# A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems

A Case Study on Apollo

Zi Peng, Jinqiu Yang, Tse-Hsun (Peter) Chen
Concordia University
Montreal, Canada
{zi_peng,jinqiuy,peterc}@encs.concordia.ca

Lei Ma
Kyushu University
Fukuoka, Japan
malei@ait.kyushu-u.ac.jp

## ABSTRACT

Autonomous Driving System (ADS) is one of the most promising and valuable large-scale machine learning (ML) powered systems. Hence, ADS has attracted much attention from academia and practitioners in recent years. Despite extensive study on ML models, it still lacks a comprehensive empirical study towards understanding the ML model roles, peculiar architecture, and complexity of ADS (i.e., various ML models and their relationship with non-trivial code logic). In this paper, we conduct an in-depth case study on Apollo, which is one of the state-of-the-art ADS, widely adopted by major automakers worldwide. We took the first step to reveal the integration of the underlying ML models and code logic in Apollo. In particular, we study the Apollo source code and present the underlying ML model system architecture. We present our findings on how the ML models interact with each other, and how the ML models are integrated with code logic to form a complex system. Finally, we inspect Apollo in a dynamic view and notice the heavy use of model-relevant components and the lack of adequate tests in general. Our study reveals potential maintenance challenges of complex ML-powered systems and identifies future directions to improve the quality assurance of ADS and general ML systems.

## CCS CONCEPTS

• **Computing methodologies** → *Machine learning*; • **Computer systems organization** → **Other architectures**.

## KEYWORDS

Autonomous driving systems, Machine learning, Model testing, Empirical study

**ACM Reference Format:**
Zi Peng, Jinqiu Yang, Tse-Hsun (Peter) Chen and Lei Ma. 2020. A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368089.3417063


*ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA*

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417063

## 1 INTRODUCTION

In recent years, the research community has made significant breakthroughs in machine learning (ML) algorithms and practical applications, e.g., image recognition [21], natural language processing [36], speech recognition [15], and health care [7]. Since the advances in ML, and especially deep learning, practitioners have proposed various solutions to solve important real-world challenges. From face recognition to autonomous driving vehicles, engineers start to design and implement large-scale complex ML-powered systems.

In particular, autonomous driving system (ADS) is one of the most eye-catching and valuable ML-powered systems. ADS aims to automate the vehicle driving process without the need of human intervention. ADS relies on various hardware (e.g., cameras, ranging or radar sensors) to collect information related to the current road condition, and uses deep learning to assist making real-time driving decisions. Thus far, ADS has shown great potentials, and business analysts have estimated that its market value may be worth hundreds of billions of US dollars [41]. However, testing and ensuring the quality of ADS is a challenging engineering problem. Failures in doing so may result in life-threatening accidents with severe consequences [40].

Many exciting recent progress [14, 27, 38, 43] has been made for the quality assurance of ML models in ADS. For example, DeepXplore [27] proposes a differential testing method to detect inconsistent results in multiple ML model version variants. DeepTest [38] generates test images by simulating camera noises to test deep learning models in ADS. Similarly, Zhang et al. [43] transform pictures taken by cameras in ADS using GAN to simulate environment condition changes, and test their corresponding effect on the result of ML models. Although prior studies have shown promising results in testing ML models for ADS systems, these studies only consider the testing of only ML models independently at unit level. However, in reality, there may be various ML models that work collaboratively based on the information received from different cameras and sensors (i.e., not only relying on pictures taken by one single camera). In addition, there may be extensive code logic that interacts with ML models to form a complex ML-powered system such as ADS. For example, ADS may utilize code logic to provide some resilience to noises in the captured images. Therefore, understanding how ML models interact with each other and their integration with code logic in ADS may open up new avenues to future research and practice on the quality assurance of ADS.

In this paper, we conduct an in-depth case study on Apollo 5.0 [3], which is one of the most advanced ADS systems in the world. Apollo is developed by Baidu Inc. and is now widely adopted by a number of world-leading automakers, some of which even start to offer autonomous taxi services [4]. We first perform a qualitative study of the ML models that are used in Apollo and their interactions with the system. We find that Apollo contains a large number of ML models (i.e., 28) that are responsible for various tasks such as traffic light recognition, lane detection, obstacle perception and detection, and trajectory prediction. Then, we study the relationship between the ML models and code logic at the integration level. We find that ML models interact with each other in diverse ways and code logic plays a significant role in such interaction. As a result, the integration of ML models and code logic has inherent complexity. Finally, we inspect Apollo's ML model usage in a dynamic view and also the current testing effort.

In summary, this paper makes the following contributions:

- We find that Apollo contains 28 ML models that are trained based on various deep learning frameworks (e.g., Caffe, Paddle, and PyTorch). Developers may use multiple ML models for the same task (i.e., through a configuration file), or use different ML models for the same task but in different scenarios (i.e., through code logic). We also highlight the potential ML maintenance challenges related to ML model maintenance and management.
- We find that Apollo relies on a high definition (HD) map to track information such as the location of traffic light. In addition, Apollo may combine information from all of the camera, LiDAR, and radar together to make a prediction result, instead of using only camera as studied in prior research [14, 27, 38, 43].
- We find that there exist diverse ways that ML models interact with each other. Such interactions are implemented by non-trivial code logic, which further complicates the overall integration in Apollo. An example of the impact of such interaction is that code logic and the performance of one or several ML model(s) may have a significant impact on the performance of other ML models.
- We categorize the diverse roles that code logic plays in ML model interaction. For example, Apollo developers leverage historical information (e.g., the color of the traffic light a second ago) and various code logic to filter out or correct results from ML models. Moreover, developers may use hard-coded threshold values to determine the cutoff values for various ML model results.
- We inspect Apollo in a dynamic view using a simulated run and find a heavy and repeated use of ML model-relevant code. We also inspect the current test effort in Apollo and notice an overall lack of adequate test.

We position this paper as a first step to understand the complexity to integrate ML models and code logic in real-world ML-powered systems. Such complexity calls for collaborative effort from both researchers and practitioners to further improve the quality assurance of ML-powered systems.

**Paper organization.** Section 2 provides a general background on ADS. Section 3 provides the results to our research questions and discussions of our findings. Section 4 studies the dynamic aspect of ML model interaction in Apollo. Section 5 describes the threats to validity of our study. Section 6 surveys related work. Finally, Section 7 concludes the paper.

## 2 AN OVERVIEW OF AUTONOMOUS DRIVING SYSTEMS

In this section, we describe a brief overview of the architecture in typical Autonomous Driving Systems (i.e, ADS).

With the performance leap of machine learning and deep learning over the past decade, their application across different domains on many challenging tasks becomes possible. Recently, both the academia and industry have started researching and developing autonomous driving systems, a representative complex ML-powered system. The main goal is to create a driverless system that can intelligently navigate a vehicle with full automation. To achieve such an ambitious goal, an autonomous driving system, in general, often contains several important high-level functionalities. First, an ADS needs to have the capability to automatically locate the autonomous driving vehicle and obtain relevant information of road and surrounding environment (e.g., number and location of lanes on a street, the current observed and upcoming status of traffic lights). Second, an ADS needs to efficiently process the information that it received in real time based on the current road condition. For example, it needs to know the distance between the autonomous driving vehicle and the vehicles around it. Third, after processing the surrounding information on road conditions, an ADS needs to make decisions on how the autonomous driving vehicle should turn, accelerate, or decelerate. Finally, an ADS sends the movement decision in the form of control signals to maneuverer a vehicle (e.g., braking, steering the wheel).

To locate the position of an autonomous driving vehicle, an ADS often leverages a high definition map (HD map) and Global Navigation Satellite System (GNSS) sensors. The HD map contains all essential road data such as road boundaries, parking areas, the number of lane lines on a street, locations of traffic lights, crosswalk, etc. GNSS sensors collect information from multiple satellites to provide a more accurate location estimation compared to regular GPS. An ADS uses GNSS to locate the current position of the autonomous driving vehicle on the HD map in centimeter-level accuracy [39]. After the autonomous driving vehicle identifies its location on the map, ADS uses various perception sensors and equipment to obtain information on the surrounding environment. ADS uses LiDAR (Light Detection and Ranging) that leverages pulsed laser to create a 3D model of the surrounding obstacles and their distance from the autonomous driving vehicle. To increase the perception accuracy, an ADS is also often equipped with radars to collect moving object signals, and uses machine learning models to process image signals sensed by cameras to identify nearby objects and obstacles. As a result, the perception component of an ADS often combines information from various sources to make a more accurate estimation on the location and movement of surrounding obstacles (e.g., cars or pedestrians). Based on the perception information, an ADS then predicts, possibly using ML models, the future trajectories of all perceived obstacles (e.g., the direction that a pedestrian is moving towards to). The planning component will then determines a reasonable trajectory according to driving scenarios, and sends the corresponding decision to the driving components, controlling the movement of a vehicle.

As a typical complex machine learning system, an ADS is composed of complicated code logic, integration of various hardware

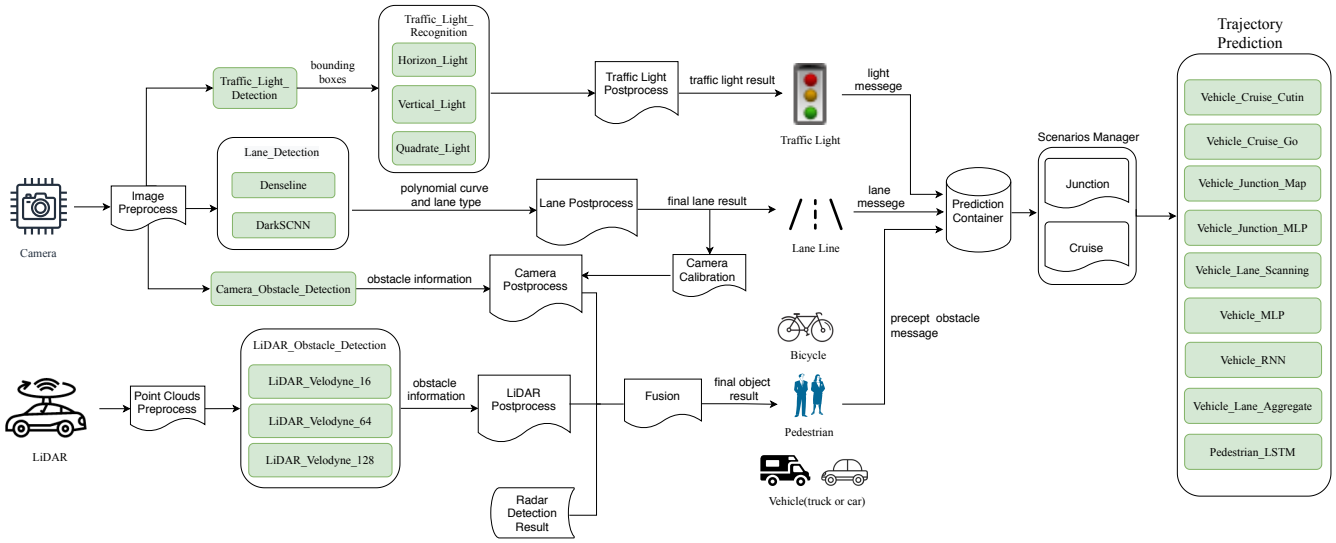


**Figure 1: The relevant components and workflow of Apollo, where the green boxes represent ML models in the system.**

**Table 1: An overview of the statistics of Apollo.**

| Language | No. files | Lines of code |
|---|---|---|
| C/C++ | 3,534 | 566,127 |
| Python | 279 | 36,233 |
| Proto Buffers | 251 | 18,964 |
| Total | 4,064 | 621,324 |

and software components with some of the decision making based on results from machine learning models. There may even be cases where multiple machine learning models collectively make a decision, resulting in a significant impact on the subsequent actions that an ADS takes. In this paper, we conduct a comprehensive study of the industrial intelligent autonomous driving system (i.e., Baidu Apollo 5.0) to perform an in-depth investigation of typical uses of machine learning in a complex ML-powered system at different levels, as well as the potential interaction of ML components with logic of traditional software components. Apollo is a world-leading commercial autonomous driving system that is open-sourced and maintained by Baidu Inc. Table 1 summarizes the overall statistics of Apollo. Apollo is deployed and tested in various cities around the world and provides several enterprise solutions. Some cities now even realize to leverage Apollo in offering autonomous taxi services [4]. Therefore, understanding the roles and how various ML models integrates and interacts in an ML-powered system can greatly facilitate potential system region quality issue detection to improve the quality and safety of ADS in practice.

## 3 CASE STUDY AND RESULTS

In this section, we study Apollo's system architecture, and the interaction between ML models and traditional software components by investigating three research questions. For each research question, we summarize the motivation, approach, results, discussion, and implication.

### RQ1: What is the ML model architecture and relevant information flow in Apollo?

**Motivation.** Recently, extensive studies have been performed on testing and analysis at ML component level [20, 27, 38, 42, 43], which is an important first step for system-level analysis. In practice, an ML model is often not used standalone. Instead, it is integrated into a larger system, wrapping with glue code, and interacting with logic of traditional software components. Up to present, it still lacks a study to perform an in-depth analysis of how an ML model is used in a complex ML-powered system, e.g., the roles of these ML models and how they interact with traditional software, together forming a larger system. This study intends to bridge this gap in performing a comprehensive study on the typical complex ML-powered system, i.e., Apollo autonomous driving system. In practice, an Autonomous Driving System (ADS) often contains multiple components where each component may use various machine learning models to make life-critical decisions. Understanding the types and functionality of these ML models would be an important first step to facilitate future research on quality assurance of ADS and ML-powered system testing in general.

**Approach.** We adopt a hybrid approach that combines both automated and manual analysis to analyze the ML models used in Apollo. First, we manually study Apollo documentation and source code to create a list of the ML frameworks that Apollo uses. During our manual analysis, we find that Apollo follows a practice where it loads external trained ML models into the system by reading the model files using file names. These model files contain a file extension based on the ML frameworks for which the model is trained. For example, in Apollo, the model trained by the Caffe framework [6] has an extension of *.caffemodel*, and the model trained by PyTorch [29] has an extension of *.pt*. Hence, in our second step, we perform the file extension matching study of the ML frameworks that Apollo uses. Then, we perform a systematic file extension searching in Apollo source code to localize each ML model. Finally, we perform

**Table 2: An overview of the machine learning models in Apollo.**

| Component | Model Name | ML Framework | Model Type | No. model | Description |
|---|---|---|---|---|---|
| Traffic Light Perception | Traffic_Light _Detection | CaffeNet | CNN | 1 | Detect traffic lights from images captured by camera. |
| | Horizon_Light | CaffeNet | CNN | 1 | Recognize horizontal traffic light. |
| | Vertical_Light | CaffeNet | CNN | 1 | Recognize vertical traffic light. |
| | Quadrate_Light | CaffeNet | CNN | 1 | Recognize quadrate traffic light. |
| Lane Perception | Denseline | TensorRT | CNN | 1 | Detect lane line from images captured by camera using denseline. |
| | DarkSCNN | CaffeNet PaddleNet | SCNN | 3 | Detect lane line from images captured by camera using dark SCNN. |
| Obstacle Perception | Obstacle_Feature _Extraction | CaffeNet | YOLO3D | 1 | Extract feature for each detected object. |
| | Camera_Obstacle _Detection | PaddleNet TensorRT | YOLO3D | 4 | Detect obstacle from images captured by camera. |
| | LiDAR_Velodyne_16 | CaffeNet | CNN | 1 | Use 16 channels Velodyne LiDAR to perceive obstacles. |
| | LiDAR_Velodyne_64 | CaffeNet | CNN | 1 | Use 64 channels Velodyne LiDAR to perceive obstacles. |
| | LiDAR_Velodyne_128 | CaffeNet PaddleNet | CNN | 2 | Use 128 channels Velodyne LiDAR to perceive obstacles. |
| Trajectory Prediction | Vehicle_Cruise_Cutin | PyTorch | MLP and CNN-1d | 1 | Predict vehicle trajectories probability for cruise cutin scenarios. |
| | Vehicle_Cruise_Go | PyTorch | MLP and CNN-1d | 1 | Predict vehicle trajectories probability for cruise scenarios. |
| | Vehicle_Junction_Map | PyTorch | CNN | 1 | Predict vehicle trajectories probability using a semantic map-based CNN model for junction scenarios. |
| | Vehicle_Junction_MLP | PyTorch | CNN+MLP | 1 | Predict vehicle trajectories probability using an MLP model for junction scenarios. |
| | Vehicle_Lane_Scanning | PyTorch | CNN | 1 | Scan lane sequences. |
| | Vehicle_MLP | N/A | MLP | 1 | Predict vehicle trajectories probability using MLP model. |
| | Vehicle_RNN | N/A | RNN | 1 | Predict vehicle trajectories probability using RNN model. |
| | Vehicle_Lane_Aggregate | PyTorch | RNN, MLP and CNN | 3 | Aggregate lane and obstacle information to predict vehicle trajectories probability. |
| | Pedestrian_LSTM | PyTorch | LSTM | 4 | Predict pedestrian trajectories using LSTM model. It contains four steps: 1) Get position embedding; 2) Get social embedding; 3) Conduct single LSTM and update hidden states; 4) generate trajectory point. |

a manual inspection to categorize each ML model and based on its purpose (e.g., the functionality of a model) and location in the source code (e.g., component in the system). Note that some ML model files may be duplicated and outdated (e.g., indicated in the source code or Apollo documentation). Therefore, we filter out such ML models during our manual study.

**Results.** Table 2 summarizes the identified ML models in Apollo. Similar to the finding in a prior study [12], we find that typical ML models, especially deep neural networks (e.g., Convolutional Neural Networks and Recurrent Neural Networks) are widely used in Apollo. We manually uncover the interaction of the components in Apollo that contain ML models (Figure 1). In total, we found 28 ML models. The ML models can be categorized into four major categories: (1) traffic light perception, (2) lane perception, (3) obstacle detection, and (4) trajectory prediction. Interestingly, for each model category, Apollo often provides several ML models (i.e., see column *No. Model* in Table 2). For example, in the Camera Perception component, there are four ML models related to obstacle detection. Developers need to choose which one of the four ML models to use, in a configuration file, when detecting obstacles.

Next, we discuss how ML models are used in each category.

***Traffic Light Perception.*** **Traffic light perception relies only on cameras and contains four different ML models for different types of traffic light.** Traffic light perception module uses cameras to detect the presence of traffic lights in a region of interest (ROI) and recognizes the color of the detected traffic light as red, yellow, green, black, or unknown. Black means the light status is uncertain (e.g., when the light is blinking). There are three major parts in the traffic light detection module:

*Identifying initial traffic light location.* ADS receives information on the presence and the position of traffic light by querying the HD Map. If a traffic light exists at the current location, then images are taken by the camera and sent to the next step for location detection.

*Traffic light location detection.* ADS uses a Faster R-FCNN (Region-based Fully Convolutional Network) model to process the images that may have a traffic light (i.e., the model in the category *TrafficLight-Detection* in Table 2). The model outputs the identified locations of the traffic light using bounding boxes with a score (e.g., the likelihood that a bounding box identifies a traffic light).

*Traffic light recognition.* Given each bounding box obtained from the previous step, traffic light recognition models then output the probabilities of the light being red, yellow, green, and black. The class with the maximum probability is regarded as the light's status.

If no probability value is larger than a predefined threshold (default value is 0.5), the color will be recognized as black (i.e., unknown). Note that, based on the shape of the traffic light (i.e., horizontal, vertical, and quadrate), Apollo would use the corresponding ML model (as shown in Table 2) for traffic light recognition.

***Lane Perception.*** **Lane perception relies on cameras and developers can specify which ML model to use in a configuration file.** Apollo has two lane detection ML models: darkSCNN [26] and denseline. After receiving the preprocessed image taken by the camera, lane detection first reads a configuration file to determine which ML model to use as well as whether to deal with image distortion. Then, the loaded ML model detects the positions of the polynomial curve and the position of the lane line. darkSCNN supports vanishing points to better detect curved lines or lane lines blocked by obstacles [26].

***Obstacle Perception and Detection.*** **ADS may combine information from various sources (e.g., several LiDAR sensors and radar) to improve perception accuracy.** Obstacles (or objects) such as cars, trucks, bicycles, or pedestrians on the road can be recognized by using cameras or LiDARs. The model output includes a 3D position of the object (e.g., projected from the 2D input image, or obtained from the point cloud generated by LiDAR), the type of the object and a confidence score, the relative position, velocity and direction that the object is moving towards, and a series of historical trajectory points of the object in the past few seconds. Below, we discuss obstacle detection using cameras and LiDAR.

*Obstacle detection using cameras.* The camera detection algorithm is designed based on a multi-task YOLO 3D Neural Network model [30–32]. Given a 2D image as the input, the output layer of the YOLO model predicts various properties of the object: object center point, object width, object height, object confidence (i.e., how likely it is an object), class confidence (e.g., how likely the object is a vehicle), the projected 3D dimensions of the object, the object orientation, and the orientation confidence. With the above-mentioned information, Apollo can transform a 2D object on the image to bounding boxes on a 3D image.

*Obstacle detection using LiDAR.* Apollo LiDAR detection component utilizes the 16, 64, 128 channel (i.e., LiDARs with different number of laser beams) Velodyne LiDARs [17] to detect an object. 3D point cloud data obtained from LiDAR sensors will be refined by removing noises, such as buildings, using the HD Map. Only the data in the ROI will be fed to an UNet FCNN model [33]) for segmentation (e.g., get attributes for each point in the point cloud). There are separate ML models for each of the 16, 64, 128 channel LiDAR. For each point, the model would output attribute information such as the probability of being a valid object, class probabilities, confidence score, object height, object heading, object center offset, etc. Then, the points that do not belong to an object (e.g., background) will be filtered out. Finally, the points that belong to the same object will be clustered together based on the attribute information. For example, points that represent the same vehicle may be clustered together. Note that each cluster would have a confidence score (i.e., calculated based on the average confidence score of every point in the cluster) and the clusters that have a score below a threshold (i.e., defined in a configuration file) would be removed. The final detected objects would correspond to the final clusters resulted from the segmentation process.

As shown in Figure 1, Apollo uses algorithms, such as Kalman filter [24], to fuse the detection results from LiDAR, camera, and radar to increase object perception and detection accuracy. Moreover, the inputs to the LiDAR segmentation models (i.e., one ML model for each of the 16, 64, 128 channel LiDAR sensor) are also the fused results from multiple sensors. In short, instead of using one source of information or relying on one model result, ADS leverages multiple data sources and combines a fused prediction result to improve accuracy.

***Trajectory Prediction.*** **Trajectory prediction involves using various ML models for different scenarios.** The trajectory prediction component predicts the trajectories of obstacles/objects, such as vehicle and pedestrian, so that ADS can plan the route accordingly and avoid collision. The trajectory prediction component involves various ML models, and it chooses the most appropriate ML model based on the perception information. With the information from the above-mentioned object perception and detection components, and the HD map, the prediction component prioritizes the sensed obstacles/objects to *caution*, *normal*, or *ignorable*. Then, for each *unignorable* object, the prediction component chooses an ML model for prediction based on information such as the type of the object (e.g., vehicle, pedestrian, bicycle, or unknown) and the status of the object (e.g., on lane, off lane, in junction, or in cruise). Note that developers can change in a configuration file where an ML model should be used in different scenarios (more discussion on this in RQ2).

*Pedestrian prediction.* Pedestrian prediction takes the pedestrian historical positions and the social information (e.g., the number and position of the people near the pedestrian, which may have an effect on human-to-human interaction) as input to the ML model [1]. Apollo uses an attention-based Long-Short Term Memory (LSTM) network [1, 16] to predict the trajectory points of the pedestrian in the next few seconds.

*Vehicle prediction.* Vehicle prediction contains two scenarios, one is when a vehicle is moving on a lane (i.e., cruising), and the other one is when a vehicle is near a road junction (e.g., a traffic light or stop sign). For a vehicle moving on the lane, the ML models combine the vehicle's state, historical movement state, and lane line information to calculate a probability that the vehicle may move to a different lane. For a vehicle that is near a road junction, the ML model predicts the intention of the vehicle (e.g., go straight or turn right) by using information such as the direction and angle of the head of the vehicle. Some ML models are also trained using data from the HD map to make more accurate prediction near road junctions (e.g., using the model "Vehicle_Junction_Map"). After predicting the intention of the vehicle, other ML models will predict the future trajectories of the vehicle in the next few seconds.

**Discussion and Implication.**

*Future studies should provide support to ML model management and evolution.* As we found in Apollo, developers can choose which ML model to use for a given task in a configuration file. However, this may cause some potential model maintenance issues. As the number of ML models increases, some models may not be as up-to-date as others (e.g., had less training data or used a less powerful algorithm).

There may be life-critical consequences when developers choose an older ML model. Moreover, deciding which ML model to use can be a challenging decision. Future studies should provide support to developers on choosing the optimal ML model for a given task under different situations (e.g., different cities), and help developers maintain variants of the models.

*ADS relies on ML models that are trained using various deep learning and machine learning frameworks.* Prior studies [13, 28] found that different frameworks may result in a model with slightly different performance. As shown in Table 2, Apollo developers use ML models that are trained using various frameworks (i.e., CaffeNet, Paddle, PyTorch, and TensorRT). Therefore, the performance of some ML models may be inconsistent and suboptimal. There may also be maintenance overhead, as developers need to maintain code that interacts with the ML models that are trained using various frameworks. Future studies may help developers reduce maintenance overhead by providing a better abstraction of different ML frameworks.

*Some functionalities of ADS rely on HD map, which may be a single point of failure.* We found that Apollo relies on the HD map for various tasks, such as knowing the estimated location of a traffic light and the information on various traffic sign (e.g., speed). Some ML models (e.g., Vehicle_Junction_Map) also make prediction based on input information received by querying the HD map. As a result, an outdated HD map may be a single point of failure that may affect the subsequent decisions of the ML models in the system. Although ADS may use vehicle-to-everything (V2X) to exchange information between the vehicle and other entities that may affect the vehicle (e.g., traffic light) [9], such services may not be widely available yet. Future studies may consider testing the impact of HD map or V2X on the decision of ADS.

*We find that there are 28 ML models in Apollo, trained using different ML frameworks. The Platform provides flexibility, supporting configurable ML model selection for a particular task.*

## RQ2: What are the relationship and interaction between code modules and ML models?

**Motivation.** The uncovered model architecture of Apollo and the results of RQ1 reveal that there exist non-trivial interactions among the ML models. Such interactions are implemented with the help of code logic. The interactions may further complicate ML model testing and pose new challenges in the integration testing between the models and code modules. For example, the output from one model may be in diverse data formats (i.e., requiring post-processing steps), become part of the input of several ML models, and require proper validation before inputting to other ML model or code logic. Therefore, to provide guidance on future research on improving the testing of AI-powered system, in this RQ, we present our analysis results on the ML model interactions and how code plays a role in such interactions, e.g., pre-, post-processing and validation steps.

**Approach.** We uncover and present the ML model architecture in Apollo as described in RQ1. The ML model dependency architecture shows a high-level interaction, such as the outcome of the traffic light detection model is part of the input for traffic light recognition model. However, the details of such interactions are lacking in the high-level model architecture. For example, what types of data are being transmitted among the models, and what functionalities the code logic can provide in such transmissions. To obtain such details, we perform a comprehensive qualitative analysis of the Apollo code base. The steps in the qualitative analysis are as follows.

```
1  //line 2-4 are from a CyberRT configuration file
2  obstacle_conf {...
3  evaluator_type: CRUISE_MLP_EVALUATOR
4  ...}
5
6  case ObstacleConf::CRUISE_MLP_EVALUATOR: {
7      evaluator_ptr.reset(new CruiseMLPEvaluator());
8  }
9  ...
10 void CruiseMLPEvaluator::LoadModels (){
11     torch_go_model_ptr_ =
12         torch::jit::load(FLAGS_torch_vehicle_cruise_go_file, ...);
13 }
14 ...
15 //Using gflags to define a string for the model file path
16 DEFINE_string(torch_vehicle_cruise_go_file, <file_to_a_ML_model>)
```

**Figure 2: An example on how Apollo manages and loads different ML models in a flexible manner.** CRUISE_MLP_EVALUATOR **is configured in line 3, initiated in line 7 and loads a model file in line 12.**

*(1) Understanding how ML models are managed, used and interact in Apollo.* To provide flexibility and better management of model variability, Apollo developers utilize a combination of configuration management (i.e., CyberRT [5]), gflags [10], and Proto Buffers [11]. Apollo first loads the configuration files to identify which ML model module to use. An *ML model module* is a code component that abstracts the common behaviors of one ML model (e.g., load, infer, and reset). In one ML model module, the file path of the used ML model is further abstracted using *gflags* as a defined string to be substituted during compilation. For the interactions among ML models, Apollo uses CyberRT and Proto Buffers to communicate among ML model modules: One ML model module publishes a message (i.e., defined by a proto file) and the other ML model module receives the published message. Figure 2 shows a simplified example of how an ML model file is loaded and managed in Apollo. Note that the code snippets are from different configuration and C++ files. Line 3 specifies the model module to use in a CyberRT configuration file. Then, the model module (*CruiseMLPEvaluator*) is initiated based on the configuration file (see line 7). In line 11-12, the model file is loaded according to a pre-defined gflags string (line 16).

*(2) Finding the model modules in use.* We started the qualitative analysis by finding the ML model modules based on tracing the model file paths in Apollo (i.e., uncovered in RQ1). On one hand, we identified the representation of the absolute path of ML model files (e.g., *traced_online_lane_enc.pt* by PyTorch). This step is necessary as Apollo developers do not directly use the absolute paths of the ML model files, but use an internal representation abstracted by gflags. On the other hand, the ML model modules are flexibly loaded in Apollo components (e.g., perception and prediction) based on CyberRT configuration files. By combining the information from both the above-mentioned steps, we find the ML model modules that are used in Apollo and their corresponding ML model files.

*(3) Tracing the code that interacts with the ML models.* We started with the ML model modules that abstract the common behaviours of ML models. In each ML model module, we first searched for the model inference code. For example, the inference API for PaddlePaddle, TensorRT, or Caffe is called Infer(). Then, we identified the code that prepares the input of the ML models (i.e., a set of feature values) and produces the output of the ML models (i.e., prediction results). Finally, we traced the backward propagation of the input and the forward propagation of the output until the propagation hits the boundary of the ML model. For the input of an ML model, the boundary is set until the code reads input from the hardware (e.g., sensors) or from another ML model (i.e., receiving messages in CyberRT). For the output, the boundary is set until the output is being passed to other code modules or other ML model modules (i.e., publishing messages in CyberRT). In short, we include all the code logic that interacts with each of the ML models, i.e., pre- and post-processing, and validation, for further manual analysis.
*(4) Tracing the data transmission among the ML models.* Apollo leverages proto files to represent data transmitted among ML models. For example, the model "LiDAR obstacle detector" outputs the type of the detected obstacle (e.g., vehicle, pedestrian, bicycle, unknown movable, unknown unmovable, or unknown), and stores the result in a message defined in the proto file (i.e., "perception_obstacle.proto"). Subsequently, the ML models in the prediction component can use the stored obstacle type as input. We manually examined the relevant proto files and recorded how these proto files are used by different model modules for communicating among various ML models.

Upon identifying the relevant artifacts (i.e., code, model modules, and proto files) by following the aforementioned steps, we performed a manual analysis on the 28 ML models in Apollo to reveal how ML models interact with each other and the key roles played by the code logic in such interaction.

**Result.** Based on the *28* models in Apollo, we summarize common ways that the ML models may interact with each other. Furthermore, by analyzing the code logic implemented for ML model interactions, we uncover the roles of such code modules in the integration. We summarize the common ML model interactions in Apollo below.
*(1)* All the output of one ML model is used *exclusively* and *entirely* by another ML model as input. For example, the output of traffic light detection model, i.e., bounding boxes and scores (the likelihood that a bounding box identifies a traffic light), is the input to one of the three ML models in traffic light recognition depending on the traffic light type.
*(2)* The output of one ML model is used to post-process the output of another ML model. For example, the lane detection models (i.e., "Lane_Detection" in Figure 1) identify lane locations and further use the information to calibrate the camera framework. The calibrated camera framework is leveraged to post-process the obstacle detection results (i.e., "Camera_Obstacle_Detection" in Figure 1).
*(3)* The outputs of several ML models are combined together as another ML model's input. For example, prediction uses an LSTM encoder to process the vehicle movement history which is detected and recorded by the obstacle perception component. The prediction also uses either LSTM or one dimensional CNN encoder to encode lane line learned from the lane perception component. The two encoders are used together in predicting the trajectory of a vehicle driving on a lane.

Given the diverse ML model interactions, code modules play non-trivial roles in such interactions. Upon identifying the code logic in ML model interactions, we categorized their roles in pre-, post-processing, and validating the input and output of ML models.
***Filtering out invalid output based on common sense.*** ML models may produce output that does not exist in reality. For such cases, developers employ simple heuristics in the code logic to only keep valid outputs. For example, in traffic light detection, the ML model outputs the boundaries of the detected traffic lights (i.e., bounding boxes), each of which is defined by four coordinates. However, some detected areas may not exist in reality (i.e., the areas of the bounding boxes are negative). Developers implement the logic in a method named *SelectOutputBoxes()* to remove the invalid detected results and only keep the valid ones.
***Selecting ML model output using hard-coded or user-specified parameters.*** To improve the accuracy of the ML model results, Apollo developers set criteria to only select certain ML model results that satisfy the defined criteria. Compared to common sense related filtering, such selection requires domain knowledge and is more related to specific user needs. Hence, some of the criteria are configurable by users. Table 3 shows a list of criteria and thresholds that Apollo developers use for different ML models. Most of the parameters are related to the confidence threshold of ML model results. There are a few exceptions based on the specific application scenarios. For example, in traffic light detection, whether bounding boxes have significant overlaps (i.e., the default value is 60%) is used to delete certain bounding boxes (e.g., they are likely to be the same traffic light). In most cases, the parameters can be configured by users, as indicated by the last column in Table 3.
***Using code logic to complement the imperfect outcome from ML models.*** The data-driven and probabilistic nature of ML models makes it unable to reach 100% general accuracy in practice. Therefore, in addition to heuristic-based filtering, the inaccurate results from ML models may be further refined by their downstream computation modules. Apollo mainly uses a classic detection-to-track approach based on Hungarian algorithm [22, 23] and Kalman filter [19] to track multiple objects. We observe that in Apollo, it is common to utilize the tracking modules (non-machine learning) to refine the outcome of the detection modules (ML models). In short, one detection module (e.g., traffic light detection) forwards the detection results to a corresponding tracking module (i.e., traffic light tracker). Then the trackers (i.e., traffic light tracker) are used to increase detection stability by remedying incorrect or missing detections, e.g., detection may fail when an object is blocked by other obstacles, and softening the jitters of detection bounding boxes. Specifically, the traffic light tracker can revise the detection results based on time-sensitive constraints. An example of such a constraint is that *"a yellow traffic light will not turn green in the next moment"*. For example, if the traffic light detection fails to detect the color at a specific timestamp, it will output either `BLACK` or `UNKNOWN_COLOR`. However, if the traffic light was detected in a stable color at a timestamp in close proximity, the detected color will be revised based on the established constraints according to the previously-detected color.

**Table 3: Hard-coded or user-specified parameters to validate and filter out output of machine-learning models in Apollo.**

| Components | Parameter Name | Default | Usage | Type |
|---|---|---|---|---|
| Traffic Light Perception | iou_thresh | 0.6 | The overlap threshold for traffic light detection bounding boxes. Bounding box has large overlap(>=0.6) with others will be dropped. | hard coded |
| Traffic Light Perception | classify_threshold | 0.5 | The decision threshold for traffic light classification. | configurable |
| Lane Detection | confidence_threshold | 0.95 | The confidence threshold for SCNN detection. | configurable |
| Obstacle Detection (Camera) | confidence_threshold | 0.4 | The confidence threshold for YOLO detection. | configurable |
| | threshold | 0.5 | Non Maximum Suppression (NMS) threshold to filter overlapped detection result. | configurable |
| | inter_cls_nms_thresh | 0.6 | nms threshold to filter overlapped result in gpu. | configurable |
| Obstacle Perception (LiDAR) | objectness_thresh | 0.5 | The objectness threshold to filter non-object in clustering. | configurable |
| | confidence_thresh | 0.1 | The detection confidence score threshold for filtering out the candidate clusters in the post-processing step. | configurable |
| | min_pts_num | 3 | In the clustering step, clusters that contains a few points (<3) are removed. | configurable |

Note that such refinement is not limited to traffic light detection. All the detection-to-track components in Apollo utilize feedback-based refinement. How exactly each refinement is conducted is also different. To give another example, the obstacle tracker uses Hungarian algorithm to associate the detected obstacles to the existing track list, then use Robust Kalman Filter to estimate and correct inaccurate motion states in the track list.

***Scenario-based ML model selection and input generation.*** As shown in Table 2, there may be separate ML models for different scenarios of the same task. For example, the trajectory prediction components contain ten evaluators in code logic to deal with different scenarios, and each scenario has a corresponding ML model. As one can see from the example in Figure 2 (i.e., Apollo uses a flexible way to manage models), one evaluator may load more than one ML models (e.g., CruiseMLP loads Cruise_Go_Model and Cruise_Cutin_Model). The code logic in the evaluator then automatically decides which ML model to use based on some dynamic variables that present the current states (e.g., whether the vehicle is on a lane). Interestingly, we also find that the inputs of the ML models may be different in different evaluators. Therefore, developers need to apply code logic to extract different features to feed into the corresponding ML model.

***Applying fusion to combine information from various sensors to improve perception accuracy.*** As described in Section 2 and RQ1, Apollo supports obstacle detection using cameras, LiDARs, and radars. However, none of the detection results is perfect due to the limitation of the sensor (e.g., the camera is sensitive to light change, and LiDAR will contain many noises for a complicated environment) and ML model inaccuracy. Hence, ADS needs to combine multiple sensor detection results to minimize the error and get a better perception result. Apollo fusion takes camera tracking result, LiDAR tracking result, and radar detection result as input. Users can define in the configuration which sensor results can be fused. Developers rely main on algorithms such as Hungarian Optimizer and Kalman filter to fuse the output. After applying fusion, objects are published as an obstacle perception message (e.g., PerceptionObstacles).

**Discussion and Implication.**

*Future studies should consider testing the use of ML models for different scenarios of the same task.* Prior ML model testing research [27, 38] often only focuses on the behaviour of a single model for one specific task (e.g., traffic light recognition). However, as we found, in ADS, there may be several ML models, and each one is tailored towards one variant of the same problem. For example, different types of traffic light use different ML models. In the case of trajectory prediction, ADS may choose the ML model depending on the state of the vehicle (e.g., at interaction or cruising). Future research should consider such design when testing ML models in different scenarios.

*Future studies should consider the complex ML model interactions when testing ML models in large systems.* Most of the current research efforts to improve ML model testing focuses on testing a standalone ML model (at unit level) without considering its interactions with other ML models. Our study shows that a large-scale ML-powered system commonly employs a diverse set of ML models and leverages the ML model interactions to implement complex functionalities. For example, Apollo utilizes multiple ML models using data from different sensors, and improves the resilience of model result in some critical decision-making processes. Considering such interactions is a pursuable direction to make testing ML model more practical in real-world systems.

*Future studies should investigate the maintenance challenges of the code logic in ML model interactions.* As we found, code logic plays important roles in ML model interactions and has close connections with the relevant ML models. The evolution of ML models is more frequent than expected [34]. The needs of exchangeable ML models for different scenarios may cause ML models to update even more frequently. As ML models evolve, it poses maintenance challenges on the code logic that should co-evolve. Failing to update the code logic accordingly as ML models evolve may introduce bugs and fail to achieve the optimal performance of ML models, and further negatively impact the entire ML-powered system.

*Future studies should consider testing the integrated behaviours of ML model and code logic.* Our study shows that certain code logic and ML models are closely dependent on each other. In some cases, the integrated behaviours of one ML model and the corresponding code logic may have a significant impact on the performance of another ML model. ML models and source may need to co-evolve. Future studies should provide approaches to better test the integrated behaviours of ML models and source code to detect regressions.

*Developers use code logic to pre- and post-process ML model input and result to improve prediction accuracy. ADS may choose which ML model to use based on different scenarios and relies on different algorithms to combine results from different ML models. Developers also provide configuration options to tune the thresholds used in ML prediction.*

## 4 THE DYNAMIC ASPECT OF APOLLO

In previous sections, we present and discuss the static view of the ML models in Apollo through a qualitative study: the model architecture and information flow, and how the code logic and ML models interact and are integrated into Apollo. In this section, we present some quantitative results from a dynamic view. We exercise and perform runtime analysis of Apollo in two settings: a production run in a simulator and in-house testing. We then study two aspects related to the deployed ML models and their relevant code logic. In particular, we first study how frequently ML models are executed in a simulated production run. Then, we inspect the current testing effort on the model relevant code logic, i.e., the code coverages of Apollo CI tests on the model-relevant code modules. We make a special focus of our study on two model-relevant components (i.e., perception and prediction as indicated in Table 2). In the rest of this section, we first describe how we exercise Apollo in the two settings. We then present the results and discuss the implications.

**Running Apollo in a simulator.** We exercised Apollo in a simulated environment using a recorded playback from previous production runs. Apollo has been deployed on roads to conduct tests in the field in major cities in China. Data is recorded from such production runs for later playback. The recorded data includes everything the deployed self-driving car would receive from the hardware (e.g., cameras, LiDAR, and Radar). A message is used for internal data transmission managed by CyberRT. For example, the LiDAR module may publish one message when it receives new data. The playback we used lasts 9.983 seconds and contains 4,418 CyberRT messages.

To assess how frequently ML models are used in the simulated production run, we inserted logging statements in each method in Apollo and analyzed the generated logs by executing Apollo in the simulator using the recorded playback. In total, we inserted logs in 1,994 methods from the perception and prediction (1,488 methods and 506 methods, respectively) components and 6,598 methods from other components. We analyzed the logs to count the unique and repeated hits of the logged methods in Apollo.

**Measuring code coverage of Apollo CI tests.** Apollo developers deploy test execution in continuous integration (CI) and then every build is exercised against the CI tests. Overall, Apollo has a total of 621 CI test files. Perception has the most test files among all modules (159 tests). Prediction is the other module that uses ML models, and contains 44 test files. We use LCOV [25] to measure the common coverage metrics of the Apollo CI tests, i.e., line, method, and branch coverage.

**Results and Implications.**

*ML model usage in a dynamic view.* We find that the methods in model-relevant components (i.e., perception and prediction) are executed *heavily* and *repeatedly* in the simulated run: 709/1,944 (35.5%) methods are executed at least once with an average of 1,965 repeated executions per method. On the other hand, only 3.5% of the methods in other components are executed at least once. Note that not each method will be exercised in a simulated run of one playback, since we only execute Apollo once with the default configuration.

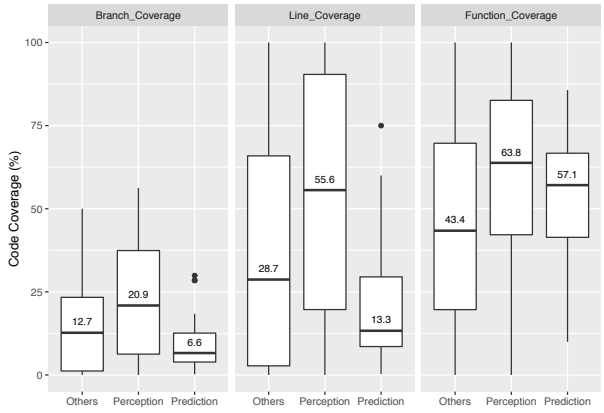


**Figure 3: Box plots of code coverages of each source code file in Apollo: perception, prediction and the other components.**

*Code coverage of code components in Apollo.* We find that, the overall code coverage achieved by Apollo CI Tests is not high: 13.30% for branch coverage, 21.50% for line coverage, and 33.90% for function coverage. It is reasonable since Apollos is still under active development, and that much of testing effort is paid to train/test ML models. However, given the importance of code logic in forming such a complex system, as our study reveals, improving code coverage is also important to guarantee the quality assurance of life-critical systems such as ADS. Furthermore, we show the breakdown (Figure 3) of the coverage metrics per source-code file in different components: perception, prediction and other code components. Perception and prediction are the only two components that use ML models and contain most of the ML model relevant code logic. Interestingly, we find that the perception component achieves considerably higher code coverages than other components, including the prediction component. Our communication with Apollo developers confirms that extra test effort is allocated to test the complexity in the perception component due to its importance as one of the upstream components in the information flow in Apollo. *However, the code coverage is still low. More testing effort is needed to improve the CI test coverage in Apollo. Nevertheless, it may be very challenging due to the complex nature of an ADS system and unprecedented coupling among ML model and code logic.*

## 5 THREATS TO VALIDITY

**External validity.** In this study, we conduct a case study on a representative complex ML-powered system, i.e., one of the state-of-the-art industrial autonomous driving systems (ADS), Apollo. Currently, Apollo has adopted by major automakers, some of which even start the deployment in some cities to offer autonomous taxi services [4]. However, our results may still not generalize to other ADS or ML systems. Our findings provide an initial overview on the

architecture of ML systems and how ML models are integrated with code logic. Since ML systems are becoming more critical in modern society and little is known how these systems are developed, future studies are needed to investigate the design and challenges of other ML systems.

**Internal validity.** Much of our analysis relies heavily on manual inspection of the Apollo source code and documentation. The first two authors of the paper thoroughly studied the architecture of Apollo and its design. Although many of our results are confirmed by Apollo developers, there may still be biases in our results. To mitigate the issue, the third and fourth authors also verify the results of the manual inspection.

## 6 RELATED WORK

In this section, we discuss the most relevant literature regarding machine learning systems in the context of autonomous driving vehicles (ADV), and empirical studies on machine learning engineering practice.

**Testing of ML Models for Autonomous Driving.** Quality issues of machine learning raise significant attention recently, and we have witnessed an increasing trend of research on testing ML models, especially on deep neural networks (DNNs) in the context of autonomous driving. DeepXplore [27] proposes a differential testing method to detect the DNNs behavior inconsistencies, which are used for ADV steering control. DeepTest [38] proposes to generate tests by basic image transformations, which simulates the real-world camera noises, to test the DNN models. DeepRoad [43] performs more advanced driving scene transformation by GAN to test the DNNs when environment condition changes occur. DeepBilllboard [46] performs testing through attacking billboard images to mislead ADV DNN controls. Similar image-based testing, Cao et al. [8] shows that LiDAR component could also be tested by performing well-designed attack techniques. Haq et al. [14] recently performs a comprehensive study to compare the offline and online testing of DNN in the context of ADV, where they found that offline testing is often more optimistic than online testing and simulation-based methods can be useful in many cases. For more comprehensive on the state-of-the-art progress on ML testing, we refer interesting readers to the recent survey [42].

Different from existing work on testing ML models, our work intends to investigate the practical ML system architecture, especially on what and how ML models are used in a complex ML system, and what is the relation and interaction of ML models with a traditional software component, together forming the whole complex ML systems like Apollo. Our results show that ML models can be used for various purposes and integrated into different parts of an ML system. Obviously, testing each ML model at the unit level is necessary, but still not enough. As a next important step, testing and more general quality assurance should also be extensively performed at the integration level and system level.

**Empirical Study on ML Engineering.** The data-driven development of ML brings new challenges to the well established traditional software system development process. Recently, we also witnessed quite a few empirical studies on ML system engineering practices, which would an important step to build high quality ML at system level. Amershi et al. [2] performs a case study of ML system development process at Microsoft, to investigate the current practices and challenges. Zhang et al. [44] investigate the common challenges in developing ML applications, by studying the posts and answers on StackOverflow and Github. As the important foundation of ML system, the ML frameworks are also studied by previous work [18, 18, 35, 37, 45], which shows many state-of-the-art ML frameworks (e.g., Tensorflow, Lucene, Mahout, Scikit-learn, Paddle, and Caffe) can contain bugs. Furthermore, Guo et al. [13] find that compatibility issue also exists among current ML frameworks, where a model can downgrade its performance when being migrated to a different ML framework or platform. This posts great concern and calls for attention that the dependency libraries of an ML system could also be an important factor that impacts its quality. Some recent work [28] starts to effectively detect the potential quality issues in the framework. When it comes to ADS, Joshua et al. [18] perform an early step study to investigate the common bug symptoms and causes. The results show that, as a complex system, the bugs in ADS can scatter across the system, and many of the bugs are still traditional software bugs.

Different from existing work, our work focuses on studying the typical architecture of a complex ML system (i.e., Apollo). We made an in-depth analysis on the roles of ML models, and how they integrate into a larger system. This provides an important step towards understanding and providing quality assurance for a complex ML system. Our study confirms the necessity of providing quality assurance for a practical ML system at multiple levels, i.e., model (unit) level, integration level and system level. New methodology, toolchains, and benchmarks would be needed, that handles both ML model and traditional software components.

## 7 CONCLUSION

In this paper, we present a case study on a representative and complex ML system, i.e., autonomous driving system (Apollo). We focus on unveiling its complex nature, i.e., various ML models and their relationship with non-trivial code logic. We present the model architecture in Apollo, and find that ML model interactions happen frequently in diverse ways. Moreover, we find that non-trivial code logic plays various roles in such interactions. Our further in-depth inspection of Apollo in a dynamic view confirms a heavy use of ML model relevant code components and reveals a lack of adequate test at integration level in a diverse way. Our findings indicate important maintenance challenges of complex ML-powered systems and call for collaborative effort to improve the quality assurance of ADS and general complex ML systems at the system level.

## ACKNOWLEDGMENTS

We thank Baidu Apollo Platform developers for their comprehensive discussion and feedback during our study, setting up the first step for systematic quality assurance of complex ML systems. Lei Ma is supported by JSPS KAKENHI Grant No. 20H04168, 19K24348, 19H04086, and JST-Mirai Program Grant No. JPMJMI18BB, Japan.

## REFERENCES

[1] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Fei-Fei Li, and Silvio Savarese. 2016. Social LSTM: Human Trajectory Prediction in Crowded Spaces. In *2016 IEEE Conference on Computer Vision and Pattern*

*Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 961–971. https://doi.org/10.1109/CVPR.2016.110

[2] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP âĂŹ19)*. IEEE Press, 291âĂŞ300. https://doi.org/10.1109/ICSE-SEIP.2019.00042

[3] Baidu. [n. d.]. Apollo. https://apollo.auto/. ([n. d.]). Last checked May 2020.

[4] Baidu. [n. d.]. Chinese internet giant Baidu offers free trial robotaxi rides through search and map apps in Changsha. https://www.scmp.com/tech/apps-social/article/3080712/chinese-internet-giant-baidu-offers-free-trial-robotaxi-rides. ([n. d.]). Last checked May 2020.

[5] Baidu. [n. d.]. CyberRT. https://cyber-rt.readthedocs.io/en/latest. ([n. d.]). Last checked May 2020.

[6] BAIR. [n. d.]. Caffe. https://caffe.berkeleyvision.org/. ([n. d.]). Last checked May 2020.

[7] BBC. 2020. *AI 'outperforms' doctors diagnosing breast cancer*. https://www.bbc.com/news/health-50857759

[8] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z. Morley Mao. 2019. Adversarial Sensor Attack on LiDAR-Based Perception in Autonomous Driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS âĂŹ19)*. 2267âĂŞ2281.

[9] S. Chen, J. Hu, Y. Shi, Y. Peng, J. Fang, R. Zhao, and L. Zhao. 2017. Vehicle-to-Everything (v2x) Services Supported by LTE-Based Systems and 5G. *IEEE Communications Standards Magazine* 1, 2 (2017), 70–76.

[10] gflags. [n. d.]. gflags. https://github.com/gflags/gflags. ([n. d.]). Last checked May 2020.

[11] Google. [n. d.]. Protocol Buffers. https://developers.google.com/protocol-buffers. ([n. d.]). Last checked May 2020.

[12] Sorin Mihai Grigorescu, Bogdan Trasnea, Tiberiu T. Cocias, and Gigel Macesanu. 2019. A Survey of Deep Learning Techniques for Autonomous Driving. *CoRR* abs/1910.07738 (2019). arXiv:1910.07738 http://arxiv.org/abs/1910.07738

[13] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An Empirical Study towards Characterizing Deep Learning Development and Deployment across Different Frameworks and Platforms. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. 810–822.

[14] F. U. Haq, D. Shin, S. Nejati, and L. C. Briand. 2020. Comparing Offline and Online Testing of Deep Neural Networks: An Autonomous Car Case Study. In *Proceedings of the 13th International Conference on Software Testing, Validation and Verification (ICST '20)*. 85–95.

[15] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[17] https://velodynelidar.com/. [n. d.]. https://velodynelidar.com/. https://velodynelidar.com/. ([n. d.]). Last checked May 2020.

[18] Garcia Joshua, Feng Yang, Shen Junjie, Almanee Sumaya, Xia Yuan, and Chen Qi-Alfred. 2020. A Comprehensive Study of Autonomous Vehicle Bugs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*.

[19] Rudolph Emil Kalman. 1960. A new approach to linear filtering and prediction problems. (1960).

[20] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 1039–1049.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90.

[22] Harold W Kuhn. 1956. Variants of the Hungarian method for assignment problems. *Naval Research Logistics Quarterly* 3, 4 (1956), 253–258.

[23] Harold W. Kuhn. 2010. The Hungarian Method for the Assignment Problem. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey (Eds.). Springer, 29–47. https://doi.org/10.1007/978-3-540-68279-0_2

[24] Geesara Kulathunga, Aleksandr Buyval, and Aleksandr Klimchik. 2019. Multi-Camera Fusion in Apollo Software Distribution. *IFAC-PapersOnLine* 52, 8 (2019), 49–54.

[25] lcov. [n. d.]. lcov. http://ltp.sourceforge.net/coverage/lcov.php. ([n. d.]). Last checked May 2020.

[26] Xingang Pan, Jianping Shi, Ping Luo, Xiaogang Wang, and Xiaoou Tang. 2018. Spatial as Deep: Spatial CNN for Traffic Scene Understanding. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI '18)*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). 7276–7283.

[27] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 1–18.

[28] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038.

[29] PyTorch. [n. d.]. PyTorch. https://pytorch.org/. ([n. d.]). Last checked May 2020.

[30] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*. 779–788.

[31] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR '17)*. IEEE Computer Society, 6517–6525.

[32] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *CoRR* abs/1804.02767 (2018). arXiv:1804.02767 http://arxiv.org/abs/1804.02767

[33] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Proceedings of the 18th International Conference on Medical Image Computing and Computer-Assisted Intervention (Lecture Notes in Computer Science)*, Nassir Navab, Joachim Hornegger, William M. Wells III, and Alejandro F. Frangi (Eds.), Vol. 9351. Springer, 234–241.

[34] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS '15)*. 2503–2511.

[35] 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 ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM '08)*. ACM, 149–157.

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS '14)*. 3104–3112.

[37] F. Thung, S. Wang, D. Lo, and L. Jiang. 2012. An Empirical Study of Bugs in Machine Learning Systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 271–280.

[38] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 303–314.

[39] Guowei Wan, Xiaolong Yang, Renlan Cai, Hao Li, Yao Zhou, Hao Wang, and Shiyu Song. 2018. Robust and Precise Vehicle Localization Based on Multi-Sensor Fusion in Diverse City Scenes. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '18)*. 4670–4677.

[40] Wired. [n. d.]. TTesla's Autopilot Was Involved in Another Deadly Car Crash. https://www.wired.com/story/tesla-autopilot-self-driving-crash-california/. ([n. d.]). Last checked May 2020.

[41] Oliver Wyman. [n. d.]. THE TRUE VALUE OF AUTONOMOUS DRIVING. https://www.oliverwyman.com/our-expertise/insights/2015/jul/automotive-manager-2015/customer/the-true-value-of-autonomous-driving.html. ([n. d.]). Last checked May 2020.

[42] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* (2020), 1–1.

[43] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). 132–142.

[44] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *Proceedings of the 30th International Symposium on Software Reliability Engineering (ISSRE '19)*. 104–115.

[45] 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)*. 129–140.

[46] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*.