Poirot: Automatic Root Cause Analysis of Safety Violations in ADS Simulation Testing via Hypothetical Reasoning

YOU LU, Fudan University, China
DINGJI WANG, Fudan University, China
KUN ZHANG, Fudan University, China
BIHUAN CHEN*, Fudan University, China
JIYAN ZHANG, Fudan University, China
XIN PENG, Fudan University, China

With the rapid development of autonomous driving systems (ADSs), it has become critical to ensure their operational safety, leading to the widespread adoption of simulation testing. While existing scenario-based simulation testing approaches have demonstrated effectiveness in detecting safety violations, they often fall short in providing insight into the underlying causes of these violations, which is an essential capability for improving the safety and reliability of ADSs. To address this limitation, we propose a two-phase novel framework, Poirot, for root cause analysis in simulation testing via hypothetical reasoning. Given a reproducible violation scenario, in the module-level analysis phase, Poirot replays the violation scenario, and identifies the faulty module by iteratively replacing an actual module with an idealized module and checking whether the violation persists. In the component-level analysis phase, depending on the identified faulty module, Poirot further applies either hypothetical reasoning with a suspicion-guided search strategy or causal analysis to narrow the fault space and pinpoint the faulty component. We evaluate Poirot with two ADSs, *i.e.,* Apollo and Autoware, on a comprehensive benchmark that includes a total of 80 real and injected faults along with their triggering scenarios. Compared with the state-of-the-art root cause analysis approaches, *i.e.,* ACAV and Rocas, Poirot improves the module-level accuracy by 187.29% on average; and identifies the faulty components at a finer granularity, achieving component-level accuracy of 90.62%. Our ablation study shows that our suspicion-guided search strategy in Poirot efficiently reduces the exploration of the fault space by 58.77%, leading to a 65.41% reduction in the time for fault localization. Finally, applied to two scenario-based simulation testing methods, *i.e.,* AvFuzzer and MoDitector, Poirot attributes 425 violation scenarios to 8 faults, cutting debugging time by 94.59% compared to manual analysis in practice.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

ACM Reference Format:
You Lu, Dingji Wang, Kun Zhang, Bihuan Chen, Jiyan Zhang, and Xin Peng. 2026. Poirot: Automatic Root Cause Analysis of Safety Violations in ADS Simulation Testing via Hypothetical Reasoning. In *Proceedings of*

*Bihuan Chen is the corresponding author.

Authors' Contact Information: You Lu, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China; Dingji Wang, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China; Kun Zhang, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China; Bihuan Chen, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China; Jiyan Zhang, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China; Xin Peng, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '26, October 3–9, 2026, Oakland, California, United States
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '26). ACM, New York, NY, USA, 23 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In recent decades, autonomous driving systems (ADSs) have undergone a remarkable evolution. These systems hold immense potential to enhance road safety, alleviate traffic congestion, and improve overall transportation efficiency, thereby revolutionizing the automotive industry [52]. Nowadays, manufacturers are making substantial investments in ADS research [23]. Despite significant breakthroughs made by industry leaders such as Tesla, Waymo, and Uber, current ADSs still struggle with corner cases and exhibit erroneous behaviors due to the inherent uncertainty in deep learning models, the complexity of architectures, and the unpredictable hazards from real-world driving environments. Such erroneous behaviors of ADSs can result in severe consequences and substantial losses, as evidenced by multiple well-documented traffic incidents [3, 32, 49]. Consequently, extensive testing is needed to ensure the safety and reliability of ADSs before their deployment.

Due to the cost-effective nature, simulation testing [16, 41, 44, 73, 76] has been widely adopted to generate diverse and challenging scenarios for ADSs with high-fidelity simulators, *e.g.,* LGSVL [38] and CARLA [17]. Recently, scenario-based simulation testing has been widely studied to generate critical scenarios for ADSs [11, 14, 27, 28, 39, 45, 67, 72, 74], and these approaches have demonstrated strong capabilities in finding safety violations. However, they generally fall short in providing explanations for why such violations occur, leaving the burden of root cause analysis to labor-intensive manual inspection. Therefore, the need for an automatic approach that can analyze the root causes and localize the faults in ADSs that lead to safety violations is paramount.

Inspired by root cause analysis approaches in deep learning models [6, 13, 62] and traditional software systems [8, 42, 70], recent years have seen attempts to introduce root cause analysis into the testing of cyber-physical systems (CPSs) [12, 30, 34, 55, 63, 68, 71]. For instance, Swarmbug [30] treats swarm robotics as a black box and measures how configurations affect system behaviors. MAYDAY [34] diagnoses accidents in robotic aerial vehicles caused by controller or mission command bugs. RVPlayer [12] replays robotic aerial vehicles with hypothetical reasoning in a simulator, testing whether accidents could be avoided by modifying certain control parameters or code.

Unfortunately, applying existing root cause analysis approaches for CPSs to ADSs is non-trivial. For one thing, ADSs comprise multiple independent modules that communicate via message passing, where a minor fault in one module can propagate into severe errors in others, increasing the difficulty of identifying the root cause. For another thing, within each module, complex components made up of deep learning models, vast amounts of code, and extensive configurations, constitute a huge fault space, thereby restricting the efficiency of root cause analysis. Given the architectural complexity of ADSs, tracing a high-level safety violation in simulation testing back to the faulty module or the faulty component is extremely challenging. As far as we know, there are only a few works [1, 10, 15, 19, 50, 58] designed for root cause analysis of safety violations in ADS simulation testing. Some works [1, 15, 19, 50] focus on identifying the external triggers or features in violation scenarios, but they cannot attribute the violations to the internal faulty modules or components in ADSs. Rocas [19] and ConfVE [10] focus on identifying misconfigurations in ADSs. However, misconfiguration is only one of many root causes, which accounts for merely 27.25% of bugs [9, 21]. ACAV [58] performs causal analysis on scenario records to identify multiple causal events (*i.e.,* wrong priority prediction, wrong trajectory prediction, improper planning and vehicle out-of-control) responsible for a collision, but it cannot attribute collisions to the deterministic module or component and is incapable of handling other types of safety violations, such as running red light.

To address these limitations, we propose Poirot, a novel framework for root cause analysis for ADS simulation testing via hypothetical reasoning. The key idea is to reason about *what the system's*

behavior would be if certain modules or components in an ADS were not faulty, and use this counterfactual perspective to pinpoint the causes of safety violations, *e.g.,* collision, running red light, crossing yellow line, and not reaching destination. By systematically constructing and testing such *what-if* scenarios, Poirot can progressively narrow the fault location from a coarse module level to a fine component level in an ADS. Specifically, Poirot operates in two phases, *i.e.,* module-level analysis and component-level analysis. In the first phase, Poirot replays the violation scenario to obtain idealized module messages from the simulator, and identifies the faulty module in the ADS by iteratively replacing actual modules with idealized modules (which are virtual modules publishing idealized messages), and checking whether the violation persists. In the second phase, if the faulty module is the localization, perception, or prediction module, Poirot first replays the violation scenario to collect idealized component messages and actual component messages in the faulty module. Then, Poirot performs differential analysis to compute the suspicion scores for components in the faulty module. Finally, Poirot applies hypothetical reasoning within the faulty module by a suspicion-guided search strategy. If the faulty module is the planning or control module, Poirot leverages causal analysis for fine-grained faulty component localization.

To evaluate the effectiveness and efficiency of Poirot, we create a benchmark that includes a total of 80 real and injected faults with their triggering scenarios on two ADSs, *i.e.,* Apollo [4] and Autoware [20]. First, we compare Poirot with the state-of-the-art root cause analysis approaches, *i.e.,* ACAV [58] and Rocas [19]. Poirot achieves 91.87% accuracy at the module level, which is 187.29% higher than those of baseline approaches on average, and 90.62% accuracy at the component level. Second, our ablation study shows that our suspicion-guided search strategy in Poirot effectively reduces the exploration of the fault space by 58.77%, resulting in a 65.41% decrease in the time required for faulty component localization. Finally, when applied to scenario-based testing methods, *i.e.,* AvFuzzer [40] and MoDitector [66], Poirot successfully attributes 425 violation scenarios to 8 faults, cutting debugging time by 94.59% compared to manual root cause analysis in practice.

In summary, the main contributions of our work are summarized as follows.

- We propose an automatic root cause analysis framework, Poirot, that leverages hypothetical reasoning by systematically constructing and evaluating counterfactual execution to identify the module and component in an ADS responsible for safety violations.
- We design a suspicion-guided search strategy to prioritize components, reducing the fault space and expediting the localization of the faulty component.
- We construct a comprehensive benchmark for root cause analysis, including a total of 80 real and injected faults across two ADSs, *i.e.,* Apollo and Autoware, along with their triggering scenarios.
- We implement a prototype of Poirot, and conduct extensive experiments to demonstrate its effectiveness and efficiency in root cause analysis of safety violations.

2 Background and Problem Formulation

We first introduce the background on multi-module ADSs, and then formulate the problem.

2.1 Multi-Module Autonomous Driving Systems

Current ADSs are typically designed with a hierarchical modular architecture to achieve both functional decoupling and extensibility [4, 18, 20]. They are generally organized into five major modules, *i.e.,* localization, perception, prediction, planning, and control. Specifically, the localization module provides accurate vehicle positions by integrating GPS, IMU, and other sensor data, which in turn offers spatial references for the perception module to construct an understanding of the environment through object detection, classification, and tracking. The prediction module consumes perception results to estimate the future trajectories of surrounding traffic participants such as

vehicles, cyclists, and pedestrians. These predictions, together with the localization information, form essential inputs to the planning module, which generates safe and efficient driving trajectories. Finally, the control module executes the planned trajectory by issuing low-level commands to vehicle actuators, ensuring precise maneuvering and closing the loop of the driving process.

Each module is further decomposed into components, with each component encapsulating specific algorithmic functionalities and specifying its dependencies via configuration files. The localization, perception, and prediction modules primarily construct an understanding of the surrounding environment. They handle heterogeneous sensor inputs from multiple modalities, and involve numerous deep learning models together with associated preprocessing and postprocessing steps, resulting in more than twenty components [4]. In contrast, the planning and control modules are largely rule-based. The planning module generally comprises two key components, *i.e.,* the planner, which selects appropriate driving behaviors (*e.g.,* going straight, stopping, yielding, and overtaking) according to scenarios, and the decider, which generates feasible candidate trajectories including waypoints and speeds. The control module is typically divided into longitudinal controller, regulating speed and acceleration, and lateral controller, governing steering for accurate trajectory tracking. Interactions among modules and components are coordinated through middleware frameworks that enable efficient message passing, most commonly based on the Robot Operating System (ROS) [56] or its optimized variant CyberRT [4]. In this design, each component operates as an independent node that publishes messages under specific topics, while subscribers automatically receive and process the transmitted messages. This mechanism reduces inter-module dependencies, enhances parallelism, and enables all exchanged messages to be recorded as driving logs for replay and validation, which is crucial for system testing, debugging, and verification.

2.2 Problem Formulation

Our goal is to automatically attribute safety violation scenarios in ADS simulation testing to the faulty modules and faulty components of the ADS under test via hypothetical reasoning. To clearly specify the problem, we first define some notations, and then formulate our problem.

Definition 1 (Scenario)**.** A scenario S is a 7-tuple $\langle d, W, E, p_{start}, p_{des}, \mathbb{T}, \mathbb{N}\rangle$, where d denotes the frame duration for the scenario; W specifies the weather condition; E denotes the Ego vehicle controlled by the ADS with starting position p_{start} and destination p_{des}; $\mathbb{T}$ denotes a set of traffic lights $\{T_0, T_1, \ldots, T_{|\mathbb{T}|-1}\}$, whose signal can be red, yellow, and green; and $\mathbb{N}$ is a set of NPCs $\{N_0, N_1, \ldots, N_{|\mathbb{N}|-1}\}$. The behavior of the Ego vehicle and NPCs can be denoted as trajectories in a simulation.

Definition 2 (Trajectory)**.** The trajectory of the Ego vehicle E or an NPC $N_h \in \mathbb{N}$ is a 2-tuple $\langle P, V\rangle$, where $P = \langle p^0, p^1, \ldots, p^{d-1}\rangle$ is a sequence of waypoints that the traffic participant follows at each timestamp during the duration d; and $V = \langle v^0, v^1, \ldots, v^{d-1}\rangle$ is a sequence of speed of the traffic participant at each timestamp during the frame duration d. For the Ego vehicle E, its initial waypoint p^0 is specified as p_{start}, and its trajectory is controlled by the ADS to reach the destination p_{des}. In contrast, for NPCs, their trajectories are recorded in the violation scenario and replayed during the root cause analysis process.

Definition 3 (Scenario Execution)**.** The execution of a scenario $\mathcal{E}(S, ADS, \mathbb{O}) \rightarrow [R, \mathbb{B}]$ returns an execution record R and a Boolean value $\mathbb{B}$ indicating whether the scenario execution suffers any safety violations against the predefined set of violation oracles $\mathbb{O}$.

We currently focus on four types of violations, *i.e.,* collision, running red light, crossing yellow line, and not reaching destination. We define their corresponding oracles $\mathbb{O}$ as follows.

- **Collision.** This oracle checks if the Ego vehicle E collides with any NPC $N_h \in \mathbb{N}$. Given the waypoints $\langle p_E^0, p_E^1, \ldots, p_E^{d-1}\rangle$ of the Ego vehicle and $\langle p_{N_h}^0, p_{N_h}^1, \ldots, p_{N_h}^{d-1}\rangle$ of any NPC N_h at each

timestamp during the simulation, the violation condition of this oracle is defined by Eq. 1,

$$\min(\{D_{E2N}(p_E^t, p_{N_h}^t) \mid 0 \le h < |\mathbb{N}|, 0 \le t < d\}) = 0 \tag{1}$$

where $D_{E2N}(p_E^t, p_{N_h}^t)$ calculates the distance between the bounding boxes of the Ego vehicle E and any NPC N_h at frame t.

- **Running Red Light.** This oracle checks if the Ego vehicle crosses the stop line at a positive speed when the traffic light signal is red. Given the waypoints $\langle p_E^0, p_E^1, \dots, p_E^{d-1}\rangle$ and the speed sequence $\langle v_E^0, v_E^1, \dots, v_E^{d-1}\rangle$ of the Ego vehicle, the traffic light $T_i \in \mathbb{T}$, and the stop line l_{st} associated with the traffic light extracted from the map, the violation condition of this oracle is defined by Eq. 2,

$$\exists T_i \in \mathbb{T}, t \in [0, d), signal(T_i, t) = red \land D_{E2l}(p_E^t, l_{st}) = 0 \land v_E^t > 0 \tag{2}$$

where $signal(T_i, t)$ returns the signal of the traffic light T_i at frame t, and $D_{E2l}(p_E^t, l_{st})$ calculates the distance between the center of the Ego vehicle and the stop line l_{st} at frame t.

- **Crossing Yellow Line.** This oracle checks if the Ego vehicle hits the yellow lines during the simulation. Given the waypoints $\langle p_E^0, p_E^1, \dots, p_E^{d-1}\rangle$ of the Ego vehicle and a set of yellow lines denoted as *Lines* extracted from the map, the violation condition of this oracle is defined by Eq. 3,

$$\min(\{D_{E2l}(p_E^t, l) \mid l \in Lines, 0 \le t < d\}) < threshold_1 \tag{3}$$

where $D_{E2l}(p_E^t, l)$ calculates the distance between the center of the Ego vehicle and the yellow line l at frame t, and $threshold_1$ is set to half width of the Ego vehicle's bounding box.

- **Not Reaching Destination.** This oracle checks if the Ego vehicle reaches the destination in the given time. Given the waypoints $\langle p_E^0, p_E^1, \dots, p_E^{d-1}\rangle$ of the Ego vehicle and its destination p_{des}, the violation condition of this oracle is defined by Eq. 4,

$$D_{E2des}(p_E^{d-1}, p_{des}) > threshold_2 \tag{4}$$

where $D_{E2des}(p_E^{d-1}, p_{des})$ calculates the final distance of the Ego vehicle to its destination, and $threshold_2$ is set to half length of the Ego vehicle's bounding box.

Given the above definitions and violation oracles, we further formulate the problem of root cause analysis in ADS simulation testing. The inputs are a target ADS and a violation scenario S_{vio}, where $\mathcal{E}(S_{vio}, ADS, \mathbb{O}) \rightarrow [R, true]$. Then, we conduct root cause analysis to understand how the ADS's behavior would be if certain module or component were not faulty. Formally, this can be expressed as a counterfactual execution $\mathcal{E}^{f \leftarrow f^*}(S_{vio}, ADS, \mathbb{O})$, where f denotes a module $\mathcal{M}_j \in \mathbb{M}$ or a component $C_k \in \mathbb{C}^{\mathcal{M}_j}$. $\mathbb{M}$ is the set of modules $\{\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_{|\mathbb{M}|-1}\}$ within the ADS, and $\mathbb{C}^{\mathcal{M}_j}$ is the set of all components $\{C_0, C_1, \dots, C_{|\mathbb{C}^{\mathcal{M}_j}|-1}\}$ within a certain module $\mathcal{M}_j$. The idea is to replace f with its expected correct functionality f^* while keeping the rest of the ADS unchanged, and then re-execute the scenario. If the violation persists, f is unlikely to be the root cause of the violation, and further analysis is required to explore other possible faulty modules or components in the ADS.

3 Methodology

We propose POIROT, a two-phase root cause analysis framework by constructing and evaluating counterfactual executions to first identify the faulty module $\mathcal{M}_{faulty} \in$ {localization, perception, prediction, planning, control} and then the faulty component $C_{faulty} \in \{C_0, C_1, \dots, C_{|\mathbb{C}^{\mathcal{M}_{faulty}}|-1}\}$ within $\mathcal{M}_{faulty}$. The key challenge lies in *(1) how to obtain idealized messages for each module and component, and (2) how to reduce the fault space*. The approach overview of POIROT is shown in Fig. 1.

Specifically, in the module-level localization phase, POIROT (*i.e.,* Step-1 in Fig. 1) replays the violation scenario to collect idealized messages from the simulator, and aligns them temporally to obtain the idealized module messages (see Sec. 3.1). Next, POIROT (*i.e.,* Step-2 in Fig. 1) constructs idealized message flows for modules in the ADS, and applies hypothetical reasoning to identify

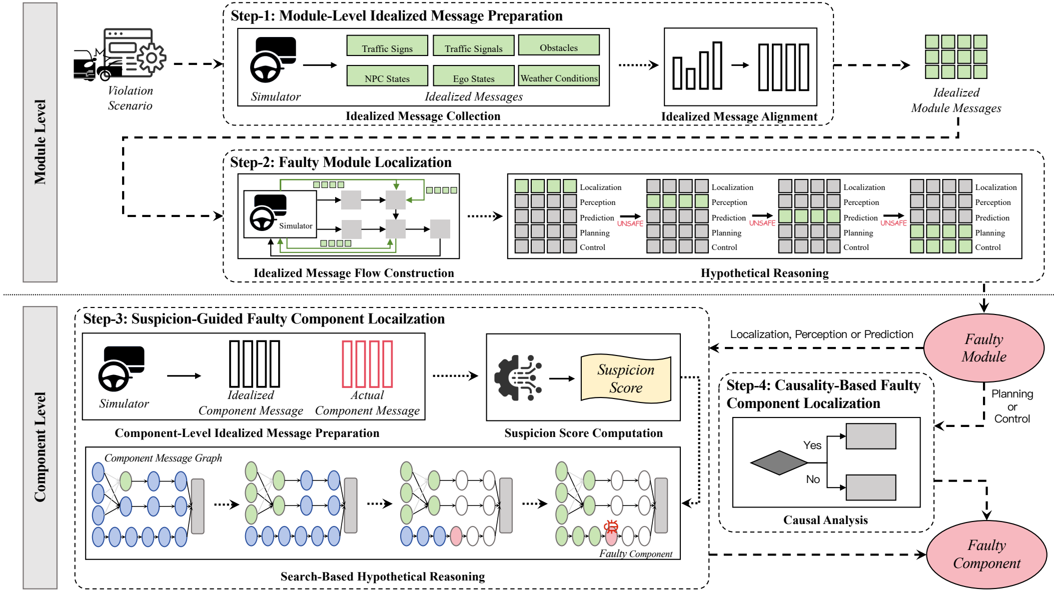


Fig. 1. Approach Overview of Poirot

$\mathcal{M}_{faulty}$ by iteratively replacing actual modules with idealized modules, which are virtual modules publishing idealized messages, and checking whether the violation disappears (see Sec. 3.2).

In the component-level localization phase, if $\mathcal{M}_{faulty} \in \{\text{localization}, \text{perception}, \text{prediction}\}$, Poirot (*i.e.,* Step-3 in Fig. 1) obtains the idealized component messages and actual component messages within $\mathcal{M}_{faulty}$ via scenario replaying. By performing differential analysis between these two types of messages, Poirot computes the suspicion score for each component in $\mathcal{M}_{faulty}$. Then, Poirot applies a suspicion-guided search strategy on the component message graph (*CMG*) of $\mathcal{M}_{faulty}$ to reduce the fault space and accelerate the identification of the faulty component C_{faulty} (see Sec. 3.3). If $\mathcal{M}_{faulty} \in \{\text{planning}, \text{control}\}$, Poirot (*i.e.,* Step-4 in Fig. 1) employs causal analysis to enable fine-grained localization of C_{faulty} (see Sec. 3.4).

3.1 Module-Level Idealized Message Preparation

Idealized module messages represent the outputs that a correctly implemented module would produce under a scenario. To derive such error-free messages that are used to replace the actual messages of a module, we first replay the violation scenario within the simulator to collect idealized messages of the modules in the ADS. Then, we perform temporal alignment to ensure that these idealized messages are temporally consistent with the actual module messages.

Idealized Message Collection. Although the simulator is typically employed to interact with the ADS through raw sensor inputs and actuator commands, we observe that a wide range of intermediate information can also be directly generated or extracted from the simulator itself. For instance, information about traffic signs, traffic signals, static obstacles, NPC states, Ego states, as well as weather conditions, can be retrieved directly via the provided APIs [17, 38]. Given that the localization, perception, and prediction modules are fundamentally designed to understand the surrounding environment, which is fully controlled by the simulator, we can collect their corresponding idealized messages in the required formats by utilizing the ground-truth information from the simulator. In contrast, the planning and control modules in the ADS involve uncertainty in motion planning and decision execution, therefore lacking externally verifiable idealized messages. To this end, we

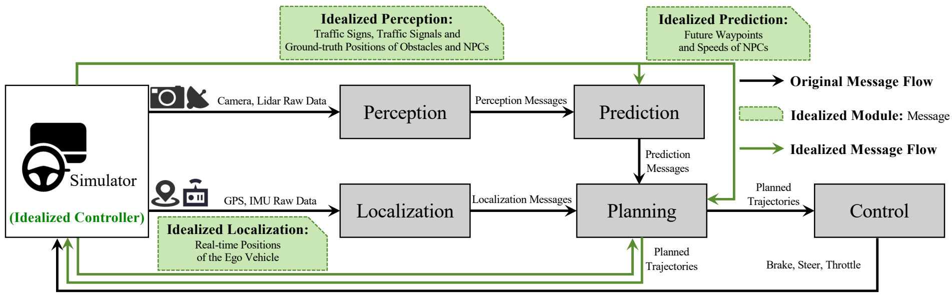


Fig. 2. Idealized Message Flow for Modules in the ADS

cannot collect their idealized messages from the simulator. Instead, we treat the simulator as an idealized controller that tracks the planned trajectory with high fidelity, thereby serving as a strong baseline to assess whether a violation stems from planning or from control execution.

Idealized Message Alignment. A critical challenge in constructing idealized module messages for localization, perception, and prediction is maintaining temporal consistency with their actual messages. In a multi-module ADS, messages are exchanged and processed asynchronously, and each module adopts its own publishing and subscribing frequency. Furthermore, generating idealized messages from the simulator (*e.g.*, extracting and formatting ground-truth states and trajectories) is not instantaneous, and thus computing these messages incurs non-negligible latency. Consequently, discrepancies naturally arise between the frequency of idealized messages extracted from the simulator and the actual messages generated by the actual modules in the ADS. Without careful handling, such inconsistencies may lead to message loss during subsequent counterfactual executions.

To mitigate this issue, we perform temporal alignment of the idealized messages derived from the simulator. Specifically, we adopt the highest message publishing frequency among all actual modules (*i.e.*, the frequency of the localization module) in the ADS as the reference frame frequency. Following previous works [15, 58], if a module publishes one or more messages within a given frame, the most recent idealized message is aligned with the beginning of that frame. Conversely, if no message is published within the frame, the last available idealized message prior to that frame is propagated forward. In addition, we pause the simulator and resume it only after the idealized messages for the current frame have been computed and aligned during counterfactual execution to ensure that the ADS always consumes temporally consistent idealized messages. As a result, we obtain temporally aligned streams of idealized module messages that can effectively replace the actual messages of modules in the ADS except for the planning module and control module. Compared to the actual module messages, which often suffer from bias and error propagation, these aligned idealized module messages are semantically aligned with the ground-truth environment information in the simulator, and thus provide a reliable baseline for isolating faulty modules.

3.2 Faulty Module Localization

To identify the faulty module responsible for a violation, we reconfigure the message flow in the ADS, constructing idealized modules which are virtual modules publishing idealized module messages for the actual modules. Then, we perform iterative hypothetical reasoning by replacing actual modules with their corresponding idealized modules and observing the execution results.

Idealized Message Flow Construction. We construct the idealized message flow for modules in the ADS by replacing the actual modules with idealized modules, which are virtual modules emulated by the simulator. For the localization, perception, and prediction modules, the simulator publishes idealized module messages obtained in Sec. 3.1. Concretely, the simulator provides (i) idealized

localization messages that encode the real-time positions of the Ego vehicle, (ii) idealized perception messages containing the traffic signs, traffic signals, and the ground-truth positions of obstacles and NPCs, and (iii) idealized prediction messages specifying the future waypoints and speeds of NPCs. Furthermore, for the planning module and the control module, since the planned trajectories of the actual planning module already represent the intended future behaviors of the Ego vehicle, these can be directly used to update the Ego vehicle's positions within the simulator. This is equivalent to assuming that the simulator acts as an idealized control module that generates commands capable of perfectly tracking the planned trajectories of the actual planning module.

Fig. 2 illustrates the constructed idealized message flow for modules in the ADS. We reconfigure the runtime message-passing mechanism so that downstream consumers of localization, perception, and prediction modules can directly subscribe to the corresponding idealized messages from the simulator, rather than the messages of the actual modules. Besides, the planned trajectories of the actual planning module are sent to the simulator to update the Ego vehicle's positions.

Message delivery is redirected at the middleware layer (*e.g.,* ROS [56] or CyberRT [2]), so that the idealized modules are transparently injected into the system. From the perspective of the downstream modules, the idealized upstream modules are indistinguishable from their original publishers, except that their outputs are error-free. This non-intrusive architecture requires no changes to the internal implementation of ADS modules, and allows us to replace actual modules with their idealized counterparts one by one at a time, thereby isolating the effect of a single module while preventing error propagation from upstream dependencies.

Hypothetical Reasoning. To identify the faulty module, we adopt a hypothetical reasoning procedure that follows the ADS pipeline. The underlying intuition is that if the violation disappears once a given actual module is substituted with the idealized module, the root cause can be attributed to that module; otherwise, the persistence of the violation indicates that the cause of the violation lies in other modules. Specifically, given a violation scenario S_{vio}, the target ADS, and the violation oracles $\mathbb{O}$, we first replace each module $\mathcal{M}_j$ in the ordered sequence of {localization, perception, prediction}, and conduct a counterfactual execution $\mathcal{E}^{\mathcal{M}_j \leftarrow \mathcal{M}_j^*}(S_{vio}, ADS, \mathbb{O})$, where $\mathcal{M}_j^*$ represents the idealized counterpart of $\mathcal{M}_j$. If $\mathcal{E}^{\mathcal{M}_j \leftarrow \mathcal{M}_j^*}(S_{vio}, ADS, \mathbb{O}) \rightarrow [R, false]$, *i.e.,* the violation is no longer observed after the replacement, we attribute the fault to $\mathcal{M}_j$. Otherwise, the faulty behavior must reside in one of the downstream modules, and our analysis continues along the sequence. Then, for the planning module and control module, we conduct a counterfactual execution by replacing the actual control module with the idealized control module. If $\mathcal{E}^{Control \leftarrow Control^*}(S_{vio}, ADS, \mathbb{O}) \rightarrow [R, true]$, *i.e.,* executing the planned trajectory with the idealized control module still results in violations, we attribute the fault to the planning module. Otherwise, we attribute the fault to the control module.

3.3 Suspicion-Guided Faulty Component Localization

Given the identified faulty module $\mathcal{M}_{faulty}$, we further conduct the component-level localization to find the faulty component C_{faulty} in $\mathcal{M}_{faulty}$. For $\mathcal{M}_{faulty} \in$ {localization, perception, prediction}, similar to the module-level localization phase, we apply hypothetical reasoning to identify C_{faulty} by replacing actual components with their corresponding idealized components and checking whether the violation persists. However, as mentioned in Sec. 2, these modules typically consist of multiple components with complex interactions. To accelerate the identification of C_{faulty} in these modules, we employ a suspicion-guided search strategy. Specifically, we prepare the idealized component messages for each component C_k in $\mathcal{M}_{faulty}$, and compute the suspicion score for C_k via differential analysis between the actual component messages and the idealized component messages. Then, we apply hypothetical reasoning to identify C_{faulty} guided by the suspicion scores.

Component-Level Idealized Message Preparation. The components in the localization, perception, and prediction modules typically have different types of outputs. Certain components decompose environmental understanding tasks into subtasks. For instance, a traffic-light detection component outputs the position of traffic lights, while a traffic-light recognition component determines their state. For such components, we can directly obtain the idealized messages from the simulator. Other components, in contrast, refine the environmental understanding tasks step by step. For instance, an obstacle estimation component provides approximate obstacle locations, whereas an obstacle detection component outputs more precise bounding boxes with locations. For this type, we adopt the accurate ground-truth object locations simultaneously as the idealized messages for both estimation and detection components. Similar to module-level idealized message preparation described in Sec. 3.1, we temporally align the idealized messages derived from the simulator with the actual component messages $A_{C_k}(t) = \{x_t^{(k)} | t \in [0, d)\}$, obtaining the idealized component messages $I_{C_k}(t) = \{\tilde{x}_t^{(k)} | t \in [0, d)\}$, where $x_t^{(k)}$ is a message of the component C_k at frame t, $\tilde{x}_t^{(k)}$ is its corresponding idealized message, and d is the frame number of the scenario.

Suspicion Score Computation. Given $A_{C_k}(t)$ and $I_{C_k}(t)$, we first compute the discrepancy between them at each frame t, and then aggregate them into a suspicion score SS_{C_k} for C_k.

Specifically, we define the frame-level discrepancy as $\delta_t^{(k)} = ||x_t^{(k)} - \tilde{x}_t^{(k)}||_2$. To account for the allowed tolerance of component outputs and heterogeneous feature scales between messages of different components, direct comparison of frame-level discrepancies across components would be misleading. To transform these frame-level discrepancies of different components into a comparable, unit-less scale, we evaluate their extremeness relative to the empirical distribution observed in normal runs. Concretely, we collect scenarios without violations during simulation testing, and compute the frame-level discrepancy of these normal runs for each component C_k. Then, we construct the empirical cumulative distribution function $\hat{F}_k$ [53] of the frame-level discrepancies for C_k. Based on $\hat{F}_k$, we compute the frame-level tail probability that the frame-level discrepancy $\delta_t^{(k)}$ is suspicious by $p_t^{(k)} = 1 - \hat{F}_k(\delta_t^{(k)})$. Higher values of $p_t^{(k)}$ indicate a stronger evidence of suspicion.

Finally, we aggregate frame-level probabilities into a single suspicion score SS_{C_k}. To emphasize deviations occurring closer to the violation, we apply exponentially decaying weights over time. Specifically, the weight assigned to frame t is defined as $w_t = \frac{\gamma^{d-t}}{\sum_{\tau=1}^{d} \gamma^{d-\tau}}, \quad \gamma \in (0, 1)$, where n is the total number of frames in the scenario and τ indexes each frame. Under this scheme, frames closer to the end of the scenario where the violation occurs receive larger weights, while earlier frames are exponentially downweighted according to the discount factor $\gamma = 0.8$ by default. The overall suspicion score SS_{C_k} of C_k is defined by Eq. 5 as follows.

$$SS_{C_k} = \sum_{t=1}^{d} w_t \, p_t^{(k)} = \sum_{t=1}^{d} \frac{\gamma^{d-t}}{\sum_{\tau=1}^{d} \gamma^{d-\tau}} \left(1 - \hat{F}_k(\delta_t^{(k)})\right) \tag{5}$$

Search-Based Hypothetical Reasoning. To reduce the fault space and accelerate the identification of C_{faulty} in $\mathcal{M}_{faulty}$, we employ a search-based hypothetical reasoning, using suspicion scores $\mathbb{SS}$ of all the components in the faulty module to select and prune components.

We first model the message flow of components defined in the configuration file within $\mathcal{M}_{faulty}$ as a component message graph $CMG = (\mathbb{C}^{\mathcal{M}_{faulty}}, M, sc)$, a directed acyclic graph with a unique sink node $sc \in \mathbb{C}^{\mathcal{M}_{faulty}}$. Here, nodes $\mathcal{M}_{faulty}$ represent components and edges M denote message flow directions. Fig. 3a shows an example of a CMG in the perception module of Apollo. We employ a counterfactual execution $\mathcal{E}^{C_k \leftarrow C_k^*}(S_{vio}, ADS, \mathbb{O}) \rightarrow [R, \mathbb{B}]$ to reason whether a component $C_k \in \mathbb{C}^{\mathcal{M}_{faulty}}$ is faulty. If the execution returns $true$, indicating that the violation persists after replacing C_k with its idealized counterpart C_k^*, we refer to C_k as a non-faulty component. Otherwise, C_k is regarded as a suspicious component.

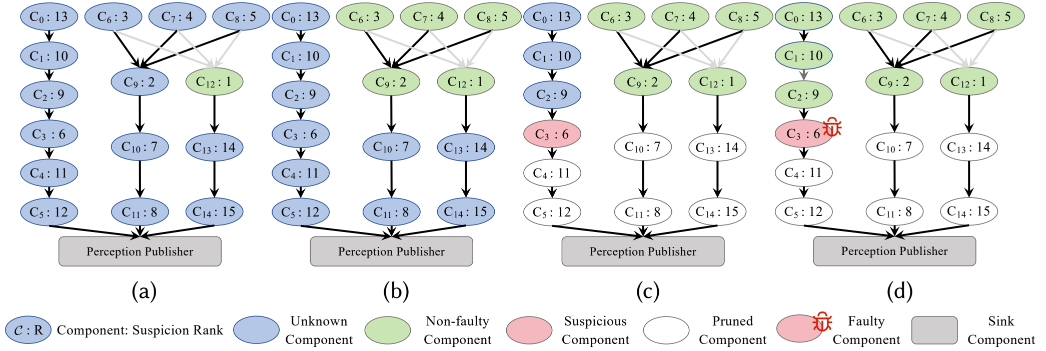


Fig. 3. Example of Search-Based Hypothetical Reasoning for the Perception Module in Apollo

Let $Pred[C_k]$ denote the set of immediate predecessor components of C_k, $Anc^*[C_k]$ denote the set of C_k and all its ancestor components, and $OD[C_k]$ denote the out-degree of C_k in the *CMG*, we present the following three lemmas.

Lemma 3.1. *For a non-faulty component C_k, $\forall u \in Pred[C_k]$, the edge $\langle u, C_k \rangle$ is not on any path from C_{faulty} to sc. Besides, if $OD[u] = 1$, then u is a non-faulty component.*

Proof. (1) If $\langle u, C_k \rangle$ is on a path from C_{faulty} to sc, then the substitution of C_k with its idealized component C_k^* would resolve the violation, which contradicts the fact that C_k is a non-faulty component. (2) If u is not a non-faulty component, then there exists a path from C_{faulty} to sc that passes through u but not through C_k, which contradicts the fact that $OD[u] = 1$ and the edge $\langle u, C_k \rangle$ is not on any path from C_{faulty} to sc. □

Lemma 3.2. *For a suspicious component C_k, $C_{faulty} \in Anc^*[C_k]$.*

Proof. If $C_{faulty} \notin Anc^*[C_k]$, then there exists a path from C_{faulty} to sc that does not pass through C_k. Since C_{faulty} is the unique faulty component, the violation should persist after replacing C_k with its idealized substitute C_k^*, which contradicts the fact that C_k is a suspicious component. □

Lemma 3.3. *For a suspicious component C_k, if $Pred[C_k] = \emptyset$, or if $\forall u \in Pred[C_k]$, u is a non-faulty component, then C_k is the faulty component.*

Proof. By Lemma 3.2, $C_{faulty} \in Anc^*[C_k]$. If $Pred[C_k] = \emptyset$, then $C_{faulty} = C_k$. Otherwise, all ancestor components of C_k have already been excluded by Lemma 3.1, hence $C_{faulty} = C_k$. □

Based on the above lemmas, the process of our search-base hypothetical reasoning, guided by suspicion scores, is presented in Algorithm 1. The input includes the component message graph $CMG = (\mathbb{C}^{\mathcal{M}_{faulty}}, M, sc)$ of the faulty module $\mathcal{M}_{faulty}$, the suspicion scores $\mathbb{SS}$ of all the components in $\mathcal{M}_{faulty}$, the violation scenario S_{vio}, the target ADS, the violation oracles $\mathbb{O}$, and the idealized component set $\mathbb{C}^{\mathcal{M}_{faulty}*}$ of all components in $\mathcal{M}_{faulty}$ (constructed similar to the idealized modules described in Sec. 3.2). The output is the identified faulty component $C_{faulty} \in \mathbb{C}^{\mathcal{M}_{faulty}}$.

First, for each component C_k in the *CMG*, we initialize its predecessors in the array *Pred*, ancestors and itself in the array Anc^*, and its out-degree in the array *OD*; and we also tag the state of C_k as UNKNOWN in the array *Label* (Line 1-4). Then, for each component with the UNKNOWN label, we conduct the main loop of suspicion-guided hypothetical reasoning (Line 5-30).

In each iteration, we select the component with the highest suspicion score among all UNKNOWN components in the *CMG* as the component for reasoning, denoted as $C_{reasoning}$, and its idealized

Algorithm 1: Search-Based Hypothetical Reasoning

Input: component message graph $CMG = (\mathbb{C}^{\mathcal{M}_{faulty}}, M, sc)$, suspicion scores $\mathbb{SS}$, violation scenario S_{vio}, target ADS, violation oracles $\mathbb{O}$, idealized components $\mathbb{C}^{\mathcal{M}_{faulty}*}$
Output: faulty component C_{faulty}

```
1  foreach C_k ∈ ℂ^{M_faulty} do
2  |   Pred[C_k], Anc*[C_k], OD[C_k] ← Initial(CMG);
3  |   Label[C_k] ← UNKNOWN;
4  end
5  while ∃C_k ∈ ℂ^{M_faulty}, Label[C] = UNKNOWN do
6  |   C_reasoning, C*_reasoning ← SelectReasoningComponent(𝕊𝕊, Label, CMG, ℂ^{M_faulty*});
7  |   if ℰ^{C_reasoning ← C*_reasoning}(S_vio, ADS, 𝕆) → [R, true] then
8  |   |   Label[C_reasoning] ← NON-FAULTY;
9  |   |   ℤ ← ∅;
10 |   |   foreach u ∈ Pred[C_reasoning] do
11 |   |   |   if OD[u] == 1 then
12 |   |   |   |   ℤ ← ℤ ∪ {u};
13 |   |   |   end
14 |   |   |   else
15 |   |   |   |   OD[u] ← OD[u] − 1;
16 |   |   |   end
17 |   |   end
18 |   |   Label ← MarkNonFaultyComponentRecursively(ℤ, Pred, Label, CMG);
19 |   end
20 |   else
21 |   |   Label[C_reasoning] ← SUSPICIOUS;
22 |   |   Label ← PruneNonAncestorComponent(Anc*[C_reasoning], label, CMG);
23 |   end
24 |   foreach C_k ∈ ℂ^{M_faulty} do
25 |   |   if Label[C_k] = SUSPICIOUS ∧ (Pred[C_k] = ∅ ∨ ∀u ∈ Pred[C_k], Label[u] = NON-FAULTY) then
26 |   |   |   C_faulty ← C_k;
27 |   |   |   return C_faulty;
28 |   |   end
29 |   end
30 end
```

counterpart $C^*_{reasoning}$ (Line 6), and then we conduct counterfactual execution by replacing $C_{reasoning}$ with $C^*_{reasoning}$. If the execution result is $[R, true]$ (Line 7- 19), we mark $C_{reasoning}$ as NON-FAULTY, and initialize an empty set $\mathbb{Z}$ to store components whose out-degree is equal to one (Line 8- 9). Then, we iterate through each predecessor component u of $C_{reasoning}$, and if $OD[u] = 1$, we add u to the set $\mathbb{Z}$. Otherwise, we decrement the out-degree of u by one, indicating that the message flow from u to $C_{reasoning}$ has no causal influence on $C_{reasoning}$ (Line 10-17). We recursively tag all components in $\mathbb{Z}$ and their ancestors as NON-FAULTY according to Lemma 3.1 (Line 18). For example, as shown in Fig. 3a, we first select $C_{reasoning} = C_{12}$ for reasoning and the violation does not disappear, indicating C_{12} is a non-faulty component and being marked as NON-FAULTY. Then, we decrement the out-degree of its immediate predecessor components C_6, C_7 and C_8 by one. Then, as shown in Fig. 3b, we select the next UNKNOWN component in the graph with the highest suspicion score, *i.e.,* C_9, for reasoning, and its reasoning result is NON-FAULTY. Since the out-degrees of C_6, C_7 and C_8 are now equal to one, we recursively mark these components as NON-FAULTY.

If the result is $[R, false]$ (Line 20- 23), we mark $C_{reasoning}$ as SUSPICIOUS and prune all components not in $Anc^*[C_{reasoning}]$ from the graph according to Lemma 3.2 (Line 21-22). For example, as

shown in Fig. 3c, after the reasoning of C_{12} and C_9, we select C_3 as $C_{reasoning}$, and the execution result is $[R, false]$, indicating C_3 is a `SUSPICIOUS` component and the faulty components must be in $Anc^*[C_3] = \{C_0, C_1, C_2, C_3\}$. Hence, we prune the graph by discarding all components not contained in $Anc^*[C_3]$, and focus exclusively on the remaining components.

After each iteration, we check whether there exists a `SUSPICIOUS` component C_k whose predecessor components are all marked as `NON-FAULTY` or have no predecessors. If such a component exists, we return it as the faulty component according to Lemma 3.3 (Line 24-29). For example, as shown in Fig. 3d, we select C_2 as $C_{reasoning}$ from the remaining `UNKNOWN` components after the reasoning of C_3, and the execution result is $[R, true]$. We mark C_0, C_1, and C_2 as `NON-FAULTY` according Lemma 3.1. At this point, C_3 is the only remaining `SUSPICIOUS` component, and its predecessor component C_2 is marked as `NON-FAULTY`, so we return C_3 as the faulty component.

3.4 Causality-Based Faulty Component Localization

For $\mathcal{M}_{faulty} \in$ {planning, control}, since we lack definitive ground truth of components in these two modules, we instead resort to causal analysis on the module outputs to identify the most likely responsible component C_{faulty} in $\mathcal{M}_{faulty}$ for the violation.

Causal Analysis. As mentioned in Sec. 2, the planning module consists of two components, *i.e.,* the planner, which is responsible for selecting high-level driving behaviors such as going straight and overtaking, and the decider, which determines detailed paths and velocity profiles of trajectories. ACAV [58] identifies the causes of collisions in the planning module by examining the relationship between the planned path and obstacles in the spatio-temporal (s-t) domain [48]. Specifically, if the ADS selects an inappropriate driving behavior such that its velocity profile is intersected with obstacles in the s-t graph, the violation is attributed to the planner. Conversely, if the high-level behavior is reasonable, but the velocity profile leaves insufficient safety margins, resulting in eventual collision, the root cause is attributed to the decider. Inspired by this principle, we extend causal attribution to cover a broader set of violation types in the planning module.

For running red light violations, if the planner selects a behavior such as going straight, the violation is attributed to the planner. If the planner correctly chooses to stop but the generated velocity profile fails to decelerate to zero, causing the vehicle to cross the stop line, the violation is attributed to the decider. A similar logic applies to violations of crossing yellow line. If the ADS explicitly decides to overtake or change lanes in a region with a solid yellow line, the violation originates in the planner. If the high-level behavior is to keep lane, but the trajectory generated by the decider crosses the yellow line, this indicates faults in the decider. In scenarios where the ADS gets stuck and fails to reach the destination, if the ADS repeatedly chooses yielding or stopping in situations where traffic conditions would allow safe progress, the cause lies with the planner. If the high-level decision is to proceed or follow, but the generated velocity profile remains excessively conservative due to overly restrictive constraints, the fault is attributed to the decider.

Finally, for the control module that executes the planned trajectory through lateral and longitudinal controllers, attribution is made by comparing the executed trajectory in the simulator with the planning output. If the actual lateral path deviates from the trajectory produced by planning, the violation is attributed to the lateral controller. If the actual velocity profile diverges from the planned velocity, the fault lies in the longitudinal controller. This systematic causal analysis provides fine-grained attribution at component level across the planning and control modules.

4 Evaluation

We have implemented a prototype of Poirot with 2,536 lines of C++ and Python code. To evaluate the effectiveness and efficiency of Poirot, we design the following three research questions.

Table 1. Distribution of Real and Injected Faults

ADS	Localization			Perception			Prediction			Planning			Control		
	Real	Injected	Total	Real	Injected	Total	Real	Injected	Total	Real	Injected	Total	Real	Injected	Total
Apollo	1	2	3	2	8	10	2	6	8	8	6	14	1	4	5
Autoware	0	3	3	2	6	8	2	6	8	9	6	15	2	4	6

- **RQ1**: How effective is Poirot in root cause analysis at module level and component level?
- **RQ2**: How efficient is Poirot in root cause analysis at module level and component level?
- **RQ3**: How could Poirot help analyze violations found by scenario-based testing approaches in practice, compared to manual analysis?

4.1 Experiment Setup

Target ADS and Simulator. We choose Apollo [4] and Autoware [20] as our target ADSs, which are the most representative industrial-grade ADSs with widespread commercialization. For Apollo, we select LGSVL 2021.3 [26, 38] as our simulation platform because LGSVL [57] offers stable connections with Apollo. For Autoware, we select CARLA 0.9.13 [17] as our simulation platform because CARLA is the most widely used open-source simulator for Autoware.

Benchmark. To evaluate the effectiveness and efficiency of root cause analysis in ADSs, we construct a benchmark containing 80 faults derived from both real and injected faults on Apollo and Autoware. The benchmark aims to provide reproducible and diverse safety violations, including collision, running red light, crossing yellow line, and failing to reach the destination, each of which is linked to a specific triggering scenario. To construct the benchmark, we first collected issues and pull requests from the GitHub repositories of Apollo and Autoware. For Apollo, we obtained a total of 1,181 issues and 906 pull requests, which were manually inspected to remove irrelevant entries. Through this process, we identified 14 safety violations for Apollo that can be stably reproduced in simulation, and validated them against the corresponding fixes suggested in the issues or pull requests, ensuring that the root causes were correctly captured and avoiding introducing new violations. Following the same procedure, we collected 15 reproducible safety violations for Autoware.

Since the number of real violations is limited, we further injected 26 faults into Apollo and 25 into Autoware following prior works [58, 65], targeting different modules to simulate representative safety violations. To ensure the injected faults are reasonable, we analyzed real faults in Apollo and Autoware, distilled common fault patterns (*e.g.,* misconfigurations, logic errors, and wrong boundaries), and injected faults accordingly. We further ensured that each injected fault (i) is triggered by at least one scenario in our benchmark, (ii) produces violations consistent with real fault patterns, and (iii) disappears after reverting the injected change. In total, the benchmark consists of 40 Apollo faults and 40 Autoware faults. The distribution of real and injected faults is summarized in Table 1, and Fig. 4 illustrates the hierarchical organization of cases by modularization granularity. Overall, our benchmark consists of 42 collision scenarios, 14 running red light scenarios, 10 crossing yellow line scenarios, and 14 scenarios where the ADS fails to reach the destination. It covers the 5 modules and 28 unique components in these two multi-module ADSs. The time duration of these scenarios ranges from 10 seconds to 35 seconds. Each case in the benchmark includes the ADS version, the violation description, the violation symptom, the triggering scenario, the fault type, the root cause (*i.e.,* the faulty module, and the faulty component), and the corresponding fix method. All the detailed information of the benchmark is available at our replication site [54].

Baseline. We compare Poirot with the state-of-the-art root cause analysis approaches of ADS simulation testing, *i.e.,* ACAV [58] and Rocas [19]. ACAV performs s-t-graph–based causal analysis on driving records to identify the functional causes of collisions. In our experiments, we used its official artifact with configurations and causal templates, and supplied the required specifications

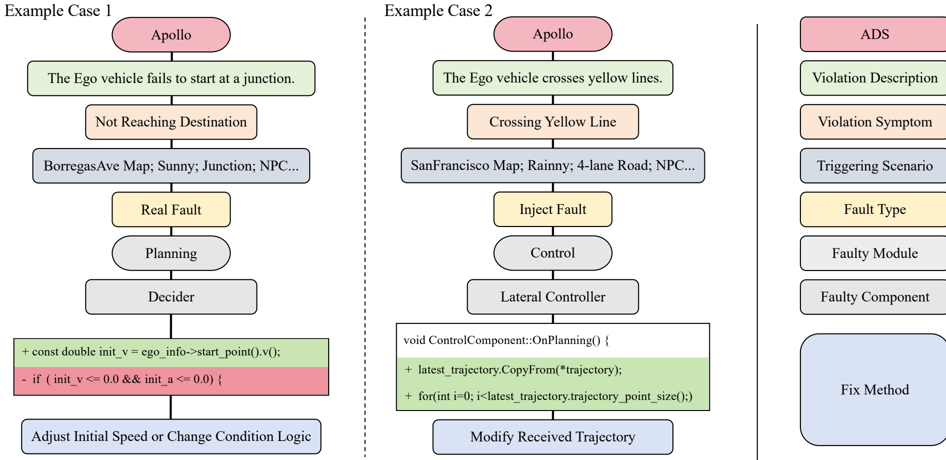


Fig. 4. Hierarchy of Cases in the Benchmark by Modularization Granularity

(*e.g.,* road topology). A case is counted as correct only if ACAV's functional cause maps to the same faulty module as our ground truth. Rocas compares message variations between two similar scenarios (*i.e.,* one resulting in an accident and one that does not), finding the triggering entity in the scenarios and tagging the first module with abrupt message changes as faulty. Since Rocas is not publicly available, we reimplement it based on the methodology described in its paper, and compare its accuracy of attributing the violations to the internal faulty modules in the ADS.

RQ Setup. For **RQ1**, and **RQ2**, we bridge the corresponding versions of Apollo with LGSVL, and Autoware with CARLA, respectively. We replay the triggering scenarios in our benchmark to reproduce the violations, and use Poirot to analyze the root causes of the violations.

For **RQ1**, we evaluate the effectiveness of Poirot in root cause analysis at both module level and component level. We measure the accuracy of Poirot in identifying the faulty module, comparing it with ACAV and Rocas, and report the component-level accuracy of Poirot.

For **RQ2**, we evaluate the efficiency of Poirot by measuring the average time it takes to localize the fault for each violation in our benchmark. We also report the per-step runtime of Poirot to provide a detailed analysis of the time overhead. Furthermore, to quantify the benefits of our suspicion-guided search strategy, we implement two variants of Poirot. Poirot$^{\flat}$ performs hypothetical reasoning by navigating the message flow in the *CMG* of the faulty module and iteratively replacing components until the faulty component is identified. Poirot$^{\natural}$ selects components solely based on their suspicion-score ranking, *i.e.,* it uses the scores without our search strategy. We compare Poirot with these two variants on violations in the localization, perception, and prediction modules by measuring the number of counterfactual executions, the average time consumption required and the proportion of the fault space explored to identify the faulty component.

For **RQ3**, we evaluate Poirot by applying it to two scenario-based testing approaches, *i.e.,* AV-Fuzzer [40] and MoDitector [66]. Specifically, we set up AV-Fuzzer and MoDitector with Apollo and the LGSVL simulator according to their documents. We run AV-Fuzzer and MoDitector for 12 hours. Then, two of the authors independently conduct manual root cause analysis, attributes the violations to the corresponding faults in Apollo to construct the ground truth. A third author is involved for a group discussion to resolve conflicts. Finally, the Cohen Kappa coefficient reaches 0.93. We report the accuracy of Poirot and the corresponding reduction in time required to find the fault.

Environment. We conduct all the experiments on Ubuntu 20.04.4 LTS servers with 4 NVIDIA GeForce RTX 3090 GPUs, Intel(R) Xeon(R) Silver 4310 @ 2.10GHz and 128GB memory.

Table 2. Results of the Effectiveness Evaluation

Level	Method	Apollo					Autoware				
		Localization	Perception	Prediction	Planning	Control	Localization	Perception	Prediction	Planning	Control
Module	ACAV	0.0%(0/3)	0.0%(0/10)	25.0%(2/8)	21.4%(3/14)	40.0%(2/5)	0.0%(0/3)	0.0%(0/8)	37.5%(3/8)	26.7%(4/15)	33.3%(2/6)
	Rocas	0.0%(0/3)	30.0%(3/10)	75.0%(6/8)	71.4%(10/14)	40.0%(2/5)	0.0%(0/3)	25.0%(2/8)	87.5%(7/8)	60.0%(9/15)	66.7%(4/6)
	Poirot	100.0%(3/3)	100.0%(10/10)	87.5%(7/8)	92.9%(13/14)	100.0%(5/5)	100.0%(3/3)	100.0%(8/8)	75.0%(6/8)	80.0%(12/15)	83.3%(5/6)
Component	Poirot	100.0%(3/3)	100.0%(10/10)	75.0%(6/8)	92.9%(13/14)	100.0%(5/5)	100.0%(3/3)	100.0%(8/8)	75.0%(6/8)	80.0%(12/15)	83.3%(5/6)

4.2 Effectiveness Evaluation (RQ1)

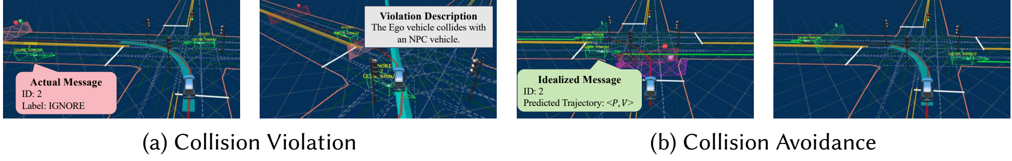
Module-Level Accuracy. Table 2 presents the effectiveness of Poirot in root cause analysis at both module level and component level. At the module level, Poirot achieves an average accuracy of 96.08% for Apollo and 87.66% for Autoware, with the highest accuracy of 100.00% on the localization and perception modules, and the lowest average accuracy of 81.25% on the prediction module. Compared with ACAV and Rocas, Poirot achieves an average accuracy improvement of 187.29%.

Component-Level Accuracy. Poirot maintains high accuracy across both ADSs. Poirot achieves an average accuracy of 93.58% for Apollo and 87.66% for Autoware, with the highest accuracy of 100.00% when identifying the faulty component in the localization and perception modules, and the lowest accuracy of 75.00% when identifying the faulty component in the prediction module.

Breakdown Analysis. For ACAV, it assumes a fault-free localization and perception module, which limits its effectiveness in identifying faults within these modules. Most of the inaccurate results stem from misdiagnosing prediction- and planning-related violations as control-related ones. Additionally, ACAV only analyzes collision violations, resulting in lower overall accuracy. In contrast, Poirot targets five modules in the ADS, and its hypothetical reasoning only requires a binary oracle. By observing whether the violation still exists through counterfactual execution, Poirot has better generalization to different types of violations. For Rocas, it first obtains a reference execution via physical mutation in the scenario and then uses the message difference to identify the initial deviating module. However, in dense and highly interactive traffic, even minor perturbations can trigger cascading effects, making it difficult to identify the start of fault propagation. Additionally, Rocas assumes violations are avoidable by tuning parameters, whereas many violations stem from logic errors, model defects, or missing functionalities. In contrast, Poirot provides a more rigorous oracle for distinguishing the relationship between internal modules and safety violations.

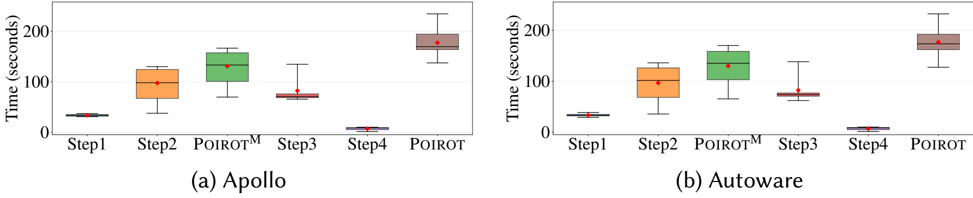
Poirot fails to achieve 100.00% accuracy when the root cause lies in the prediction, planning, and control modules. We manually inspected the cases where Poirot reported wrong results, and found that the inaccuracies mainly arise from the following reasons. First, fault tolerance and redundancy across modules can mask the true fault, leading to misattribution. For example, the planning module often incorporates fallback strategies to tolerate the uncertainty of upstream learning-based components. During counterfactual execution, when we replace the prediction module with an idealized prediction module, the ADS may switch to a different planning branch and thereby bypass latent faults in the planning module, causing the fault to be incorrectly attributed to the prediction module and failing to detect the downstream module faults. To this end, three prediction-related cases and four planning-related cases are misattributed to their upstream modules. Second, for the planning and control modules, using the simulator as an idealized control module may ignore real-world control constraints (*e.g.,* vehicle dynamics and actuation latency), which introduces one misclassification of control-related case at the boundary between the planning and control module.

Case Study. Fig. 5 shows an example of our root cause analysis on a real collision scenario in Apollo. In Fig. 5a, the Ego vehicle collides with an NPC vehicle at the intersection due to the faulty prediction module, which incorrectly assumes that the NPC vehicle will not interact with the Ego vehicle and labels it as *IGNORE*. We then sequentially replace the localization and perception modules, but the violation persists. Instead, as shown in Fig. 5b, when we replace the output of the prediction



(a) Collision Violation (b) Collision Avoidance

Fig. 5. Example of Counterfactual Execution by Replacing Prediction Module in Apollo



(a) Apollo (b) Autoware

Fig. 6. Results of the Efficiency Evaluation

module with the idealized prediction messages, including the NPC vehicle's future trajectories and removing the *IGNORE* tag. The ADS is able to stop and subsequently accelerate through the intersection safely after the NPC vehicle passes. Further component-level analysis pinpoints the faulty component within the prediction module as the *ObstacleFilter*, which labels distant vehicles as *IGNORE* without accounting for vehicle speed, ultimately failing to predict their future trajectories.

Summary. POIROT effectively identifies the faulty module with an average gain of accuracy by 187.29% across modules, compared to the baseline approach, and further localizes the faulty component within the identified faulty module with a high accuracy by 90.62%.

4.3 Efficiency Evaluation (RQ2)

Fig. 6 presents the time overhead of POIROT, along with a breakdown of each step for identifying faulty modules and components in Apollo and Autoware. Specifically, at the module level, POIROT (*i.e.,* the POIROT^{M} in Fig. 6) requires an average of 130.90 seconds for Apollo and 130.13 seconds for Autoware to identify the faulty module. Idealized message collection from the simulator (*i.e.,* Step-1) takes an average of 33.43 seconds across the two ADSs, which takes up 15.21% of the total runtime, while the conduction of multiple counterfactual executions (*i.e.,* Step-2) takes an average of 97.09 seconds across the two ADSs, accounting for 44.19% of the total runtime.

At the component level, the root cause analysis for the localization, perception, and prediction modules (*i.e.,* Step-3) takes an average of 82.19 seconds due to multiple counterfactual executions, accounting for 37.41% of the total runtime, while the component-level root cause analysis for the planning and control modules (*i.e.,* Step-4) only requires an average of 6.97 seconds.

Overall, the most time-consuming step in POIROT are conducting multiple counterfactual executions (*i.e.,* Step-2 and Step-3), which involves re-running the entire ADS simulation multiple times. Therefore, the scope reduction of counterfactual execution by suspicion-guided search strategy is necessary to make the fault localization affordable. Furthermore, we compare POIROT with two variants, POIROT$^{\flat}$ and POIROT$^{\natural}$, to evaluate the benefits of our suspicion-guided search strategy.

Table 3 shows that both variants degrade the efficiency of POIROT, with the most significant efficiency loss observed when the search-based strategy is removed. POIROT$^{\flat}$ requires the largest number of counterfactual executions and incurs the highest time cost. This is because every candidate component along the message flows is considered in turn, which significantly increases the number

Table 3. Results of the Efficiency Evaluation for the Suspicion-Guided Search Strategy in Poirot

Tool	**Ablation Item**		**Average Execution Times**		**Average Time Consumption (s)**		**Fault Space Exploration (%)**	
	Suspicion Score	*Search-Based Strategy*	Apollo	Autoware	Apollo	Autoware	Apollo	Autoware
Poirot$^\flat$			8.14	8.11	240.86	234.41	73.99	77.63
Poirot$^\natural$	✓		6.24	4.74	180.33	152.26	58.93	46.57
Poirot	✓	✓	2.81	2.63	82.11	82.26	30.41	32.11

Table 4. Results of the Usefulness Evaluation on Module-Level Fault Localization

Module	**Manual Analysis**	**Poirot**	**Precision**	**Recall**	**Accuracy**
Perception	104	105	93.33%	94.23%	92.94%
Prediction	92	89	95.51%	92.39%	
Planning	159	159	93.71%	93.71%	
Control	70	72	87.50%	90.00%	

of trials. Introducing suspicion scores in Poirot$^\natural$ partially alleviates this issue, reducing the average number of counterfactual executions by 23.34% on Apollo and 41.55% on Autoware, and decreasing time cost by 25.13% and 35.05%, respectively. However, since Poirot$^\natural$ does not employ our search-based strategy, it may still suffer from the huge fault space. In contrast, Poirot, compared with Poirot$^\natural$, reduces the average number of counterfactual executions by 49.74%, and decreases time cost by about 50.22%. Besides, Poirot reduces the proportion of the fault space explored to identify the faulty component by 48.40% on Apollo and 31.05% on Autoware, compared with Poirot$^\natural$.

Summary. Poirot incurs moderate time overhead for both module- and component-level root cause analysis on Apollo and Autoware, where the dominant cost comes from multiple counterfactual executions that require re-running the ADS simulation. To make this time cost affordable, Poirot leverages a suspicion-guided search strategy to enhance the efficiency of faulty component localization by 58.77% in fault space exploration and 65.41% in time consumption, respectively.

4.4 Usefulness Evaluation (RQ3)

After 12 hours of simulation-based testing, AV-Fuzzer and MoDitector generated a total of 665 and 837 scenarios, respectively, uncovering 245 and 180 safety violations. Following more than 672 hours (*i.e.,* four weeks) of manual analysis, we ultimately obtained 425 violation scenarios and eight unique faulty components across four modules because the faulty modules and some faulty components were common across the violation scenarios produced by the two scenario-based testing approaches. We then applied Poirot to these violation scenarios for root cause analysis, and compared the results with manual analysis. Table 4 reports results of Poirot and manual analysis at the module-level. Poirot attributes 395 out of 425 violation scenarios to the correct faulty modules. Overall, Poirot achieves an accuracy of 92.94%, a macro-averaged precision of 92.51% and a macro-averaged recall of 92.58%. Besides, with respect to the component-level analysis, Poirot identifies 8 faulty components, and attributes 382 out of 425 violation scenarios to the correct faulty components. Overall, Poirot achieves an accuracy of 89.88%, a macro-averaged precision of 88.80% and a macro-averaged recall of 89.00%. The detailed results of the component-level fault localization are available at our replication site [54] due to the space limitation.

In terms of efficiency, Poirot required only 15.45 hours in total to identify the faulty modules and 20.93 hours to identify the faulty components, reducing the time required by at least 94.59% compared to manual analysis. This substantial reduction in analysis time highlights the practical value of Poirot for accelerating the debugging and validation process in ADSs.

Summary. Poirot effectively provides reliable fault localization for safety violations in ADS simulation testing, attributing a total of 425 violations to 8 unique faulty components, reducing the time consumption by at least 94.59% compared to manual analysis.

4.5 Threats to Validity

First, the benchmark's inherent limitations pose a potential threat to validity. For real faults collected from issues and pull requests, we can reconstruct only a limited subset of driving-violation scenarios, since many reported faults (*e.g.*, build, documentation, or UI issues) are outside the scope of ADS safety violations. In addition, reports often omit sufficient details on how violations are triggered or resolved, which lowers the reconstruction success rate and increases reproduction effort. For injected faults, they may not fully reflect real-world fault characteristics in ADSs. To reduce this threat, we adopt injected faults from prior work [58, 65], covering misconfigurations, logic errors, and missing checks. These faults yield violations similar to those induced by real faults, enabling a meaningful evaluation of root cause analysis tools in ADS simulation testing.

Second, the diversity of violations poses a potential threat to validity. To mitigate this, our current implementation supports four violation types, which cover the most frequently observed safety violations in real-world driving and those exposed in our benchmark. Extending Poirot to other violation types does not require architectural changes. In practice, it only entails integrating a violation-specific oracle to monitor the relevant signals and determine whether the violation still exists after replacing the modules or components during hypothetical reasoning. Such oracles are widely available in prior ADS testing literature [59]. Therefore, Poirot can be readily generalized to a broader range of safety violations by incorporating the corresponding oracles.

Third, the limitation to analyze multi-module or multi-component faults poses a potential threat to validity. However, our benchmark construction process and the reproduction of real fault scenarios indicate that the most violations are typically triggered by a single fault. Moreover, prior studies [9, 21] suggest that existing bugs in multi-module ADSs can generally be resolved through a single fix within the corresponding module or component. Importantly, even when a violation results from multi-fault interactions, Poirot is designed to identify the start module or component whose abnormal behavior propagates to the violation. After fixing this first faulty module, Poirot can be re-run on the fixed ADS to further localize the remaining faults.

Finally, the general applicability of Poirot constitutes a potential threat. To address this, we evaluate Poirot on two representative ADSs and two simulation platforms. Modern multi-module ADSs generally share architectural similarities with Apollo and Autoware, suggesting that our approach is broadly applicable. Additionally, we demonstrate Poirot's applicability across two simulation testing approaches, *i.e.,* AV-Fuzzer and MoDitector, further validating its practical relevance.

5 Related Work

Root Cause Analysis in ADSs. Scenario-based testing approaches [11, 14, 27, 28, 33, 39, 40, 45, 47, 59–61, 67, 72, 74, 75] have been widely explored to generate diverse driving scenarios for ADSs, aiming to detect safety violations. However, their focus lies in determining whether the ADSs fail to meet the test oracles, without revealing the underlying causes of the violations. Recent works [22, 66] attempt to generate violation scenarios triggered by specific ADS modules. In contrast, Poirot focuses on post-accident analysis to identify the causes of safety violations in simulation testing, which is complementary to the testing approaches mentioned above. Some studies prioritize critical driving records [15] or extract key violation features [50], but treat ADSs as black boxes, lacking links between violations and internal components. ARIEL [1] repairs the interaction features of a rule-based ADS without identifying the causes of safety violations in ADS simulation testing.

Rocas [19] and ConfVE [10] detect misconfigurations by mutating configuration parameters in the ADS. Lu et al. [46] propose a LLM-based approach to automatically diagnose the safety violation by reliability determination and crash classification. These works diverge from our research goal of attributing safety violations to the inner faulty components. ACAV [58] applies causality analysis to locate faulty modules, but cannot map collisions to specific components. Based on this work, Poirot attributes four types of violations via hypothetical reasoning, *i.e.,* collision, running red light, crossing yellow line, and not reaching destination, conducting root cause analysis at the component level. Compared with [65], which relies on idealized substitutes but suffers from large fault spaces, our work adopts a suspicion-guided search strategy to improve efficiency and provides a comprehensive benchmark for root cause analysis of safety violations in ADS simulation testing.

Root Cause Analysis. Root cause analysis is essential for debugging failures in software systems [35, 42, 70], with existing approaches including log-based [36, 37, 51, 69] and replay-based [7, 64] causality analysis. Chen et al. [8] apply counterfactual reasoning to momentum wheels for root cause analysis. Li et al. [42] employ causal inference with intervention recognition for online service systems, and Liu et al. [43] focus on large-scale microservice systems. Besides, some works explore the bias in deep learning models [5, 6, 13, 62]. Recently, several works [24, 55, 63, 68, 71] investigate root cause analysis in cyber-physical systems such as robots [25, 31] and drones [29]. Swarmbug [30] treats swarm robotics as a black box to investigate configuration effects on system behavior. MAYDAY [34] applies program analysis to diagnose controller or mission command bugs in drones. RVPlayer [12] replays drone missions in simulation, using hypothetical reasoning to assess whether parameter or code changes could prevent accidents. However, these approaches are not designed for safety violation attribution in ADS simulation testing, while our work applies hypothetical reasoning to identify the root causes.

6 Conclusion

We have proposed and implemented Poirot, a framework for root cause analysis in ADS simulation testing via hypothetical reasoning, along with a comprehensive benchmark for evaluation. Large-scale experiments have been conducted to demonstrate the effectiveness and efficiency of Poirot.

Data Availability

The source code of Poirot and our benchmark is available at our replication site [54].

References

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2020. Automated repair of feature interaction failures in automated driving systems. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis.* 88–100.

[2] Baidu Apollo. 2024. *CyberRT.* Retrieved August 12, 2025 from https://carla.org/

[3] Daniel Atherton. 2022. Incident 434: Sudden Braking by Tesla Allegedly on Self-Driving Mode Caused Multi-Car Pileup in Tunnel. In *AI Incident Database*, Khoa Lam (Ed.). Responsible AI Collaborative. Retrieved February 13, 2023 from https://incidentdatabase.ai/cite/434/

[4] Baidu. 2022. *Apollo: An open autonomous driving platform.* Retrieved August 12, 2025 from https://github.com/ApolloAuto/apollo

[5] Elias Bareinboim and Jin Tian. 2015. Recovering causal effects from selection bias. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29.

[6] Shruti Bothe, Usama Masood, Hasan Farooq, and Ali Imran. 2020. Neuromorphic AI empowered root cause analysis of faults in emerging networks. In *2020 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom).* 1–6.

[7] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. 2011. Intrusion recovery for database-backed web applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles.* 101–114.

[8] Siya Chen, Guang Jin, and Chunhui Ji. 2023. Root-Cause Analysis of Momentum Wheel Fault Based on Counterfactual Reasoning. In *Proceedings of the 14th International Conference on Reliability, Maintainability and Safety.* IEEE, 284–287.

[9] Yuntianyi Chen, Yuqi Huai, Yirui He, Shilong Li, Changnam Hong, Qi Alfred Chen, and Joshua Garcia. 2025. A Comprehensive Study of Bug-Fix Patterns in Autonomous Driving Systems. *Proc. ACM Softw. Eng.* 2, FSE (2025), 380–402.

[10] Yuntianyi Chen, Yuqi Huai, Shilong Li, Changnam Hong, and Joshua Garcia. 2024. Misconfiguration software testing for failure emergence in autonomous driving systems. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1913–1936.

[11] Mingfei Cheng, Yuan Zhou, and Xiaofei Xie. 2023. Behavexplor: Behavior diversity guided testing for autonomous driving systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 488–500.

[12] Hongjun Choi, Zhiyuan Cheng, and Xiangyu Zhang. 2022. Rvplayer: Robotic vehicle forensics by replay with what-if reasoning. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium*.

[13] Juan Correa and Elias Bareinboim. 2017. Causal effect identification by adjustment under confounding and selection biases. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[14] Jiarun Dai, Bufan Gao, Mingyuan Luo, Zongan Huang, Zhongrui Li, Yuan Zhang, and Min Yang. 2024. SCTrans: Constructing a Large Public Scenario Dataset for Simulation Testing of Autonomous Driving Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[15] Yao Deng, Xi Zheng, Mengshi Zhang, Guannan Lou, and Tianyi Zhang. 2022. Scenario-based test reduction and prioritization for multi-module autonomous driving systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 82–93.

[16] Wenhao Ding, Chejian Xu, Mansur Arief, Haohong Lin, Bo Li, and Ding Zhao. 2023. A Survey on Safety-Critical Driving Scenario Generation—A Methodological Perspective. *IEEE Transactions on Intelligent Transportation Systems* 24, 7 (2023), 6971–6988.

[17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[18] erdos project. 2022. *Pylot.* Retrieved August 12, 2025 from https://github.com/erdos-project/pylot

[19] Shiwei Feng, Yapeng Ye, Qingkai Shi, Zhiyuan Cheng, Xiangzhe Xu, Siyuan Cheng, Hongjun Choi, and Xiangyu Zhang. 2024. ROCAS: Root Cause Analysis of Autonomous Driving Accidents via Cyber-Physical Co-mutation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1620–1632.

[20] The Autoware Foundation. 2023. *Welcome to the Autoware Foundation.* Retrieved August 12, 2025 from https://autoware.org/

[21] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A Comprehensive Study of Autonomous Vehicle Bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 385–396.

[22] Luca Giamattei, Antonio Guerriero, Roberto Pietrantuono, and Stefano Russo. 2024. Causality-driven testing of autonomous driving systems. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–35.

[23] Catherine Gray. 2022. *Top 10 companies developing autonomous vehicle technology.* Retrieved August 25, 2024 from https://aimagazine.com/technology/top-10-companies-developing-autonomous-vehicle-technology

[24] Zhijian He, Yao Chen, Enyan Huang, Qixin Wang, Yu Pei, and Haidong Yuan. 2019. A system identification based oracle for control-cps software fault localization. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 116–127.

[25] Md Abir Hossen, Sonam Kharade, Bradley Schmerl, Javier Cámara, Jason M O'Kane, Ellen C Czaplinski, Katherine A Dzurilla, David Garlan, and Pooyan Jamshidi. 2023. CaRE: Finding Root Causes of Configuration Issues in Highly-Configurable Robots. *IEEE Robotics and Automation Letters* 8, 7 (2023), 4115–4122.

[26] Yuqi Huai. 2023. *SORA-SVL.* Retrieved January 12, 2025 from https://www.ics.uci.edu/~yhuai/SORA-SVL/

[27] Y. Huai, S. Almanee, Y. Chen, X. Wu, Q. Chen, and J. Garcia. 2023. scenoRITA: Generating Diverse, Fully Mutable, Test Scenarios for Autonomous Vehicle Planning. *IEEE Transactions on Software Engineering* 49, 10 (2023), 4656–4676.

[28] Yuqi Huai, Yuntianyi Chen, Sumaya Almanee, Tuan Ngo, Xiang Liao, Ziwen Wan, Qi Alfred Chen, and Joshua Garcia. 2023. Doppelgänger Test Generation for Revealing Bugs in Autonomous Driving Software. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering*. 2591–2603.

[29] Upasita Jain, Marcus Rogers, and Eric T Matson. 2017. Drone forensic framework: Sensor and data identification and verification. In *Proceedings of the IEEE Sensors Applications Symposium*. 1–6.

[30] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. 2021. Swarmbug: debugging configuration bugs in swarm robotics. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 868–880.

[31] Eliahu Khalastchi and Meir Kalech. 2018. On fault detection and diagnosis in robotic systems. *Comput. Surveys* 51, 1 (2018), 1–24.

[32] Srishti Khemka. 2021. Incident 347: Waymo Self-Driving Taxi Behaved Unexpectedly, Driving away from Support Crew. In *AI Incident Database*, Khoa Lam (Ed.). Responsible AI Collaborative. Retrieved August 13, 2025 from

https://incidentdatabase.ai/cite/347/

[33] Seulbae Kim, Major Liu, Junghwan" John" Rhee, Yuseok Jeon, Yonghwi Kwon, and Chung Hwan Kim. 2022. Drivefuzz: Discovering autonomous driving bugs through driving quality-guided fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1753–1767.

[34] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave Jing Tian, and Dongyan Xu. 2020. From control model to program: Investigating robotic aerial vehicle accidents with {MAYDAY}. In *Proceedings of the 29th USENIX Security Symposium*. 913–930.

[35] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2010. Intrusion recovery using selective re-execution. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. 89–104.

[36] Srinivas Krishnan, Kevin Z Snow, and Fabian Monrose. 2010. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM conference on Computer and communications security*. 50–60.

[37] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition.. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, Vol. 16.

[38] LGSVl. 2021. *SVL Simulator: An Autonomous Vehicle Simulator.* Retrieved August 12, 2025 from https://github.com/lgsvl/simulator/releases/tag/2021.3

[39] Changwen Li, Joseph Sifakis, Qiang Wang, Rongjie Yan, and Jian Zhang. 2023. Simulation-Based Validation for Autonomous Driving Systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 842–853.

[40] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2020. Av-fuzzer: Finding safety violations in autonomous driving systems. In *Proceedings of the 2020 IEEE 31st International Symposium on Software Reliability Engineering*. 25–36.

[41] Li Li, Wu-Ling Huang, Yuehu Liu, Nan-Ning Zheng, and Fei-Yue Wang. 2016. Intelligence testing for autonomous vehicles: A new approach. *IEEE Transactions on Intelligent Vehicles* 1, 2 (2016), 158–166.

[42] Mingjie Li, Zeyan Li, Kanglin Yin, Xiaohui Nie, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2022. Causal inference-based root cause analysis for online service systems with intervention recognition. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3230–3240.

[43] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. 2021. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering*. 338–347.

[44] Guannan Lou, Yao Deng, Xi Zheng, Mengshi Zhang, and Tianyi Zhang. 2022. Testing of autonomous driving systems: where are we and where should we go?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 31–43.

[45] Chengjie Lu. 2023. Test Scenario Generation for Autonomous Driving Systems with Reinforcement Learning. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings*. 317–319.

[46] You Lu, Yifan Tian, Yuyang Bi, Bihuan Chen, and Xin Peng. 2024. Diavio: LLM-Empowered Diagnosis of Safety Violations in ADS Simulation Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–388.

[47] Yixing Luo, Xiao-Yi Zhang, Paolo Arcaini, Zhi Jin, Haiyan Zhao, Fuyuki Ishikawa, Rongxin Wu, and Tao Xie. 2021. Targeting requirements violations of autonomous driving systems by dynamic evolutionary search. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering*. 279–291.

[48] Interactive Mathematics. 2024. *Velocity (s-t) Graphs.* Retrieved August 12, 2024 from https://www.intmath.com/kinematics/1-velocity-graphs.php

[49] Sean McGregor. 2018. Incident 4: Uber AV Killed Pedestrian in Arizona. In *AI Incident Database*, Sean McGregor (Ed.). Responsible AI Collaborative. Retrieved August 13, 2025 from https://incidentdatabase.ai/cite/4

[50] Neelofar Neelofar and Aldeida Aleti. 2024. Identifying and explaining safety-critical scenarios for autonomous vehicles via key features. *ACM Transactions on Software Engineering and Methodology* 33, 4 (2024), 1–32.

[51] Peter Ohmann and Ben Liblit. 2017. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. *Automated Software Engineering* 24, 4 (2017), 865–904.

[52] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. 2016. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles* 1, 1 (2016), 33–55.

[53] Francesca Pianosi and Thorsten Wagener. 2015. A simple and efficient method for global sensitivity analysis based on cumulative distribution functions. *Environmental Modelling & Software* 67 (2015), 1–11.

[54] Poirot. 2026. *Poirot.* Retrieved January 27, 2026 from https://poirot4ads.github.io/Poirot/

[55] Christopher M Poskitt, Yuqi Chen, Jun Sun, and Yu Jiang. 2023. Finding causally different tests for an industrial control system. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. 2578–2590.

[56] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *Proceedings of the ICRA workshop on open source software*, Vol. 3. 5.

[57] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. 2020. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. *arXiv preprint arXiv:2005.03778* (2020).

[58] Huijia Sun, Christopher M Poskitt, Yang Sun, Jun Sun, and Yuqi Chen. 2024. ACAV: a framework for automatic causality analysis in autonomous vehicle accident recordings. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[59] Yang Sun, Christopher M Poskitt, Jun Sun, Yuqi Chen, and Zijiang Yang. 2022. LawBreaker: An approach for specifying traffic laws and fuzzing autonomous vehicles. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[60] Haoxiang Tian, Yan Jiang, Guoquan Wu, Jiren Yan, Jun Wei, Wei Chen, Shuo Li, and Dan Ye. 2022. MOSAT: finding safety violations of autonomous driving systems using multi-objective genetic algorithm. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 94–106.

[61] Haoxiang Tian, Guoquan Wu, Jiren Yan, Yan Jiang, Jun Wei, Wei Chen, Shuo Li, and Dan Ye. 2022. Generating Critical Test Scenarios for Autonomous Driving Systems via Influential Behavior Patterns. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[62] Ruibo Tu, Kun Zhang, Bo Bertilson, Hedvig Kjellstrom, and Cheng Zhang. 2019. Neuropathic pain diagnosis simulator for causal discovery algorithm evaluation. *Advances in Neural Information Processing Systems* 32 (2019).

[63] Meriel Von Stein and Sebastian Elbaum. 2021. Automated environment reduction for debugging robotic systems. In *Proceedings of the 2021 IEEE International Conference on Robotics and Automation*. 3985–3991.

[64] Gregory Walkup, Sriharsha Etigowni, Dongyan Xu, Vincent Urias, and Han W Lin. 2020. Forensic investigation of industrial control systems using deterministic replay. In *Proceedings of the 2020 IEEE Conference on Communications and Network Security*. 1–9.

[65] Ziwen Wan, Yuqi Huai, Yuntianyi Chen, Joshua Garcia, and Qi Alfred Chen. 2024. Towards Automated Driving Violation Cause Analysis in Scenario-Based Testing for Autonomous Driving Systems. *arXiv preprint arXiv:2401.10443* (2024).

[66] Renzhi Wang, Mingfei Cheng, Xiaofei Xie, Yuan Zhou, and Lei Ma. 2025. MoDitector: Module-Directed Testing for Autonomous Driving Systems. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 137–158.

[67] Tong Wang, Taotao Gu, Huan Deng, Hu Li, Xiaohui Kuang, and Gang Zhao. 2024. Dance of the ads: Orchestrating failures through historically-informed scenario fuzzing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1086–1098.

[68] Tomoya Yamaguchi, Bardh Hoxha, Danil Prokhorov, and Jyotirmoy V Deshmukh. 2020. Specification-guided software fault localization for autonomous mobile systems. In *Proceedings of the 18th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 1–12.

[69] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*. 143–154.

[70] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering* 28, 2 (2002), 183–200.

[71] Renbin Zhang, Zongze Cao, and Kewei Wu. 2020. Tracing and detection of ICS anomalies based on causality mutations. In *Proceedings of the IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*. 511–517.

[72] Xudong Zhang and Yan Cai. 2023. Building critical testing scenarios for autonomous driving from real accidents. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 462–474.

[73] Xinhai Zhang, Jianbo Tao, Kaige Tan, Martin Törngren, José Manuel Gaspar Sánchez, Muhammad Rusyadi Ramli, Xin Tao, Magnus Gyllenhammar, Franz Wotawa, Naveen Mohan, Mihai Nica, and Hermann Felbinger. 2023. Finding Critical Scenarios for Automated Driving Systems: A Systematic Mapping Study. *IEEE Transactions on Software Engineering* 49, 3 (2023), 991–1026.

[74] Xiaodong Zhang, Wei Zhao, Yang Sun, Jun Sun, Yulong Shen, Xuewen Dong, and Zijiang Yang. 2023. Testing automated driving systems by breaking many laws efficiently. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 942–953.

[75] Ziyuan Zhong, Gail Kaiser, and Baishakhi Ray. 2023. Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1860–1875.

[76] Ziyuan Zhong, Yun Tang, Yuan Zhou, Vania de Oliveira Neves, Yang Liu, and Baishakhi Ray. 2021. A survey on scenario-based testing for automated driving systems in high-fidelity simulation. *arXiv preprint arXiv:2112.00964*

(2021).