A Comprehensive Study of Bug-Fix Patterns in Autonomous Driving Systems

YUNTIANYI CHEN, University of California, Irvine, USA
YUQI HUAI, University of California, Irvine, USA
YIRUI HE, University of California, Irvine, USA
SHILONG LI, University of California, Irvine, USA
CHANGNAM HONG, University of California, Irvine, USA
QI ALFRED CHEN, University of California, Irvine, USA
JOSHUA GARCIA, University of California, Irvine, USA

As autonomous driving systems (ADSes) become increasingly complex and integral to daily life, the importance of understanding the nature and mitigation of software bugs in these systems has grown correspondingly. Addressing the challenges of software maintenance in autonomous driving systems (e.g., handling real-time system decisions and ensuring safety-critical reliability) is crucial due to the unique combination of real-time decision-making requirements and the high stakes of operational failures in ADSes. The potential of automated tools in this domain is promising, yet there remains a gap in our comprehension of the challenges faced and the strategies employed during manual debugging and repair of such systems. In this paper, we present an empirical study that investigates bug-fix patterns in ADSes, with the aim of improving reliability and safety. We have analyzed the commit histories and bug reports of two major autonomous driving projects, Apollo and Autoware, from 1,331 bug fixes with the study of bug symptoms, root causes, and bug-fix patterns. Our study reveals several dominant bug-fix patterns, including those related to path planning, data flow, and configuration management. Additionally, we find that the frequency distribution of bug-fix patterns varies significantly depending on their nature and types and that certain categories of bugs are recurrent and more challenging to exterminate. Based on our findings, we propose a hierarchy of ADS bugs and two taxonomies of 15 syntactic bug-fix patterns and 27 semantic bug-fix patterns that offer guidance for bug identification and resolution. We also contribute a benchmark of 1,331 ADS bug-fix instances.

CCS Concepts: • **Software and its engineering** → *Software libraries and repositories*; *Software defect analysis.*

Additional Key Words and Phrases: Bug-fix pattern, Autonomous driving systems, Empirical study

ACM Reference Format:
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, Article FSE018 (July 2025), 23 pages. https://doi.org/10.1145/3715733

1 Introduction

The rise of Autonomous Vehicles (AVs) signifies a crucial transformation in the realm of transportation. With more than 50 companies such as Ford, Toyota, Tesla, and Waymo [1, 2, 10, 13–16, 26] actively developing AVs, these vehicles are swiftly becoming a significant part of our daily lives.

Authors' Contact Information: Yuntianyi Chen, University of California, Irvine, Irvine, USA, yuntianc@uci.edu; Yuqi Huai, University of California, Irvine, Irvine, USA, yhuai@uci.edu; Yirui He, University of California, Irvine, Irvine, USA, yiruih@uci.edu; Shilong Li, University of California, Irvine, Irvine, USA, shilonl2@uci.edu; Changnam Hong, University of California, Irvine, Irvine, USA, changnah@uci.edu; Qi Alfred Chen, University of California, Irvine, Irvine, USA, alfchen@uci.edu; Joshua Garcia, University of California, Irvine, Irvine, USA, joshug4@uci.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTFSE018
https://doi.org/10.1145/3715733

This surge in development and deployment underscores the importance of understanding and maintaining the software that powers these complex cyber-physical systems. In the intricate ecosystem of AV software development, a critical yet often overlooked aspect is the pattern of bug fixes. While the initial focus of AV development has been on achieving functional and Operational Design Domain (ODD) [32] goal which specifies the operating conditions under which an ADS can operate safely, it is equally important to understand and repair the software bugs that inevitably arise. We define a bug-fix pattern (BFP) as a recurring repair solution applied to similar bug types, capturing the modifications needed to resolve the bug. The significance of bug-fix patterns lies in their ability to reveal underlying system vulnerabilities and inform more effective automated repair strategies. This understanding is not only critical for enhancing software reliability and performance but also crucial for ensuring the safety of AV operations. As AVs find their way onto public roads, the stakes of software malfunctions escalate, necessitating a thorough and detailed study of bug repair.

One of the critical gaps in previous research [30, 43, 49, 53, 64] on bug-fix patterns is the limited focus on syntactic bug-fix patterns, primarily at the code or statement level, often overlooking the semantic aspects (e.g., affected algorithms or domain-specific components) of the projects under study. Although some existing empirical studies have investigated bug-fix patterns or strategies in a variety of domains including the deep learning stack [38], federated learning systems [33], and deep learning libraries [42], none of them have focused on bug-fix patterns in the ADS domain.

Our research centers on two widely-used [11, 12], production-grade (i.e., used by some commercial companies) ADSes—Baidu Apollo [19], which have reached mass production agreements with Ford and Volvo [12], and Autoware [22], which was selected by the US Department of Transportation for intelligent transportation solutions [23]. These systems represent the forefront of ADS technology and offer a large-scale dataset of bug history for understanding bug-fix patterns in real-world, high-stakes, or virtual-simulation environments. The exploration of BFPs in these systems is crucial, as it provides insights into the challenges and strategies employed in maintaining software that directly impacts the safety and reliability of AVs. The importance of this study is further highlighted by the safety-critical nature of autonomous driving. As AVs continue to integrate into the fabric of daily transportation, the implications of software faults range from minor malfunctions to catastrophic failures, highlighting the necessity of meticulous software maintenance and bug fixing. While automated tools for bug detection and repair show promise, a comprehensive understanding of BFPs, encompassing both syntactic and semantic aspects, is crucial to advance the ADS domain.

In addressing this need, our study not only investigates the common syntactic BFPs in Apollo and Autoware but also delves into the semantic information underlying these patterns. This approach allows for a more nuanced understanding of the bugs, their causes, and the most effective strategies for their repair. We further introduce the concept of modularization granularity as a structured hierarchy of ADS bugs at various levels of abstraction. This hierarchy is essential for the systematic analysis and debugging of ADS and serves as a foundational structure for developing robust software systems. From both syntactic and semantic levels, our study formulates a more effective response to the complexities of ADS development and bug repair, potentially leading to resilient and fault-tolerant system designs. Our proposed hierarchical classification of ADS bug-fix pattern study includes 15 syntactic and 27 semantic bug-fix patterns, as well as root causes, symptoms, modules, sub-modules, related algorithms, and detailed bug-fix actions. This taxonomy is instrumental for developers, testers, and researchers in developing automated bug detection and repair tools for ADS. Bug-fix patterns are the building blocks for repairing programs. Our taxonomy and statistics provide an in-depth understanding of common BFPs in ADSes, which ensures comprehensive coverage and paves the way for targeted improvements and innovations in debugging or repair techniques tailored for ADS. Specifically, our taxonomy can inform the design of new testing techniques, such as mutation testing, and the benchmark we propose can serve as a valuable resource for failure

prediction research. The methodology of this study involves a detailed analysis of the collected data to identify common BFPs and understand their underlying causes and symptoms. Our research advances software engineering for ADS in the following key aspects:

- We present the first empirical study of bug-fix patterns in the ADS domain through an analysis of 1,331 fixed bugs from 15,099 pull requests in two open-source ADSes, offering a foundational understanding of the challenges and strategies in this domain.
- We conduct the first study that introduces and differentiates the concepts of syntactic and semantic bug-fix patterns, which serve as a practical guide for ADS software maintenance and identify opportunities for improved bug identification and repair for future work.
- We propose a hierarchy of ADS bug-fix pattern study classified by modularization granularity. We also contribute a benchmark of 1,331 ADS bug-fix instances, which is publicly accessible [4].

2 Background

2.1 Autonomous Driving Systems

An ADS aims to achieve high automation levels for vehicles to automatically run on roads. The Society of Automotive Engineers defines six levels of autonomous driving, from Level 0, with no assistance systems, to Level 5, which represents fully autonomous driving [51]. Baidu Apollo [21] and Autoware [22] achieve Level 4 [9], which refers to a high degree of automation where the vehicle can handle all aspects of driving in certain environments without human intervention, but a human override is still an option. The core modules and components are listed in Table 1.

Table 1. Modules in Autonomous Driving Systems

Module	Description
Planning	Makes decisions for the AV to execute, such as cruising or stopping.
Perception	Processes data from the surrounding environment detected by sensors.
Prediction	Receives obstacle information and predicts its future motion.
Control	Enforces the planned trajectory with lateral and longitudinal control.
Localization	Provides location, heading, and velocity information of the AV.
Simulator	Tests the ADS in a virtual environment, replicating real-world scenarios.
Sensing	Detects obstacles and traffic to understand the environment.
CAN Bus	Handles communication between software and vehicle.
HMI	Collects and visualizes the status and interfaces of the system.
HD Map	Includes lane geometries and locations of traffic control devices.
System	Coordinates the integration and operation of all AV modules.
Infrastructure	Includes robotics middleware to support communication among modules.
Utilities&Tools	Provides necessary utilities, such as sensor calibration tools.
Docker	Includes the Docker image housing an instance of the ADS.
Build	Compiles and integrates the AV software, enabling smooth deployment.
Drivers	Contains the hardware drivers necessary for operating the AV.
Documentation	Provides guides, specifications, and manuals detailing the ADS.

2.2 Bug Study of Autonomous Driving Systems

Understanding the nature of defects in ADSes is essential for enhancing their safety and performance. A comprehensive analysis of prior work [37] provides a foundational understanding by categorizing ADS bugs from the root causes that reflect errors in code and symptoms manifested as incorrect behaviors or errors during runtime as presented in Table 2 and Table 3.

Table 2. Root Causes of Bugs in the ADS

Type	Description
Alg	Flawed logic that requires a comprehensive correction.
Num	Incorrect numerical calculations, values, or usage.
Assi	Wrong variable assignments or initializations.
MCC	Absence of necessary conditional statements.
Data	Incorrect data structure definitions or pointer misuse.
Exter-API	Incorrect usage of interfaces of other systems or libraries.
Inter-API	Misuse of interfaces of other components within the ADS.
ICL	Incorrect condition logic or faulty conditional expressions.
Conc	Mismanagement of concurrency mechanisms like threads.
Mem	Misuse of memory or improper memory management.
Doc	Incorrect manuals, tutorials, or code comments.
Config	Incorrect system setup or build settings in configurations.
OT	Infrequent and uncategorized issues.

Table 3. Symptoms of Bugs in the ADS

Type	Description
Crashes	Improper termination of an ADS.
Hangs	The system becomes unresponsive but remains running.
Build	Errors hinder the compilation or installation of ADS modules.
DGUI	Erroneous output on a GUI, visualization, or HMI.
Cam	Errors prevent image capture by the camera.
Stop	Incorrect behaviors during stopping or parking.
LPN	Incorrect lane positioning, maintenance, or navigation.
SVC	Incorrect AV speed and velocity management.
TLP	Incorrect behaviors involving handling of traffic lights.
Lau	ADS or component fails to start or launch.
Turn	AV behaves incorrectly when making a turn.
Traj	Incorrect trajectory prediction results.
IO	Incorrect input/output operations to files or devices.
LOC	Inaccurate multi-sensor fusion-based localization.
SS	Behaviors that affect the safety or security of its passengers.
OP	Incorrect processing or handling of obstacles on the road.
Logic	Incorrect behaviors that do not terminate the program.
Doc	Errors in the documentation or comments.
UN	Symptoms that are unreported and cannot be identified.
OT	Infrequent symptoms that do not fit into above categories.

2.3 Bug-Fix Patterns

The concept of bug-fix patterns (BFPs) emerges as a critical tool in the software engineering domain, offering a structured approach to diagnosing and rectifying common defects in software systems [30, 49]. These patterns, distilled from the collective experience of developers, encapsulate proven solutions to recurring problems, thereby streamlining the debugging and maintenance processes. In the context of ADS, where software reliability and safety are essential, the identification and application of BFPs become even more vital. Autonomous driving systems, with their integration of complex algorithms, sensor data processing, and real-time decision-making, present a unique set of challenges for software maintenance. The diversity of bugs in ADS—ranging from algorithmic inaccuracies and sensor data misinterpretations to issues in system integration and performance—necessitates a specialized set of BFPs tailored to these specific problems. For instance, a bug-fix pattern aimed at addressing inaccuracies in environmental perception might involve strategies for sensor data fusion enhancement or algorithmic refinement for object detection.

3 Methodology

3.1 Dataset

In our study of bug-fix patterns, we collected a dataset that encompasses 19,379 commits, 9,708 merged pull requests, and 3,532 issues for Apollo [21], as well as 22,825 commits, 5,391 merged pull requests, and 559 issues for Autoware. All these commits, issues, and pull requests were established on or prior to November 13, 2023, as detailed in Table 4.

Table 4. Statistics of ADS Dataset

System	Start Date–End Date	Commits	Issues	Pull Requests	Bug Fixes
Apollo	07/04/2017–11/13/2023	19,379	3,532	9,708	338
Autoware	08/25/2015–11/13/2023	22,825	559	5,391	993

Considering the objective to identify bug-fix patterns in ADSes, we collected closed and merged single-file pull requests that fix bugs. Such pull requests enable us to (1) scrutinize the modified source code, associated issues, and developer discussions; and (2) avoid too many bugs and bug-fix patterns that occurred in one pull request to interfere with the evaluation and analysis. Note that GitHub's pull requests serve diverse functions, such as introducing new features, bug fixes, or refactoring. We implemented a technique to maximize the capture of bug-fix pull requests. In doing so, we employed an approach similar to those of previous research [37, 41, 58, 63] by curating a list of bug-associated keywords like "fix", "repair", "defect", "error", "bug", "issue", "mistake", "incorrect", "fault", and "flaw", subsequently

searching for these terms in both titles and content. Pull requests with any of these keywords were classified as initial bug-fix requests. We further filtered out 62 pull requests that were not real bug fixes even with keywords and repeated pull requests derived from different branches during labeling by analyzing their description and intentions. This methodology yielded 338 and 993 integrated pull requests for Apollo and Autoware, respectively, that fit our criteria.

3.2 Hierarchy of ADS Bug-Fix Pattern Study by Modularization Granularity

In this study, we aim to analyze the nature of ADS bug-fix patterns by examining them through seven distinct perspectives: (i) the root causes indicative of developer errors in the codebase; (ii) the symptoms of these bugs, as evidenced by aberrant behaviors, system failures, or runtime errors; (iii) the specific ADS module and sub-module where the bug is localized; (iv) the algorithm or function that the bug is related to; (v) the syntactic bug-fix pattern; (vi) the semantic bug-fix pattern; and (vii) the bug-fix action related to the specific bug-fix pattern, root cause, sub-module, or algorithm.

We introduce the concept of *Modularization Granularity* as a means to dissect and understand the architecture of ADS, offering a clear depiction of the structural layers of software development into a hierarchy of granularity that spans from broad architectural modules to the minute intricacies of code statements. Modularization granularity is essential to decompose complex systems into manageable segments. We identify this granularity through various levels of labels, each serving its unique role in the modular architecture. Figure 1 shows the hierarchy along with two examples presenting how we label the bug fixes in the Planning module.

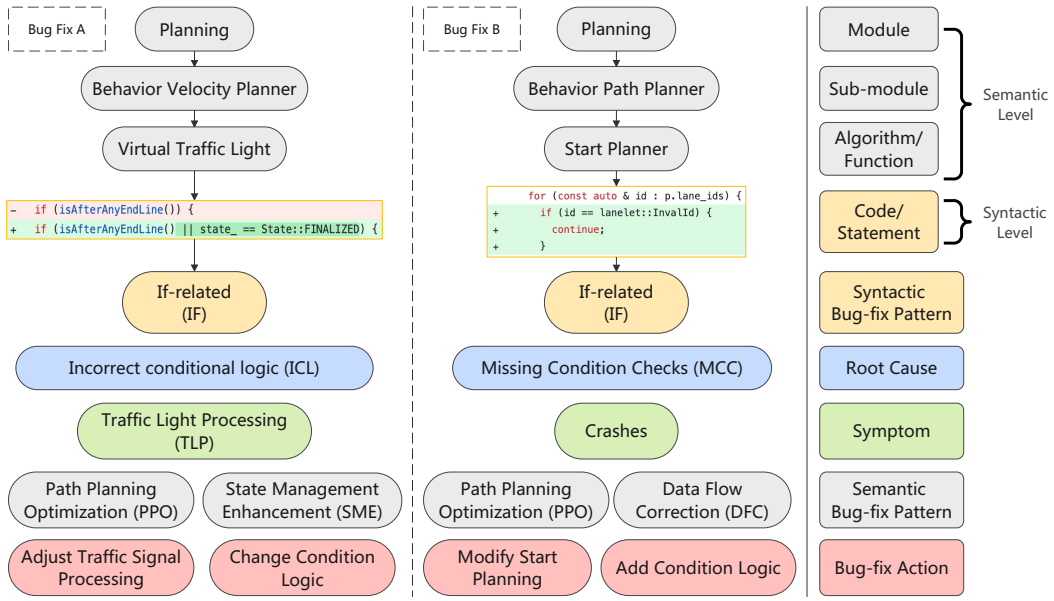

Fig. 1. Hierarchy of ADS Bug-Fix Pattern by Modularization Granularity

On the semantic level, we have *Module* and *Sub-module* labels which indicate the major components and their subdivisions respectively, facilitating a high-level understanding and management of the system. We also have *Algorithm/Function* labels that reveal the intricate details of the system's functionality and operational logic. Finally, at the most granular level, we have the *Statement/-Code* labels, which are indicative of the syntactic construction of the software. *Syntactic BFPs* are categorized according to code and statement changes, while *Semantic BFPs* are analyzed from a

high-level comprehensive understanding of ADS misbehaviors. *Bug-fix actions* are determined through an in-depth examination of other labels, which describe the actionable strategies made to bugs, providing detailed steps and modifications necessary to correct errors, adjust code behavior, and improve system functionality. For each pull request, we also label the *Symptom* to identify the observable effects of bugs and the *Root cause* that delves into their underlying reasons to help understand the relations between bug-fix patterns and bug types. This hierarchical framework allows us to trace symptoms back to their root causes and categorize them into our taxonomy of bug-fix patterns. This structured analysis informs our findings by providing a nuanced view of how different types of bugs arise, propagate through, and influence the system.

While distinguishing the severity of individual bug-fix patterns may be challenging, the design of our modularization granularity hierarchy allows us to use the symptoms to assess severity. For instance, the bugs that have Crashes symptoms are clearly more severe than those with DGUI or no symptoms. We could use symptoms and corresponding violations to assess the severity of bugs.

The hierarchy has practical applications for both researchers and practitioners. For researchers, it provides a robust framework for studying bug-fix patterns in other domains. Its top three semantic levels and semantic bug-fix patterns can be customized to suit specific application domains, making it broadly applicable. For example, adapting the hierarchy more broadly to cyber-physical systems or robotics could provide valuable insights into their unique bug-fix challenges. This flexibility underscores the hierarchy's potential to standardize bug-fix analysis across diverse systems, aiding researchers in labeling and analyzing bug-fix data systematically. For practitioners, the hierarchy serves as a diagnostic tool to guide debugging and repair processes. By mapping the symptom and root cause of a bug to specific semantic or syntactic bug-fix patterns, the hierarchy helps practitioners identify actionable bug-fix actions. Furthermore, it connects these patterns to affected modules, sub-modules, algorithms, and code statements, providing targeted guidance for debugging and repair. The hierarchy's adaptability also ensures its applicability to other modular architectures.

3.3 Classification and Labeling

To mitigate potential subjective biases during the labeling phase, we allocated each of the 1,331 merged bug-fix pull requests to two co-authors, who are also contributors and developers of both Apollo and Autoware open-source projects and possess enough background in the ADS domain. We employed the open-coding strategy of intercoder reliability [48] to help strengthen the labeling process. Our methodology necessitated that each co-author independently assess the bug, which involved meticulous examination of the source code, commit logs, code reviews, pull request information, and associated issue descriptions to recognize the labeling items.

In our work, we commenced with the root cause and symptom taxonomies in a previous ADS bug study [37] and also taxonomy of generic bug-fix patterns focusing on the syntactic level [30, 43, 49, 53] as a foundation for ADS bug analysis. The taxonomy of root causes was subsequently augmented by employing an open-coding paradigm, thereby broadening the spectrum. For the pull request that eluded classification within the foundational taxonomy, each co-author designated a label for it. Post-labeling, co-authors collaboratively reconciled any discrepancies in their classifications. We added two root causes, **Syntax, Naming, and Typography (SNT)** that involves errors in the basic structure of the code, including syntactical mistakes, naming conventions, and typographical errors, and **Dependency Issues (DEP)** related to importing, versioning, and managing dependencies.

For semantic bug-fix patterns and bug-fix actions, none of the previous work could provide a useful taxonomy due to the domain-specific nature of ADS. In this research, the open coding process was employed to refine and identify distinct semantic bug-fix patterns and bug-fix actions. A preliminary investigation was conducted involving two co-authors who independently examined

the bug fixes to establish a tentative classification framework. Each rater recommended a series of categories, which were later amalgamated and refined during a face-to-face session attended by all contributing authors. This meeting served as a platform to validate and integrate the classification schemes proposed by the individual raters. This reconciliation process led to updates in the classification scheme. We validated the final classification by consulting with researchers in the ADS domain and developers from Apollo and Autoware. Their expertise helped refine the classification scheme, and we relabeled the affected pull requests accordingly. The classification result is presented in Section 4. Our findings indicated that a single bug origin may result in multiple symptoms and bug-fix patterns. Therefore, certain bugs were cataloged into multiple categories, unlike the previous study [37] only considered a single symptom for each bug. In the process of labeling, two co-authors, possessing expertise in ADS, were engaged to categorize the bug-fix patterns according to the specified schema. The degree of concordance between these raters was quantified using Cohen's Kappa coefficient [59], which was used by a recent bug-fix pattern study [42]. In instances of labeling discrepancies, periodic discussions were conducted to achieve reconciliation. Throughout this process, the Kappa score persistently exceeded 80%, indicative of a robust understanding and unanimous agreement among the raters [46].

4 Taxonomy

4.1 Bug-Fix Patterns at the Syntactic Level

We adopted the classification from previous work [43, 49] as it comprehensively covered the syntactic BFPs observed in Apollo and Autoware. We also added new patterns like Syntax (SYN), *Document Fix (DOC)*, and *Library (LIB)* to describe bug repairs that are missing in previous studies but relevant for ADSes. We enumerate the syntactic bug-fix patterns in the following paragraphs.
If-related (IF). Modifications in the conditional logic of code through addition, modification, or removal of *if* statements to ensure that operations are executed under the correct conditions.
Assignment (AS). Modifications in the assignment statements, including changes in the expressions on the right-hand side of an assignment, to ensure correct value assignment and operation.
Method Call (MC). The MC pattern involves adjustments in the way methods are invoked, which include changing the number and type of parameters or the values passed to it.
Method Declaration (MD). Changes in Method Declarations involve alterations in the method's interface, such as the number of parameters, parameter types, or return type.
Sequence (SQ). Changes in the order of a series of operations or method calls.
Loop (LOOP). Modifications in the conditions or the structure of loop statements.
Return (RT). This pattern encompasses modifications in the return statements of functions, ensuring that functions return the correct values or types as expected.
Local Variable (LV). Changes to the declaration or initialization of local variables within functions.
Non-source-code Variable (NV). NV adjustments pertain to changes in parameters that are not part of the source code but are essential for the configuration of the software.
Library (LIB). Fixes involve the addition, removal, or modification of libraries used in the project.
Syntax (SYN). This pattern addresses issues like incorrect indentation in Python, syntax misuse, or typographical errors that can affect the code execution.
Try-Catch (TRY). This pattern involves the addition or removal of try-catch blocks, which are vital for error handling and exception management without crashing.
Switch (SW). Add or remove switch branches or change the conditions within the switch block.
Class Field (CF). CF fixes pertain to modifications in the class fields or attributes that are present in any object of a class, and whose lifetime is the same as the object lifetime.
Document Fix (DOC). Corrections or updates in documentation or comments within the code.

4.2 Bug-Fix Patterns at the Semantic Level

The semantic level of bug-fix patterns in ADS delves into the deeper, meaning-oriented aspects of software bugs, focusing on how these bugs affect the behavior and functionality of the system as a whole. Unlike syntactic bug-fix patterns, which are concerned with the structure and form of the code, semantic bug-fix patterns address the logic, algorithms, and operational semantics of the ADS software, sometimes including multiple syntactic patterns within a single semantic pattern. To comprehensively understand these patterns, we categorize them into *domain-specific* and *domain-independent* bug-fix patterns. Domain-specific patterns address issues unique to autonomous driving technology, while domain-independent patterns apply to a broader range of software systems. We enumerate our identified ADS semantic bug-fix patterns in the remainder of this section.

4.2.1 Domain-Specific Semantic Bug-Fix Patterns. Domain-specific patterns are uniquely tailored to address the problems inherent to ADSes. These patterns often highlight the intricate interactions between sub-modules and external factors like road conditions, traffic regulations, and sensor data.
Path Planning Optimization (PPO). Refine the algorithms related to path and velocity planning in ADS, including enhancements to the lane-change strategies, obstacle avoidance, or navigation in complex environments. The code snippet provided illustrates a specific bug fix of PPO:

```
+ overwriteStopPoint(clipped,traj_smoothed);
  traj_smoothed.insert(traj_smoothed.begin(),traj_resampled->begin(),
    traj_resampled->begin()+ *traj_resampled_closest);
- overwriteStopPoint(*traj_resampled,traj_smoothed);
```

In this fix, the function `overwriteStopPoint` is being called with updated parameters while being moved forward. By adjusting the source of the stop point data and how the trajectory is assembled, the updated code aims to improve the vehicle's path planning accuracy, making the stop points more precise and better aligned with the actual route and obstacles.
Sensor Data Interpretation (SDI). Address issues in interpreting data from sensor input (e.g., LiDAR, radar, and cameras), ensuring accurate and reliable environmental perception.
Control System Adjustment (CSA). Optimize vehicle's control systems, such as steering, braking, and throttle, which are vital to ensure smooth and safe vehicle operation.
Predictive Algorithm Enhancement (PAE). Improve the algorithms responsible for predicting potential hazards or traffic conditions for proactive safety measures and efficient route planning.
SLAM Algorithm Refinement (SAR). Refine Simultaneous Localization and Mapping (SLAM) algorithms used in the Localization module to enhance map creation and position estimation.
Environmental Adaptability (EA). EA involves enhancements in the system's ability to adapt to different environmental conditions, such as varying weather, lighting, or road surface conditions, which are critical for the reliability of ADSes. The code snippet below illustrates an EA bug fix:

```
  if ((rclcpp::Time(msg->stamp)-rclcpp::Time(latest_perception_msg_.stamp)).
    seconds() > perception_time_tolerance_) {
-   latest_external_msg_.signals.clear();}
+   latest_perception_msg_.signals.clear();}
```

The repair suggests the original code was incorrectly resetting wrong signals when a delay in perception message updates was detected. By clearing correct signals, the system ensures outdated perception data does not interfere with the ability to adapt to current environmental conditions.
Firmware-Software Harmonization (FSH). Align the vehicle's firmware with higher-level software systems, ensuring that hardware-software interactions are seamless and efficient.
Module Integration and Interaction (MII). Improve the integration and interaction among various ADS modules for ensuring cohesive and harmonious system operations.
Safety Protocol Enhancement (SPE). Enhance safety features to ensure compliance with emerging safety standards and regulations, as well as to address newly identified safety concerns.

Communication Protocol Refinement (CPR). Enhance communications for vehicle-to-vehicle (V2V) and vehicle-to-everything (V2X) interactions to ensure robust and secure data transmission.
Real-time Data Processing Improvement (RDPI). Improvements in algorithms for ensuring efficient handling of real-time data across modules and sub-modules for better vehicle control.
Autonomous Decision-Making Improvement (ADMI). Enhancements in ADS decision-making algorithms, ensuring that the vehicle makes safe and logical decisions in real-time traffic scenarios.
Simulation Performance Optimization (SPO). Improve the simulator's performance to enable more efficient virtual testing. This pattern could involve code optimizations to reduce computational load, enhance rendering speeds, and decrease latency in the simulation environment.

4.2.2 Domain-Independent Semantic Bug-Fix Patterns. Domain-independent semantic bug-fix patterns play a vital role in the overall functionality and robustness of ADSes. These patterns are not unique to ADS but are critical for ensuring the software's operational efficacy and reliability.
Data Flow Correction (DFC). Address issues in the way data is passed and used throughout the system. Rectify the data flow between functions to ensure data is correctly processed and utilized.
Logic Amendment (LA). Changes in the underlying logic of the code, which is applied when existing logic leads to incorrect or undesired behaviors to realign with the intended functionality.
State Management Enhancement (SME). Refine how states (e.g., vehicle state or sensor state) are tracked, updated, and managed within the system to ensure correct state-dependent functionalities.
Interface Consistency (IC). Interface consistency fixes are applied to standardize and streamline the interactions between different software modules or components.
Resource Management Improvement (RMI). Optimize the utilization of system resources like memory, processing power, and network bandwidth that can impact system performance.
Concurrency Control (CC). Address issues arising from simultaneous operations or multi-threaded environments, ensuring concurrent processes do not lead to conflicts, deadlocks, or inconsistency.
Error Handling Refinement (EHR). Improve the system's ability to detect, report, and recover from errors. Robust error handling can prevent cascading failures and improve system resilience.
Configuration and Environment Management (CEM). Update configuration or environment options to ensure that the ADS operates correctly and is adaptable to different operational contexts.
Security Strengthening (SS). Enhance the system's defense against external threats and vulnerabilities, including security flaws, which is important in ADS due to its autonomous nature.
User Interface Adjustment (UIA). Changes to the human-machine interface of ADS, aimed at improving usability, providing clearer information, or enhancing the overall user experience.
Dependency Update (DU). Update or modify external libraries or components to ensure compatibility and that the system benefits from the latest improvements of its dependencies.
Build and Compilation Enhancement (BCE). This pattern aims to improve the efficiency, reliability, and accuracy of the build and compilation processes in software development.
Debugging Tools Improvement (DTI). DTI involves integrating better logging systems, visual debugging aids, and automated analysis tools to help identify and diagnose issues.
Documentation Update and Clarification (DUC). This pattern involves correcting and updating the documentation to ensure that it reflects the current state of the project accurately.

5 Experimental Results

This paper investigates bug-fix patterns in ADSes to address the following research questions:

RQ1 (Bug-fix pattern frequency): What are the most common bug-fix patterns in the ADS?

This question aims to unravel the prevalent methodologies employed to address syntactic bugs—those related to the structure or format of the code—and semantic bugs, which concern the meaning or behavior of the code within ADS.

RQ2 (Bug-fix action frequency): What are the most common bug-fix actions in the ADS?

This question investigates the frequency of specific bug-fix actions and strategies in the ADS. By categorizing these actions into domain-specific and domain-independent fixes, the study aims to provide actionable insights into which strategies are most frequent in bug fixing.

RQ3 (Bug-fix patterns across root causes): How do semantic bug-fix patterns vary across different root causes of bugs in the ADS?

This analysis delves into the diversity of bug origins within ADS and examines the correspondence between these root causes and the bug-fix patterns applied. Understanding this correlation is important for tailoring debugging strategies to the specific nature of a bug's root cause.

RQ4 (Bug-fix patterns across symptoms): How do semantic bug-fix patterns vary across different symptoms in the ADS?

This question focuses on the manifestation of bugs through various symptoms within ADS. By mapping bug-fix patterns to specific symptoms, the study aims to elucidate targeted strategies for symptom-based debugging, fostering symptom-oriented guidance to ADS optimization.

RQ5 (Bug-fix patterns across ADS modules): Are semantic bug-fix patterns different across various modules in the ADS?

ADS encompasses a spectrum of modules, this question explores whether the bug-fix patterns employed are module-specific, reflecting the unique challenges and requirements of each module.

5.1 RQ1: Bug-Fix Pattern Frequency

For RQ1, we analyzed the frequency and types of common bug-fix patterns in autonomous driving systems, specifically focusing on Apollo and Autoware. The results, presented in Table 5 and Table 6, categorize the bug-fix patterns into syntactic and semantic types.

In both ADSes, syntactic BFPs, particularly 23.9% for Method Call (MC) and 18.3% for If-related (IF), are most frequently observed. These patterns typically involve adjustments in the code's basic structure and flow, suggesting a recurring need for attention to fundamental coding practices in ADS development. The MC pattern represents nearly one-quarter of all bug-fix patterns in both Apollo (26.9%) and Autoware (23.1%). This suggests that method invocation errors are particularly challenging in ADSes. Given that method calls often involve integrating multiple components, it hints at the complexity and interdependencies within these systems. This could imply a need for better modularization or more robust testing around component interactions.

Table 5. Syntactic Bug-Fix Patterns in ADSes

Syntactic Bug-Fix Pattern	Apollo	Autoware	Both
If-related (IF)	13.6%	19.5%	18.3%
Assignment (AS)	15.6%	14.7%	14.8%
Method Call (MC)	26.9%	23.1%	23.9%
Method Declaration (MD)	1.8%	1.9%	1.9%
Sequence (SQ)	2.8%	2.4%	2.5%
Loop (LOOP)	3.8%	6.0%	5.5%
Return (RT)	4.0%	8.4%	7.6%
Local Variable (LV)	9.5%	12.6%	12.0%
Non-source-code Variable (NV)	7.3%	5.9%	6.1%
Library (LIB)	2.8%	1.8%	2.0%
Syntax (SYN)	1.8%	0.1%	0.4%
Try-Catch (TRY)	0.6%	0.2%	0.3%
Switch (SW)	0.2%	0.1%	0.1%
Class Field (CF)	0.0%	0.1%	0.1%
Documentation (DOC)	9.1%	3.2%	4.4%

Finding 1: Syntactic bug-fix patterns like 18.3% of If-related (IF) and 23.9% of Method Call (MC) are the most common types in ADSes, which indicates that issues related to control flow decisions and incorrect method invocations are prevalent.

Moreover, domain-specific semantic bug-fix patterns, including 22.9% Path Planning Optimization (PPO) and 6.5% Module Integration and Interaction (MII), also feature prominently in our findings. These patterns often involve complex modifications and a deep understanding of the system's operational logic. The complexity and critical nature of these fixes, particularly in MII, reveal the intricacies involved in ADS software maintenance, highlighting the necessity for robust architectures. This complexity underscores the importance of both software researchers and practitioners in directing attention to the integration, modular architecture, and encapsulation of ADS [55, 56]. Moreover, the significant presence of MII-related issues suggests an impending need to prioritize integration testing in ADS, an area that currently lacks comprehensive study but is crucial for vehicle safety and performance [44].

Finding 2: Semantic bug-fix patterns like 22.9% fixes of Path Planning Optimization (PPO) and 6.5% of Module Integration and Interaction (MII) are the most common domain-specific ones. The prominence of MII-related fixes indicates the need for improved module integration testing techniques and advanced modular ADS software architecture design and encapsulation.

The domain-independent semantic patterns, such as 13.6% of Data Flow Correction (DFC) are indicative of foundational challenges in ensuring cohesive system operation. The prevalence of DFC in bug-fix patterns highlights a critical area of system functionality that often is understudied, pointing to a gap in data flow-oriented testing within the software testing domain [54, 60]. This gap suggests the need for more comprehensive data-flow testing strategies that move beyond traditional code coverage to enhance data-flow coverage and system reliability.

Finding 3: The prominence of domain-independent semantic bug-fix patterns like 13.6% of Data Flow Correction (DFC) indicates the need for more research on data flow-based testing (e.g., data-flow coverage) for ADSes, which may be more computationally expensive but possibly worthwhile for these safety-critical ADSes.

Table 6 also indicates that Autoware has a higher ratio of Path Planning Optimization (PPO) bug-fix patterns, possibly due to its specific development focus and project needs since Autoware has more code lines and sub-modules in the Planning module than Apollo does. Autoware also claims to be a customizable and easily extendable Planning development platform [25]. Its maintainers accepted more pull requests from community contributors around the world while Apollo's maintainers tended to merge more pull requests from Baidu employees. Apollo's higher number of build, debug, and documentation-related instances reflect different project workflows and priorities, such as a stronger emphasis on ensuring robust documentation and debugging processes to facilitate collaboration and maintenance.

Table 6. Semantic Bug-Fix Patterns in ADSes

Semantic Bug-Fix Pattern	Apollo	Autoware	Both
Domain-specific Semantic Patterns			
Path Planning Optimization (PPO)	9.2%	27.3%	22.9%
Sensor Data Interpretation (SDI)	2.0%	2.4%	2.3%
Control System Adjustment (CSA)	1.2%	1.5%	1.4%
Predictive Algorithm Enhancement (PAE)	2.0%	0.6%	0.9%
SLAM Algorithm Refinement (SAR)	3.8%	2.9%	3.1%
Environmental Adaptability (EA)	4.7%	4.0%	4.2%
Firmware-Software Harmonization (FSH)	2.2%	0.4%	0.8%
Module Integration and Interaction (MII)	12.4%	4.6%	6.5%
Safety Protocol Enhancement (SPE)	0.0%	0.7%	0.5%
Communication Protocol Refinement (CPR)	0.5%	0.6%	0.6%
Real-time Data Processing Improvement (RDPI)	2.8%	0.7%	1.2%
Autonomous Decision-Making Improvement (ADMI)	1.3%	0.3%	0.5%
Simulation Performance Optimization (SPO)	0.2%	1.5%	1.1%
Domain-independent Semantic Patterns			
Data Flow Correction (DFC)	11.2%	14.4%	13.6%
Logic Amendment (LA)	2.0%	2.1%	2.1%
State Management Enhancement (SME)	3.2%	6.1%	5.4%
Interface Consistency (IC)	0.3%	0.7%	0.6%
Resource Management Improvement (RMI)	1.2%	0.4%	0.6%
Concurrency Control (CC)	0.3%	0.3%	0.3%
Error Handling Refinement (EHR)	4.0%	3.3%	3.5%
Configuration and Environment Management (CEM)	7.2%	7.4%	7.3%
Security Strengthening (SS)	0.5%	0.1%	0.2%
User Interface Adjustment (UIA)	2.5%	2.4%	2.5%
Dependency Update (DU)	5.2%	3.0%	3.6%
Build and Compilation Enhancement (BCE)	8.5%	5.4%	6.2%
Debugging Tools Improvement (DTI)	4.0%	3.3%	3.5%
Documentation Update and Clarification (DUC)	7.5%	3.5%	4.5%

This could be due to Apollo's larger user base and its need for clear documentation and thorough debugging to support a wide range of developers and applications. Additionally, Apollo's extensive integration with various hardware and software components, reflected by the Module Integration and Interaction (MII) patterns of 12.4% in Apollo compared to 4.6% in Autoware, may require more frequent build and debug fixes to maintain compatibility and performance. Autoware just finished its major version update to Autoware.universe [17], the tutorial and documentation are still under construction and waiting to be completed in the future.

Finding 4: Autoware shows a higher ratio of Path Planning Optimization (PPO), due to its customizable Planning platform and larger Planning module, which accepts more contributions from the open-source community. Apollo emphasizes build, debug, and documentation-related fixes, reflecting its broader user base and the need for thorough debugging and integration with diverse hardware and software. Apollo's focus on Module Integration and Interaction (MII) patterns also highlights the complexity of maintaining compatibility across its components.

5.2 RQ2: Bug-Fix Action Frequency

This research question explores the prevalence of domain-specific and domain-independent bug-fix actions in ADS, aiming to categorize them into frequently occurring strategies and identify patterns that could guide targeted repair methods. By quantifying and analyzing these actions, we aim to discern which bug-fix actions are most effective and frequently utilized across ADSes.

Table 7. Domain-Independent Bug-Fix Actions with Occurrences

Bug-Fix Action	#N	Description
Adjust Return Values	187	Add, delete, or change the return values, which is crucial for fixing incorrect output.
Update System Config	177	Add, remove, or change the system configuration or settings to ensure correct behavior.
Fix API Misuse	166	Correction of misused API functions to improve system interoperability.
Add Condition Logic	147	Add conditional statements to handle special cases or edge conditions in the code logic.
Change Condition Logic	123	Modify existing conditional logic to correct bugs or optimize decision-making pathways.
Update Comments	109	Update code comments or documentation to clarify or correct misunderstandings.
Modify Debug Handler	96	Enhance the error handling or debugging mechanisms to improve diagnostic ability.
Update Data Values	84	Adjustments to how data values are handled or calculated within the code.
Modify Syntax Error	66	Corrections of syntax errors or typos that may cause compilation or runtime errors.
Move Statements	62	Relocation of code statements for optimization or to resolve execution order issues.
Modify GUI Property	58	Changes or additions to graphical user interface elements or properties.
Change Library Import	49	Add, delete, or change the libraries importing to resolve dependency issues.
Optimize Script Tools	46	Improve the efficiency or functionality of scripts or tools used within the system.
Update Dependency	42	Add or remove project dependencies to resolve conflicts or ensure proper library linking.
Modify Parameter Path	39	Changes to parameter paths within functions or systems to resolve accessibility issues.
Add/Delete/Modify Parameter Declaration	39	Adjust variable or parameter declarations to fix scoping or initialization problems.
Change Data Processing	31	Modifications to how data is processed, often to fix errors or improve efficiency.
Fix Memory Issue	30	Resolution of memory management issues, such as leaks or improper allocation.
Modify Command	29	Update commands or options to resolve functional issues or improve user interaction.
Change Data Type	28	Changes in data types to resolve type mismatches or improve data integrity.

Table 7 presents the data of frequent domain-independent bug-fix actions that occurred more than 20 times. The most common actions involve adjustments to return values (187 occurrences) and system configurations or settings (177 occurrences), which emphasize the need for fine-tuning output and system behavior. This reflects the dynamic nature of ADS, where precise control over return values and system configurations is essential for performance and safety. API misuse fixes (166 occurrences) are also frequent, suggesting that proper API usage is critical in ADS development. Errors in API functions can lead to cascading issues in the system, necessitating frequent corrections.

Additionally, logical condition changes and additions (270 occurrences in total) indicate that much of the bug-fixing effort in ADS involves improving decision-making logic, highlighting the complexity of handling edge cases and optimizing pathways.

The frequency of certain bug-fix actions, such as return value adjustments and API misuse corrections, suggests that developing specialized automated repair tools for these categories could substantially reduce manual debugging efforts. Additionally, the prominence of logical condition modifications implies that enhancing logic-checking mechanisms could improve overall system reliability and performance. By identifying recurring actions, this study provides actionable insights for prioritizing repair strategies that target the most frequent and critical bug types in ADS.

Finding 5: The most frequent bug-fix actions in ADS are related to adjustments of return values and configurations, indicating the critical need for precise control over output and system behavior. Frequent API misuse fixes also highlight the importance of proper API usage, as errors can propagate and cause system-wide issues. Logical condition changes and additions point to the complexity of decision-making logic in ADS, emphasizing the need to handle edge cases.

As shown in Figure 2, we also examine the frequency of domain-specific bug-fix actions in ADSes, focusing on the most frequent actions that occur more than 20 times and highlighting their importance within the ADS context. The domain-specific data clearly illustrates that Planning is the most critical ADS module, with a diverse range of sub-modules contributing to the system's overall functionality and 12 frequent bug-fix actions, such as *Modify Avoidance Processing* (98 occurrences) and *Adjust Stop/Park Handling* (88 occurrences), point to its centrality in ADS operations. Planning sub-modules are integral to both path planning and lane-change processing, such as obstacle avoidance and velocity handling, reflecting their crucial role in vehicle decision-making. In addition to Planning, the Perception module stands out as the second most important and complex module of the ADS, with 4 frequent bug-fix actions. The Perception module plays a critical role in integrating data from object detection sensors and software fusion algorithms, making it essential for accurately interpreting the environment. The complexity of this module lies in its dual functionality—relying on both hardware (sensors) and software (fusion algorithms)—to ensure real-time object detection, mapping, and environmental understanding within the ADS.

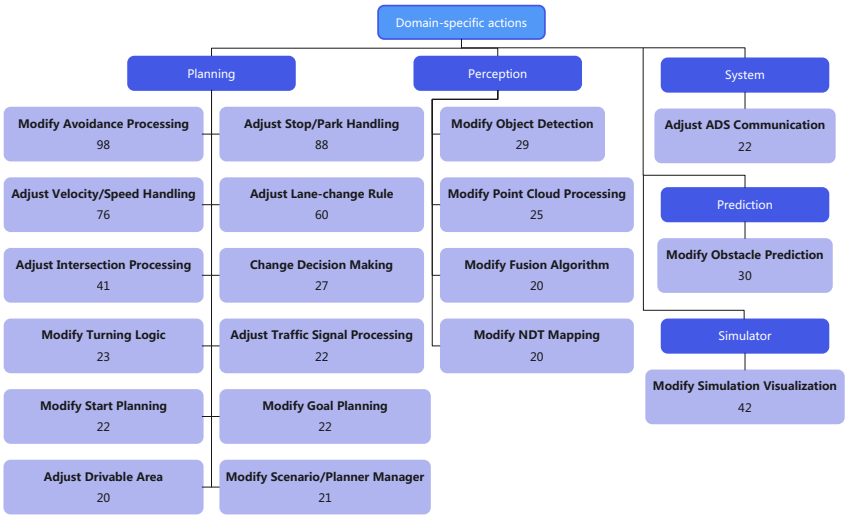

Fig. 2. Domain-Specific Bug-Fix Actions with Occurrences

Object and obstacle processing emerges as one of the most important ADS functionalities, with bug-fix actions like 98 occurrences of *Modify Avoidance Processing*, 30 of *Modify Obstacle Prediction*, and 29 of *Modify Object Detection* being highly frequent. This demonstrates that ensuring system's ability to detect, predict, and avoid obstacles is a top priority. These actions span multiple ADS modules, including Perception (object detection), Planning (avoidance), and Prediction (obstacle prediction), highlighting the complexity and interdependence of object handling across the ADS.

Finding 6: Object and obstacle processing is a major focus, emphasizing the importance of detecting, predicting, and avoiding obstacles. The frequent occurrence of object-processing actions across multiple ADS modules reflects the interdependence and complexity of handling obstacles, highlighting the priority of maintaining robust object-handling mechanism.

Autonomous vehicle maneuvers play an important role in ADS path planning. Frequent bug-fix actions like *Adjust Stop/Park Handling* (88 occurrences), *Adjust Velocity/Speed Handling* (76 occurrences), and *Modify Turning Logic* (23 occurrences) stress the importance of refining these maneuvers to avoid accidents and ensure smooth navigation. The complexity of handling speed and velocity, in particular, indicates a need for ongoing refinement in dynamic vehicle control.

Lane-related bug-fix actions also feature prominently, with *Adjust Lane-change Rule* (60 occurrences), *Modify Goal Planning* (22 occurrences), *Modify Start Planning* (22 occurrences), and *Adjust Drivable Area* (20 occurrences) indicating the system's reliance on precise lane generation and start and destination planning for safe and efficient travel. Lane changes, drivable area processing, and goal planning are all essential for adaptive route selection, further emphasizing the importance of these components within the broader ADS planning framework.

The *Modify Simulation Visualization* (42 occurrences) demonstrates the critical need for simulator reliability in ADS development. Simulations are used to test and verify system behaviors before deployment in the real world. A stable and accurate simulation environment is vital for scenario-based testing, such as obstacle detection and collision avoidance, to ensure that the ADS performs as expected in diverse conditions. High-frequency modifications in this area signal the need to continuously refine simulation environments for more reliable and user-friendly ADS simulators.

Finding 7: Frequent maneuver-related bug-fix actions (187 occurrences) highlight the need for precise control to ensure safe navigation and prevent accidents. Lane-related actions (102 occurrences) underscore the importance of accurate lane and route planning for adaptive and efficient travel. The simulation-related action (42 occurrences) emphasizes the critical role of simulation in verifying ADS behaviors under various conditions. Frequent adjustments to the simulator highlight the ongoing need to enhance its stability and accuracy for reliable testing.

5.3 RQ3: Bug-Fix Patterns Across Root Causes

For both ADSes, we identified a total of 1,331 instances of these root causes. Figure 3 correlate semantic BFPs with root causes. The Algorithm (Alg) root cause has a significant occurrence of 887 bug fixes and spans across 23 types of semantic bug-fix patterns, which underlines the complexity and multifaceted nature of these bugs. This observation, aligned with the detailed classification of the affected algorithm or functions for pull requests in Table 8, where *Evasive Actions*, particularly related to obstacle processing, emerges as the most corrected subgroup, underscoring the criticality of refining obstacle avoiding algorithms in ADS. The dataset we collected presents a compelling case for developing learning-based automated program repair (APR) approaches, particularly as the top five affected algorithms—*Avoidance*, *Lane Change*, *Scene Intersection*, *Start Planner*, and *Goal Planner*—point to specific areas where such APR strategies could be most beneficial.

Finding 8: Algorithm-related bug fixes, with a total of 877 occurrences, are characterized by various semantic patterns that require multifaceted code modifications to address the underlying incorrect algorithm logic. This complexity, notably present in the *Evasive Actions* subgroup, suggests an opportunity for advancing APR techniques for obstacle processing algorithms.

For the Config root cause, 121 occurrences of Configuration and Environment Management (CEM), 51 of Build and Compilation Enhancement (BCE), and 35 of Module Integration and Interaction (MII) dominate semantic patterns. This trend suggests that configuration-related bugs not only require changes in system configuration files but also demand adjustments in build files and hardware settings. This highlights the broader scope of configuration bugs, which extend beyond code to include systemic and environmental factors. According to recent surveys in automated program repair [39, 62], although APR has made strides in addressing code-level issues, the aspect of configuration repair remains relatively unexplored, necessitating focused efforts to fill this gap.

Table 8. Classification of Affected Algorithm Groups with Frequency

Algorithm Group	Subgroup	Total (%)	Example
Planning & Navigation	Route planning and goal setting	41 (5.5%)	Start Planner
	Lane-related maneuvers	54 (7.2%)	Lane Change
	Evasive actions	82 (11.0%)	Avoidance
	Parking and stopping maneuvers	25 (3.4%)	Pull Over
	Intersection handling	40 (5.4%)	Scene Intersection
	Route management	44 (5.9%)	Mission Planner
Control & Actuation	Motion control	18 (2.4%)	MPC Controller
	Command decision systems	18 (2.4%)	Turn Signal Decider
Perception & Sensing	Sensor data processing	3 (0.4%)	Pointcloud Preprocessor
	Sensor data fusion	6 (0.8%)	Perception Fusion
	Object and environment detection	30 (4.0%)	CV Tracker
Prediction	Entity Behavior Prediction	12 (1.6%)	Map Based Prediction
	Traffic and Environmental Prediction	3 (0.4%)	Traffic Light Predictor
Mapping & Localization	Map handling and management	34 (4.6%)	PNC Map
	Localization and position estimation	8 (1.1%)	EKF Localizer
System Management & Infrastructure	System monitoring and management	24 (3.2%)	System Error Monitor
	Data management and configuration	9 (1.2%)	CSV Loader
Simulation, Testing, & Evaluation	Simulation and testing environments	7 (0.9%)	Planning Simulator
	Performance evaluation and analysis	7 (0.9%)	Trajectory Evaluator
User Interface & Communication	HMI and display systems	4 (0.5%)	RTC Manager Panel
	Communication and data exchange	2 (0.3%)	Cyber Channel
Emergency Handling & Safety	Emergency response	10 (1.3%)	Emergency Handler
	Risk checking and safety planning	3 (0.4%)	Path Safety Checker

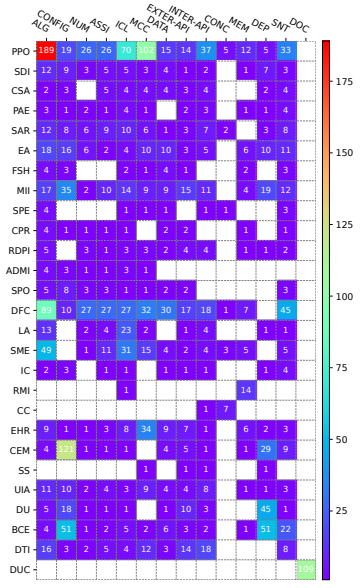

Fig. 3. Relation between root causes and Semantic BFPs for both ADSes

Finding 9: Most configuration-related bugs are fixed by Configuration and Environment Management (CEM), Build and Compilation Enhancement (BCE), and Module Integration and Interaction (MII) patterns (207 occurrences in total), indicating the need for comprehensive fixes that span both source code and configurations. This situation emphasizes the critical gap in automated testing and program repair research concerning ADS configuration.

5.4 RQ4: Bug-Fix Patterns Across Symptoms

As shown in Figure 4, this research question illustrates the relationship between bug symptoms observed in Apollo and Autoware and semantic BFPs, with 2473 symptom instances in total.

The occurrence of domain-specific bug-fix pattern Path Planning Optimization (PPO) and domain-independent pattern Data Flow Correction (DFC) across various symptoms, such as Crashes, Stop and parking (Stop), Lane Positioning and Navigating (LPN), Speed and Velocity Control (SVC),

Turning (Turn), Trajectory (Traj), Obstacle Processing (OP), and Logic, signifies the critical nature of these patterns in addressing diverse and complex issues in different driving scenarios.

Finding 10: The frequent occurrence of Data Flow Correction (DFC) (364 occurrences) in diverse symptoms underscores the importance of data-flow handling for reliable autonomous vehicle operations, which suggests the need for robust data-flow testing and correction for ADSes.

The 30 instances of Debugging Tools Improvement (DTI) bug fixes, primarily in Display and GUI (DGUI) symptoms, demonstrate that ADS developers value debugging methods that are specialized to their domain, but further value effective visualizations for debugging concerns. Existing debugging techniques [20, 24, 27, 47], such as log analysis, breakpoint debugging, and unit testing, provide a broad foundation for software diagnosis. However, the absence of dedicated research on convenient debugging tools for ADS reveals a critical gap, where the intricate and real-time nature of ADS presents unique challenges such as complex sensor data integration, safety-critical decision-making, and dynamic environmental interaction. Improved visualization tools for debugging may consider offering a more intuitive understanding of ADS behavior and error states.

Finding 11: The prevalence of 30 Debugging Tools Improvement (DTI) bug fixes in DGUI symptoms highlights the critical need for ongoing improvements in debugging tools and user interfaces within autonomous vehicle systems.

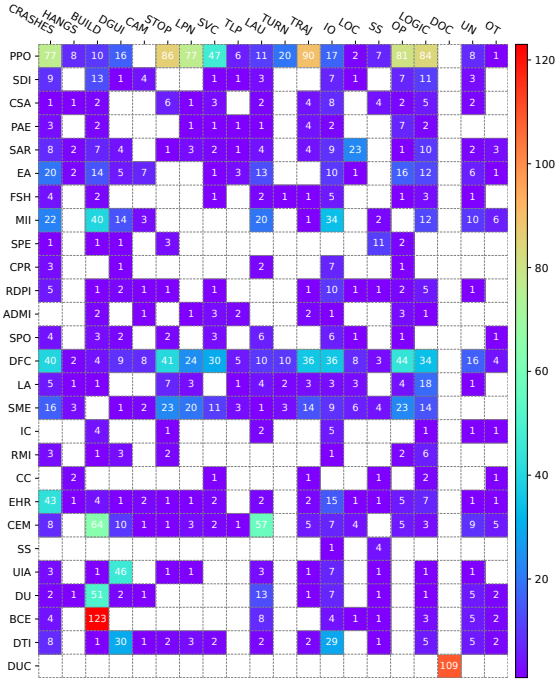

Fig. 4. Relation between Symptoms and Semantic Bug-Fix Patterns for Apollo and Autoware

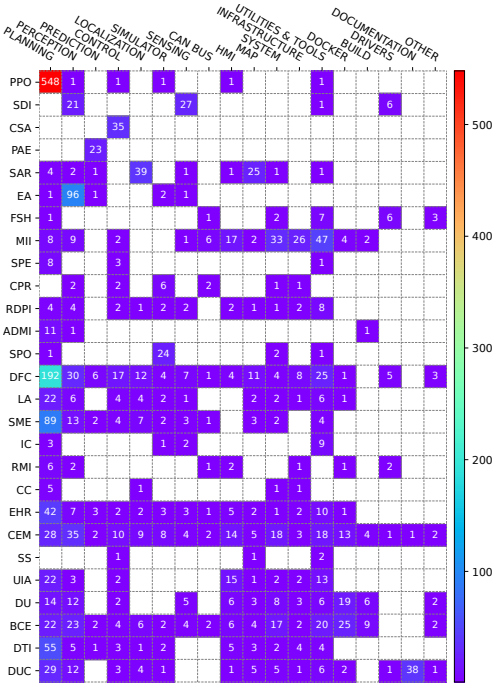

Fig. 5. Relation between Modules and Semantic Bug-Fix Patterns for Apollo and Autoware

State Management Enhancement (SME) bug-fix patterns are most commonly observed in Stop and Parking (Stop) (23 occurrences), Obstacle Processing (OP) (23 occurrences), and Lane Positioning and Navigating (LPN) (20 occurrences) symptoms. Effective state management is crucial for ensuring seamless transitions between different operational modes, such as transitioning from

driving to parking or navigating through obstacles. This necessity points to the intricate interplay between state control mechanisms and the vehicle's capacity to interpret and react to dynamic driving environments, thereby stressing the importance of sophisticated state handling strategies to mitigate potential risks and enhance driving safety. These scenarios, which necessitate precise state transitions such as moving from forward to reverse for parking or adjusting to dynamic obstacles, highlight the critical role of advanced state management.

Finding 12: The concentration of State Management Enhancement (SME) patterns in scenarios like Stop and Parking, Obstacle Processing, and Lane Positioning and Navigating underscores the importance of robust state management for AV adaptability to diverse driving conditions.

The significant count of 178 STOP-related bug-fix patterns in AVs underscores the complex and persistent issues in stopping and parking functionalities, revealing systemic vulnerabilities and the need for refined parking algorithms and community concerns. These patterns suggest that current methodologies may fall short in addressing the intricate interactions among subsystems essential for efficient stop and parking operations. Few existing papers focus on enhancing the parking and stopping of the ADS [28, 40], this gap in research and development highlights the urgency of advancing testing frameworks and enhancing safety standards to address the operational complexities and potential hazards inherent in AV stop and parking mechanisms.

Finding 13: The presence of 178 STOP-related bug-fix patterns and few AV studies focusing on such functionalities reveals a major gap of stop and parking optimization in ADS research.

5.5 RQ5: Bug-Fix Patterns Across ADS Modules

This RQ investigates the degree of repair efforts associated with ADS modules. Figure 5 presents the occurrences of semantic bug-fix patterns within distinct modules of Apollo and Autoware.

In the ADS domain, the Planning module is distinguished by its coverage of 22 types of semantic BFPs, with PPO manifesting 548 times, making it one of the most frequently occurring patterns in ADS modules. The Planning module is also the most complicated module, with a total of 50 sub-modules for Apollo and 51 for Autoware. With its diverse sub-modules like trajectory generation and velocity planning, the Planning's complexity and centrality in ADS underscore the necessity for improved design and modularization for enhanced maintenance and functionality.

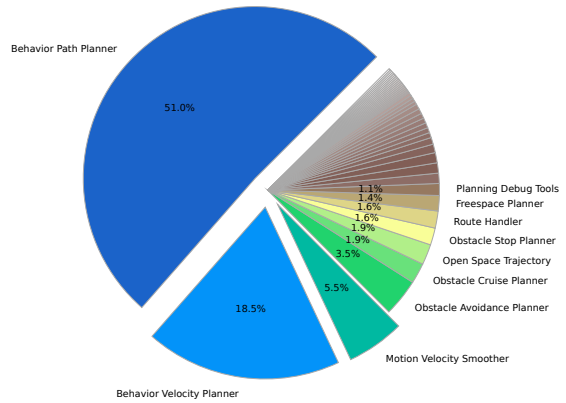

Fig. 6. Distribution of Sub-modules in Planning Module (Labels less than 1.0% are hidden)

The Planning's intricacy is further illuminated by the frequent occurrence of PPO and its subdivision into numerous sub-modules, as highlighted in Figure 6, such as the *Behavior Path Planner* (51.0%), whose dominance signifies its critical role in determining the optimal path. Its frequent interaction with other subsystems makes it a focal point for potential errors. The *Behavior Velocity Planner* at 18.5% further emphasizes the complexity of real-time decision-making in velocity adjustments, affecting safety and performance. *The Motion Velocity Smoother* at 5.5% plays a crucial role in ensuring the vehicle's movements remain smooth and predictable, reducing jerky motions that could compromise passenger comfort and safety. This intricate structure of interdependent

sub-modules and the transition of Apollo to fragment modules into smaller packages in *v9.0* [3, 18] reflects the growing need for refined architectural and modularization approaches in ADS, particularly to enhance the Planning's efficiency and maintainability in handling real-world challenges like obstacle avoidance and smooth velocity transitions. The Planning's complexity, as evidenced by the high frequency of semantic bugs in its core sub-modules, indicates that further research into modularization could lead to significant performance improvements and error reduction.

Finding 14: The prominent occurrence of 548 instances of semantic bug-fix patterns and about 50 sub-modules in the Planning module underscores its centrality and complexity in both ADSes. Key sub-modules about path, velocity, and avoidance planning contribute significantly to this complexity due to their high functional load and frequent interaction with other sub-modules. The varied roles of these sub-modules highlight the need for a more refined software architecture and improved granular modularization to facilitate easier maintenance and system upgrades.

Domain-specific semantic BFPs are predominantly associated with particular modules, whereas domain-independent BFPs are more evenly spread across various modules. For instance, SLAM Algorithm Refinement (SAR) is specific to the Localization and Environmental Adaptability (EA) to the Perception. This distribution implies domain-specific issues are closely tied to the functionalities of their respective modules, reflecting adherence to the principle of single responsibility [45], where each module is closely tied to distinct functionalities, necessitating precise, module-specific solutions. Conversely, domain-independent patterns, embodying cross-cutting concerns [34, 35], pervade various modules, suggesting a need for overarching strategies that address these broader issues.

Finding 15: While domain-specific bug-fix patterns tend to coalesce to their most semantically similar ADS module, domain-independent ADS bug-fix patterns cross-cut modules.

Configuration and Environment Management (CEM) is present across all modules, with a notable prevalence in the Perception module (35 occurrences). This indicates that configuration and environmental factors are universally important across modules, with the Perception module being particularly sensitive to these aspects. The pervasive nature of CEM patterns suggests that consistent and accurate configuration and environment management are critical for the seamless operation of autonomous driving systems, especially in processing and interpreting sensory data.

Data Flow Correction (DFC), with a total of 330 occurrences in all modules, stands out as the most common domain-independent bug-fix pattern, indicating that issues related to data handling and processing are pervasive across various modules. This prevalence suggests that ensuring the accuracy and efficiency of data flow is a critical concern in ADSes, reflecting the need for robust mechanisms to manage and correct data flow across the system's diverse components.

Finding 16: The prominence of Data Flow Correction (DFC) as a domain-independent BFP highlights the critical need for accurate and efficient data flow management across the diverse components of ADSes. This prevalence indicates a pressing demand for advanced APR heuristics and debugging tools specifically designed to address the complex data-flow challenges in ADS.

6 Discussion

The prevalence of Data Flow Correction (DFC) underscores the pivotal role of accurate data-flow handling in the ADS (Findings 3, 10, and 16). DFC's significance across various operational contexts reveals that data-flow issues are a major factor in system bugs, necessitating advanced testing and repair strategies. The need for improved tracking and visualization of data-flow is further backed by our study's results, as ADS engineers appear to already struggle from these issues (Finding 11).

In terms of software testing, our study reveals the need for a greater focus on ADS module integration testing (Finding 2) and for more general software testing focused on understudied scenario types, e.g., AV stopping and parking (Findings 12 and 13). In fact, the many sub-modules of Planning indicate that there are a wide variety of opportunities to potentially handle many sub-functionalities of Planning and test them in tandem (e.g., obstacle processing and avoidance (Findings 6 and 8) and parking (Findings 7 and 12)) and with other modules (e.g., Planning and Control along the interactions between software, hardware, and the physical environment (Finding 2)).

The manner in which ADS domain-specific BFPs tend to coalesce in semantically similar ADS modules, while domain-independent BFPs cross-cut those modules (Finding 15), along with certain ADS modules (e.g., Planning) being architected into many sub-modules (Finding 14), suggests that ADS engineers highly value properly-encapsulated and well-architected ADS modules. We are not aware of a previous study that correlated bug fixing and software architecture in such a manner. Future research may consider whether machine learning-based modules (e.g., Prediction and Perception) may benefit in terms of bug fixing from explicit modularization and encapsulation.

The dataset in this study could serve as a benchmark to facilitate future automated program repair research in the ADS domain. The high occurrences of certain bug-fix actions, such as *Adjust Return Values* and *Fix API Misuse* (Finding 5), suggest that developing specialized automated repair tools for these categories could substantially reduce manual debugging efforts. Additionally, the prominence of logical condition modifications implies that enhancing logic-checking mechanisms could improve overall system reliability and performance. By identifying recurring actions, this study provides actionable insights for prioritizing repair strategies that target the most frequent and critical bug types in ADS.

7 Threats to Validity

Internal Threats. The main internal threat to the validity of this study is the potential subjective bias or errors in bug classification. To mitigate this, we began our labeling process using established classification schemes from the literature [52, 57] and employed an open-coding scheme [29] to expand these initial frameworks. We also concentrated on real bug fixes by selecting only closed and merged pull requests. Each bug was independently examined and labeled by two ADS developers, both of whom have contributed to Apollo and Autoware, with any discrepancies resolved through discussion until consensus was achieved. In addition, discussions with Apollo and Autoware developers were held to refine our classification, further reducing subjective biases and errors.

Additionally, our approach to identifying bugs relies heavily on keywords in the context of pull requests. This method might result in false positives since developers sometimes mistakenly mark feature additions as bug fixes. We mitigated this by filtering out false positives during labeling.

External Threats. The primary external threat concerns the reliability of our dataset. To address this, our data collection includes all pull requests, commits, and issues from the inception of the subject ADSes until the major version release at the end of 2023. This comprehensive dataset aligns with similar methods [36, 37, 41, 58, 63] used in other bug studies to identify bug-fix pull requests. Another threat is the open-source nature of the ADSes under study. Our study involves two widely-used representatives—Apollo and Autoware, developed independently and featuring extensive code bases, which may not reflect the bug-fix patterns prevalent in closed-source environments due to differing development processes. As for the generalizability, despite the study encompassing only two systems, they have significant industry and research usage, including the US government, Google, Ford, and various car manufacturers [2, 5–8, 12, 15, 23], and have been developed, iterated, and studied for years, accumulating more research studies, issues, and pull requests than any other small-scale open-source ADS projects. The two projects we selected represent the forefront of

open-source ADS development. Furthermore, most ADSes, including Apollo and Autoware, share the similar reference architecture and are developed from the ROS architecture [50].

Besides, the number of labeled bugs in our study (1,331) is comparable to other bug-fix pattern studies in different domains (e.g., 970 in deep neural networks [42], 446 in deep learning stack [38], and 395 in federated learning systems [33]), underscoring the representativeness of our findings.

8 Related Work

ADS Bug Study. ADSes have been a focus of recent software research, particularly concerning software bugs. Garcia et al. [37] provided a comprehensive study of bugs in AVs, which has become increasingly relevant as these systems become more prevalent in the automotive industry. Building on this, Zampetti et al. [61] conducted an empirical characterization of software bugs in open-source cyber-physical systems, which include ADS. Their study contributes to the understanding of the challenges and common issues in the development and maintenance of these complex systems.

Bug-Fix Patterns. Syntactic bug-fix patterns primarily involve changes in the code structure and statements. Pan et al. [49] were among the pioneers in exploring this area, providing foundational insights into the nature and taxonomy of syntactic bug fixes. This study laid the groundwork for subsequent research, including Zhong et al. [64], who conducted an empirical study on real bug fixes, focusing on the features of Java projects. Soto et al. [53] looked deeper into bug fixes by examining patterns, replacements, deletions, and additions. Following this, Campos et al. [30, 31] conducted large-scale observational studies to identify common BFPs, which were instrumental in cataloging the recurrent patterns in bug fixes across various software. Additionally, Islam et al. [43] introduced more types of syntactic BFPs based on the study of Pan et al. [49] and provided an in-depth analysis of BFPs, particularly focusing on the structural edits and nesting levels in code.

For semantic bug-fix patterns, Islam et al. [42] explored the bug fixes in deep neural networks. Huang et al. [38] presents a comprehensive study on characterizing symptoms, root causes, and fix patterns of dependency bugs across the deep learning stack. Du et al. [33] complements empirical bug studies in deep learning systems by studying six popular federated learning systems.

9 Conclusion

In the rapidly evolving landscape of autonomous driving systems, ensuring the reliability and safety of these systems is essential. Our empirical study, which focuses on bug-fix patterns in two major autonomous driving systems, Apollo and Autoware, has provided a comprehensive insight into the hierarchy of ADS bugs and their fix patterns in this domain. The dominant bug-fix patterns identified, encompassing issues related to path planning optimization, data flow correction, and if-related statements, underscore the multifaceted challenges developers face in maintaining and improving these systems. Our proposed taxonomies of 15 syntactic and 27 semantic bug-fix patterns serve as foundational resources for developers and researchers in the ADS domain by identifying and providing a benchmark of 1,331 bug fixes. In future work, we aim to use the results and dataset we collected from the bug-fix pattern study for facilitating automatic program repair, enhancing debugging tools, training targeted developers, and cross-domain applications in the ADS domain.

10 Data Availability

The dataset and additional analytical figures are available at [4].

Acknowledgments

The authors thank the anonymous reviewers for their constructive feedback and appreciate the support of NSF 2346561, 2443763, CNS-2145493, CNS-1929771, DARPA 230939, and USDOT 69A3552348327 for the Carmen+ University Transportation Center.

References

[1] April 2022. Toyota Follows Tesla In Developing A Vision-Based Self-Driving System. https://bit.ly/3TQQNuw
[2] December 2020. 40+ Corporations Working On Autonomous Vehicles. https://bit.ly/3PdBpGu
[3] December 2023. Baidu launches Apollo Open Source Platform 9.0. https://autonews.gasgoo.com/m/70030171.html
[4] February 2025. Artifacts of the ADS Bug-Fix Pattern Study. https://doi.org/10.5281/zenodo.14807539
[5] January 2018. You can take a ride in a self-driving Lyft during CES - The Verge. https://bit.ly/3Rh1p6s
[6] July 2018. Baidu starts mass production of autonomous buses. https://bit.ly/3oJXrSy
[7] July 2018. Waymo's autonomous cars have driven 8 million miles on public roads. https://bit.ly/43uSNwb
[8] June 2019. CARMA Overview. https://bit.ly/44JQ423
[9] June 2021. Baidu is building Level 4 autonomous robotaxis called Apollo Moon in China. https://bit.ly/3HQ46qH
[10] June 2022. GM's Cruise starts charging fares for driverless rides in San Francisco. https://reut.rs/40rtGJu
[11] March 2022. Udacity: Self-Driving Fundamentals: Featuring Apollo. https://bit.ly/3RnMVC0
[12] November 2018. Baidu hits the gas on autonomous vehicles with Volvo and Ford deals. https://tcrn.ch/3EL8Prj
[13] October 2020. Ford unveils new self-driving test vehicle for 2022 launch. https://cnb.cx/3zi2AIQ
[14] October 2021. Tesla Sold 2 Million Electric Cars: First Automaker To Reach Milestone. https://bit.ly/3G1zDVm
[15] September 2017. The 18 Companies Most Likely to Get Driverless Cars on the Road First. https://bit.ly/3sFsNRo
[16] September 2017. Waymo and Intel Collaborate on Self-Driving Car Technology. https://bit.ly/4d71Mbg
[17] September 2022. Past, Present and the Future of Autoware - Autoware. https://bit.ly/4clFkKW
[18] September 2024. Apollo Architecture Evolution. https://bit.ly/3PnTVwL
[19] September 2024. Baidu Apollo: An open autonomous driving platform. https://en.apollo.auto/apollo-self-driving
[20] September 2024. Debug Autoware - Autoware Documentation. https://bit.ly/3zkkTRg
[21] September 2024. GitHub - Apollo: An open autonomous driving platform. https://bit.ly/3XSQUa5.
[22] September 2024. GitHub - Autoware: An open-source software project for autonomous driving. https://bit.ly/3MIaiCH
[23] September 2024. GitHub - CARMA Platform. https://bit.ly/45GZGvN
[24] September 2024. How to Debug Apollo. https://bit.ly/3TNhvpq
[25] September 2024. Planning component design - Autoware Documentation. https://bit.ly/3SDUzrE
[26] September 2024. Waymo - Self-Driving Cars - Autonomous Vehicles. https://waymo.com/
[27] 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 *IEEE Intelligent Vehicles Symposium Workshops, IV 2021 - Workshops, Nagoya, Japan, July 11-17, 2021*. IEEE, 197–202. doi:10.1109/IVWORKSHOPS54471.2021.9669254
[28] Holger Banzhaf, Dennis Nienhüser, Steffen Knoop, and Johann Marius Zöllner. 2017. The future of parking: A survey on automated valet parking with an outlook on high density parking. In *IEEE Intelligent Vehicles Symposium, IV 2017, Los Angeles, CA, USA, June 11-14, 2017*. IEEE, 1827–1834. doi:10.1109/IVS.2017.7995971
[29] Erik Blair. 2015. A reflexive exploration of two qualitative data coding techniques. *Journal of Methods and Measurement in the Social Sciences* 6, 1 (2015), 14–29.
[30] Eduardo Cunha Campos and Marcelo de Almeida Maia. 2017. Common Bug-Fix Patterns: A Large-Scale Observational Study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.). IEEE Computer Society, 404–413. doi:10.1109/ESEM.2017.55
[31] Eduardo Cunha Campos and Marcelo de Almeida Maia. 2019. Discovering common bug-fix patterns: A large-scale observational study. *J. Softw. Evol. Process.* 31, 7 (2019). doi:10.1002/SMR.2173
[32] Krzysztof Czarnecki. 2018. Operational design domain for automated driving systems. *Taxonomy of Basic Terms ", Waterloo Intelligent Systems Engineering (WISE) Lab, University of Waterloo, Canada* (2018).
[33] Xiaohu Du, Xiao Chen, Jialun Cao, Ming Wen, Shing-Chi Cheung, and Hai Jin. 2023. Understanding the Bug Characteristics and Fix Strategies of Federated Learning Systems. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1358–1370. doi:10.1145/3611643.3616347
[34] Marc Eaddy, Alfred Aho, and Gail C. Murphy. 2007. Identifying, Assigning, and Quantifying Crosscutting Concerns. In *First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM '07)*. 2–2. doi:10.1109/ACOM.2007.4
[35] Marc Eaddy, Thomas Zimmermann, Kaitin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. 2008. Do Crosscutting Concerns Cause Defects? *IEEE Trans. Software Eng.* 34, 4 (2008), 497–515. doi:10.1109/TSE.2008.36
[36] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.).

IEEE Computer Society, 509–519. doi:10.1109/ASE.2017.8115662

[37] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 385–396. doi:10.1145/3377811.3380397

[38] Kaifeng Huang, Bihuan Chen, Susheng Wu, Junming Cao, Lei Ma, and Xin Peng. 2023. Demystifying Dependency Bugs in Deep Learning Stack. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 450–462. doi:10.1145/3611643.3616325

[39] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. *CoRR* abs/2303.18184 (2023). doi:10.48550/ARXIV.2303.18184 arXiv:2303.18184

[40] André Ibisch, Stefan Stümper, Harald Altinger, Marcel Neuhausen, Marc Tschentscher, Marc Schlipsing, Jan Salinen, and Alois C. Knoll. 2013. Towards autonomous driving in a parking garage: Vehicle localization and tracking using environment-embedded LIDAR sensors. In *2013 IEEE Intelligent Vehicles Symposium (IV), Gold Coast City, Australia, June 23-26, 2013*. IEEE, 829–834. doi:10.1109/IVS.2013.6629569

[41] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 510–520. doi:10.1145/3338906.3338955

[42] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: fix patterns and challenges. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1135–1146. doi:10.1145/3377811.3380378

[43] Md Rakibul Islam and Minhaz F. Zibran. 2020. How bugs are fixed: exposing bug-fix patterns with edits and nesting levels. In *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini (Eds.). ACM, 1523–1531. doi:10.1145/3341105.3373880

[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, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 31–43. doi:10.1145/3540250.3549111

[45] Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.

[46] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[47] Pascal Minnerup, David Lenz, Tobias Kessler, and Alois Knoll. 2016. Debugging Autonomous Driving Systems Using Serialized Software Components. *IFAC-PapersOnLine* 49, 15 (2016), 44–49. doi:10.1016/j.ifacol.2016.07.612 9th IFAC Symposium on Intelligent Autonomous Vehicles IAV 2016.

[48] Cliodhna O'Connor and Helene Joffe. 2020. Intercoder Reliability in Qualitative Research: Debates and Practical Guidelines. *International Journal of Qualitative Methods* 19 (2020), 1609406919899220. doi:10.1177/1609406919899220 arXiv:https://doi.org/10.1177/1609406919899220

[49] Kai Pan, Sunghun Kim, and E. James Whitehead Jr. 2009. Toward an understanding of bug fix patterns. *Empir. Softw. Eng.* 14, 3 (2009), 286–315. doi:10.1007/s10664-008-9077-5

[50] 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 *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[51] Christina Rödel, Susanne Stadler, Alexander Meschtscherjakov, and Manfred Tscheligi. 2014. Towards Autonomous Cars: The Effect of Autonomy Levels on Acceptance and User Experience. In *Proceedings of the 6th International Conference on Automotive User Interfaces and Interactive Vehicular Applications, Seattle, WA, USA, September 17 - 19, 2014*, Linda Ng Boyle, Gary E. Burnett, Peter Fröhlich, Shamsi T. Iqbal, Erika Miller, and Yuqing Wu (Eds.). ACM, 11:1–11:8. doi:10.1145/2667317.2667330

[52] Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, 2008, Kaiserslautern, Germany*, H. Dieter Rombach, Sebastian G. Elbaum, and Jürgen Münch (Eds.). ACM, 149–157. doi:10.1145/1414004.1414030

[53] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, 512–515. doi:10.1145/2901739.2903495

[54] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *ACM Comput. Surv.* 50, 1 (2017), 5:1–5:35. doi:10.1145/3020266

[55] Jie Tang, Shaoshan Liu, Songwen Pei, Stéphane Zuckerman, Chen Liu, Weisong Shi, and Jean-Luc Gaudiot. 2018. Teaching Autonomous Driving Using a Modular and Integrated Approach. In *2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC 2018, Tokyo, Japan, 23-27 July 2018, Volume 1*, Sorel Reisman, Sheikh Iqbal Ahamed, Claudio Demartini, Thomas M. Conte, Ling Liu, William R. Claycomb, Motonori Nakamura, Edmundo Tovar, Stelvio Cimato, Chung-Horng Lung, Hiroki Takakura, Ji-Jiang Yang, Toyokazu Akiyama, Zhiyong Zhang, and Kamrul Hasan (Eds.). IEEE Computer Society, 361–366. doi:10.1109/COMPSAC.2018.00057

[56] Ömer Sahin Tas, Florian Kuhnt, J. Marius Zöllner, and Christoph Stiller. 2016. Functional system architectures towards fully automated driving. In *2016 IEEE Intelligent Vehicles Symposium, IV 2016, Gotenburg, Sweden, June 19-22, 2016*. IEEE, 304–309. doi:10.1109/IVS.2016.7535402

[57] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*. IEEE Computer Society, 271–280. doi:10.1109/ISSRE.2012.22

[58] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 805–816. doi:10.1145/2786805.2786850

[59] Susana M. Vieira, Uzay Kaymak, and João M. C. Sousa. 2010. Cohen's kappa coefficient as a performance measure for feature selection. In *FUZZ-IEEE 2010, IEEE International Conference on Fuzzy Systems, Barcelona, Spain, 18-23 July, 2010, Proceedings*. IEEE, 1–8. doi:10.1109/FUZZY.2010.5584447

[60] Elaine J. Weyuker. 1990. The Cost of Data Flow Testing: An Empirical Study. *IEEE Trans. Software Eng.* 16, 2 (1990), 121–128. doi:10.1109/32.44376

[61] Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. 2022. An empirical characterization of software bugs in open-source Cyber-Physical Systems. *J. Syst. Softw.* 192 (2022), 111425. doi:10.1016/J.JSS.2022.111425

[62] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2024. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 55:1–55:69. doi:10.1145/3631974

[63] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 129–140. doi:10.1145/3213846.3213866

[64] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 913–923. doi:10.1109/ICSE.2015.101

Received 2024-09-13; accepted 2025-01-14