAR	Noise	Trigger 5	OK
56	LiDAR	Severe	Trigger 1	Timeout
57	LiDAR	Severe	Trigger 2	Timeout
58	LiDAR	Severe	Trigger 3	Timeout
59	LiDAR	Severe	Trigger 4	Timeout
60	LiDAR	Severe	Trigger 5	Timeout
61	GNSS	Silent	Trigger 1	OK
62	GNSS	Silent	Trigger 2	OK
63	GNSS	Silent	Trigger 3	OK
64	GNSS	Silent	Trigger 4	OK
65	GNSS	Silent	Trigger 5	OK
66	GNSS	Noise	Trigger 1	OK
67	GNSS	Noise	Trigger 2	OK
68	GNSS	Noise	Trigger 3	OK
69	GNSS	Noise	Trigger 4	OK
70	GNSS	Noise	Trigger 5	OK
71	GNSS	Severe	Trigger 1	OK
72	GNSS	Severe	Trigger 2	OK
73	GNSS	Severe	Trigger 3	OK
74	GNSS	Severe	Trigger 4	OK
75	GNSS	Severe	Trigger 5	OK
76	IMU - Gyroscope	Severe	Trigger 1	Timeout
77	IMU - Gyroscope	Severe	Trigger 1	Collision
78	IMU - Gyroscope	Severe	Trigger 1	Out

Fault ID	Location	Type	Trigger	Outcome
79	IMU - Gyroscope	Severe	Trigger 1	Collision
80	IMU - Gyroscope	Severe	Trigger 1	Collision
81	IMU - Gyroscope	Severe	Trigger 2	Out
82	IMU - Gyroscope	Severe	Trigger 2	Collision
83	IMU - Gyroscope	Severe	Trigger 2	Collision
84	IMU - Gyroscope	Severe	Trigger 2	Collision
85	IMU - Gyroscope	Severe	Trigger 2	Timeout
86	IMU - Gyroscope	Severe	Trigger 3	Out
87	IMU - Gyroscope	Severe	Trigger 3	Collision
88	IMU - Gyroscope	Severe	Trigger 3	Out
89	IMU - Gyroscope	Severe	Trigger 3	Collision
90	IMU - Gyroscope	Severe	Trigger 3	Collision
91	IMU - Gyroscope	Severe	Trigger 4	Collision
92	IMU - Gyroscope	Severe	Trigger 4	Timeout
93	IMU - Gyroscope	Severe	Trigger 4	Collision
94	IMU - Gyroscope	Severe	Trigger 4	Collision
95	IMU - Gyroscope	Severe	Trigger 4	Timeout
96	IMU - Gyroscope	Severe	Trigger 5	Collision
97	IMU - Gyroscope	Severe	Trigger 5	Collision
98	IMU - Gyroscope	Severe	Trigger 5	Timeout
99	IMU - Gyroscope	Severe	Trigger 5	Timeout
100	IMU - Gyroscope	Severe	Trigger 5	Out

Instituto Superior
de Engenharia
Politécnico de Coimbra