Branch-Level Fault Localization in ADS Planning via Temporal Coverage Analysis

SANGMIN WOO, KAIST, Republic of Korea
DOHYUN KIM, KAIST, Republic of Korea
DONGHWAN SHIN*, University of Sheffield, United Kingdom
YONGDAE KIM*, KAIST, Republic of Korea

Planning failures in Automated Driving Systems (ADS) are increasingly detected through simulation-based testing, yet localizing their root causes within planning code remains a major challenge. Planning modules execute complex rule-based decision logic over hundreds of frames in a closed-loop interaction with the environment, where faults trigger observable failures only after temporal gaps and under specific execution contexts. These characteristics make traditional spectrum-based fault localization ineffective, as faulty behavior is obscured by execution-level coverage aggregation and limited test diversity. In this paper, we study the problem of debugging planning failures and present a temporal coverage analysis approach for localizing faults in rule-based planning modules. Our key insight is that, while execution-aggregated coverage masks fault behavior, frame-level execution dynamics reveal distinctive temporal signatures that indicate when and how faulty branches activate. Leveraging this insight, our approach first identifies a suspicious frame using planning semantics, and then ranks candidate branches by analyzing their execution behavior within a localized temporal window. We evaluate our approach on 221 reproducible non-collision Apollo planning failures, covering immobility and emergency mission failures. Our results show that temporal coverage analysis enables accurate suspicious-frame identification and substantially reduces branch inspection effort compared to oracle-based and random baselines, effectively localizing faults from a single failing execution. We further analyze failure cases that lack observable execution signals to clarify the fundamental limits of execution-based localization. Overall, this work demonstrates that temporal execution analysis provides a practical and effective foundation for debugging planning failures in rule-based ADS planning modules.

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

Additional Key Words and Phrases: Automated Driving System, Debugging, Fault Localization

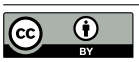
ACM Reference Format:
Sangmin Woo, Dohyun Kim, Donghwan Shin, and Yongdae Kim. 2026. Branch-Level Fault Localization in ADS Planning via Temporal Coverage Analysis. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA044 (October 2026), 23 pages. https://doi.org/10.1145/3832135

1 Introduction

Automated Driving Systems (ADS) have rapidly advanced toward commercial deployment, yet safety-critical failures continue to occur in real-world operation [9, 22, 35, 39]. Among these, planning failures such as unexpected stops at intersections, failure to complete lane changes, or

*Corresponding authors

Authors' Contact Information: Sangmin Woo, KAIST, Daejeon, Republic of Korea, wsm723@kaist.ac.kr; Dohyun Kim, KAIST, Daejeon, Republic of Korea, dohyunjk@kaist.ac.kr; Donghwan Shin, University of Sheffield, Sheffield, United Kingdom, d.shin@sheffield.ac.uk; Yongdae Kim, KAIST, Daejeon, Republic of Korea, yongdaek@kaist.ac.kr.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA044
https://doi.org/10.1145/3832135

inability to execute emergency maneuvers remain particularly problematic because they directly affect vehicle behavior and safety.

The planning module is a critical debugging target responsible for the decision-making that determines a vehicle's motion and trajectory. The planning module periodically computes trajectories based on surrounding objects, localization, and map data, with each computation cycle referred to as a frame. While recent research has explored learning-based approaches to planning modules, many production ADS still rely on rule-based decision logic to ensure safety and interpretability. Consequently, effective fault localization techniques for identifying faults within rule-based planning logic remain essential for ADS development and deployment. While prior work has demonstrated that such failures can be detected through systematic testing and large-scale simulation, understanding why they occur and localizing their root causes remain challenging [25, 28, 31, 44, 47].

Most existing debugging tools localize faults only at module or component granularity [21, 42, 45], leaving developers to manually inspect hundreds of candidate branches (i.e., control-flow edges of conditional statements) in complex planning logic. Traditional spectrum-based fault localization (SBFL) techniques are ill-suited to this setting [1, 23, 24, 51, 52]. SBFL relies on statistical comparison across multiple independent executions, whereas planning failures often manifest as temporal anomalies within a single execution. A branch may execute correctly in one frame but contribute to failure several frames later due to accumulated internal state and evolving environmental context.

Debugging the planning module presents two fundamental challenges. First, temporal gaps exist between fault execution and observable symptoms. For example, vehicles may stop due to trajectory optimization failures caused by unstable vehicle states from faulty decision logic executed many frames earlier. Without automated assistance, developers must manually trace execution backward through long decision sequences to identify faults. Second, planning failures arise from closed-loop, temporally chained decision making where discrete events across frames interact through accumulated state. A branch that executes correctly in one frame may only lead to failure after a sequence of subsequent planning decisions and environment feedback. As a result, faults are often context- and history-dependent, becoming observable only under specific temporal conditions. This closed-loop execution fundamentally violates SBFL's core assumption that faulty code consistently correlates with failing executions across runs, explaining why execution-level coverage aggregation obscures fault-specific behavior in the planning module.

In this work, we focus on the debugging problem for planning failures, rather than on failure detection itself. To study this problem concretely, we ground our analysis in planning failures reported in recent work on Apollo, an industry-grade module-based ADS [25]. We successfully reproduced 221 non-collision planning failures (immobility and emergency mission failures), enabling systematic investigation of temporal fault behavior in planning logic.

Through this investigation, we observe that while coverage aggregated over entire executions obscures fault-specific behavior, frame-level execution dynamics expose when and how faults activate. A faulty branch may execute correctly in most frames and even in other test cases, but contribute to failure only when executed under a specific accumulated internal state. This state is carried by contextual variables that track the history of prior planning decisions, such as routing records or active behavioral modes. As these variables are gradually corrupted over successive frames, the branch can eventually trigger a fault under conditions that would otherwise appear benign. The resulting faults exhibit characteristic temporal behaviors, such as sudden coverage changes at specific frames, that remain invisible when execution coverage is aggregated. This makes frame-level analysis more suitable for localization than cross-execution statistical comparison.

Based on this observation, we present a temporal coverage analysis technique for localizing faults in rule-based planning modules. Our approach has four stages: manual analysis extracts planning semantics and diagnostic variables; runtime data generation collects per-frame traces and

branch coverage; suspicious frame localization bridges the temporal gap by identifying the likely fault-related frame; and suspicious branch localization ranks candidate branches using frame-level coverage behavior. This enables branch-level localization from a single failing execution without requiring multiple passing or failing test variants.

This paper makes the following key contributions:

- Through a systematic analysis of reproduced planning failures, we characterize major failure categories and their distinct temporal execution behaviors, providing the empirical foundation for our approach.
- We propose a temporal coverage analysis method that shifts fault localization from cross-execution statistical comparison to intra-execution temporal analysis, enabling branch-level localization from a single failing execution.
- We evaluate our approach on 221 reproducible Apollo planning failures with manually identified fault-triggering frames and faulty branches as ground truth, showing that it accurately identifies suspicious frames and substantially reduces branch inspection effort compared to baselines.

2 Background

2.1 Software Architecture of ADS

Many open-source ADS platforms [2, 5, 12], as well as production-level autonomous driving services [16, 49], adopt a modular software architecture. This architecture typically consists of distinct modules for perception, prediction, planning, and control that operate in a closed-loop cycle, continuously processing sensor data to generate safe driving maneuvers.

Each cycle begins with the perception module processing sensor measurements to detect surrounding objects, their positions and types, along with traffic signals and road signs. The prediction module then forecasts the future behavior of surrounding objects using the perception output. The planning module generates a safe and efficient trajectory based on environmental information from previous modules and map data. Finally, the control module produces acceleration, braking, and steering commands to execute the planned trajectory. These commands actuate vehicle movement, and the next cycle begins by capturing the updated environment.

2.2 Planning Module of ADS

The planning module of ADS creates a safe and efficient driving trajectory for the ego vehicle (the vehicle being controlled by the ADS), taking into account the ego vehicle's current state, destination, and the environment surrounding it. Two primary approaches exist for implementing planning modules: rule-based planning and learning-based planning [20]. Rule-based approaches use explicitly defined logic and decision rules derived from expert knowledge. Learning-based approaches employ machine learning models trained on driving data to generate trajectories. This work focuses on rule-based planning systems, which are widely used in open-source ADS platforms. Rule-based planning modules typically employ a hierarchical, multi-stage architecture to generate safe trajectories. Open-source platforms like Apollo, Autoware, and Pylot implement three main stages: route planning, behavioral planning, and motion planning [2, 5, 10, 12, 13, 17].

Route planning generates a global path (i.e., the route provided by the routing module that specifies the sequence of road segments to reach the destination) from the current position to the destination by applying graph search algorithms (e.g., Dijkstra, A*) to road network data. Behavioral planning manages high-level decisions through state-based logic. For example, Apollo uses an explicit scenario system where each scenario encodes decision sequences for specific traffic situations. Consider a stop sign scenario that defines how the vehicle should behave when encountering a stop sign. The scenario specifies the sequence of actions to decelerate, stop before

the line, wait for clearance, and then proceed. Finally, motion planning refines the trajectory by optimizing path and speed profiles while performing collision avoidance based on surrounding object states. Faults at any planning stage can cause safety-critical failures at the system-level, necessitating systematic debugging approaches.

3 Characterizing Planning Failures

3.1 Reproducing Planning Failures

To study planning failures concretely, we require a dataset of failures that can be systematically reproduced and analyzed under closed-loop execution. We collected planning failures from recent work on automated testing of ADS that targets planning failures [25, 43, 47]. Three sources provided initial failure cases; however, two sources [44, 46] exhibited reproducibility challenges due to simulator version incompatibilities, deprecated resources, or behavioral differences in our environment. The remaining source, SafePlanner [26], provides a publicly available dataset of 520 end-to-end failure-inducing simulation inputs evaluated on Apollo. It also identified emergency mission failures, such as pulling over at hazardous locations or incorrectly repeating already-completed emergency missions, expanding the scope of known planning failures. This dataset serves as the starting point of our study. To isolate faults originating within the planning module from upstream perception errors, we followed prior work [44, 46] and replaced the perception module with a 3D ground truth sensor during reproduction [29]. This setup ensures that any observed failure is attributable to the planning module's own decision logic, rather than to misdetections or noise in perception. Using this setup, we successfully reproduced 221 planning failure cases. Notably, collision-related failures present in SafePlanner's original dataset disappeared after reproduction with ground truth perception, confirming that those collisions had originated from perception-planning interactions rather than from planning logic alone. Our approach does not target collision failures, as they involve multi-module interactions that require fundamentally different analysis. This scope offers a key analytical advantage: failures that persist under ground-truth perception are more directly tied to planning-module decision logic, allowing the analysis to focus on planning-level fault behavior without cross-module attribution.

The 221 reproduced failures consist of two failure types: immobility failures and emergency mission failures. Immobility failures occur when the ego vehicle permanently stops and cannot proceed, typically detected through oracle checks that classify the vehicle as immobilized after remaining stationary for a specified duration [25, 43, 47]. Emergency mission failures occur when the vehicle fails to execute critical safety maneuvers such as emergency pull-over [25]. Through systematic investigation of these cases, we made a key observation: there exists a temporal gap between when a fault executes and when its effects manifest as system-level symptoms. Furthermore, some cases exhibited repeated planning decisions such as continuous rerouting attempts, suggesting that fault activation depends on accumulated internal state rather than isolated code paths. These observations motivate frame-level temporal analysis rather than traditional aggregated coverage methods. We illustrate this with a concrete example in the following section.

3.2 Illustrative Example

Figure 1 illustrates a representative failure example actually discovered in a testing study of Apollo [25]. In this case, the ego vehicle starts in the fourth lane and receives a destination requiring lane changes to the third lane, followed by a left turn at the intersection **(Figure 1 (a))**. However, the ego vehicle fails to change lanes due to an NPC (Non-Player Character) vehicle traveling in the third lane **(Figure 1 (b))**. Finally, the ego vehicle forcibly attempts to turn left despite failing to reach the leftmost lane, resulting in immobilization within the intersection **(Figure 1 (c))**.

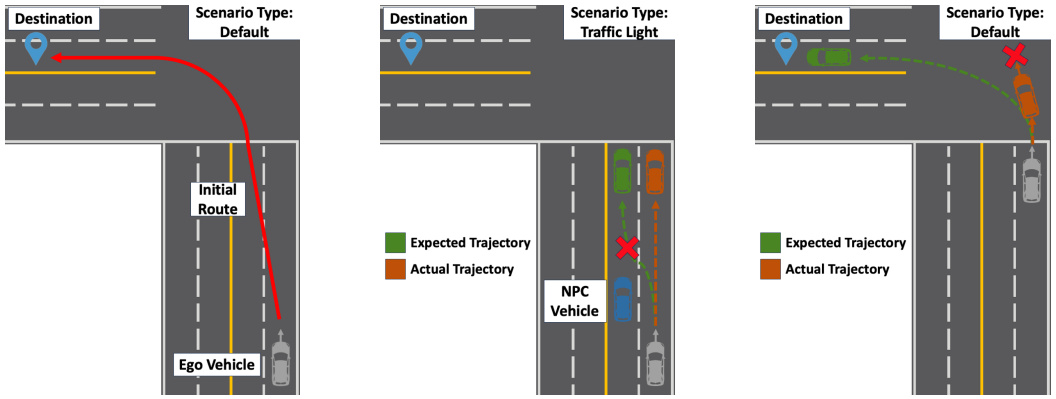


Fig. 1. Progress of the illustrative example.

```
1  if (IsDifferentRouting(last_routing, current_routing)) {
2      last_routing = current_routing;
3      planning_context.Clear(); // Bug: Also clears rerouting_info inside
4      init_planner(); // Force to the default scenario
5  }
6  // ... (omitted for brevity)
7  cooldown_time = 3.0;
8  rerouting_info = planning_context.rerouting();
9  current_time = NowInSeconds();
10 if (rerouting_info.has_last_rerouting_time() &&
11     (current_time - rerouting_info.last_rerouting_time() < cooldown_time)) {
12     return true; // Skip rerouting during cooldown
13 }
14 if (!Rerouting(planning_context)) {
15     return false; // Failed to send request
16 }
17 rerouting_info.set_last_rerouting_time(current_time);
18 return true;
```

Fig. 2. Simplified code showing the bug of the illustrative example.

This failure stems from path generation failure due to improper rerouting repetition. Figure 2 shows the simplified version of the faulty code. Lines in Figure 2 are executed in order at every frame as mentioned in Section 2.1. The planning module first checks whether the incoming route has changed **(line 1)**. If so, at this point, it updates the last routing info, clears the planning context, and re-initializes the planning module. After a series of planning processes, it determines whether rerouting is needed. As a final criterion, it sets a 3-second cooldown time to prevent too frequent rerouting. Specifically, it checks whether less than 3 seconds have elapsed since the last rerouting request. If rerouting has already been executed within 3 seconds, it terminates by returning true immediately **(line 7-13)**. Otherwise, it executes rerouting and updates the last rerouting time with the current time **(line 14-18)**.

In the illustrative example, this logic leads to the following failure. When the ego vehicle reaches the stop line without changing lanes, the planning module determines that the initial route (the red line in Figure 1 (a)) is infeasible and triggers a rerouting request to the routing module. Note that in Apollo, route planning is handled by a separate routing module. As a result, the planning module generates an alternative route that takes a detour to reach the destination. However, when the new route arrives in the next frame, the planning module incorrectly clears the last rerouting timestamp along with the planning context **(line 1-3)**; this is a logical flaw. This causes it to fail to recognize that rerouting has already been requested, repeatedly sending new requests **(line 15)**. Each request triggers init_planner() **(line 4)**, which resets the active scenario to the default,

conflicting with the traffic-light scenario that should be active at the intersection. This scenario oscillation causes the vehicle's pose and heading to become misaligned as it vacillates within the intersection. Consequently, the vehicle attempts the path generation in an unstable state, providing invalid inputs to the path optimization algorithm.

This example reveals two major debugging challenges in ADS planning modules.

First, a temporal gap exists between the observation of system-level failure and the faulty code execution. Consider a developer debugging the failure shown in Figure 1. The developer would naturally examine the point where the system-level failure is observed—the frame where the vehicle stopped. However, this reveals only that path optimization failed due to invalid state inputs, providing no clues to the fault occurred in past frames. Tracing back to discover the rerouting logic flaw requires manually analyzing hundreds of frames, making fault localization extremely time-consuming and error-prone. This temporal gap makes manual fault localization extremely challenging in practice.

Second, planning faults exhibit context dependency, where the same code behaves differently depending on execution history and environment. The same branch may behave benignly in most execution contexts but trigger failure only when executed after a specific sequence of prior decisions has created a problematic state. For example, in the case shown in Figure 1, executing the rerouting logic when rerouting is necessary is not problematic. The fault manifests only when a failure to clear internal data causes the rerouting logic to execute repeatedly across successive frames, accumulating an invalid state.

As a result, traditional SBFL becomes ineffective for planning module debugging due to two core assumptions that are fundamentally violated in this setting. First, SBFL assumes that faulty code consistently correlates with failing executions across independent test runs. However, in ADS planning, the same branch may execute correctly in most frames and even across other test cases, contributing to failure only when executed under a specific accumulated internal state. This means that a faulty branch, i.e., the branch-level code location responsible for the observed failure, carries no consistent statistical signal that SBFL can exploit. Second, SBFL requires a set of passing executions as a reference for comparison. Constructing appropriate passing cases for ADS planning failures is difficult because valid reference executions must preserve the same driving context while removing only the fault-triggering condition. These two violations explain why execution-level coverage aggregation obscures fault-specific behavior in the planning module, and motivate our shift to intra-execution temporal analysis as an alternative foundation for fault localization.

These challenges highlight the need for a fundamentally different approach to fault localization in ADS planning modules. In the following, we formalize the debugging problem and present our approach that addresses these challenges.

3.3 Problem Formulation

Section 3.2 illustrates that planning faults are temporal and context-dependent. This makes execution-aggregated branch coverage insufficient for localization because it obscures the frame at which a branch becomes fault-relevant. We therefore formulate the problem as branch-level localization using frame-level coverage within a single failing case.

We target branch-level localization because rule-based planning modules encode driving decisions through conditional control flow. A module- or component-level diagnosis is too coarse for debugging, as it still leaves developers with many decision points to inspect. In contrast, a branch corresponds to a concrete decision outcome in the planning code, making it a practical unit for explaining how a planning decision contributed to the observed failure.

We now define the key terms used in this formulation. A *failure* is an observable system-level symptom of the planning module, such as permanent immobilization or failure to complete an

emergency maneuver. A *fault* is the code-level cause within the planning module that leads to the observed failure.

In rule-based planning modules, vehicle behavior is determined by conditional decision logic, such as if-statements and switch cases. We define a *branch* as a static control-flow edge corresponding to a possible outcome of such conditional logic, such as the true or false edge of an if-statement or a case edge of a switch statement. Each instrumented branch is assigned a unique identifier, regardless of whether it is taken in a particular frame. A branch is *covered* in a frame if execution reaches the corresponding conditional logic and takes that edge during the planning cycle; otherwise, its coverage value for that frame is zero. Since this work performs branch-level localization, we operationalize a fault as a *faulty branch*. A *faulty branch* is a branch whose covered or uncovered behavior across frames contributes to the observed planning failure.

To formalize the input to this localization problem, we follow the simulation-input formulation used by SafePlanner [25], from which our reproduced failure cases are drawn. This formulation captures the ego vehicle, destination, and surrounding objects that together reproduce a planning failure.

Definition 3.1 (Failure Case). A failure case is defined by the simulation input that reproduces the observed planning failure: $F = (s_{ego}, d, P)$, where s_{ego} is the initial ego vehicle state, d is the destination, and $P = \{(s_1, a_1), (s_2, a_2), \ldots, (s_{N_P}, a_{N_P})\}$ is the set of surrounding objects with their initial state s_i and maneuver a_i. Here, each state represents the object's position, rotation, and velocity. Note that $s_{ego} \notin P$, as the ego vehicle is controlled by the ADS.

Problem Statement. Given a failure case F and the source code of the target planning module M, the goal is to assign suspiciousness scores to the branches of M and induce an inspection order in which the faulty branch appears as early as possible. Developers inspect branches following this order to identify the branch responsible for the observed failure. Accordingly, the objective is to reduce the number of branches that must be inspected before reaching the faulty branch.

4 Approach

Figure 3 shows the overview of our approach, which consists of four stages. First, *Manual Analysis* (Section 4.1) extracts planning semantics from the source code of the target ADS. Planning semantics consist of two components: decision logic, the rules governing how the module decides when to stop or handle emergencies; and diagnostic variables, the runtime signals exposing fault-related state changes. Second, *Runtime Data Generation* (Section 4.2) instruments the planning module based on the extracted semantics, executes each failure case in simulation, and collects per-frame diagnostic traces and branch coverage. Third, *Suspicious Frame Localization* (Section 4.3) takes the diagnostic traces and failure type as input and classifies the observed failure into a failure category. It then identifies the frame where the fault likely originated, bridging the temporal gap between fault execution and failure observation. Finally, *Suspicious Branch Localization* (Section 4.4) analyzes frame-level temporal coverage behavior within a window around the suspicious frame and produces a ranked list of candidate branches for inspection.

4.1 Manual Analysis

In the manual analysis stage, developers instantiate the planning semantics introduced above for the target ADS. This step is performed once for the target ADS and target failure scope, rather than separately for each failure case. We first inspect semantic sources that govern the target failure behavior: stop logic for immobility and stop-related failures, and emergency handling logic for emergency mission failures. These sources identify the code regions and decision mechanisms that should be analyzed for localization. We then select diagnostic variables from these regions and

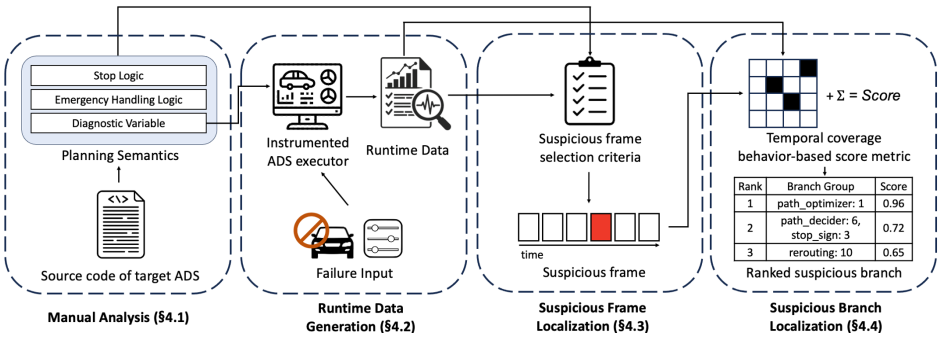


Fig. 3. Overview of our approach.

instrument them as runtime signals, which provide the evidence used by the subsequent localization stages. Once these variables are instrumented, the remaining stages automate trace collection, suspicious-frame identification, and branch ranking for each failure case.

We instantiate this one-time extraction process on Apollo (v7.0) [2], one of the most established open-source ADS platforms and the target ADS of this work. For our target failure types, stop logic and emergency handling logic are the primary semantic sources, directly governing the vehicle stopping and emergency behaviors that underlie these failures. Diagnostic variables are selected to capture the runtime state changes associated with these decision points.

Stop Logic. Stop logic refers to the planning logic that determines when and where the vehicle should halt during normal driving, such as for obstacles, traffic rules, or planning fallback. This logic is directly related to immobility failures, because an unintended or repeated stop decision can cause the vehicle to remain permanently stopped.

Apollo implements three stop policies: obstacle-based, traffic rule-based, and fallback trajectory stops. For obstacle and traffic rule-based stops, Apollo generates stop decisions containing the stop position, object identifier, and type for obstacles or traffic rules (represented as virtual objects). The nearest stop decision is selected to generate the trajectory. Fallback stops occur when path generation fails (Section 3.2), where optimization failure triggers a fallback trajectory that halts the vehicle.

Emergency Handling Logic. Emergency handling logic refers to safety-critical maneuver logic invoked under abnormal or emergency conditions, such as emergency stop or emergency pull-over. In general, emergency mechanisms enable vehicles to safely exit dangerous conditions through controlled stopping maneuvers. Analyzing this mechanism helps identify failures in safety-critical operations.

Apollo implements two emergency scenarios, emergency pull-over and emergency stop, both triggerable manually through the human-machine interface or automatically by the planning module. Emergency pull-over maneuvers the vehicle to the rightmost lane, identifying a suitable parking position that is sufficiently distant from the current location and outside intersections. Emergency stop halts the vehicle in its current lane by calculating an appropriate stop position based on speed and location, then decelerating accordingly.

Diagnostic Variables. After analyzing stop logic and emergency handling logic, we select diagnostic variables as runtime signals that expose the state changes produced by these decision mechanisms. These variables help identify when and where faults occur during execution. They encompass decision-making events, computation outcomes, location information, and control flow transitions that collectively reveal how the planning module behaves frame by frame.

In Apollo's planning module, we extract diagnostic variables for each failure type. For stop-related failures (immobility & emergency stop), we monitor stop decisions to identify when permanent stops originate and path optimization status to detect planning failures that trigger fallback trajectories. For emergency pull-over failures, we track the pull-over position variable to identify when invalid positions are generated or modified. Finally, motivated by Section 3.2, we extract routing requests and scenario transitions to capture control flow changes like scenario oscillation.

Applicability beyond Apollo. Although our implementation targets Apollo, adapting the manual analysis to another rule-based hierarchical planner does not require redesigning the localization procedure. The developer needs to identify the platform-specific counterparts of the same semantic sources and runtime signals used in our Apollo analysis. For example, Autoware Universe [5] implements stop logic through behavior velocity planning modules such as Traffic Light, Stop Line, and Intersection, which correspond to Apollo's traffic-rule and obstacle-based stop policies. It also implements emergency handling logic through Minimal Risk Maneuver behaviors, such as emergency stop and comfortable stop, which correspond to Apollo's emergency stop scenario. Similarly, diagnostic variables can be selected from Autoware's topic-based interfaces and internal planning-module variables to expose when these decision mechanisms affect runtime behavior. Thus, applying our approach to another platform mainly requires mapping the same semantic roles, namely stop logic, emergency handling logic, and diagnostic variables, to that platform's implementation, while the subsequent runtime collection, suspicious-frame localization, and branch ranking stages remain unchanged. The exact diagnostic variables and supported failure types may differ across platforms; for instance, Autoware's pull-over MRM is currently under development, whereas Apollo provides a fully implemented emergency pull-over scenario.

4.2 Runtime Data Generation

The runtime data generation stage reproduces failure cases to collect runtime data. Following the identification of diagnostic variables, the planning module is instrumented to collect their runtime values during execution. Branch coverage can be measured using existing coverage tools if available, or through custom instrumentation that records branch execution status at each frame. The target ADS is rebuilt with the instrumented source code and deployed with a compatible simulator, creating an execution environment that captures diagnostic and coverage information.

Each failure case $F = (s_{ego}, d, P)$ is executed by feeding input $s_{ego}, d,$ and P to the simulator. The simulator generates sensor data and environmental state, which the instrumented planning module processes frame by frame. At each frame, our instrumentation records (1) diagnostic variables' runtime values and (2) executed branch identifiers. Execution continues until failure occurs, producing complete traces from the initial state to the manifestation of failure. The runtime output of F is formally defined as follows:

Definition 4.1 (Runtime Output). Executing a failure case F produces runtime output from the target ADS, denoted as $O = (D, C, \Sigma, f)$, where $D = (d_1, d_2, \ldots, d_n)$ is the diagnostic trace and d_i contains the values of diagnostic variables at frame i, $C = (c_1, c_2, \ldots, c_n)$ is the coverage trace where c_i records the branches executed at frame i, $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_{m'})$ is the sequence of active behavioral states selected during execution, and $f \in \{\text{Immobility}, \text{MissionFailure}\}$ is the failure type. Here, n denotes the total number of frames and m' the number of scenario selections. Each σ_i represents the high-level driving behavior mode that the planning module operates under at that point; in Apollo, these correspond to explicitly defined scenarios such as the stop sign scenario (Section 2.2).

4.3 Suspicious Frame Localization

Suspicious frame localization (SFL) identifies the frame where the fault likely originated by analyzing diagnostic traces through the lens of planning semantics. As discussed in Section 3.2, planning faults often manifest long after the actual faulty execution, creating a temporal gap that makes direct inspection of the failure point uninformative. Rather than pinpointing the exact fault location, SFL estimates the frame at which the fault-related planning decision is most likely to have occurred by reasoning about when the corresponding stop or emergency handling logic would have been invoked based on the observed diagnostic trace.

To do so, SFL first classifies the observed failure into one of four failure categories, $\mathcal{K} = \{\text{ObstacleStop, Rerouting, Optimization, Pull-Over}\}$. Note that these failure categories are distinct from the failure types $f \in \{\text{Immobility, MissionFailure}\}$ introduced in Section 3.1: while failure types describe the system-level observable outcome, failure categories provide a finer-grained characterization of the underlying planning behavior that led to the failure, enabling category-specific backtracking criteria. Formally, given Σ, f, and D, SFL outputs a suspicious frame t_{sus} and a failure category κ, denoted as $(t_{sus}, \kappa) = \text{SFL}(\Sigma, f, D)$, where $\kappa \in \mathcal{K}$.

Because the temporal gap between faulty execution and observable failure varies by failure category, SFL applies category-specific criteria to identify t_{sus}. Each criterion is derived from the stop logic and emergency handling logic extracted in Section 4.1, reasoning about when the fault-related planning decision is most likely to have first affected the diagnostic trace:

- **ObstacleStop.** If the diagnostic trace indicates that the final stop is caused by an obstacle stop decision, SFL backtracks to the frame where the corresponding stop decision was first generated, eliminating the delay introduced by prolonged inactivity detection.
- **Rerouting.** If the diagnostic trace shows repeated rerouting requests occurring within a short interval, SFL selects the frame of the second request, which violates Apollo's expected rerouting interval.
- **Optimization.** If the diagnostic trace indicates path optimization failure, SFL selects the first failing frame, capturing the vehicle state that triggered the optimization failure.
- **Pull-Over.** If the diagnostic trace indicates an abnormal pull-over behavior, such as an unstable or incorrectly determined pull-over position, SFL selects the frame where the final pull-over position is computed, capturing the point at which the wrong decision was made.

The identified failure category and suspicious frame are passed to the next stage to guide branch-level fault localization.

4.4 Suspicious Branch Localization

Suspicious branch localization (SBL) is the final stage of the approach that identifies the specific faulty code near the suspicious frame. As discussed in Section 3.2, faulty branches in ADS often execute correctly in some contexts, making them indistinguishable from benign code when using traditional coverage-based methods that aggregate data at the case level. SBL addresses this by analyzing frame-level temporal coverage behavior within a localized window around the suspicious frame, detecting temporal execution anomalies that correspond to fault activation.

In SBL, we focus on branch coverage instead of line coverage because planning modules employ complex rule-based structures where individual branches represent distinct decision logic, and line-level granularity creates excessive ties in suspiciousness rankings.

Given C, κ, and t_{sus}, SBL scores candidate branches and outputs ranked branch groups for inspection, denoted as $B_{rank} = \text{SBL}(C, \kappa, t_{sus})$. The grouping is used only to consolidate indistinguishable branches; the localization target remains an individual faulty branch. Algorithm 1 shows pseudocode for SBL. It takes as input the coverage trace C, failure category κ, and suspicious frame t_{sus}. It defines

Algorithm 1 Suspicious Branch Localization

Require: Coverage trace $C = (c_1, c_2, \ldots, c_n)$;
failure category κ; suspicious frame t_{sus}
Ensure: Ranked branch groups B_{rank}
1: $W \leftarrow [t_{sus} - N,\ t_{sus} + N]$ ▷ *Define analysis window*
2: $\mathcal{B} \leftarrow \{\, b \mid \exists i \in W,\ b \in c_i \,\}$ ▷ *Collect branches in window*
3: **for** $b \in \mathcal{B}$ **do**
4: $H_b \leftarrow (h_b(i))_{i \in W}$, where $h_b(i) = \mathbf{1}[b \in c_i]$ ▷ *Compute coverage patterns*
5: **end for**
6: $\mathcal{G} \leftarrow \textsc{GroupByPattern}(\{(b, H_b) \mid b \in \mathcal{B}\})$ ▷ *Group by pattern*
7: **for** $G_k \in \mathcal{G}$ **do**
8: Select any representative branch $b \in G_k$ ▷ *Compute group scores*
9: $\rho_k \leftarrow Score_\kappa(b)$
10: **end for**
11: $B_{rank} \leftarrow \textsc{Sort}_{\prec}(\{(G_k, \rho_k)\})$ ▷ *Rank branch groups*
12: **return** B_{rank}

an analysis window of radius N around t_{sus} and collects all branches executed within it **(line 1-2)**. For each branch, it computes a binary coverage pattern recording per-frame execution status **(line 3-5)**, then groups branches that share identical coverage patterns to consolidate the ranked output **(line 6)**. Each group is scored using the failure-category-specific metric $Score_\kappa$ **(line 7-10)**, detailed in Section 4.4.1. Groups are ranked by descending score, with ties broken by group size **(line 11-12)**.

Table 1. Mapping from failure types to temporal coverage behaviors and scoring metrics.

Failure Type	Failure Category	Coverage Behavior	Intuition	Score Metric
Immobility	ObstacleStop	Step	Stop decision persists across frames	$\text{Score}_{\text{step}}$
	Optimization	Spike	Optimization failure appears at a single frame	$\text{Score}_{\text{spike}}$
	Rerouting	Flap	Faulty rerouting causes oscillatory behavior	$\text{Score}_{\text{flap}}$
Mission Failure	Pull-Over	Spike	Invalid pull-over position appears at a single frame	$\text{Score}_{\text{spike}}$

4.4.1 Temporal Coverage Behavior-Based Score Metric. The design of our scoring metric is grounded in the mapping between failure categories and their expected temporal coverage behaviors, which is derived from the planning semantics extracted in Section 4.1. Each failure category exhibits a characteristic coverage behavior near the suspicious frame, reflecting how the corresponding planning decision manifests in the execution trace. Table 1 summarizes this mapping, which guides the selection of the appropriate scoring metric for each failure category.

Our scoring metric analyzes temporal coverage behaviors within a localized window around the suspicious frame rather than aggregating execution frequencies across test cases. Here, the coverage behavior of a branch refers to its per-frame execution pattern within the analysis window, captured as a binary sequence indicating whether the branch was executed at each frame. We define failure-category-specific expected behaviors based on planning semantics, then score branches by how well they match these behaviors near t_{sus}.

Let $\mathcal{B}$ denote the set of all branches observed in the coverage trace C within the analysis window $W = [t_{sus} - N, t_{sus} + N]$ where, $N \in \mathbb{N}$ denotes the radius of the analysis window. For each branch $b \in \mathcal{B}$, we use a binary indicator $h_b(i)$ where $h_b(i) = 1$ if branch b executes at frame i and $h_b(i) = 0$ otherwise. To weight frames near the suspicious frame more heavily, we apply a distance-weighted factor $w(i) = \frac{1}{1+|i-t_{sus}|}$, which decreases with distance from t_{sus}.

For each failure category κ, we define a corresponding sub-score metric designed to detect branches exhibiting the expected coverage behavior. SBL selects the appropriate sub-score metric based on κ from SFL and scores each branch $b \in \mathcal{B}$ accordingly. The following subsections describe each expected coverage behavior and its corresponding metric.

Spike Behavior. A spike is characterized by a single-frame deviation from the dominant execution status near t_{sus}. Optimization and Pull-Over categories correspond to this temporal coverage behavior according to the planning semantics. In the case of Optimization, an occurrence of a path optimization failure triggers the fallback trajectory, which occurs at a single frame. For Pull-Over, the calculation of an initial abnormal pull-over position or the unexpected change of pull-over position is executed in a single frame. Because faulty branches in these categories are expected to exhibit such spike behavior, we design a score metric to capture these spikes as follows:

$$Score_{spike}(b) = \max\left(\delta_b^{spike}(i) \cdot w(i)\right), \qquad i \in [t_{sus} - N + 1, t_{sus} + N - 1].$$

where $\delta_b^{spike}(i)$ is an indicator for the spike of a branch b at frame i defined as

$$\delta_b^{spike}(i) = \mathbf{1}\big(h_b(i-1) = h_b(i+1) \neq h_b(i)\big), \qquad i \in [t_{sus} - N + 1, t_{sus} + N - 1].$$

Step Behavior. The second behavior is a step. This behavior captures temporal coverage behaviors where execution status changes once and remains stable thereafter. This behavior corresponds to the ObstacleStop failure category. When the ego vehicle stops permanently due to a stop decision, the branch including that stop decision also executes permanently. A faulty branch that causes the execution of this branch is expected to show the same coverage behavior, and we design a sub-score metric to capture this. The sub-score metric is formalized as follows:

$$Score_{step}(b) = \max\left(\delta_b^{step}(i) \cdot w(i)\right). \qquad i \in [t_{sus} - N + 1, t_{sus} + N].$$

where $\delta_b^{step}(i)$ is an indicator for the step behavior of a branch b at frame i defined as

$$\delta_b^{step}(i) = \mathbf{1}\Big(h_b(i-1) \neq h_b(i) \ \wedge \ \forall j \in W,\ j > i,\ h_b(j) = h_b(i)\Big), \qquad i \in [t_{sus} - N + 1, t_{sus} + N].$$

Flap Behavior. The characteristic of the last behavior is different from the others. While the other behaviors above were derived from the planning semantics, this behavior is derived from the observation of the scenario oscillation of the motivating example in Section 3.2. A flapping behavior refers to a coverage pattern where a branch frequently alternates between executed and not-executed states across frames within the analysis window, reflecting the oscillatory nature of the faulty decision logic. We expect that the faulty branch causing the scenario oscillation must have a flapping coverage behavior near t_{sus}, so that we design a sub-score metric to capture this flapping.

$$Score_{flap}(b) = \frac{1}{|W| - 1} \sum \mathbf{1}\big(h_b(i) \neq h_b(i-1)\big). \qquad i \in W \setminus \{t_{sus} - N\}.$$

To sum up, we design three sub-score metrics in our implementation, and SBL selects the corresponding sub-score metric for the failure category κ to calculate the score of a branch b as follows:

$$Score_{\kappa}(b) = \begin{cases} Score_{spike}(b) & \text{if } \kappa \in \{\text{Optimization, Pull-Over}\}, \\ Score_{step}(b) & \text{else if } \kappa = \text{ObstacleStop}, \\ Score_{flap}(b) & \text{otherwise } \kappa = \text{Rerouting}. \end{cases}$$

4.4.2 Branch Grouping and Tie Breaking. Although the localization target is an individual faulty branch, multiple branches can exhibit identical frame-level coverage patterns within the analysis window. Because such branches are indistinguishable by coverage alone, our implementation collapses them into branch groups for inspection. This grouping is used only to avoid redundant inspection and does not change the localization target: the ground-truth target remains an individual faulty branch.

Formally, for each branch $b \in \mathcal{B}$, we define its coverage pattern vector over the analysis window W as

$$\mathbf{v}(b) = \big(h_b(i)\big)_{i \in W} \in \{0,1\}^{|W|}.$$

Two branches b and b' are said to be equivalent if they share the same coverage pattern, i.e., $b \sim b' \iff \mathbf{v}(b) = \mathbf{v}(b')$. This equivalence relation partitions the set of all branches $\mathcal{B}$ into a set of coverage pattern groups: $\mathcal{G} = \{G_1, G_2, \ldots, G_M\}$ where each group is defined as

$$G_k = \{\, b \in \mathcal{B} \mid \mathbf{v}(b) = \mathbf{v}_k \,\},$$

and $\mathbf{v}_k$ denotes a unique coverage pattern observed in W.

Since all branches in the same group share identical coverage behavior, they receive the same suspiciousness score. Accordingly, we define the score of a group G_k as

$$Score_\kappa(G_k) = Score_\kappa(b), \qquad \forall b \in G_k,$$

where $Score_\kappa(b)$ is computed using the sub-score metric selected for the failure category κ. In practice, the score of each group is computed using an arbitrary representative branch from the group.

To reduce debugging effort without altering the underlying suspiciousness metric, we use the group size only as a secondary criterion when multiple groups receive the same score. Let $|G|$ denote the number of branches in group G and let $\varepsilon \geq 0$ be a tie tolerance. We rank groups using the following lexicographic order:

$$G_a \prec G_b \iff \Big(Score_\kappa(G_a) > Score_\kappa(G_b)\Big) \vee \Big(\big|Score_\kappa(G_a) - Score_\kappa(G_b)\big| \leq \varepsilon \wedge |G_a| < |G_b|\Big).$$

Finally, after computing the suspiciousness score for each branch group, SBL produces a total ordering over the groups to guide fault inspection. Since multiple groups may receive identical scores due to identical or highly similar coverage behaviors, SBL applies a deterministic ranking rule to resolve such ties. The rule prioritizes groups with higher suspiciousness scores, and when scores are indistinguishable, it favors smaller groups to reduce the number of branches that developers must inspect.

$$B_{rank} = \mathrm{sort}_{\prec}\big(\{(G_k, \rho_k) \mid G_k \in \mathcal{G},\ \rho_k = Score_\kappa(G_k)\}\big).$$

5 Evaluation

In the evaluation section, we answer the following research questions (RQs):

- **RQ1:** How accurately does SFL identify the suspicious frame compared to an oracle-based baseline?
- **RQ2:** How much does SBL reduce branch inspection effort while localizing faulty branches within the identified suspicious frame?
- **RQ3:** How does the approach facilitate the localization of complex faults that are challenging for traditional spectrum-based fault localization methods?

RQ1 evaluates the accuracy of SFL in identifying the suspicious frame for known failures, comparing it against an oracle-based baseline that identifies the suspicious frame from the observable failure oracle without leveraging planning semantics. Since the performance of SBL highly depends on the accurate detection of the suspicious frame, this evaluation establishes the foundation for assessing the overall effectiveness of our approach. RQ2 assesses the overall fault localization capability of

our approach at the branch level. The result of RQ2 demonstrates whether our semantic-guided approach can effectively pinpoint faulty branches and provide actionable insights for developers. Finally, RQ3 presents a detailed case study that demonstrates how our approach localizes faults in practice. This case study illustrates the concrete debugging process enabled by our approach and explains why it can detect faults that traditional spectrum-based methods would miss.

5.1 Experiment Setup

Target ADS. Our experiment targeted Baidu Apollo 7.0 [2], a major open-source level-4 ADS platform with a modular architecture. Apollo provides a well-structured rule-based planning module that handles various driving scenarios including emergency scenarios. We selected Apollo 7.0 for compatibility with the SafePlanner failure dataset [25].

Instrumentation. We manually analyzed the planning module source code of Apollo 7.0 and extracted planning semantics from it. We then instrumented the source code to collect the runtime values of the diagnostic variables. For diagnostic variables, we defined a Protocol Buffer schema (.proto file) [37] to serialize all extracted diagnostic variables, along with a collector class for runtime recording. Then, we inserted macro functions at their assignment points to record values at each frame. For branch coverage, we instrumented all source files implementing scenarios, stages (high-level behavior sequences), tasks (atomic planning operations), and traffic rules. These files are selected because they contain the implementation of stop logic and emergency handling logic.

Simulator. We used the LGSVL simulator, which is known as the most compatible simulator with Apollo [30]. Unfortunately, the official development team had discontinued the simulator, and the cloud server that provided the necessary resources to run the simulator had been shut down. Nevertheless, the open-source SORA-SVL project has maintained the cloud infrastructure and can serve as a local cloud alternative [55]. We constructed our execution environment with Apollo 7.0, LGSVL simulator, and the SORA-SVL local cloud.

Dataset. We evaluated our approach on planning failures from SafePlanner [25], which provided 520 initial failure scenarios discovered through end-to-end testing. We reproduced these failures using a 3D ground truth sensor to replace the perception module, successfully reproducing 221 cases that manifest as immobility or emergency mission failures in the planning module. For each reproducible failure, we manually debugged the case and localized the faulty branch to establish ground truth for evaluating fault localization accuracy. We considered a branch faulty if it was causally related to the observed faulty behavior, with causality confirmed either through manual code analysis or by verifying that its modification eliminated the failure. To further validate our causal analysis, we reported faulty branches identified during this process to the Apollo repository as GitHub issues, excluding cases for which we found an existing issue already reporting the same bug. We grouped failures sharing the same faulty branch, and Table 2 summarizes the resulting 221 reproduced failures, with the Faulty Branch column listing the filename and branch identifier for each fault. Detailed analyses (failure videos, triggering conditions, fault causes, and potential fixes) are available on our companion website (Section 8).

Table 2. Reproducible failure cases from prior work.

Failure No.	Description	Type	Faulty Branch	# of Cases
1	The ego vehicle fails to overtake an NPC and stops near an intersection	I	path_lane_borrow_decider: 26	46
2	Path optimization fails due to a map or NPCs	I	piecewise_jerk_path_optimizer: 10	34
3	Path optimization fails near intersection after lane change failure	I	rerouting: 10	25
4	The ego vehicle permanently stops at a stop sign after rerouting	I	scenario_manager: 134	69
5	The ego vehicle repeatedly tries to pull over, although it already pulled over successfully	M	path_bounds_decider: 40	25
6	The ego vehicle pulls over inside an intersection	M	path_bounds_decider: 51	22

I: Immobility, M: Mission Failure

Parameter Setting. Our implementation has one main parameter, the analysis window radius N. We set $N = 20$ based on our preliminary evaluation to balance context coverage and ranking precision. This value spans approximately 3–4 seconds of simulated time, capturing the typical temporal scope of fault manifestation in planning modules. Section 5.3 provides a detailed sensitivity analysis.

5.2 RQ1: Accuracy of SFL in Suspicious Frame Identification

While no existing technique directly targets frame-level localization for non-collision ADS failures, we construct a fair oracle-based baseline from the closest available idea in ADS debugging. MoDitector is not designed for frame-level fault localization, but it uses module-specific oracles to focus diagnosis on a window near the system-failure timestamp [48]. Inspired by this failure-window idea, our *oracle-based baseline* uses only observable failure timing, without planning semantics.

To make the baseline consistent with the evaluated failures, we instantiate it using SafePlanner's failure oracles. For immobility failures, the baseline selects the first frame of the 20-second window in which the ego vehicle remains below 0.3 m/s [25]. For emergency mission failures, where no sustained immobility window exists, the baseline uses the final frame of the execution log as the suspicious frame. We evaluate SFL against this baseline to test whether planning semantics identify fault-relevant frames more accurately than failure-oracle timing alone.

Metric. We compare SFL and the oracle-based baseline using two metrics:

- **Distance** Δ: The absolute difference between the detected suspicious frame and the ground truth fault-triggering frame is $\Delta = |t_{sus} - t_{gt}|$, where t_{gt} is the position of the actual faulty frame.
- **In-k accuracy**: The percentage of cases where $\Delta \leq k = 20$. This indicates whether the fault-triggering frame falls within the subsequent SBL analysis window, allowing SBL to analyze the relevant temporal context around the fault.

Table 3. Accuracy of Suspicious Frame Localization.

		Ours		Oracle-Based Baseline	
Failure No.	**# of Cases**	**Avg. Diff.**	**In-20 (%)**	**Avg. Diff.**	**In-20 (%)**
1	46	0.98	100.0	144.37	4.3
2	34	0.00	100.0	92.24	11.8
3	25	2.08	100.0	171.72	0.0
4	69	13.80	100.0	78.94	11.6
5	25	0.00	100.0	12.56	88.0
6	22	1.00	100.0	171.18	0.0
Overall	221	4.85	100.0	106.77	16.3

Result. Table 3 shows the results. Our approach achieves an average distance of $\Delta = 4.85$ frames across all 221 cases, which, given a per-frame processing time of 150–200 ms in our experiments, translates to under 1 second of execution time, showing that SFL localizes suspicious frames close to the actual fault. All cases achieve In-20 = 100%, placing the fault-triggering frame within the subsequent SBL analysis window and allowing SBL to analyze the relevant temporal context around the fault. In contrast, the oracle-based baseline achieves $\Delta = 106.77$ with only 16.3% In-20, showing that backtracking from observed stopping behavior alone is insufficient to accurately identify the suspicious frame. This confirms that planning semantics are critical for bridging the temporal distance between fault execution and failure observation, a gap the oracle-based baseline cannot recover.

Failure 4 exhibits larger Δ values due to the temporal gap between rerouting detection and its impact on scenario selection. SFL identifies the suspicious frame at the second abnormal rerouting

request (t_{sus}). However, the faulty branch resides in the scenario selection logic that determines whether to activate the stop sign scenario. This selection logic is only invoked when the vehicle needs to choose a scenario, several frames after the rerouting occurs. Thus, there is a delay between when rerouting is triggered (t_{sus}) and when the faulty branch in the scenario selection logic is actually executed and affected by the rerouting (t_{gt}). Despite this temporal distance, all Failure 4 cases satisfy In-20, ensuring the faulty branch remains within the analysis window.

The answer to RQ1 is that SFL achieves 100% In-20 accuracy with an average distance of 4.85 frames, successfully localizing all suspicious frames within the subsequent branch analysis window, substantially outperforming the oracle-based baseline (16.3% In-20 and Δ=106.77).

5.3 RQ2: Effectiveness of the Approach

This RQ evaluates how accurately SBL identifies the faulty branch within the localized suspicious frame. As with RQ1, no existing technique serves as a suitable baseline, but for different reasons. Traditional SBFL methods aggregate coverage across multiple test cases to rank suspicious code, while our approach analyzes frame-level temporal coverage behavior within a single execution [1, 23, 24, 51, 52]. Additionally, prior ADS debugging work targets different failure types (collision) or operates at coarser granularities (module or component level) [14, 21, 42, 45]. Instead, we employ a random ordering heuristic as our baseline to provide a controlled comparison for evaluating SBL's effectiveness.

Metric. We evaluate the approach using the following metrics:

- **Hit@K (Top-K)**: The percentage of cases where the faulty branch is ranked at position K or higher (i.e., $r \leq K$ where r is the rank). We report Hit@1, Hit@3, Hit@5, and Hit@10 to evaluate ranking effectiveness at different thresholds. This metric has recently been used in fault localization studies [27, 32, 38, 50, 53].
- **Inspection effort $E_{B_{rank}}$**: The total number of branches a developer must inspect from the top-ranked group through the faulty group, defined as $E_{B_{rank}} = \sum_{i=1}^{r} |G_i|$ where G_r is the group containing the faulty branch. While we measure this at the branch level, similar metrics (e.g., mean average rank and mean first rank) are widely used in fault localization literature [27, 32, 38, 50].

Using these metrics, we compare our approach against a random baseline to assess its effectiveness.

Baseline. We use a random ordering heuristic as the baseline of our evaluation. We apply SFL within a window of size $N = 20$, group branches by identical coverage patterns, and randomly select candidates for inspection. We report the average results across 50 independent trials. This baseline is fair since it operates within the same search space as SBL, differing only in the selection strategy, thus isolating the effectiveness of SBL's ranking over random selection.

Table 4. Top-k analysis on Suspicious Branch Localization (Ours vs. Baseline).

Failure No.	# Cases	Capture Rate (%)		Hit@10 (%)		Hit@5 (%)		Hit@3 (%)		Hit@1 (%)		$E_{B_{rank}}$	
		Ours	Rand.	Ours	Rand.	Ours	Rand.	Ours	Rand.	Ours	Rand.	Ours	Rand.
1	46	100.0	100.0	97.8	97.2	78.3	68.1	45.7	40.0	0.0	13.6	35.24	100.17
2	34	100.0	100.0	100.0	80.2	91.2	40.0	82.4	23.0	23.5	6.8	8.76	83.52
3	25	100.0	100.0	100.0	59.7	100.0	28.9	100.0	17.8	92.0	6.3	2.84	101.13
4	69	100.0	100.0	100.0	66.2	59.4	34.1	11.6	20.6	0.0	6.8	14.96	108.69
5	25	100.0	100.0	100.0	95.2	100.0	70.4	100.0	42.2	100.0	12.8	1.0	77.09
6	22	0.0	0.0	N/A		N/A		N/A		N/A		N/A	
Overall	221	90.0	90.0	89.6	70.8	71.5	42.2	48.4	25.1	25.3	8.2	37.13	111.14

Result. The evaluation result for RQ2 is presented in Table 4. The approach successfully localizes the faulty branch for 90% of cases (199 out of 221), with an average inspection effort of $E_{B_{rank}}$ = 37.13 branches, substantially outperforming the random baseline ($E_{B_{rank}}$ = 111.14, a 3× reduction). This substantially reduces debugging effort compared to analyzing the entire planning module. This demonstrates that SBL's temporal coverage analysis significantly improves fault localization efficiency compared to random selection within the same search space. Excluding Failure 6 cases (where localization failed due to the faulty branch never being executed), the average further drops to 15.32 branches.

The Hit@K analysis demonstrates the ranking effectiveness at different thresholds, with consistent improvements over the random baseline. Hit@1 achieves 25.3% (vs. 8.2% for random), meaning a quarter of faults are ranked as the top candidate. Hit@3 and Hit@5 reach 48.4% and 71.5% respectively (vs. 25.1% and 42.2% for random), indicating that developers can locate most faults by inspecting only the top few branch groups. Hit@10 achieves 89.6% (vs. 70.8% for random), closely matching the overall success rate. Note that Hit@1 and Hit@3 occasionally favor the random baseline in a few cases (3 instances), which is expected: our method sometimes assigns higher scores to non-faulty branch groups with prominent temporal coverage behavior, while random selection may occasionally hit the correct answer by chance. However, the $E_{B_{rank}}$ metric reveals the true picture, where our method consistently outperforms the baseline across all failure types.

The results vary across failure cases. Failures 1-5 achieve 100% localization success, while Failure 6 shows 0% success rate. Among the successful failures, inspection effort ranges from $E_{B_{rank}}$ = 1.0 (Failure 5) to 35.24 (Failure 1), reflecting differences in code complexity and temporal coverage behavior distributions of non-faulty branches.

Notably, Failure 1 and Failure 4 show poor performance compared to other failures. In case of Failure 1, the faulty frames of many cases are located at the very first frame, so dozens of branches related to the configuration and module initialization are included inside W. For Failure 4, it has a frame gap between the t_{sus} and t_{gt} as explained in Section 5.2, causing several non-faulty branches to exhibit more prominent temporal coverage behavior than the faulty branch within the analysis window. These issues lead to degradation of the Top-K analysis and $E_{B_{rank}}$.

Failure 6 exposes a limitation of our approach. The root cause lies in the logic determining the pull-over position. When Apollo searches for a suitable location, it first checks for nearby intersections to avoid invalid placement. However, it uses an excessively small search radius (1 meter). Consequently, it fails to detect the nearby intersection and generates a pull-over position inside it. The faulty branch, which would be selected when an intersection is detected, is never executed throughout the analysis window because no intersection is detected. This results in a monotonic zero coverage pattern lacking any distinctive characteristics (spikes, steps, or flaps) that our temporal coverage behavior-based metrics target (Section 4.4.1). This limitation is inherent to execution-based localization methods and does not arise from our scoring design or window selection. This failure case motivates future research on distinguishing faulty non-execution from benign non-execution.

Table 5. Evaluation of Suspicious Branch Localization for various window sizes.

N	Capture Rate (%)	Hit@10 (%)	Hit@5 (%)	Hit@3 (%)	Hit@1 (%)	$E_{B_{rank}}$
10	66.1	64.7	59.7	52.5	25.3	75.30
15	87.3	84.6	65.2	48.4	25.3	41.74
20	90.0	89.6	71.5	48.4	25.3	37.13
25	90.0	88.2	73.3	48.0	14.9	38.98
30	90.0	88.2	73.8	36.7	13.1	39.31

Sensitivity Analysis. We conducted a sensitivity analysis to evaluate the impact of the analysis window radius, N. The same metrics are calculated for 5 different Ns ($N = 10, 15, 20, 25$, and 30). Table 5 presents the results. The results reveal a clear trade-off between context coverage and ranking precision.

Smaller windows ($N \leq 15$) provide insufficient context, missing branches whose execution behaviors span multiple frames. This significantly reduces the capture rate—the percentage of cases where the faulty branch is executed at least once within the analysis window—to 66.1% at $N = 10$. Larger windows ($N \geq 25$) maintain a high capture rate (90%) but degrade ranking precision, as evidenced by declining Hit@1 (14.9% at $N = 25$ vs 25.3% at $N = 20$) and Hit@3 metrics.

$N = 20$ achieves the best balance, with 90% capture rate, 37.13 average inspection effort, and strong ranking performance across all Hit@K thresholds. This window size effectively captures the temporal scope of fault manifestation while maintaining ranking precision. Based on these results, we use $N = 20$ as it achieves the best balance between localization success and ranking precision.

> The answer to RQ2 is that the approach successfully localizes 90% of faulty branches with an average inspection effort of 37.13 branches, substantially outperforming random selection ($E_{B_{rank}} = 111.14$; a 3× reduction). This confirms that analyzing temporal coverage behavior significantly improves fault localization accuracy.

```
idx_after = 0;
// path_bound is a list of vectors representing the lateral boundaries for each s.
// Search for the first element farther than pull_over_position
while (idx_after < path_bound.size() && std::get<0>(path_bound[idx_after]) < pull_over_position.s())
    ++idx_after;
idx_before = idx_after - 1; // Bug: What if idx_after is 0?
// Is the pull_over_position within the drivable area of adjacent indices?
if (std::get<1>(path_bound[idx_before]) <= pull_over_position.l() &&
    std::get<2>(path_bound[idx_before]) >= pull_over_position.l() &&
    std::get<1>(path_bound[idx_after]) <= pull_over_position.l() &&
    std::get<2>(path_bound[idx_after]) >= pull_over_position.l()) // Suspicious branch
    return idx_after;
return -1;
```

Fig. 4. Simplified code showing the bug of the case study.

5.4 RQ3: Qualitative Analysis of Fault Localization

Through a representative case study, we demonstrate how our approach facilitates debugging and explain why classic SBFL methods are fundamentally incompatible with ADS failure analysis. We examine Failure 5, an emergency mission failure where the ego vehicle incorrectly repeats the pull-over procedure despite having already completed it successfully.

We began the debugging by inspecting the branches in the first-ranked group in B_{rank}. For this case, B_{rank} = {(st_driving_limits:1, 0.263), (path_bounds_decider:40, 0.198), . . .}. We checked the first suspicious branch, but this branch appears unrelated to the failure, as it only selects which kinematic equation to use for computing vehicle dynamic bounds. After several debugging attempts, we shifted focus to the second group. The second branch belongs to a function that checks whether the pull-over position lies within the current drivable area. Figure 4 shows the simplified code of this function.

The temporal coverage reveals a critical clue with a spike behavior where the branch remains uncovered for a single frame, causing the function to return −1. This return value signals that the pull-over position is outside the drivable area, triggering Apollo to search for a new position and causing the repetitive behavior.

The fault originates from the boundary search logic **(line 4-6)**. When the ego vehicle slightly overshoots the pull-over position, even the first boundary element lies beyond it, assigning idx_before $= -1$. This negative index causes an invalid array access, leading the branch condition to evaluate incorrectly and return -1 despite the ego vehicle having nearly stopped at the correct position.

This case illustrates the challenges of applying traditional SBFL to ADS debugging. The fault is revealed through a distinctive temporal coverage behavior (a spike in this case) within a single execution, whereas traditional SBFL relies on aggregating branch execution frequency across multiple independent test runs [52]. Such aggregation obscures critical information. For example, the faulty branch at **(line 8)** executes normally in most frames but fails at the faulty frame, diluting its suspiciousness when aggregated. Aggregation can further obscure faults by mixing branches from temporally unrelated execution points. While carefully designed reference cases might enable cross-case comparison, creating appropriate passing cases for diverse ADS failures remains difficult. Our approach addresses this by analyzing a narrow window around the suspicious frame and detecting specific temporal behavior to distinguish fault-related behaviors from coincidental execution behaviors.

The answer to RQ3 is that our approach successfully facilitates fault localization through temporal coverage behavior analysis, and demonstrates why traditional SBFL cannot effectively handle ADS planning failures.

5.5 Threats to Validity

The primary threat to the generality of our study is the approach. We evaluated our approach only on Apollo 7.0 + LGSVL simulator [2, 30], while there are several combinations of ADS and simulators. However, as we mentioned in Section 5.1, the majority of planning failures are discovered in Apollo. Furthermore, emergency mission failures have been reported only in Apollo across existing ADS testing literature [25, 28, 31, 47, 48]. As described in Section 4.1, our planning semantics are extracted based on logic commonly present in rule-based planning architectures. This suggests that the same approach can be applied to other rule-based planning platforms, though evaluation on additional ADS platforms and simulators would further strengthen the generality of our findings.

Another threat to external validity is the absence of direct baseline comparisons with other ADS debugging approaches. While several prior works address ADS debugging, they differ in target modules, failure types, and fault granularity, making direct comparison challenging [3, 4, 8, 14, 15, 19, 21, 36, 40, 42, 45]. Adapting these approaches for consistent comparison would strengthen our evaluation. Similarly, traditional SBFL techniques [1, 23, 24, 51, 52] cannot serve as baselines. Our approach analyzes temporal coverage behavior within a single execution, whereas SBFL requires multiple passing test cases for comparison. Applying SBFL at the frame level would require labeling each frame as passing or failing, which is impractical for ADS failures that lack a clear manifestation at every frame.

The primary internal validity threat concerns the accuracy of manually identified faulty branches. Although we verified our ground truth by causally analyzing and successfully repairing failures through modifying the identified branches, we cannot guarantee these repairs are optimal or do not affect other planning behaviors. Future work should establish systematic validation through comprehensive test suites or formal verification methods to ensure root cause correctness.

Another limitation is that failures from PlanFuzz and LawBreaker could not be reproduced due to unavailable experimental environments and missing assets [44, 46], while some SafePlanner failures could not be reproduced under ground truth perception [25, 26]. Successfully reproducing these additional failure cases would allow us to validate our approach on a broader range of failure scenarios, further strengthening its generality.

6 Related Work

Failure of Planning Module. Prior ADS testing research has discovered various planning module failures [25, 28, 31, 44, 47]. Li et al. [31] introduced AV-Fuzzer, targeting collision failures in Apollo via genetic algorithm-based fuzzing. Wan et al. [47] introduced the concept of immobility failures and discovered multiple such failures, along with mission failures, in Apollo and Autoware through guided fuzzing. Wang et al. [48] proposed MoDitector, which uses a module-specific oracle to generate failure scenarios and pinpoint the faulty module simultaneously. Kim et al. [28] and Sun et al. [44] developed DriveFuzz and LawBreaker for Autoware and Apollo respectively, discovering collision, immobility, and traffic violations in each platform. More recently, Kim et al. [25] proposed SafePlanner, which extracts scenario transition state machines and identifies incompleteness in emergency scenarios, expanding the understanding of mission failures. While this line of work has made significant strides in discovering planning failures, the subsequent debugging process remains largely manual. Our work complements these testing efforts by providing a systematic approach to fault localization in planning modules.

Debugging of ADS Failures. Despite this progress in failure discovery, few works perform in-depth debugging or root cause analysis of the failures found. Huang et al. [21] proposed ADSDx, which localizes the faulty module via causal analysis by replacing each module's output with ground truth, but cannot identify code-level faults. Wan et al. [45] applied similar causal analysis at the finer-grained component level, though it requires multiple replays and still relies on classic localization for code-level attribution. Sun et al. [42] reduced the search space of frames using features from Apollo's internal messages and Spatio-Temporal safety specifications, but this is specialized for collision failures and cannot address immobility or mission failures. Feng et al. [14] localized faulty parameters for collision failure by monitoring module outputs and solving an optimization problem over the component's parameter space; while potentially adaptable to immobility and mission failures, it targets parameters rather than code-level logic errors. Other efforts [3, 4, 8, 15, 19, 36, 40] have explored various debugging and fault categorization techniques, but differ from our work in scope and methodology. Beyond ADS-specific debugging, prior CPS work has studied debugging and fault localization for Simulink/Stateflow models and control software, using temporal-logic specifications, trace diagnostics, slicing, statistical debugging, and specification-guided analysis [6, 7, 33, 41, 54], or system-identification oracles, parameter localization, and feedback-based testing [11, 18, 34]. These studies highlight the importance of temporal traces and physical behavior, but they generally target models, signals, parameters, or subsystems and often require formal specifications or multiple executions. In contrast, our work focuses on ADS planning code and localizes branch-level faults from a single failing execution.

7 Conclusion

This paper proposes a temporal coverage analysis approach for localizing branch-level faults in rule-based planning modules. We extract planning semantics from the source code of the planning module to identify suspicious frames, then analyze frame-level temporal coverage behavior within a localized window to rank candidate branches, addressing the temporal gap and closed-loop nature of planning decisions that make traditional fault localization ineffective. On 221 reproducible planning failures in Apollo, our approach achieves 100% suspicious frame accuracy and successfully localizes faulty branches for 90% of cases with an average inspection effort of 37.13 branches, a 3× reduction compared to the random baseline. This demonstrates that temporal coverage analysis provides an effective foundation for debugging planning failures from a single failing execution.

8 Data Availability

The source code and experimental data are available at https://github.com/RomainLettuce/TCADS. Videos and detailed cause analyses for the six failure groups in Table 2 are available on our companion website: https://sites.google.com/view/adsfaultlocalization.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(RS-2024-00347407). Donghwan Shin was supported by the Engineering and Physical Sciences Research Council (EPSRC) [EP/Y014219/1].

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46. doi:10.1109/PRDC.2006.18

[2] ApolloAuto. 2025. Apollo. https://github.com/ApolloAuto/apollo. GitHub repository.

[3] Paolo Arcaini, Alessandro Calò, Fuyuki Ishikawa, Thomas Laurent, Xiao-Yi Zhang, Shaukat Ali, Florian Hauer, and Anthony Ventresque. 2021. Parameter-based testing and debugging of autonomous driving systems. In *2021 IEEE Intelligent Vehicles Symposium Workshops (IV Workshops)*. IEEE, 197–202. doi:10.1109/IVWorkshops54471.2021.9669254

[4] Paolo Arcaini, Xiao-Yi Zhang, and Fuyuki Ishikawa. 2022. Less is more: Simplification of test scenarios for autonomous driving system testing. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 279–290. doi:10.1109/ICST53961.2022.00037

[5] Autowarefoundation. 2025. autoware. https://github.com/autowarefoundation/autoware. GitHub repository.

[6] Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Ničković. 2018. Localizing faults in Simulink/Stateflow models with STL. In *Proceedings of the 21st international conference on hybrid systems: computation and control (part of cps week)*. 197–206. doi:10.1145/3178126.3178131

[7] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Ničković. 2021. CPSDebug: Automatic failure explanation in CPS models. *International Journal on Software Tools for Technology Transfer* 23, 5 (2021), 783–796. doi:10.1007/s10009-020-00599-4

[8] Johannes Betz, Alexander Heilmeier, Alexander Wischnewski, Tim Stahl, and Markus Lienkamp. 2019. Autonomous driving—a crash explained in detail. *Applied Sciences* 9, 23 (2019), 5126. doi:10.3390/app9235126

[9] CBS News. 2025. Tesla robotaxis face scrutiny after erratic driving caught on camera during Austin pilot. https://www.cbsnews.com/news/tesla-robotaxis-austin-texas-highway-traffic-safety/. Accessed: 2026-01-29.

[10] Hatem Darweesh, Eijiro Takeuchi, Kazuya Takeda, Yoshiki Ninomiya, Adi Sujiwo, Luis Yoichi Morales, Naoki Akai, Tetsuo Tomizawa, and Shinpei Kato. 2017. Open source integrated planner for autonomous navigation in highly dynamic environments. *Journal of Robotics and Mechatronics* 29, 4 (2017), 668–684. doi:10.20965/jrm.2017.p0668

[11] Jyotirmoy Deshmukh, Xiaoqing Jin, Rupak Majumdar, and Vinayak Prabhu. 2018. Parameter optimization in control software using statistical fault localization techniques. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 220–231. doi:10.1109/ICCPS.2018.00029

[12] Erdos-project. 2023. pylot. https://github.com/erdos-project/pylot. GitHub repository.

[13] Haoyang Fan, Fan Zhu, Changchun Liu, Liangliang Zhang, Li Zhuang, Dong Li, Weicheng Zhu, Jiangtao Hu, Hongye Li, and Qi Kong. 2018. Baidu Apollo EM Motion Planner. arXiv:1807.08048 [cs.RO] doi:10.48550/arXiv.1807.08048

[14] 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. doi:10.1145/3691620.3695530

[15] 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. doi:10.1145/3377811.3380397

[16] General Motors. 2022. GM Brings Powerful New Compute Architecture to Ultra Cruise. https://news.gm.com/newsroom.detail.html/Pages/news/us/en/2022/jan/0106-ultracruise.html. Accessed: 2026-01-26.

[17] Ionel Gog, Sukrit Kalra, Peter Schafhalter, Matthew A Wright, Joseph E Gonzalez, and Ion Stoica. 2021. Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 8806–8813. doi:10.1109/ICRA48506.2021.9561747

[18] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 116–127. doi:10.1109/ICSE.2019.00029

[19] Wenkui Hou, Wanyu Li, and Pengyu Li. 2023. Fault diagnosis of the autonomous driving perception system based on information fusion. *Sensors* 23, 11 (2023), 5110. doi:10.3390/s23115110

[20] Jun Hu, Yuefeng Wang, Shuai Cheng, Jinghan Xu, Ningjia Wang, Bingjie Fu, Zuotao Ning, Jingyao Li, Hualin Chen, Chaolu Feng, et al. 2025. A survey of decision-making and planning methods for self-driving vehicles. *Frontiers in Neurorobotics* 19 (2025), 1451923. doi:10.3389/fnbot.2025.1451923

[21] Zongan Huang, Jiarun Dai, Mingyuan Luo, Yuqi Chen, Huijia Sun, Yuan Zhang, and Min Yang. 2025. ADSDx: Towards Automated Accident Diagnosis for High-level Autonomous Driving Systems. *ACM Transactions on Software Engineering and Methodology* (2025). doi:10.1145/3783993

[22] Interesting Engineering. 2024. Amazon's robotaxis make risky intersection stops, prompting 332-vehicle software recall. https://interestingengineering.com/transportation/amazon-robotaxis-software-recall. Accessed: 2026-01-29.

[23] Tom Janssen, Rui Abreu, and Arjan JC Van Gemund. 2009. Zoltar: A toolset for automatic fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 662–664. doi:10.1109/ASE.2009.27

[24] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282. doi:10.1145/1101908.1101949

[25] Dohyun Kim, Sanggu Han, Sangmin Woo, Joonha Jang, Jaehoon Kim, Changhun Song, and Yongdae Kim. 2026. SafePlanner: Testing Safety of the Automated Driving System Plan Model. arXiv:2601.09171 [cs.SE] doi:10.48550/arXiv.2601.09171

[26] Dohyun Kim, Sanggu Han, Sangmin Woo, Joonha Jang, Jaehoon Kim, Changhun Song, and Yongdae Kim. 2026. SafePlanner Website. Accessed: 2026-01-15. https://sites.google.com/view/safe-planner/home

[27] Donguk Kim, Minseok Jeon, Doha Hwang, and Hakjoo Oh. 2025. PAFL: Enhancing Fault Localizers by Leveraging Project-Specific Fault Patterns. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 1378–1405. doi:10.1145/3720526

[28] 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. doi:10.1145/3548606.3560558

[29] LGSVL. 2021. lgsvl PerceptionSensor3D. https://github.com/lgsvl/PerceptionSensor3D. GitHub repository.

[30] LGSVL. 2021. simulator. https://github.com/lgsvl/simulator. GitHub repository.

[31] 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 *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*. IEEE, 25–36. doi:10.1109/ISSRE5003.2020.00012

[32] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673. doi:10.1109/ICSE43902.2021.00067

[33] Bing Liu, Lucia, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability* 26, 6 (2016), 431–459. doi:10.1002/stvr.1605

[34] Claudio Mandrioli. 2022. *Control-Theoretical Perspective in Feedback-Based Systems Testing*. Doctoral Thesis (compilation).

[35] Sean O'Kane. 2023. Cruise recalls entire fleet after robotaxi ran over, dragged pedestrian. https://techcrunch.com/2023/11/08/cruise-recalls-entire-fleet-after-robotaxi-ran-over-dragged-pedestrian/. Accessed: 2026-01-29.

[36] Stefan Orf, Sven Ochs, Jens Doll, Albert Schotschneider, Marc Heinrich, Marc René Zofka, and J Marius Zöllner. 2024. Modular fault diagnosis framework for complex autonomous driving systems. In *2024 IEEE 20th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 1–8. doi:10.1109/ICCP63557.2024.10793003

[37] Protocolbuffers. 2026. protobuf. https://github.com/protocolbuffers/protobuf. GitHub repository.

[38] Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation. *Proc. ACM Softw. Eng.* 1, FSE, Article 86 (July 2024), 23 pages. doi:10.1145/3660793

[39] Reuters. 2025. Baidu Robotaxi Falls Into Construction Pit in China, Raising Safety Concerns. https://www.usnews.com/news/world/articles/2025-08-08/baidu-robotaxi-falls-into-construction-pit-in-china-raising-safety-concerns. Accessed: 2026-01-24.

[40] Andrew Roberts, Mohammad Reza Heidari Iman, Mauro Bellone, Tara Ghasempouri, Jaan Raik, Olaf Maennel, Mohammad Hamad, and Sebastian Steinhorst. 2024. ADAssure: Debugging methodology for autonomous driving control algorithms. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6. doi:10.23919/DATE58400.2024.10546519

[41] Nikhil Kumar Singh and Indranil Saha. 2020. Specification-guided automated debugging of CPS models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4142–4153. doi:10.1109/TCAD.2020.3012862

[42] 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. doi:10.1145/3597503.3639175

[43] 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. doi:10.1145/3551349.3556897

[44] Yang Sun, Christopher M Poskitt, Jun Sun, Yuqi Chen, and Zijiang Yang. 2022. LawBreaker Website. Accessed: 2026-01-15. https://lawbreaker2022.github.io/

[45] 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:2401.10443 [cs.SE] doi:10.48550/arXiv.2401.10443

[46] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred Chen. 2022. PlanFuzz Website. Accessed: 2026-01-15. https://sites.google.com/view/av-ioat-sec/planfuzz

[47] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred Chen. 2022. Too Afraid to Drive: Systematic Discovery of Semantic DoS Vulnerability in Autonomous Driving Planning under Physical-World Attacks. arXiv:2201.04610 [cs.CR] doi:10.48550/arXiv.2201.04610

[48] Renzhi Wang, Mingfei Cheng, Xiaofei Xie, Yuan Zhou, and Lei Ma. 2025. MoDitector: Module-Directed Testing for Autonomous Driving Systems. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 137–158. doi:10.1145/3728876

[49] Waymo. 2021. The Waymo Driver Handbook: Teaching an autonomous vehicle how to perceive and understand the world around it. https://waymo.com/blog/2021/10/the-waymo-driver-handbook-perception. Accessed: 2026-01-26.

[50] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus A Haryono, Yuan Tian, Hafil Noer Zachiary, and David Lo. 2022. XAI4FL: Enhancing spectrum-based fault localization with explainable artificial intelligence. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 499–510. doi:10.1145/3524610.3527902

[51] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308. doi:10.1109/TR.2013.2285319

[52] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. doi:10.1109/TSE.2016.2521368

[53] Yiqian Wu, Yujie Liu, Yi Yin, Muhan Zeng, Zhentao Ye, Xin Zhang, Yingfei Xiong, and Lu Zhang. 2025. SmartFL: Semantics Based Probabilistic Fault Localization. *IEEE Transactions on Software Engineering* 51, 7 (2025). doi:10.1109/TSE.2025.3574487

[54] Tomoya Yamaguchi, Bardh Hoxha, Danil Prokhorov, and Jyotirmoy V Deshmukh. 2020. Specification-guided software fault localization for autonomous mobile systems. In *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 1–12. doi:10.1109/MEMOCODE51338.2020.9315067

[55] YuqiHuai. 2025. YuqiHuai SORA-SVL. https://github.com/YuqiHuai/SORA-SVL. GitHub repository.

Received 2026-01-30; accepted 2026-06-25