vehicles

MDPI

Article

Implementation of Re-Simulation-Based Integrated Analysis System to Evaluate and Improve Autonomous Driving Algorithms

Soobin Jeon [1], Junehong Park [2] and Dongmahn Seo [1,*]

[1] School of Computer Software, Daegu Catholic University, Gyeongsan-si 38430, Republic of Korea; marsberry@cu.ac.kr
[2] Department of Computer Software, Daegu Catholic University, Gyeongsan-si 38430, Republic of Korea; jppark@cu.ac.kr
* Correspondence: sarum@cu.ac.kr; Tel.: +82-53-850-2755

Abstract: Autonomous driving technology requires rigorous testing and validation of perception, decision-making, and control algorithms to ensure safety and reliability. Although existing simulators and testing tools play critical roles in algorithm evaluation, they struggle to satisfy the demands of complex, real-time systems. This study proposes a re-simulation-based integrated analysis system designed to overcome these challenges by providing advanced visualization, algorithm-testing, re-simulation, and data-handling capabilities. The proposed system features a comprehensive visualization module for real-time analysis of diverse sensor data and ego vehicle information, offering intuitive insights to researchers. Additionally, it includes a flexible algorithm-testing framework that abstracts simulator-specific dependencies, enabling seamless integration and evaluation of algorithms in various scenarios. The system also introduces robust re-simulation capabilities, enhancing algorithm validation using iterative testing based on real-world or simulated sensor data. To address the computational demands of high-frequency sensor data, the system employs optimized data-handling mechanisms based on shared memory, thereby significantly reducing latency and improving scalability. The proposed system overcomes critical challenges faced by existing alternatives by providing a robust, efficient, and scalable solution for testing and validating autonomous-driving algorithms, ultimately accelerating the development of safe and reliable autonomous vehicles.

Keywords: autonomous driving analysis; simulation and re-simulation; data processing optimization

Citation: Jeon, S.; Park, J.; Seo, D. Implementation of Re-Simulation-Based Integrated Analysis System to Evaluate and Improve Autonomous Driving Algorithms. *Vehicles* **2024**, *6*, 2209–2227. https://doi.org/10.3390/vehicles6040108

Academic Editor: Deogratias Eustace

Received: 5 November 2024
Revised: 17 December 2024
Accepted: 21 December 2024
Published: 22 December 2024

1. Introduction

Autonomous driving technology is emerging as a core topic of research that is in active development in various fields, including automotive, drones, and robotics. Ongoing advancements in the primary technological domains within autonomous driving—perception, decision-making, and control—have made real-world validation possible. For instance, companies such as Waymo and Apollo have demonstrated the effectiveness of their autonomous driving technologies using real-world pilot services, which are regarded as pivotal examples of commercialization [1–3]. However, existing autonomous driving technology still cannot ensure complete driving safety, which has led to the postponement of commercialization goals for some companies and even complete abandonment of plans for full autonomy in some cases. The primary reason behind such delays is the direct relationship between autonomous driving technology and safety incidents, implying that passenger and pedestrian safety cannot be fully guaranteed in commercial deployments [4].

Ensuring safety in autonomous driving systems requires rigorous testing and validation of perception, decision-making, and control algorithms to reduce false positive rates and enhance reliability. Autonomous driving simulators, such as CARLA [5], Autoware [6], Gazebo [7], and Apollo [8], have been used widely to evaluate the performance of such

algorithms in controlled virtual environments. These simulators provide high-fidelity environments, accurate vehicle dynamics, sensor models, and realistic traffic scenarios, improving testing efficiency significantly. By enabling rapid generation and repeated testing of extreme scenarios that are difficult or unsafe to reproduce in real-world conditions, simulators address the long cycles, high resource demands, and low reproducibility of physical testing [9,10].

In addition to simulators, testing tools that leverage hardware-in-the-loop (HIL) and software-in-the-loop (SIL) frameworks have become the standard in validating and improving autonomous driving algorithms. HIL connects real-world hardware to simulated environments, enabling algorithm evaluation under realistic conditions, whereas SIL focuses on testing software in virtual environments, offering a cost-effective solution during early stage development [11–13]. These tools support open-loop and closed-loop testing, simulations, and real-time visualization.

Currently, a significant portion of research on autonomous driving relies on simulators and testing tools. However, these tools often struggle to address the demands of complex, real-time systems. They lack the ability to provide closed- and open-loop residuals, where the algorithm outputs are transmitted back into the simulation environment for iterative validation [9]. This limitation restricts the effectiveness of advanced algorithm testing owing to the difficulty of evaluating the dynamic influence of updates on subsequent simulation steps, making it challenging to enhance the reliability and performance of autonomous driving algorithms in complex scenarios. Additionally, researchers often rely on logged data from real-world scenarios or generated by simulators for the development and validation of algorithms. However, utilizing such data in algorithm testing typically requires either controlling the simulator using application programming interfaces (APIs) provided by the simulator or developing separate tools to process log data, which can be time-consuming and resource-intensive [11,13,14]. While large organizations may overcome such barriers by leveraging appropriate technical expertise and resources, smaller research teams or individual researchers face significant challenges.

Further, most simulators provide C++ or Python APIs for external control [14]; Python-based systems are widely used because of their rapid prototyping capabilities and rich API ecosystems. However, Python's Global Interpreter Lock (GIL) reduces multithreading efficiency [15], and multiprocessing introduces delays owing to inter-process communication (IPC) [16], creating bottlenecks while handling large-scale data. These challenges are particularly problematic during the processing of high-frequency data streams generated by modern sensors, underscoring the need for more flexible and optimized data handling tools and mechanisms.

To address the aforementioned challenges, this paper proposes a re-simulation-based integrated analysis system designed to streamline the testing and validation of algorithms for autonomous driving research. The main contributions of the proposed system are as follows.

- Re-simulation capability based on sensor data: The system exhibits re-simulation functionality that enables the evaluation of algorithms using sensor data collected from real-world scenarios or simulators. This capability enables the replay of logged sensor data, facilitating repeatable and consistent experiments. Specific scenarios, such as traffic incidents or edge cases, can also be recreated reliably to evaluate algorithm performance under challenging conditions (Section 3.1).
- Algorithm Testing Support: The proposed system provides an interface to support the development of flexible algorithms. This interface abstracts complexities associated with diverse data formats and simulator configurations, allowing developers to focus solely on the design and refinement of the algorithms. By decoupling the development process from specific environmental dependencies, such as the type of simulator or data structure, the system minimizes technical barriers and facilitates a streamlined algorithm-testing workflow (Section 3.2).

- Optimization of Data Processing Architecture: The proposed system employs a shared memory mechanism to optimize the handling of large-scale sensor data, reduce latency, and improve processing efficiency. By minimizing reliance on traditional IPC methods, such as data serialization and deserialization, the system eliminates unnecessary overhead and ensures seamless data flow. This approach enhances its ability to process high-frequency sensor streams, such as three-dimensional (3D) light detection and ranging (LiDAR) data exceeding 1 GB/s, without performance degradation. These optimizations are particularly advantageous for real-time applications, where timely data processing is critical for high accuracy and reliability of the algorithm (Section 3.3).
- Comprehensive Visualization: The proposed system provides real-time visualization tools that enable intuitive analysis of sensor data and algorithm outputs. These tools are designed to support various sensor modalities, including LiDAR, cameras, inertial measurement units, and ego vehicle information, and present data in both two-dimensional (2D) and 3D formats. The visualization module includes features such as real-time data rendering, customizable views, and integrated analysis (Section 4).

The proposed system addresses critical limitations of existing simulators and testing tools by integrating comprehensive visualization, flexible algorithm testing support, optimized data handling, and re-simulation capabilities. Based on these advancements, the proposed system not only reduces technical and resource barriers but also provides a streamlined and adaptive environment for testing and refining algorithms. This enables researchers to focus on algorithm development more effectively, accelerating the iteration process and facilitating the development of more efficient, reliable, and robust methodologies for autonomous driving systems.

The remainder of this paper is organized as follows. Existing simulators and testing tools are discussed in Section 2, highlighting their limitations. The structure of the proposed system is presented in Section 3, focusing on its support for algorithm testing and methods for optimizing data handling. The implemented system is introduced in Section 4, and experimental results are presented. Finally, the paper is concluded in Section 5 with a summary of the findings and future research directions.

2. Related Works

2.1. Overview of Existing Simulators for Autonomous Driving

In the rapidly evolving field of autonomous driving, rigorous testing and validation of algorithms have become critical to ensure the safety and reliability of self-driving systems. To satisfy these demands, various simulators and testing tools have been developed to enable researchers to evaluate algorithms in controlled and repeatable virtual environments. Simulators such as CARLA [5], Autoware [6], Gazebo [7], and LGSVL [17] have been widely adopted owing to their ability to replicate complex traffic scenarios, realistic sensor models, and vehicle dynamics. These tools, combined with frameworks such as HIL and SIL, bridge the gap between virtual simulations and real-world applications by integrating hardware components and refining algorithms through iterative testing. In this section, we investigate existing simulators, analyze the challenges that researchers face when utilizing these tools, and identify the research desideratum that this study addresses.

CARLA [5] provides urban driving simulations with customizable APIs, supports diverse sensor models, such as LiDAR and cameras, and enables scenario creation using its robot operating software (ROS) 1 [18] bridge. However, its visualization capabilities are limited and require external tools for detailed analysis. Autoware, an ROS-based framework, allows the visualization of sensor data and vehicle information, but relies on log-based data, limiting its flexibility for evaluation in custom scenarios [9].

Gazebo, which is compatible with ROS and ROS2, supports physics-based simulations and various sensor models but lacks the rendering quality required for high-fidelity urban simulations comparable to those of CARLA or AirSim [19]. LGSVL provides a robust simulation platform with features such as digital twin synchronization, enabling the integration of simulated and real-world vehicle data. However, the discontinuation of its services and

the additional costs associated with extended features pose challenges for its long-term usability. Apollo [10] enables modular algorithm testing and simulation of realistic traffic scenarios but faces challenges owing to its closed ecosystem. AirSim excels in high-quality rendering based on Unreal Engine but lacks advanced traffic management, limiting its applicability to complex urban environments.

Commercial platforms, such as Cognata [20], Autonomy [21], and NVIDIA DRIVE Constellation [22], provide high-fidelity simulations but are expensive and exhibit restricted source code access, limiting their accessibility to small-scale researchers.

Table 1 compares open-source and commercial analysis tools and simulation services. Most simulators provide extensive and flexible APIs for various autonomous driving applications. However, they exhibit zero or limited re-simulation (closed-loop or open-loop) capability and analysis functions for results obtained using new algorithms. Although simulators such as CARLA and Autoware partially provide simple code for re-simulation, their limited functionality limits their effective use. Consequently, additional testing tools are often required to analyze and evaluate autonomous driving algorithms thoroughly, thereby increasing the complexity and time required for experimentation. This highlights the need for a more integrated analysis system to streamline algorithm evaluation and validation.

Table 1. Comparison of autonomous driving analysis tools and simulation services.

Project	Open-Source	Data Visualization	SIL Support	HIL Support	Language	Algorithm Execution and Re-Simulation
Gazebo [7]	O	△	O	△	C++, Python	×
CARLA [5]	O	O	O	△	C++, Python	△
Autoware [9]	O	×	O	△	Python	△
Apollo [10]	O	×	O	△	C++, Python	×
LGSVL [17]	△	O	O	△	C#, Python	×
AirSim [19]	O	O	O	△	C++, C#, Python, Java, Matlab	×
Cognata [20]	×	O	O	O	N/A	×
Ansys Autonomy [21]	×	O	O	O	C++, Python	×
NVIDIA DRIVE constellation [22]	×	O	O	O	C++, Python	×
Ours	O	O	O	×	Python	O

O: Feature fully supported. △: Feature partially supported. ×: Feature not supported.

2.2. Challenges of Python-Based Re-Simulation

Testing tools for autonomous driving are increasingly utilizing HIL and SIL frameworks for algorithm validation and improvement. HIL connects real-world hardware with simulated environments, enabling realistic algorithm evaluation, whereas SIL focuses on testing software in virtual environments, supporting early stage development and cost-efficient testing [12,14]. Tools such as Siemens Simcenter Amesim and Prescan exemplify these approaches by offering detailed co-simulations of vehicle dynamics and environmental models.

Modern simulators, such as CARLA, Gazebo, and Autoware, integrate HIL and SIL frameworks to provide comprehensive testing environments. This integration combines realistic simulation scenarios with iterative hardware and software testing, addressing the increasing complexity of autonomous systems and enhancing the validation processes [11,12,14].

Python has become the preferred programming language for re-simulation tasks in autonomous driving research. One of its primary advantages is its ease of use, which enables it to support rapid prototyping. Its straightforward syntax and libraries, such as NumPy, Pandas, and PyTorch, enable quick implementation and algorithm iteration, thereby reducing development time. Additionally, Python's broad compatibility with leading simulators, including CARLA, Gazebo, and Autoware, simplifies the integration process and minimizes development overhead.

Despite these benefits, Python-based systems face critical challenges, particularly in handling high-frequency sensor data and supporting real-time re-simulation. One significant limitation is the Global Interpreter Lock (GIL), which restricts Python's ability to execute multiple threads concurrently. This limitation significantly hampers performance in parallelized environments where real-time data processing is required. Another challenge arises from the IPC overhead incurred when Python's multiprocessing framework is used to share data between processes. The serialization and deserialization required for IPC introduce substantial latency because large-scale sensor data, such as LiDAR and radar streams, require conversion into transferable formats for shared and reconstruction at the receiving end. This process not only increases the computational load but also reduces the overall efficiency in time-sensitive applications. Finally, Python's memory inefficiency stemming from its dynamic typing and high-level data structures leads to reduced scalability and slower processing speed in large-scale simulations [15,16].

These challenges profoundly affect re-simulation tasks, which involve replaying recorded scenarios to validate algorithms iteratively. High-frequency sensor data, such as LiDAR streams exceeding 1 GB/s, are particularly difficult to process owing to GIL and IPC constraints. Further, the synchronization of multimodal data, such as LiDAR, radar, and camera streams, requires precise coordination, which is computationally intensive and error-prone in Python-based systems. In addition, Python's memory inefficiencies hinder the scalability of re-simulations, limiting its effectiveness in complex traffic scenarios or on large datasets [16].

Several solutions have been proposed to overcome these limitations. One approach involves the optimization of performance-critical code sections via low-level integration using tools such as Cython3 or Pybind11, effectively bypassing Python's GIL and enhancing the processing speed [16]. GPU acceleration is another promising approach that allows computationally intensive tasks to be offloaded to GPUs using libraries such as CuPy13 or TensorRT10, which improves the performance of large-scale re-simulation tasks significantly [15]. Finally, leveraging vectorized operations provided by libraries such as NumPy and SciPy minimizes Python-level loops, enhancing overall efficiency and scalability.

Although these solutions address certain challenges, achieving real-time performance and scalability in Python-based systems remains difficult. To address this issue, we propose leveraging shared memory mechanisms to overcome IPC limitations and enhance data processing efficiency.

3. Structure of Re-Simulation-Based Integrated Analysis System

In this study, we propose a simulation-based real-time algorithm analysis system that addresses the limitations of existing simulators and testing tools in autonomous driving environments. This system provides autonomous driving developers with a hassle-free development environment that allows them to focus solely on designing and improving their algorithms. In addition, the system enables rapid and efficient algorithm development and validation by offering a Python-based evaluation framework.

The proposed system aims to achieve the following objectives:

- Support comprehensive visualization tools to enable intuitive analysis of diverse sensor data, algorithm outputs, and vehicle behavior in 2D and 3D environments;
- Provide a flexible algorithm-testing framework that abstracts simulator-specific dependencies, allowing seamless evaluation and refinement of algorithms in diverse scenarios;
- Enable re-simulation capabilities for replaying and testing real-world or simulated sensor data iteratively, ensuring consistent validation under controlled conditions;
- Improve data processing efficiency and scalability using optimized data-handling mechanisms, including reduced latency and support for high-frequency sensor streams.

First, as depicted in Figure 1, we provide a comprehensive visualization tool that enables real-time visualization of sensor and ego vehicle data measured in real-world scenarios or generated by simulators. Although this functionality may not differ significantly from existing simulator-based algorithm analysis tools, it focuses on delivering

as much variable information as possible in real time to enhance user experience and decision-making.

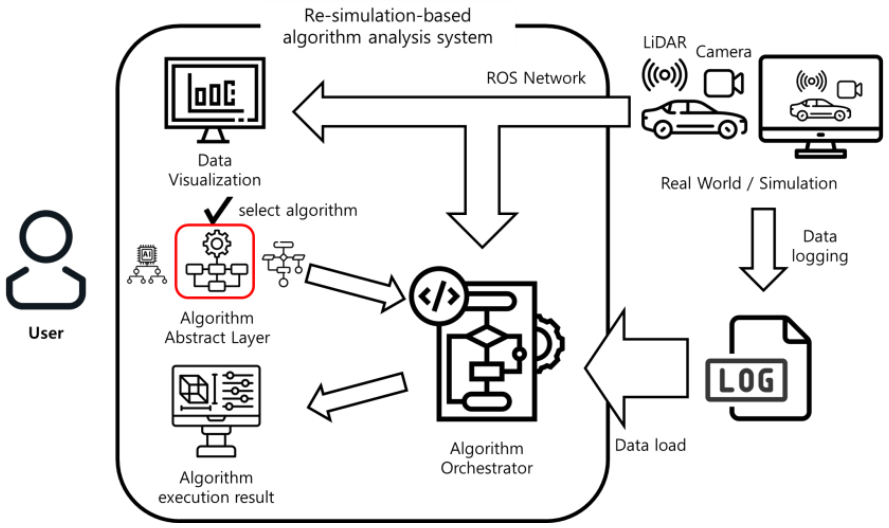

Figure 1. Overview of re-simulation based integrated analysis system.

Second, we provide an algorithm abstraction framework (AAF) that abstracts dependencies on specific simulators and data formats, enabling seamless integration and evaluation of algorithms in diverse scenarios. The AAF allows researchers to focus on algorithm development and refinement, promotes innovation, and reduces the complexity of adapting to various simulation environments.

Third, we enable re-simulation capabilities that facilitate the replay and iterative testing of real-world or simulated sensor data. This feature ensures the consistent validation of algorithm performance under controlled conditions, allowing researchers to recreate complex scenarios, identify potential weaknesses, and improve the reliability and robustness of algorithms.

Finally, we enhance data-processing efficiency and scalability by implementing optimized data-handling mechanisms tailored for high-frequency sensor streams. Constructed using Python3, the proposed system leverages its simplicity and extensive library ecosystem, which is widely favored in autonomous driving research. To overcome the limitations of Python, such as GIL and IPC overhead, shared memory mechanisms are employed to minimize latency and improve performance. This approach yields a scalable environment for the efficient processing of complex sensor data, allowing researchers to focus on algorithm development and validation without computational constraints. By addressing these challenges, this system can be expected to accelerate innovations in autonomous driving research.

3.1. Architecture of the Proposed System

The structure of the proposed system is illustrated in Figure 2. The Vehicle and Sensor layers manage various types of vehicle and sensor information abstractly. They include components such as an Actual Sensor Adapter for real-world sensors, a Virtual Sensor Adapter for simulated sensor data, a Controller for managing vehicle operations, and a Data Gather module for collecting relevant data from the sensors. The system utilizes ROS topics to transmit sensor data streams, including LiDAR point clouds, camera image frames, IMU data, and so on. Each topic is uniquely identified by a name and metadata such as timestamps and sensor IDs, ensuring synchronization across system modules. This mechanism facilitates real-time data visualization, logging, and re-simulation.

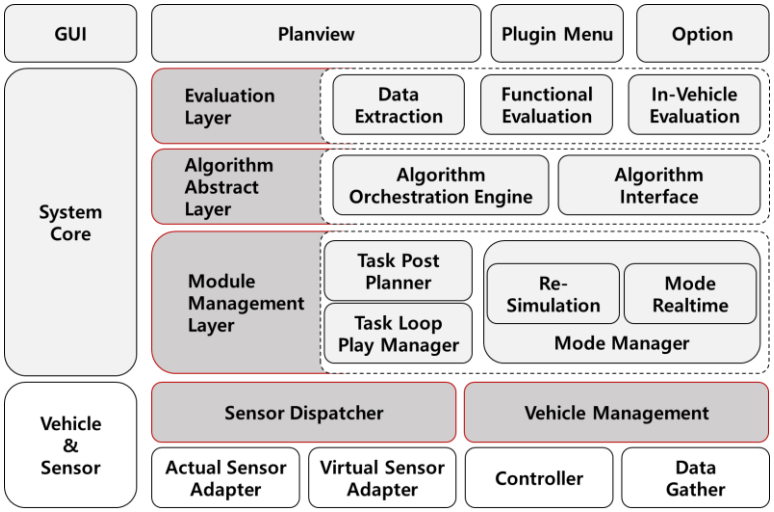

Figure 2. Re-simulation-based integrated analysis system architecture.

The System Core layer comprises three main sublayers: the Module Management Layer (MML), Algorithm Abstract Layer (AAL), and Evaluation Layer (EL). MML distributes tasks to threads and manages log data playback using Task Post Planner and Task Loop Play Manager, which create tasks based on the number of currently subscribed sensors. The Mode Manager within the MM defines the operational mode of the current system, enabling toggling between real-time data and log data playback modes. Additionally, Mode Simulation and Mode Realtime Play functions provide options for simulating or viewing data in real time.

The AAL, which includes the Algorithm Interface and Algorithm Orchestration Engine, facilitates the application of various algorithms and supports re-simulation. It enables the execution of algorithms on both logged data for re-simulation and real-time data, allowing researchers to view algorithm outputs in real time. The Algorithm Interface provides a programming interface (e.g., a class interface) that allows users to apply custom algorithms directly to the tool. This interface is categorized with respect to key areas of autonomous driving—perception, planning, and control—enabling users to receive relevant data and perform re-simulation using their algorithms within these domains. The Algorithm Orchestration Engine manages the execution and integration of multiple algorithms within the re-simulation pipeline. It ensures seamless data flow between different pipeline stages by standardizing data formats and coordinating inter-stage communication. By adapting to computational demands dynamically, it enables stable performance and efficient resource utilization across the perception, planning, and control stages.

The EL includes tools for assessing logged data, algorithm results, and other analysis metrics. It allows users to add or remove evaluation metrics based on their functional requirements. The system also supports real-time visualization and evaluation using real-time and logged data through data extraction, functional evaluations, and in-vehicle evaluations.

The GUI layer presents data, menus, and options to the user. This layer includes the Planview module, which visualizes point-cloud data in 2D or 3D form depending on the selected ROS topic or log data, as well as additional modules such as the Plugin Menu and Option for user configuration customization.

The proposed system architecture integrates various components to provide an efficient analysis environment, facilitating real-time data visualization, re-simulation, and algorithm testing with respect to multiple vehicles and control systems.

3.2. Algorithm Abstract Layer

AAL is designed to enable researchers to focus on algorithm development without worrying about experimental environments, such as simulators, sensors, or data structures. It includes an algorithm abstraction layer and a re-simulation pipeline to facilitate this

process. As depicted in Figure 3b, the re-simulation pipeline of AAL is divided into three stages corresponding to different types of autonomous driving algorithms—perception, planning, and control. The pipeline processes input data in a step-by-step manner, simulating conditions similar to those in actual autonomous driving environments. Each stage of the pipeline can either operate independently or run concurrently with the other stages, thereby providing flexibility for diverse research requirements. AAL also supports algorithm development using an abstract algorithm-based interface and the direct implementation of the resulting algorithms in the desired stage of the pipeline for testing and validation.

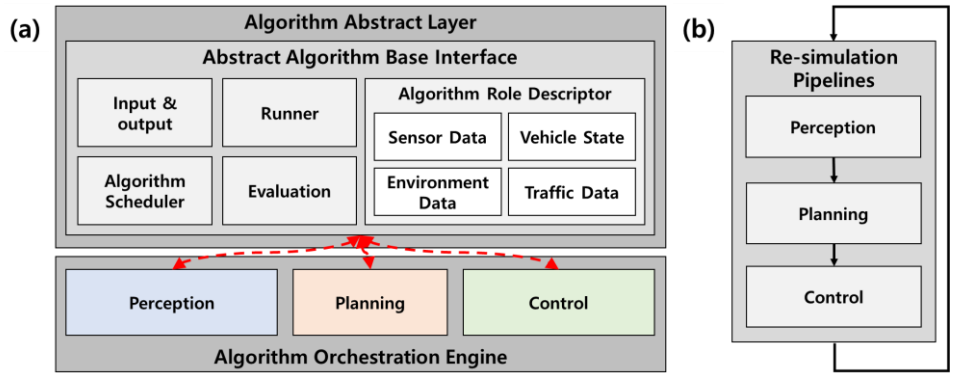

Figure 3. (**a**) Architecture of AAL, including Algorithm Interface and Orchestration Engine. (**b**) Re-simulation pipeline.

As depicted in Figure 3a, AAL supports autonomous driving algorithm development using an abstracted Algorithm Framework, which is further divided into the Abstract Algorithm Base (AAB) and the Algorithm Role Descriptor (ARD). AAB defines the essential functional units required for algorithms to operate within the re-simulation pipeline, serving as the foundational class that may be extended for specific implementations. The ARD acts as a descriptor that categorizes algorithms into one of the three pipeline stages: perception, planning, or control. Because each stage exhibits distinct input/output data requirements, execution procedures, and evaluation methods, AAL provides foundational definitions within these abstract classes. These definitions ensure consistency with respect to different stages while offering flexibility for developers to customize and extend the algorithm functionality to satisfy the unique demands of their autonomous driving scenarios.

The Algorithm Orchestration Engine (AOE) serves as the central component for managing the execution, coordination, and integration of algorithms within the re-simulation pipeline. It ensures smooth operation by overseeing workflow execution, enabling the seamless application of algorithms developed using AAB and ARD and optimizing resource allocation for efficient and reliable processing across all stages of the pipeline. AOE oversees the workflow coordination of algorithms across the three pipeline stages. It manages both sequential and parallel executions, ensuring that the output of one stage is correctly formatted and provided as input to the next stage, maintaining the logical flow of the pipeline. To support the integration and proper operation of algorithms developed based on AAB and ARD within the pipeline, AOE ensures seamless inter-stage communication. Leveraging a standardized interface, it facilitates the application of these algorithms, ensuring correct and efficient functionality within the re-simulation pipeline. Additionally, AOE ensures resource optimization by monitoring computational resource usage and adjusting execution priorities dynamically. This capability allows the system to handle high-frequency sensor data and intensive algorithm computations effectively, ensuring the fulfilment of real-time processing requirements without introducing bottlenecks. By integrating these functions, AOE ensures a robust, efficient, and scalable orchestration of the re-simulation pipeline.

Table 2 outlines the input and output data abstraction interfaces required for algorithm operation. The Input interface categorizes data into components based on their source and role within the algorithm pipeline. The Sensor Data component identifies input data from sensor modalities, such as LiDAR, cameras, and radar, and includes essential parameters, such as resolution, frame rate, and timestamps, to ensure consistency across algorithm stages. The Vehicle State component provides real-time updates on key parameters, such as position, velocity, acceleration, and heading, which are crucial for perception and planning processes. On the Output side, the Algorithm Results component delivers processed data, such as object detection outputs, planned paths, or control commands, tailored to the requirements of the downstream pipeline stages. The output structure is designed to maximize compatibility with other components, thereby facilitating seamless integration and efficient data flow. This abstraction ensures standardization of both input and output data, enabling algorithms to operate independently of specific sensor configurations or vehicle platforms, while simultaneously maintaining scalability and adaptability in diverse testing environments.

Table 2. Abstracting interface data for algorithm application via AAL.

Interface	Data Type	Fields and Descriptions
Input	SensorData	- sensor_id: Unique sensor ID - sensor_type: Enum [LiDAR, Camera, Radar, etc.] - resolution: Tuple [width, height] - frame_rate: Float - data: Array [Any] - timestamp: Float - sensor_location: Tuple [x, y, z]
	VehicleState	- position: Tuple [x, y, z] - velocity: Tuple [vx, vy, vz] - acceleration: Tuple [ax, ay, az] - heading: Float
	EnvironmentData	- weather: Enum [Sunny, Rainy, Foggy, Snowy, etc.] - lighting: Enum [Daylight, Night, Twilight, etc.] - road_condition: Enum [Dry, Wet, Icy, etc.]
	TrafficData	- traffic_lights: List [Tuple [id, state: Enum [Red, Yellow, Green]]] - nearby_vehicles: List [DetectedObject]
Output	DetectedObject	- object_id: Int - type: Enum [Vehicle, Pedestrian, Bicycle, etc.] - confidence: Float - bounding_box: Tuple [x, y, width, height] - position: Tuple [x, y, z] - velocity: Tuple [vx, vy, vz] - orientation: Tuple [yaw, pitch, roll] - acceleration: Tuple [ax, ay, az]
	PlannedPath	- waypoints: List [Tuple [x, y, z, timestamp]]
	ControlCommands	- throttle: Float - brake: Float - steering_angle: Float

3.3. Optimization of Data Processing Architecture Based on Shared Memory

Figure 4a depicts the data processing architecture of the proposed system. The data flow represents the entire process from the sensors to the System Core. The system leverages ROS topics for efficient transmission of sensor data streams, such as LiDAR and camera data. Each ROS topic contains structured information, including unique identifiers

and timestamps, enabling precise data synchronization and reducing processing delays during real-time operations. These data types are transmitted to ensure accurate synchronization and processing, supporting real-time visualization and re-simulation. The data flow represents the entire process from the sensors to the System Core. It includes the following components in sequence: (1) Sensor Data Scraper, (2) Data Logger, (3) Local File Loader, (4) Sensor Dispatcher for data processing, and (5) System Core. The architecture involves three main processes in the aggregate. Communication between these processes is achieved through pipes, with queues between each data transfer stage to buffer data.

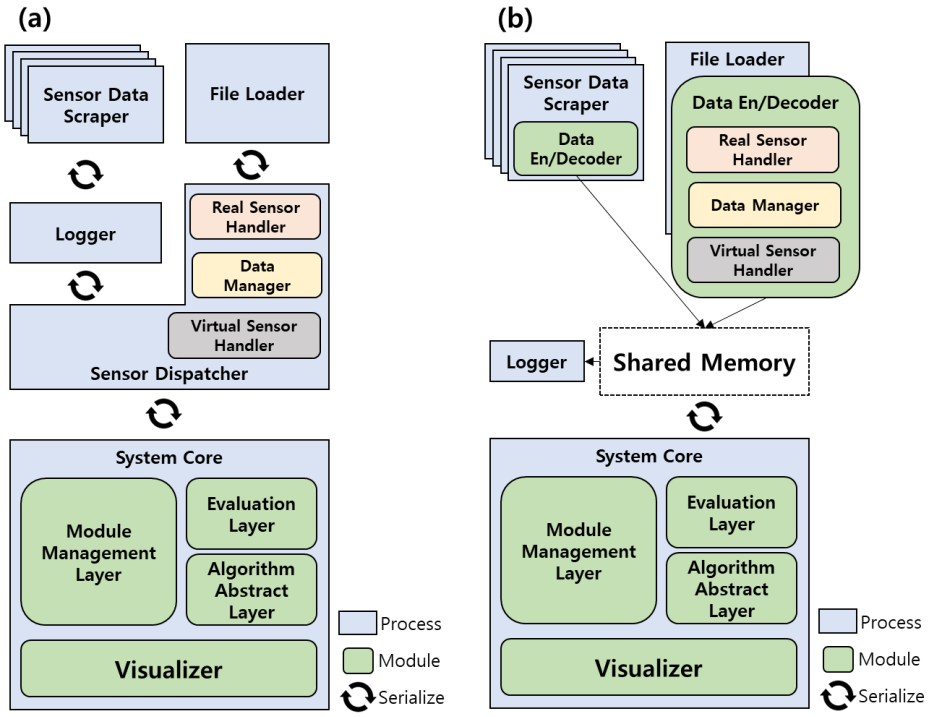

Figure 4. Optimized data processing: (**a**) legacy and (**b**) optimized.

In particular, Sensor Data Scraper (1) increases the number of processes dynamically depending on the types of sensor data being received. Only one of Sensor Data Scraper (1) and Local File Loader (3) operates at any given time based on the user's command mode (real-time data mode or replay mode). In this study, we modify Sensor Data Scraper to operate on the ROS platform, enabling the system to receive vehicle data from various systems. The Sensor Data Scraper interface also supports adaptability to various sensor data types, allowing the system to handle most sensor data without requiring user-defined additions.

System Core (5) comprises the Module Manager, Algorithm Enhancer, Evaluation, and Visualizer, which collectively manage real-time and re-simulation processes. This structure allows users to develop algorithms using the provided interfaces and validate them based on vehicle data within the analysis system.

As depicted in Figure 4a, the proposed system organizes the data processing pipeline into four distinct stages, and connects them sequentially to manage data flow efficiently. To validate the system, it is tested using four different sensors—3D LiDAR, 2D LiDAR, a camera, and an IMU. However, the test results reveal delays in processing sensor data, leading to algorithmic imbalance and visualization errors. Unnecessary processes within the data pipeline are hypothesized to have contributed to these delays, with the following root causes:

- Increase in sensor data volume and communication speed limit of the pipe;
- Python IPC and serialization/deserialization overhead;
- Unnecessary repetition in the data pipeline.

First, as the volume of sensor data inputs increases (approximately 1 GB/sec), the amount of data to be processed reaches the communication speed limit of the pipe, causing delays as the pipeline struggles to manage the high data throughput.

Second, in Python, IPC requires data serialization and deserialization (pickling/unpickling). Pickling converts Python objects into byte streams for storage or IPC. This process involves serializing and deserializing various data types, which is computationally intensive. With more extensive or more frequent data, pickling can impose an additional load on the CPU and memory, potentially slowing down overall data processing.

Third, the pipeline of the proposed system includes redundant steps, in which each process serializes and deserializes the same type of data (sensor data) unnecessarily. For instance, sensor data (e.g., labeled "A") are transmitted from Sensor Data Scraper to System Core without modification, yet repeated serialization and deserialization occur, even though the data for "A" remain unchanged. This redundancy wastes resources and disrupts dataflow synchronization.

To address problems with existing systems, this paper implements an optimized data processing architecture, as depicted in Figure 4b. To minimize unnecessary IPC, pipes are eliminated and shared memory is used for data transfer. Although the use of shared memory improves data processing efficiency by reducing the need for continuous data serialization and IPC, it introduces additional challenges related to memory management and data consistency.

The first issue arises from the fixed memory space required to handle data volume. Although this enables faster I/O by eliminating unnecessary communication processes, it involves constant memory consumption. The system modularizes the data processing steps to reduce the shared data volume, enabling the sensor data collection module to preprocess the data before storing them in shared memory. This approach minimizes the amount of data stored in shared memory, thereby improving memory utilization.

The second issue involves potential synchronization problems between processes due to the use of shared memory. In this system, data flow in a single direction—from the sensor collection process to the main control process. Data loss may occur if the main control process cannot maintain the processing speed of sensor data. To prevent this, we introduce a synchronization flag into the shared memory. The main control process checks and updates the flag based on the data ID to determine whether the latest data have been read. If the sensor data collector detects that the latest data in the shared memory have not been read, it temporarily stores the incoming data in a buffer. This mechanism is handled within the timing-update modules of the sensor data collection process and the main controller.

This optimized approach addresses the two primary challenges of fixed memory allocation and inter-process synchronization, thereby enhancing the overall efficiency and reliability of data handling in the proposed analysis system.

4. Implementation and Results

In Section 3, we described the architecture and design methodology of the proposed re-simulation-based integrated analysis tool for autonomous driving algorithms. This tool provides a foundational environment that enables quick and easy acquisition of sensor data and implementation of algorithms without environmental constraints. This section presents the implementation details and experimental results of the proposed analysis tool.

4.1. Implementation Results of the Analysis Tools

The analysis tool is developed using Python3, which is selected for its simplicity, extensive library ecosystem, and compatibility with autonomous vehicle analysis tools, such as ROS. Most simulators, such as CARLA and Autoware, use Python APIs, enabling rapid algorithm evaluation. In addition, the use of Python aligns with the design of AAL, providing an efficient and streamlined development environment for the creation and refinement of algorithms within the proposed framework.

Table 3 presents the development environment of the proposed integrated analysis system. The system implements a re-simulation tool based on Python and uses PyQt5, a graphical user interface (GUI) toolkit implemented as a Python plugin, to configure the GUI environment and visualize object data. A separate visualization library is used to render 3D spatial information data. Development took place in the Ubuntu 20.04 environment to support ROS, which is widely used in autonomous vehicle analysis tools.

Table 3. Development environment of simulation-based integrated analysis system.

Category	Technology/Tool	Version/Details
Programming Language	Python	3.8
Framework/ Middleware	ROS	Noetic Ninjemys
	PyQt	5.14.1
	Vispy	0.12.1 (3D Geographical Visualization)
	PyQtGraph	0.12.4 (2D Data Visualization)
Operating System	Ubuntu	20.04 LTS

The analysis tool interface consists of a top menu bar and a data visualization window. The top menu bar provides options for log data playback, ROS topic playback, and simulation controls, as shown in the unified Figure 5. The Camera Window in the top-left corner displays real-time video data or pre-recorded video data through the File, Simulation, and Re-simulation menus. LiDAR sensor data are displayed in the central Planview area, outputting point-cloud data in both 2D and 3D formats. The Object Visible drop-down menu at the bottom of the window allows users to specify the data to be visualized, while the Convert 2D Planview button converts 3D point-cloud data into a top-down view. The visualization integrates data from sensors such as cameras, LiDAR, and IMU, with point-cloud colors and positions determined by distance and reflectivity to enable detailed visualization of structures, buildings, and terrain. IMU data are used to align vehicle motion and orientation, ensuring accurate visualization of the surrounding environment.

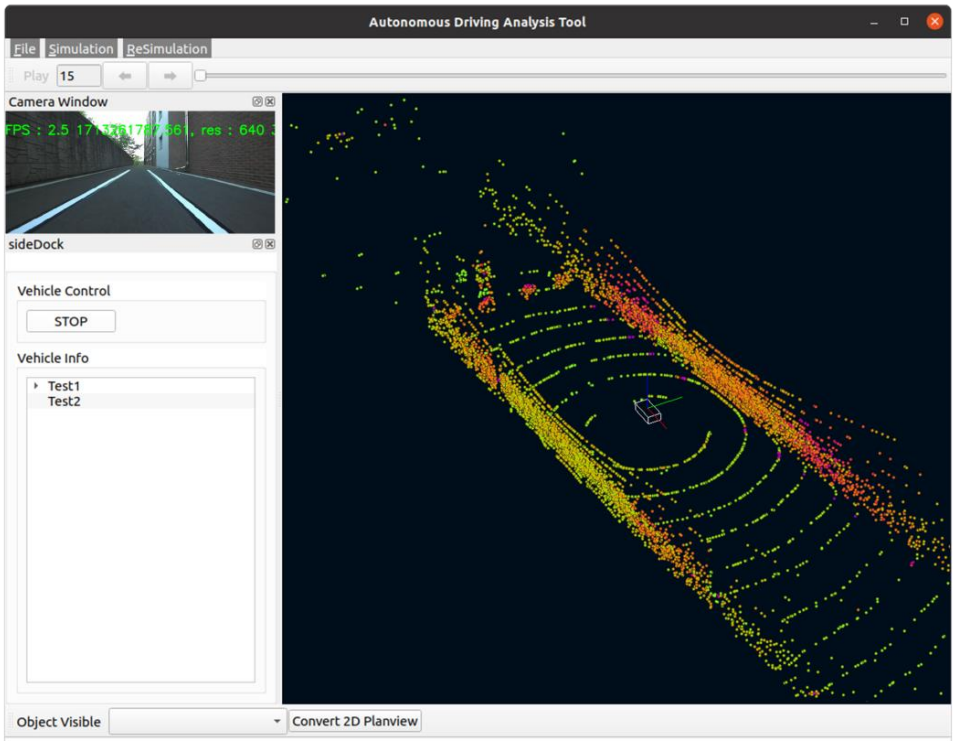

Figure 5. Main interface of the proposed analysis tool.

As depicted in Figure 6, an interface is provided to enable users to select the visualization topic from the currently published ROS topics. The box on the left displays a list of topics currently published in ROS. Topics that can be visualized using the analysis tools are highlighted in red. Clicking on a topic that can be visualized shifts the topic from the box to the right. Each of the ClearTopics, Run, and Run All Available Topics buttons initializes all topics currently being displayed, visualizes the selected topic, and displays all topics that can be visualized among the published topics. Using the log box at the bottom, users can check the status of the rospy library for using ROS data types in the current system, the status of the ROS Master for managing published topics, and the status of topics currently being visualized.

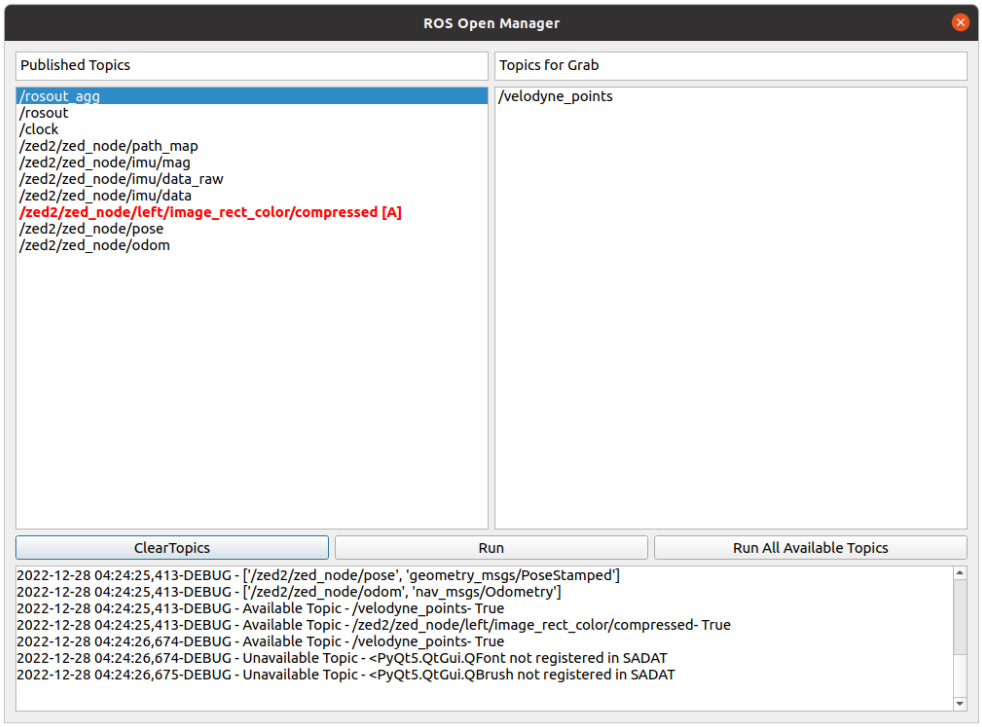

Figure 6. Interface to select ROS topics. Topics highlighted in red indicate currently active topics among those that the sensor can publish.

4.2. Algorithm Development

This section outlines the algorithm development process, emphasizing the design and implementation of the AAL. The AAL provides a modular and flexible framework that allows developers to efficiently integrate, test, and refine their algorithms. By decoupling algorithm logic from simulator-specific dependencies and data formats, the AAL ensures scalability, reusability, and consistency in algorithm development workflows.

This section covers the following aspects:

- AlgorithmBase Class: Introduction to the foundational abstract class that defines core functionalities, including sensor data processing, vehicle state updates, and algorithm execution.
- Example Implementation: Simple demonstration of the ObjectDetection Algorithm implementation, showcasing how the AlgorithmBase class can be used to process multiple sensor inputs and perform basic object detection tasks.
- System Integration: Explanation of how algorithms developed using the AAL are tested and evaluated within the re-simulation pipeline, ensuring their seamless integration into the system.

Figure 7 presents the AlgorithmBase class as the core component of the AAL. It provides a foundational framework for algorithm development by defining key functions

such as Process_sensor_data() for sensor data handling and Run_Algorithm() for implementing the algorithm logic. Developers can inherit and extend this class to specify the sensors to be used in Process_sensor_data() and define their custom algorithms within Run_Algorithm(). This approach ensures seamless integration and compatibility with the re-simulation pipeline. The output of the algorithm can be defined flexibly within the Run_Algorithm() function according to the developer's intent. The resulting outputs are stored in the output_data field, ensuring consistency in data management. Developers can utilize the output interface defined in Table 2 to standardize the output data types. This approach ensures seamless integration, compatibility with the re-simulation pipeline, and a flexible environment for algorithm development and testing.

```
CLASS AlgorithmBase:
    METHOD __init__():
        INITIALIZE vehicle_state, environment_data, traffic_data as None
        INITIALIZE sensor_data_list as an empty list
        INITIALIZE output_data as an empty list

    METHOD process_vehicle_state(vehicle_state):
        SET self.vehicle_state = vehicle_state

    ABSTRACT METHOD Process_sensor_data(sensor_data_list):
        // To be implemented: Receive and process multiple sensor data

    ABSTRACT METHOD Run_Algorithm():
        // To be implemented: Run the core algorithm logic

    METHOD run():
        IF vehicle_state, environment_data, traffic_data are not set:
            PRINT "Error: States are not initialized"
            RETURN
        CALL Run_Algorithm()

    METHOD Get_Algorithm_Output():
        RETURN output_data
```

Figure 7. AlgorithmBase Pseudocode Representing the AAL for Algorithm Development.

This Figure 8 illustrates a simple example implementation of the AlgorithmBase framework through the ObjectDetectionAlgorithm. The pseudocode demonstrates how multiple sensor inputs are processed to simulate object detection. This serves as a template for algorithm developers to implement their own algorithms for tasks such as perception, planning, and control, showcasing the flexibility and reusability of the AAL.

```
CLASS ObjectDetectionAlgorithm EXTENDS AlgorithmBase:
    METHOD Process_sensor_data(sensor_data_list):
        STORE sensor_data_list in self.sensor_data_list

    METHOD Run_Algorithm():
        PRINT "Running object detection algorithm..."

        FOR each sensor_data in self.sensor_data_list:
            CALL _detect_objects(sensor_data)

    PRIVATE METHOD _detect_objects(sensor_data):
        FOR i FROM 1 TO 3:
            CREATE a dummy DetectedObject with:
                object_id = Random ID
                obj_type = "Vehicle"
                confidence = Random value between 0.8 and 1.0
                bounding_box = Random coordinates
                position = sensor_data's location + offset
            ADD DetectedObject to self.output_data
```

Figure 8. ObjectDetectionAlgorithm pseudocode demonstrating an example implementation using AlgorithmBase.

Figure 9 depicts an interface that provides the ability to conduct simulations and check the results based on ROS logging data and implementation algorithms. The Load rosbag

File button loads the .bag files in the local path. The path to the selected file is displayed to the right of the button. The Algorithm Category and Test Algorithm drop-down menus below provide the ability to choose the type of algorithm to be applied as well as the detailed algorithm. The Algorithm Test button applies the selected algorithm to every frame of log data based on the selected log data file and algorithm, and creates a result file. Users can play and stop, frame-by-frame search, and section search by scrolling using the Replay/Pause, Step over, and Step down buttons below, as well as by scrolling horizontally.

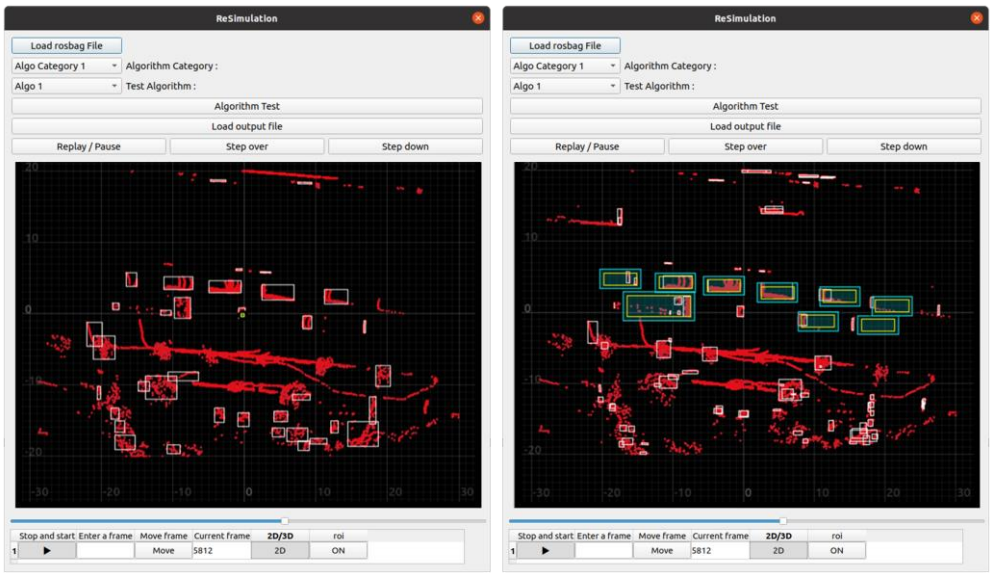

Figure 9. Re-simulation tool interface: algorithm results (**left**) and ground truth data (**right**) visualized side by side for performance comparison and export.

Algorithms developed with the AAL are tested and evaluated using the re-simulation pipeline. This pipeline runs the algorithms on recorded log data, visualizes the outputs, and compares them with ground truth data, ensuring smooth integration into the system. The left image displays the results of an executed algorithm, where bounding boxes are drawn based on the algorithm's output data. The right image combines these results with the ground truth data, enabling users to visually compare and evaluate the algorithm's performance. By leveraging the AAL, users can generate various types of outputs, such as detection results, which are immediately visualized on the interface in real time. Additionally, the generated output data can be exported in commonly used formats, including CSV and Excel, for further analysis and documentation.

The evaluation metrics are kept flexible, allowing users to define or select metrics that match their experimental goals. Common examples include Accuracy, Precision and Recall, Latency (Execution Time), and Root Mean Square Error (RMSE). The system does not enforce specific metrics, making it adaptable to various algorithms and use cases, and allowing users to design experiments that suit their specific needs.

4.3. Performance Evaluation of Data Processing Architecture

In this paper, a comparative data processing test is performed between existing systems and the proposed improved system to evaluate the effectiveness of the optimized data processing system in the implemented system. The experimental environment used for this analysis, including hardware and software configurations, is summarized in Table 4. For this purpose, three key metrics related to data handling are measured and analyzed—the average error between the data generated by Sensor Data Scraper and those processed using the existing data processing module, the average error between the data generated by Sensor Data Scraper and those processed by the improved data processing module, and the cumulative throughput error and decline rate.

Table 4. Experiment environment.

Parameter	Details
Experimental Method	Measurement of average processing error and cumulative throughput error
Key Metrics	- Average error with existing data processing module - Average error with improved data processing module - Cumulative throughput error and decline rate
Hardware	Nvidia Jetson Xavier, Velodyne VLP-16, RPLidar A3, Logitech C920 Cam, Razor 9DoF IMU
Software	ROS for data handling and ID assignment
Duration per Test	30 min
Total Repetitions	30 runs

For this analysis, the experimental configuration is taken to comprise the following components. The hardware consists of an Nvidia Jetson Xavier as the main processing unit, along with multiple sensors—a Velodyne VLP-16 for 3D LiDAR data, an RPLidar A3 for 2D LiDAR data, a Logitech C920 camera for visual data, and a Razor 9DoF IMU for motion and orientation data. The software platform is based on ROS, which handles sensor data acquisition and ID assignment processes.

In the experiment, each sensor assigns a unique ID to the data received by the sensor control unit and increases it with each new data entry. This ID assignment allows us to track the data flow throughout the system. The difference between the ID of the latest data at the sensor control unit and that of the data processed by the main control unit is recorded corresponding to each second. This ID difference reflects the delay in data processing between the sensor control and the main control units, providing a quantitative measure of system performance. Each test is conducted over 30 min with 30 repetitions to ensure statistical reliability.

Figure 10 shows the cumulative message loss for LiDAR topic messages over a 30-min period using the existing data processing module. The high loss rate highlights the inefficiencies in handling high-frequency sensor streams. This graph describes the cumulative difference in the number of data units between the Sensor Data Scraper and the Data Processing Modules over time. The *X*-axis represents time in seconds, whereas the *Y*-axis represents the difference in data count (number of data units).

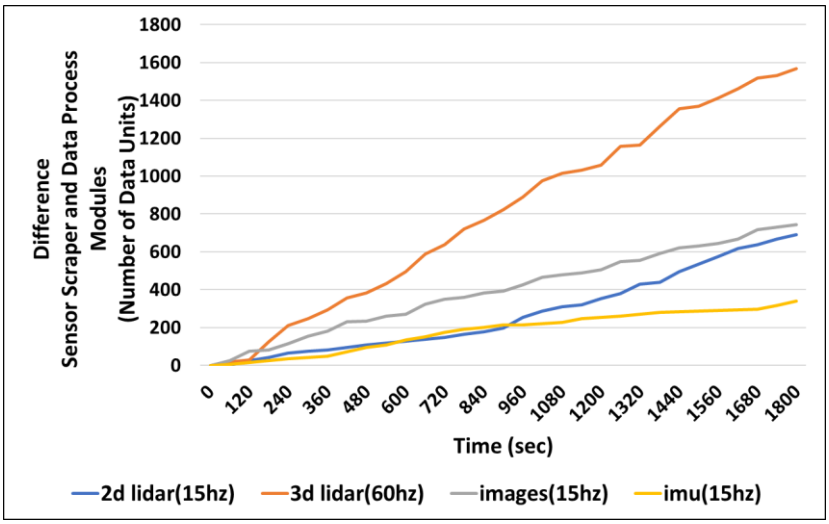

Figure 10. Average error with existing data processing module.

In Figure 10, 3D LiDAR (60 Hz) exhibits the highest error accumulation, with a steep increase in discrepancy over time. The 2D LiDAR (15 Hz), camera (15 Hz), and IMU (15 Hz) sensors also exhibit similar cumulative errors, albeit at lower rates. The delay in data processing becomes more significant with respect to all sensors, particularly for

high-frequency and high-data-volume 3D LiDAR data. This pattern suggests that the existing module struggles to process both high-frequency and large-volume data streams efficiently, particularly corresponding to 3D LiDAR. The results highlight the limitations of existing data processing modules in handling high-frequency sensors with substantial data sizes, leading to significant processing delays and potential data loss.

Figure 11 presents the cumulative message loss for LiDAR topic messages using the improved data processing module. The significant reduction in loss demonstrates the enhanced data handling capacity and reliability of the proposed system. As in Figure 11, this graph presents the difference in data count between the Sensor Data Scraper and the Data Processing Modules over time. The *X*-axis represents time in seconds, and the *Y*-axis represents the cumulative difference in data units, but on a much smaller scale, with the maximum error bounded by 3.5 data units.

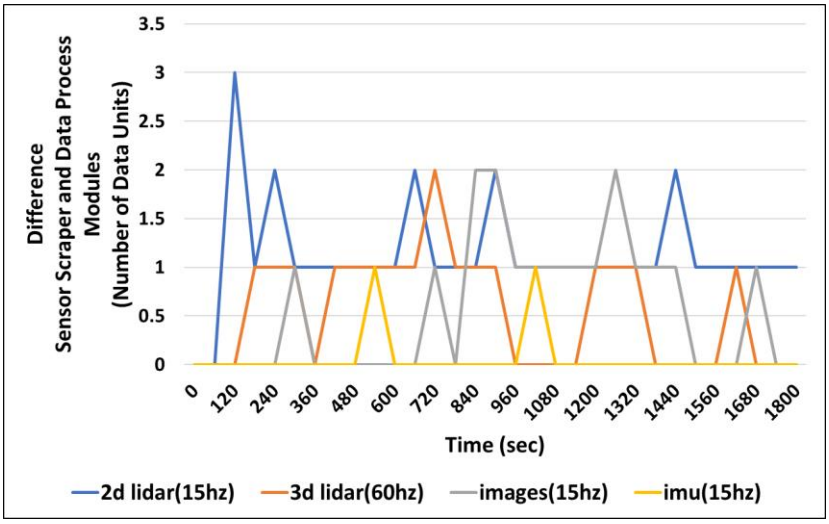

Figure 11. Average error observed with improved data processing module.

Most sensors are observed to exhibit minimal processing delays with the improved module, with the cumulative error remaining close to zero throughout the graph. Although minor discrepancies are observed occasionally, they are short-lived and non-recurrent, indicating that the optimized module can handle data more effectively. The improved module's ability to maintain low error rates for all sensors, even at higher frequencies, demonstrates its enhanced performance and stability compared to the existing module.

Table 5 lists the cumulative throughput errors and decline rates corresponding to various sensors yielded by the legacy and improved data processing modules. This table highlights performance differences in handling data obtained from the RPLidar A3, Velodyne VLP-16, Logitech C920, and 9DoF IMU sensors.

Table 5. Cumulative throughput error and decline rate.

	RPLidar A3	**Velodyne VLP-16**	**Logitech C920**	**9DoF IMU**
Existing	9560	24,889	12,250	5596
Improved	35	17	17	2
Decline rate	95.59%	99.93%	99.86%	99.96%

The experimental results confirm that the existing processing method produces up to 1500 errors depending on the sensor. The cumulative message loss of 1400 observed in the system represents the total number of missed LiDAR topic messages over a 30-min test period. This highlights the limitations of the legacy system in handling high-frequency sensor streams. In contrast, the improved processing technique reduces the error count

significantly, with most sensors recording fewer than 20 errors. The decline rate in cumulative throughput error between the legacy and improved modules further emphasizes the impact of the optimization. Velodyne VLP-16 exhibits a decline rate of 99.93%, RPLidar A3 exhibits a decline rate of 95.59%, Logitech C920 records a decline rate of 99.86%, and 9DoF exhibits a decline rate of 99.96%. This demonstrates the enhanced efficiency and reliability of the optimized data processing system in handling high-frequency and high-volume sensor data.

5. Conclusions

In this study, a re-simulation-based real-time algorithm analysis system is designed and implemented to address the limitations of existing simulators and testing tools in autonomous driving environments. Building on existing analysis tools and simulations that partially support algorithm application and reproducibility, the proposed system implements these functions in complete detail, addressing the desideratum left by existing solutions.

To address the issues identified in the data processing pipeline, including delays caused by increased sensor data volume, communication speed limits, and Python IPC overheads, the system is completely restructured to minimize IPC and reduce serialization/deserialization overhead. In particular, a shared memory area is designated to replace certain IPCs, thereby enhancing data flow efficiency. In addition, the pipeline is streamlined by eliminating unnecessary repetitions during data handling stages. These optimizations are observed to yield significant improvements in processing speed and reduce discrepancies between live sensor data and visualized outputs, thereby enhancing the overall system performance and reliability.

In future works, we intend to implement a closed-loop simulation that allows algorithm operation results obtained from the integrated analysis tool to be transmitted back into the log data. This enhancement will enable simulations to incorporate outcomes from avoidance driving and route planning algorithms, thereby providing a more comprehensive environment for testing and evaluation of autonomous driving technologies.

Author Contributions: Conceptualization, S.J. and D.S.; methodology, S.J.; software, S.J. and J.P.; validation, S.J., J.P. and D.S.; formal analysis, S.J.; investigation, S.J.; resources, S.J.; writing—original draft preparation, S.J., D.S. and J.P.; writing—review and editing, S.J.; supervision. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Acknowledgments: This research was supported by the "Regional Innovation Strategy (RIS)" through the National Research Foundation of Korea (NRF), funded by the Ministry Education (MOE) (2022RIS-006).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Rajabli, N.; Alauddin, F.; Hossain, M.S.; Muhammad, G. Software verification and validation of safe autonomous cars: A systematic literature review. *IEEE Access* **2020**, *9*, 4797–4819. [CrossRef]
2. Chen, L.; Wu, P.; Chitta, K.; Jaeger, B.; Geiger, A.; Li, H. End-to-end autonomous driving: Challenges and frontiers. *IEEE Trans. Pattern Anal. Mach. Intell.* **2023**, arXiv:2306.16927. [CrossRef] [PubMed]
3. Kim, S.W.; Philion, J.; Torralba, A.; Fidler, S. DriveGAN: Towards a controllable high-quality neural simulation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Nashville, TN, USA, 20–25 June 2021; pp. 5820–5829.
4. Kwak, M. Safety Design, Verification, and Validation Technology of Autonomous Shuttle System. *AUTO JOURNAL: J. Korean Soc. Automot. Eng.* **2022**, *44*, 26–31.
5. Dosovitskiy, A.; Ros, G.; Codevilla, F.; Lopez, A.; Koltun, V. CARLA: An open urban driving simulator. In Proceedings of the Conference on Robot Learning, Mountain View, CA, USA, 13–15 November 2017; pp. 1–16.

6. Kato, S.; Tokunaga, S.; Maruyama, Y.; Maeda, S.; Hirabayashi, M.; Kitsukawa, Y.; Azumi, T. Autoware on board: Enabling autonomous vehicles with embedded systems. In Proceedings of the 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS), Porto, Portugal, 11–13 April 2018; IEEE: New York, NY, USA, 2018; pp. 287–296.
7. Koenig, N.; Howard, A. Design and Use Paradigms for Gazebo: An Open-source Multi-Robot Simulator. In Proceedings of the International Conference on Intelligent Robots and Systems, Sendai, Japan, 28 September–2 October 2004.
8. Apollo. Available online: https://developer.apollo.auto/index.html (accessed on 27 October 2024).
9. Zhang, T.; Liu, H.; Wang, W.; Wang, X. Virtual Tools for Testing Autonomous Driving: A Survey and Benchmark of Simulators, Datasets, and Competitions. *Electronics* **2024**, *13*, 3486. [CrossRef]
10. Sohrabi, S.; Khodadadi, A.; Mousavi, S.M.; Dadashova, B.; Lord, D. Quantifying the automated vehicle safety performance: A scoping review of the literature, evaluation of methods, and directions for future research. *Accid. Anal. Prev.* **2021**, *152*, 106003. [CrossRef] [PubMed]
11. Fan, W.; He, H.; Lu, B. Online Active Set-Based Longitudinal and Lateral Model Predictive Tracking Control of Electric Autonomous Driving. *Appl. Sci.* **2021**, *11*, 9259. [CrossRef]
12. Silva, I.; Silva, H.; Botelho, F.; Pendão, C. Realistic 3D Simulators for Automotive: A Review of Main Applications and Features. *Sensors* **2024**, *24*, 5880. [CrossRef] [PubMed]
13. Chen, Q.; Yang, S.; Du, S.; Tang, T.; Chen, P.; Huo, Y. LiDAR-GS: Real-time LiDAR Re-Simulation Using Gaussian Splatting. *arXiv* **2024**, arXiv:2410.05111.
14. Son, T.D.; Bhave, A.; Van der Auweraer, H. Simulation-Based Testing Framework for Autonomous Driving Development. In Proceedings of the 2019 IEEE International Conference on Mechatronics (ICM), Ilmenau, Germany, 18–20 March 2019; IEEE: New York, NY, USA, 2019; pp. 576–583.
15. Woo, J.; Choi, H.; Lee, J. Empirical Performance Analysis of Collective Communication for Distributed Deep Learning in a Many-Core CPU Environment. *Appl. Sci.* **2020**, *10*, 6717. [CrossRef]
16. Jones, M.; Hurck, P.; Phelps, W.; Salgado, C.W. PyPWA: A Software Toolkit for Parameter Optimization and Amplitude Analysis. *Nucl. Instrum. Methods Phys. Res. Sect. A* **2024**, *1062*, 169150. [CrossRef]
17. Rong, G.; Shin, B.H.; Tabatabaee, H.; Lu, Q.; Lemke, S.; Možeiko, M.; Kim, S. LGSVL simulator: A high fidelity simulator for autonomous driving. In Proceedings of the 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC), Rhodes, Greece, 20–23 September 2020; IEEE: New York, NY, USA, 2020; pp. 1–6.
18. Quigley, M.; Gerkey, B.; Conley, K.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A.Y. ROS: An open-source Robot Operating System. In Proceedings of the ICRA Workshop on Open Source Software, Kobe, Japan, 12–17 May 2009.
19. Shah, S.; Dey, D.; Lovett, C.; Kapoor, A. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics: Results of the 11th International Conference*; Springer International Publishing: Cham, Switzerland, 2018; pp. 621–635.
20. Cognata. Available online: https://www.mdstech.co.kr/cognata (accessed on 27 October 2024).
21. Sovani, S. Simulation Accelerates Development of Autonomous Driving. *ATZ Worldw.* **2017**, *119*, 24–29. [CrossRef]
22. Vukić, M.; Grgić, B.; Dinčir, D.; Kostelac, L.; Marković, I. Unity Based Urban Environment Simulation for Autonomous Vehicle Stereo Vision Evaluation. In Proceedings of the 2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 20–24 May 2019; IEEE: New York, NY, USA, 2019.