openpilot System Architecture Analysis
by: Team Horizon

Authors

Ali Sheikhi	alish@my.yorku.ca
Erika Grandy	egrandy@my.yorku.ca
Jaskirat Pabla	pabla1@my.yorku.ca
Mate Korognai	korognaimat@gmail.com
Nabi Khalid	khalid18@my.yorku.ca
Natalie Dean	natrdean@my.yorku.ca
Shaharyar Choudhry	Sharry09@my.yorku.ca
Suha Siddiqui	suhasid@my.yorku.ca

Abstract

This report delves into the software architecture of openpilot, an open-source third-party advanced driver assistance system (ADAS) known for its widespread compatibility with over 250 car models capable of using the Comma 3/3x module. openpilot supports features such as Automated Lane Centering, Adaptive Cruise Control, Lane Change Assist, and Driver Monitoring. This document focuses on unravelling openpilot's architectural composition by looking through verifiable documentation and project structure. The order of information present is presented as follows: openpilot components, architectural styles, concurrencies present, system evolution, use cases, and lessons learned throughout the making of this document.

This document discusses various architectural styles that could be present in openpilot and discusses why each one was used. To do this, the system has been split into these subsystems: sensors, actuators, neural network routes, localization, calibration, controls, logging, miscellaneous services and hardware. From these subsystems, the extracted architectural styles discussed are Layered, Implicit Invocation - Event Based, Implicit Invocation - Publish & Subscribe, Process Control - Closed Loop Feedback, Process Control - MAPE-K, Client-Server, Pipe and Filter and Repository. The uses of these are described using diagrams extracted and remade from online documentation of openpilot accompanied by newly made diagrams that propose how the system interacts between different modules.

After looking at the architecture the report looks at how concurrency is present among the interacting subsystems. The concurrency allows for parallel execution of reading information, processing information and responding to information which ensures that openpilot can complete its functionality in real-time. This is then followed by a segment on how the system has evolved thanks to its open-source nature as well as the contribution of the team members over the years. Thanks to being open-source it enables continuous improvements through community contributions. Additionally, its large scale and wide breadth offer a diverse team of Full Stack Developers, Car Interface Engineers, Machine Learning Specialists and such working together to share their knowledge and expertise.

Lastly, use cases and lessons learned will be discussed. The use cases present possible interaction modes between the user, the openpilot system and the car to accurately capture what the user needs from the system. They highlight the interfaces that are present to the user and showcase how the system responds to these interfaces. Lessons learned will talk about what the team would do differently given the chance to do the project again.

Introduction and Overview

The purpose of the report is to outline the conceptual architecture of the openpilot software, particularly on version 0.9.5. The report will convey the functionality, components, and interactions between the components of the software. The report will also discuss the concurrency and evolution of the system. Consequently, the report will illustrate sequence diagrams, architectural composition diagrams, and use cases demonstrating the architectural design, communication and flow of data amongst the components in the system. The document also comments on the division of responsibilities amongst the participating developers as well as the lessons learned.

What does the system do (its functionality)?

The features of openpilot software include Adaptive Cruise Control (ACC), Automated Lane Centering (ALC), Forward Collision Warning (FCW), Lane Departure Warning (LDW), and camera-based Driver Monitoring (DM). The system functionally assists drivers with maintaining a safe distance and speed limit, predicting the possibility of collision, if the drivers are leaving the lane, and alerting distracted or sleeping drivers. The renowned applications of openpilot include a diverse range of models for Honda, Toyota, Hyundai, Nissan, Kia, Chrysler, Lexus, Acura, Audi, VW, Ford, and many more [6][23][22].

How is it broken into interacting parts? What are the parts? How do they interact?

The report is a breakdown of openpilot's software architecture styles used to implement the software. The report will discuss the high abstractions of the software speaking on its sensor-actuator mechanism and its decision-making model. The system at its highest level shows that the software takes input from the physical environment and utilizes that information to make decisions. The output of such software can range from many of the implemented features of the openpilot software. The report also discusses other styles used like layered architecture, event-based implicit invocation, publish-subscribe implicit invocation, process-control loop feedback, process control MAPE-K, Client-Server Pipe and Filter and Repository. Other design components include the usage of open-source libraries such as panda, opendbc, boardd, and cereal. The components detailed in this report interact with each other using sensors installed in car models such as cameras, steering wheel sensors, etc., to make artificial-intelligence-based decisions. The input is directly fed into an artificial neural network called "comma.ai" that will be able to implement safety features that openpilot ensures which is discussed in the next section. Moreover, the following are all components involved in the function of openpilot: sensors, actuators, neural network routes, localization, calibration, controls, logging, miscellaneous services and hardware. An important part to take note of is the hardware on which the openpilot can run the software: comma two (NEOS), comma three, and Linux PC. Comma two is Android based and comma three is Ubuntu-based operating systems [21]. This is only valid in older versions, a newer version (the version being inspected in this document) of openpilot highlights that the support only extends to Comma 3/3x on the Github repository README.md.

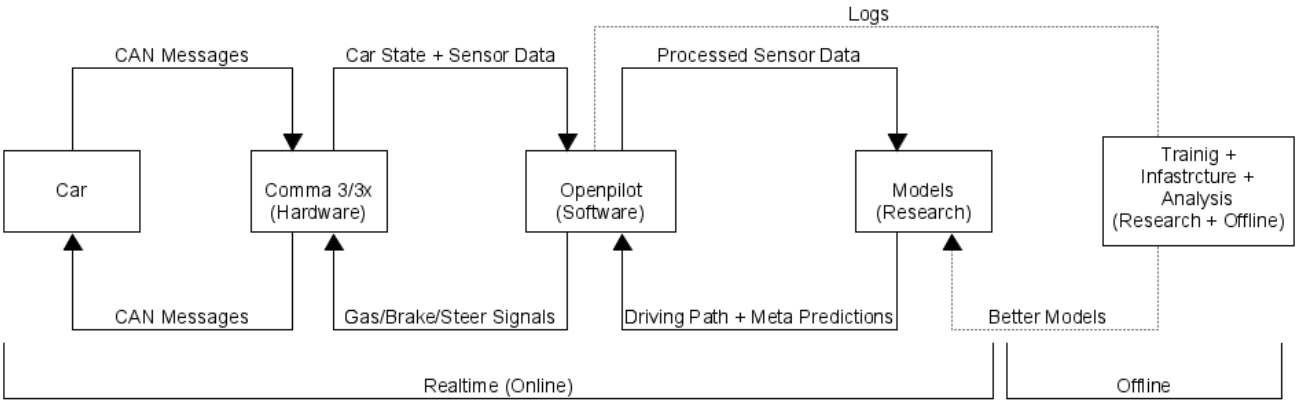

Figure 1: Interaction of technical components and breakdown of parts comprising openpilot 0.9.5 [21]

Architecture

This section looks at all the architectural styles present in the subsystem and then tries to bring it together as a whole at the end. It begins by looking at a higher level of abstraction, then breaking it down from there and inspecting each component one by one, highlighting possible architectural styles that are used in openpilot from online documentation and project structure.

High Abstraction - Layered Architecture

At its highest level, the system can be broken down into three main interacting parts: sensors, actuators, decision-making model and the actuators. They communicate through channels that interpret data and translate it into a readable form for each component. At its highest level, the system works as follows. After gathering sensor data, they transmit this information to the decision-making component in a comprehensible format. The decision maker uses the provided information and decides on a course of action that the actuators should take. It sends this course of action in an understandable format and the actuators perform the given task. The structural layout of these components is using a layered style architecture. As proof of this, consider that the sensors and actuators never communicated directly but only through the decision-maker. They may use the same channel to communicate and translate data into a readable format for the decision-maker, however, they do not communicate directly. Thus, at its highest level, the following system diagram can be a valid structural representation.

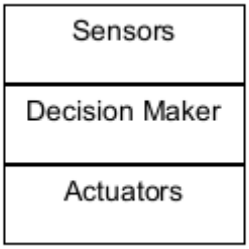

Figure 2: High-level System Diagram of openpilot

Layered Architecture

The layered architecture style refers to architecture that uses separate ordered layers. It is a common architecture used within software that has a high-level layer, such as an application UI, as well as lower layers, such as hardware components.

openpilot is an excellent example of layered architecture, due to its use of hardware components. The Comma 3/3X, the required hardware component that runs openpilot, provides a 2160x1080 OLED display, allowing for user interaction. It includes features such as GPS with WiFi or LTE connection. It also has a variety of cameras, including three 1080p cameras, a dual-cam 360° vision camera, and a narrow camera [20]. One of the 1080p cameras is faced towards the inside of the car, responsible for ensuring the driver remains attentive.

On the other end, this device is attached to the vehicle's OBD-II port. This allows for access to the vehicle's built-in features and data, such as the built-in cruise control or electronic steering, braking, and accelerating [19]. Above the hardware/physical layer, openpilot makes use of a Hardware Abstraction layer, carState, which allows for enabling high quality ports to a variety of hardware devices, such as new vehicles [21]. Above these hardware layers sit drivers, and then the application code, including the logic and controllers. The top layer is the application itself, such as the UI of the Comma 3X. Figure 3 highlights how these components interact.

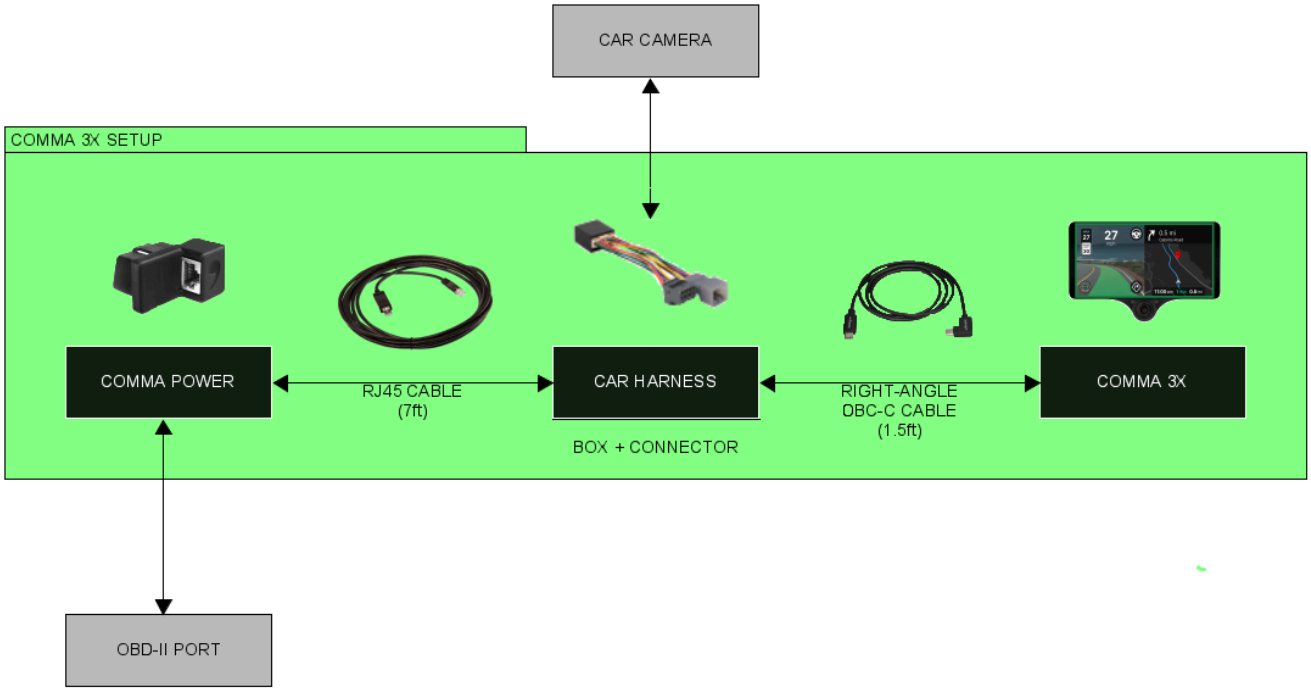

Figure 3: *Comma 3X Installation Diagram* [18].

Implicit Invocation: Event-Based and Publisher Subscriber
External Interfaces/Communication Between Layers

The communication between the Sensor, Decision Maker and Actuator components uses a combination of the following open-source projects: panda, opendbc, boardd, and cereal. Below is a diagram of how they communicate, with each section being described further below.

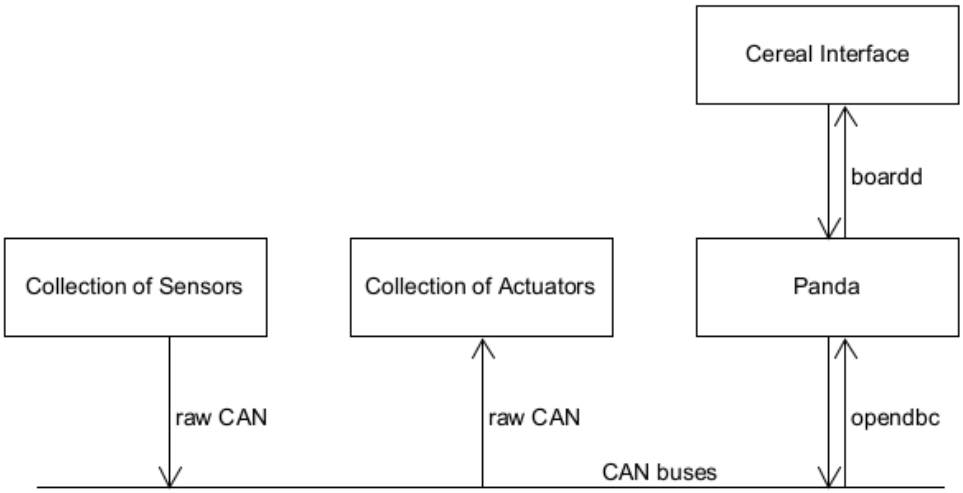

Figure 4: Communication Between openpilot and the Car

panda

Panda is a crucial component of openpilot as it is the main facilitator of data reading and writing to the sensors and actuators. It accomplishes this by interpreting and transmitting Controller Area Network (CAN) and CAN Flexible Data (CAN FD) messages through dbc files. Panda's way of communication is through the use of buses, and thus the main architectural style used for Panda is Implicit Invocation: Event-Based [17]. The basis of the architecture for Panda has been taken from *Distributed and Modular CAN-Based Architecture for Hardware Control and Sensor Data Integration* [16] and *How openpilot works in 2021* [21]. Figure 4 describes this event-based architectural style, as all components are connected to CAN busses which facilitate the communication.

opendbc

OpenDBC stands for the open database for CAN (Controller Area Network) messages. It is a crucial component of the openpilot system, serving as the backbone for understanding and interpreting the CAN bus traffic of a vehicle. The CAN bus is a robust vehicle bus standard designed to allow microcontrollers and devices to communicate with each other's applications without a host computer. OpenDBC provides a human-readable format for encoding information necessary to decode the messages passed along a vehicle's CAN bus. Given that a vehicle might have multiple CAN buses, each is represented by its own DBC (Database CAN) file. These DBC files are essential for the openpilot system to interpret and interact with the vehicle's operational parameters correctly.

OpenDBC interacts with hardware devices like Panda to connect the vehicle's CAN network to a computer, enabling the capture and analysis of CAN messages. The Cabana tool is then used to visualize this data and assist in the reverse-engineering process, allowing for the modification and creation of DBC files. These DBC files are then utilized by openpilot to interpret the vehicle's CAN messages correctly, enabling advanced driver assistance functionalities. Figure 4 displays the proposed structural position of opendbc.

boardd

The component boardd facilitates communication between the Panda module and the communication module Cereal. It is the backbone of the communication between the vehicle's sensors and actuators, and the decision-maker. Figure 4 displays the proposed structural position of boardd.

cereal

Cereal is a publisher/subscriber messaging specification for robotic systems [21]. Cereal provides a specification for exchanging messages between robotics systems and enables interprocess communication [15]. Its messaging specification uses an Event struct in storing data in packets and uses Cap'n Proto, a fast data interchange format and capability-based remote procedure call system [1], in transferring data. Its interprocess communication subsystem implements an Implicit Invocation: Publish-Subscribe architecture. It uses the messaging library ZeroMQ to implement sockets for the transfer of packets between processes [14]. It also contains the custom library msgq which builds the Pub-Sub architecture on top of shared memory that can bypass the kernel [15]. Figure 4 displays the proposed structural position of cereal.

Implicit Invocation - Event-Based

Implicit Invocation refers to the style that allows for 'on the fly' reconfiguration, in which loosely coupled components trigger one another. The Event-Based variant specifically refers to the use of events emitting an asynchronous trigger, with other components receiving the events over an event bus.

Within openpilot, the implicit invocation can be seen within the Panda component, which is responsible for reading from sensors and writing to actuators. The data is stored in CAN-based messages, which make use of event buses for communication. Various buses make up this Event-Based Architecture, and they are all routed toward panda. Sensors and actuators can send and receive data respectively through buses. They are loosely coupled and thus the individual sensors and actuators would never have to communicate.

Implicit Invocation - Publish Subscribe

The Publish-Subscribe variant of Implicit Invocation refers to the architecture style where messages are sent and received by having Publishers maintain a list of subscriptions and broadcast messages and having Subscribers register with the Publishers to receive specific broadcasted messages.

openpilot implements the Publish-Subscribe architecture in its message specification and interprocess communication component, Cereal. Cereal enables sensor processes to send data to other processes by publishing it in packets, and the other processes receive the data by subscribing to those packets and receiving it when it's published [15]. In a use case of the Cereal subsystem, a sensor process would read measurements from an inertial measurement unit and publish them in a packet. A calibration process and localization process would both have subscribed to the packet and would then receive the IMU measurements when they are published [15].

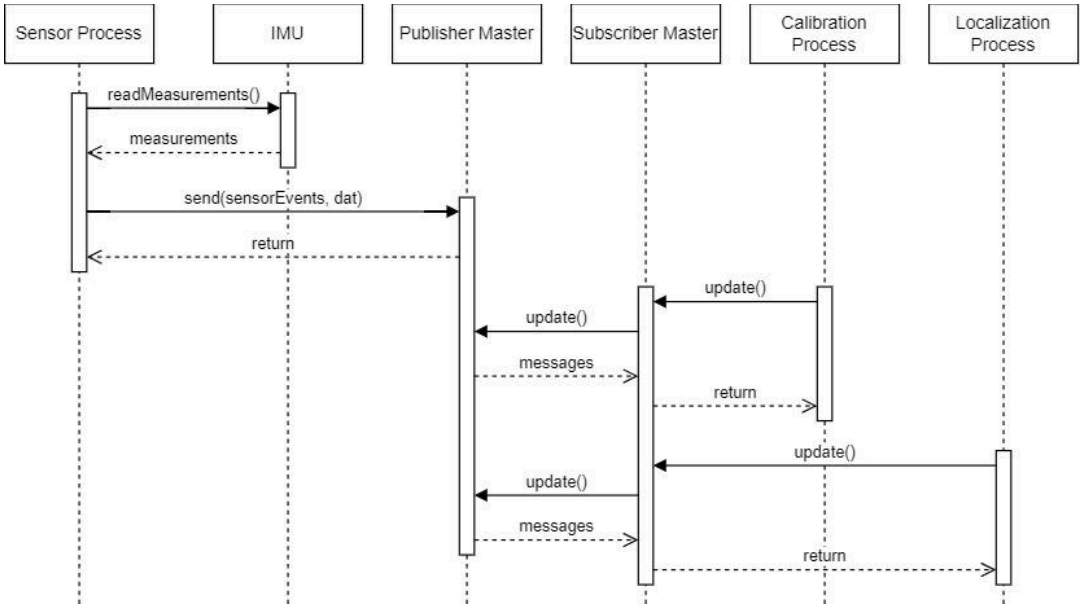

Figure 5: Cereal Sequence Diagram

Process Control - Closed-Loop Feedback

Process Control refers to the architecture style where there is a specific process, or plant, that requires maintenance at a specified value. Closed-loop feedback is a specific type of process control, in which the reads the current output value to determine what adjustments need to be made to achieve the desired value. The controller is responsible for taking in input, as well as the current output, and making changes to reach the desired value.

openpilot makes use of numerous process controllers, for different aspects of driver support. One example is within Adaptive Cruise Control (ACC), in which the desired variable would be the set speed. The controller for speed would attempt to ensure the actual speed matches the desired value. It makes use of the closed-loop feedback, reading the current speed to determine which adjustments can be made to hit the desired speed, such as greater acceleration, less acceleration, or braking.

The Lane Keeping Assistant System (LKAS) would also make use of the process control architecture similarly. It would examine the distance from either side of the acceptable lane, with the desired value being specific distances on the left and right sides of the vehicle. It would read the actual distances via the feedback loop, and adjust steering as required.

Arguably, the process-control architecture is the most important one involved in the openpilot system. The openpilot developers made use of this architecture as it is a well-known solution for systems that require sensors and actuators, and require a setpoint value. In terms of vehicle control, it also is expected to use closed-loop feedback as opposed to open-loop, as it's crucial to know what the current speed and location of the vehicle are before making adjustments.

Process Control - MAPE-K

MAPE-K is an acronym for the architecture style known as Monitor-Analyze-Plan-Execute, over shared Knowledge. This architecture operates as a feedback loop, allowing for adaptive systems to make changes while using knowledge. One

approach to this architecture is to make use of a Machine Learning Model as the knowledge component.

An example of where openpilot uses machine learning is in the Laneless Mode, which uses machine learning to predict where humans would normally drive. This model is not informed about specifics such as lane lines, intersections, lane changes, curves, or any other traffic rules - rather it is to learn them based on real data about how people drive. The goal of this methodology is to create a model that will not freeze when in an unclear situation, such as approaching a dirt road with no line markings. openpilot claims that this method also results in smoother driving, including driving more naturally through intersections, highway exits, and merges [13].

The below image, from the University of British Columbia, outlines the structure of MAPE-K that makes use of machine learning. This **may apply** to openpilot, with the 'target system' being the vehicle. The system monitors and analyzes via sensors and cameras, makes use of the machine learning model (knowledge) to assist in planning and executing decisions, and writes the data to the accelerator, steering, or brakes. The way that the MAPE-K structure operates is presented in Figure 7 below.

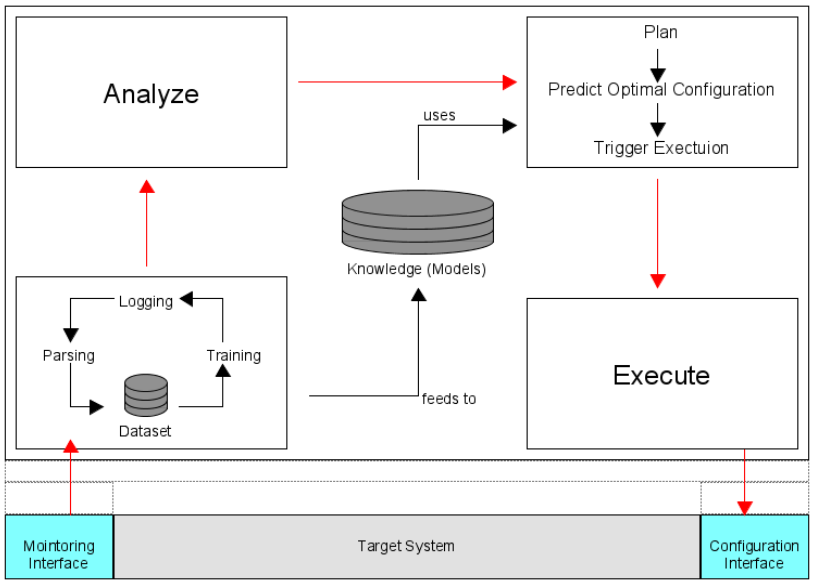

Figure 6: MAPE-K with Machine Learning [12]

Client-Server

Client-server refers to the architecture style that involves standalone remote components (servers) that are accessed by clients that make calls via the network.

openpilot states that, by default, all driving data is uploaded to their servers [11], where it is also accessible via Connect, an application to control the Comma 3. openpilot states that this data is used to train their machine learning models, by providing it with as much sample data as possible, improving openpilot as a whole. This supports the client-server architecture style, with there being a main server for data, and the vehicle's Comma 3X device acting as the client to send information. Communication is done via LTE or WiFi.

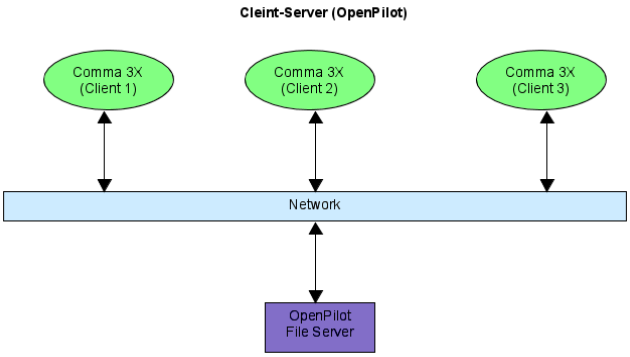

Figure 7: Client-Server Architecture

Other Architectural Styles

Pipe and Filter

openpilot is very likely to have internal pipelines to handle data, as many diagrams include an ordered sequence of events. These are usually put into the order: the sensor detects the external environment and sends the data to the decision maker module then processes that data and responds by sending data to the actuators that they must execute. However, there is a need to ensure they are stateless filters, and that they are unaware of up/downstream, for the pipe and filter style to apply.

Repository

Although there was a lack of documentation on repository style (that the team looked through) it is everyone's understanding that there must be a database somewhere when data is uploaded to a server and used for training models. More could be discovered in Assignment 2.

System Evolution

How does the system evolve? What is the control and data flow among parts?

The openpilot system evolves through a structural release process, with multiple iterations enhancing and adding features. For example, the recent 0.9.5 release brought architectural changes to driving models moving from EfficientNets to FastViTs. These models now use a Hybrid Vision Transformer architecture and perform much better in all tests [10]. It also introduces new navigation instructions and lateral planning, moving planning components inside the model. Tools like MetaDrive and Pytest were added to simplify simulation and testing. Multilingual support was expanded, and UI updates were made to improve user experience. The car interface was refined with extensive fuzzing tests and a more thorough car interface. These changes reflect a commitment to ongoing improvement and innovation. Each release builds on the last to provide a more robust and capable system.

Additionally, beyond the structured release process, the openpilot system also evolves through various other aspects. One of these aspects is that openpilot is an open source software project. This means that any developer is free to view, modify, or enhance the project. This open-source nature allows for a collaborative environment by enabling continuous improvements through contributions from the open source community. This enhances the openpilot system as developers use this opportunity to deploy bug fixes, create new features, clean up code, etc. It is important to note that all new changes are reviewed by the internal team before any changes are made permanent. Furthermore, alongside community contributions, internal employees are also

actively working on improving/evolving the system. There are several employees present who are all assigned to different areas such as Full Stack Developer, Car Interface Engineer, Production Engineer, Infrastructure Engineer, Technician, etc [9]. Moreover, The users of openpilot also play an essential role in evolving the system. When a user uses openpilot, specific data is uploaded to their servers. The specific data being collected from the users include road-facing cameras, CAN, GPS, IMU, magnetometer, thermal sensors, crashes, and operating system logs. This user data is leveraged by the openpilot team to improve and train better models so that openpilot is better for everyone [11].

In the openpilot system, control and data flow involve sensors gathering environmental data, which is processed by the onboard computer. The data flows through components like Panda for vehicle communication, with decisions made based on input from cameras and other sensors. These decisions are executed as control commands are sent to the vehicle's actuators. The architecture supports concurrent processing, with real-time data analysis and decision-making for functions like adaptive cruise control and lane keeping. The components within openpilot communicate with each other through a combination of messaging, protocols and interfaces. Cereal acts as a messaging framework, facilitating inter-process communication through a publisher-subscriber model. Boardd manages the connection between the openpilot software and the Panda hardware. Opendbc interprets vehicle CAN messages into a human-readable format, using DBC files. Which Panda then utilizes to communicate with the vehicle's CAN network, reading sensor data, and sending control commands. This architecture ensures seamless data flow which ensures that the openpilot software receives all of the necessary information from the different subsystems to make informed decisions [21].

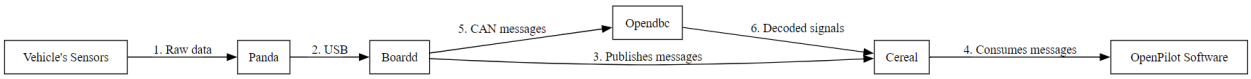

Figure 8: Communication Flow among openpilot Components

What are the implications for the division of responsibilities among developers?

Openpilot has a modular architecture that consists of four main components: vision, planning, control, and interface. Each component has a different role and responsibility in the system, and they communicate through a message passing framework called cereal [8]. The vision component is responsible for processing the camera input and producing a high-level representation of the road scene, such as the lane boundaries, the lead vehicle, and the traffic signs. The vision component uses a deep neural network called Supercombo, which is trained on millions of miles of driving data collected by openpilot users [7]. The planning component is responsible for generating a desired trajectory for the car to follow, based on the current state of the car and the environment. The planning component uses another deep neural network, which takes the output of Supercombo and produces a sequence of steering and acceleration commands [7]. The control component is responsible for executing the commands from the planning component and sending them to the car's actuators. The control component also monitors the car's state and the driver's engagement, and intervenes if necessary. The control component uses a combination of PID controllers, Kalman filters, and finite state machines [8].

The division of responsibilities among these components implies that the developers of openpilot need to have different skills and expertise, depending on which component they work on. For example, the developers of the vision and planning components need to have a strong background in machine learning, computer vision, and data analysis, while the developers of the control and interface components need to have a good understanding of the car's dynamics, the

human factors, and the safety requirements. Moreover, the developers need to collaborate and coordinate their work across the components, ensuring that they are compatible and consistent with each other. This requires a high level of communication, documentation, and testing [8].

The implications for the division of responsibilities among participating developers of openpilot are both positive and negative. On the positive side, dividing the work into different components allows the developers to focus on their areas of expertise and interest, and to leverage the existing tools and frameworks that suit their needs. This can lead to higher quality, efficiency, and innovation in each component. Moreover, having a modular architecture enables the developers to reuse and integrate their code with other open source projects, creating a network effect and a larger community of contributors and users.

On the negative side, dividing the work into different components also introduces some challenges and risks for the developers. For example, they need to ensure that their code is compatible and consistent with the other components, and that they follow the common standards and guidelines of the project. This requires a lot of communication, coordination, and testing among the developers, which can be time-consuming and difficult. Furthermore, they need to deal with the security and legal issues that may arise from using or contributing to open source software, such as licensing, compliance, and vulnerability management. These issues can be complex and costly to resolve, and may affect the reputation and trustworthiness of the project

---

The derivation process explained and alternatives (if available) discussed

The derivation process of openpilot's architectural composition was done by purely looking at the documentation and project structure. The project structure refers to how collections of files were separated in openpilot's Github repository. This means that we have done our analysis without looking at any source code. Thus, the proposed architectural styles could be seen as our suggestion for how a system like openpilot should be designed without actually implementing it. A lot of connections that we cannot notice exist, and this will be the topic of Assignment 2.

One alternative that we considered before dwelling into the architectural composition of openpilot is that it could be implemented using a Microservice architecture style. We did not discuss the Microservice style during lectures, but our team has experience with it from prior courses. Microservices implement the system as a collection of services that do processing without interacting with other services. They rely on information by directly grabbing it from sensors and directly sending instructions to actuators (or perhaps a repository). However, they each perform a different function. The advantage of Microservices is that they improve modularity, enhance scalability, and increase reliability. The disadvantages include increased complexity, coordination and consistency. Implementing it this way would allow each part to be independent which could work well for a system comprising multiple functions. An advanced driver assistance system is one such system.

Concurrency

What concurrency if any is present?

Yes, there is an indication of concurrency present. The fact that openpilot takes on input from many different sources such as cameras and sensors that come installed in the vehicle. All sensors are constantly monitoring for a combination of inputs which could result in a unique output variable such as an alert mechanism or notification. Moreover, another example of concurrency is that the decision-making model most likely uses the collected input to send data

to different components of the software which could use overlapping input from the same sensors. This means that there may be multiple decision-making components implemented in the software for the features present in the software. For example, ALC and LDW both use the same input variables but output two different results. ALC automatically centers vehicles while LDW is responsible for warning the driver of unintentional lane changes [5]. For proof of concept, this is a diagram created and published by individuals attending [4]. Accordingly, Figure 11 illustrates concurrency where interacting systems have many events occurring at the same time. The arrows in each subsection occur simultaneously.

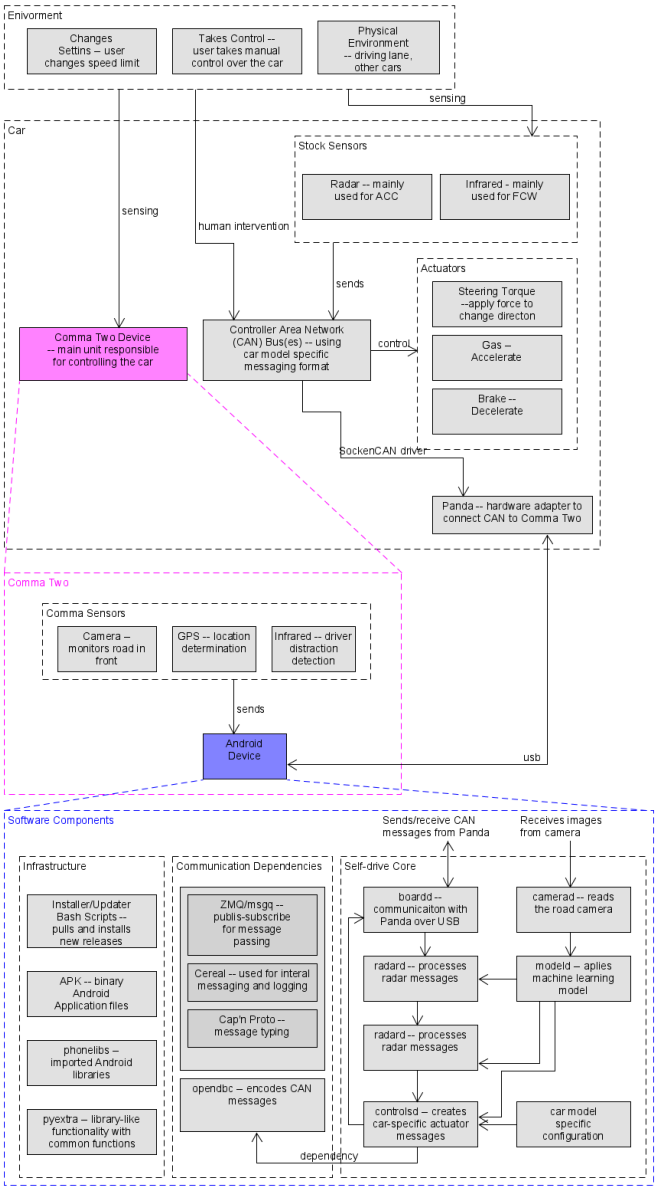

Figure 9: Delft University architectural diagram demonstrating areas of concurrency [3]

The sequence diagram shows how different processes of openpilot communicate with each other and with the car. The ui handles user input, such as engaging or disengaging the system. The controlsd is the main controller of the system which receives the car state and sends the actuator commands. The plannerd is responsible for planning the desired trajectory and speed

of the vehicle, based on the car state and the neural network models. The radard is responsible for processing the radar data and other environmental readings, such as the lead car. The car is the actual vehicle that openpilot is controlling which sends the car state to controlsd and receives the actuator commands from it.

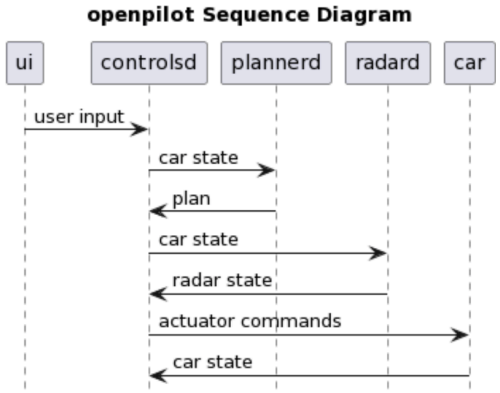

Figure 10: openpilot Sequence Diagram of Radar Subcomponents

Use Cases

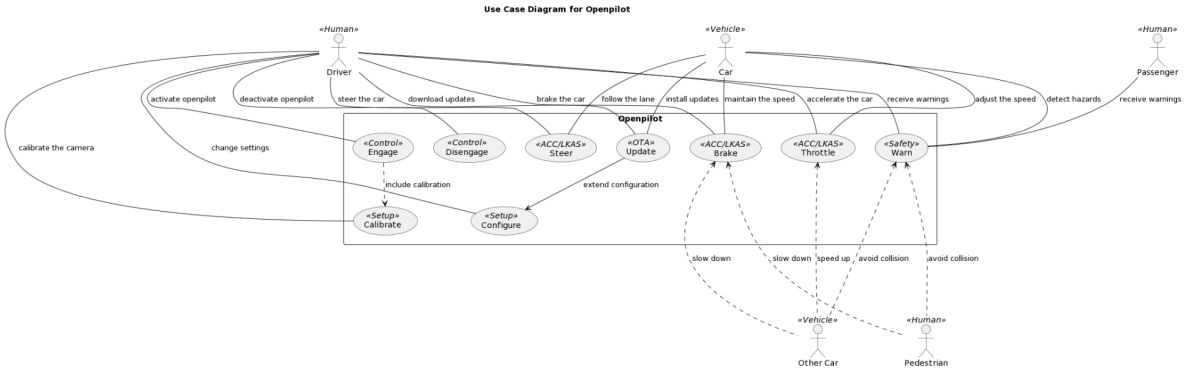

Figure 11: openpilot Use Case Diagram

Data Dictionary

AGNOS: Openpilot's new Ubuntu based OS. Runs on Comma 3.

boardd: A service that communicates with the panda and sends steering, gas, and brake commands to the car. It also reads car state such as speed, gear, and gas.

CAN-FD: A protocol for communication between car components. Faster CAN.

cereal: A messaging specification and library for inter-process communication

comma 3/3x: A newer version of the comma 2 device. It has a faster processor, a better cooling system, and a higher resolution screen.

controlsd: A service that controls the car's actuators to follow the plan. It also handles the user's input and the openpilot's status.

plannerd: A service that plans the desired car state for the next few seconds. It computes the desired speed, acceleration, and steering angle.

NEOS: The operating system that runs on the comma devices. It is based on Android 10 with some customizations and optimizations for openpilot. Runs on Comma 2.

opendbc: A collection of DBC files for different car models. DBC files describe how messages are encoded on the CAN bus for different cars.
panda: A universal car interface that connects your car to openpilot. It can act as a CAN bridge or a black box. It is responsible for reading/ data from sensors and writing data to the actuators.
radard: A service that communicates with the car's radar.
ui: A service that renders the user interface on the device's screen. It shows the camera feeds, the openpilot's status, and the alerts.

Naming Conventions

ADAS - Advanced Driver Assistance System
CAN - Controller Area Network
CAN-FD - CAN Flexible Data
GPS - Global Positioning System
IMU - Inertial measurement unit

Conclusions

Exploring openpilot revealed the use of an array of diverse architectural styles: Layered, Implicit Invocation - Event Based, Implicit Invocation - Publish & Subscribe, Process Control - Closed Loop Feedback, Process Control - MAPE-K, Client-Server, Pipe and Filter and Repository. These insights were gained from a thorough examination of documents, supported by a certain amount of interference. As the source code was not inspected for this project, some of the architectural styles listed may or may not be components of openpilot. More will be discovered in Assignment 2.

Lessons Learned

There were various things that the team would change if given the chance to redo the project. First and foremost, we would spend less time looking at the project structure of the Github repository. The reason for this is that the structure of openpilot and its nature as an open-source project is much different than what we are used to in academia. The variety of documentation for a massive safety-critical system such as openpilot is as vast as it is wide. There is a lot of information on a lot of subsystems but there is much less documentation on the overall structure as each subsystem is substantial.

In terms of learning more about openpilot, information has been gathered in terms of the limitations. For example, openpilot is not currently aware of the speed limit given signage, but only if using navigation. Additionally, when changing lanes, it cannot see next to you or check your blindspot - and urges drivers to only nudge the wheel to initiate lane change if they've already done these checks manually. openpilot may also not function as intended in low-light, weather conditions, or with bright oncoming headlights [2].

References

[1] "Introduction," Cap'n Proto: Introduction, https://capnproto.org/ (accessed Feb. 13, 2024).
[2] "Openpilot documentation," docs.comma.ai, https://docs.comma.ai/LIMITATIONS.html (accessed Feb. 13, 2024).
[3] "Delft students on software architecture," Delft Students on Software Architecture - DESOSA 2020, https://desosa.nl/ (accessed Feb. 13, 2024).

[4] “From vision to architecture: How to use openpilot and live,” From Vision To Architecture: How to use openpilot and live - DESOSA 2020, https://desosa.nl/projects/openpilot/2020/03/11/from-vision-to-architecture (accessed Feb. 13, 2024).
[5] “Lane departure warning system,” Wikipedia, https://en.wikipedia.org/wiki/Lane_departure_warning_system#:~:text=This%20system%20uses%20infrared%20sensors,alerts%20the%20driver%20of%20deviations (accessed Feb. 13, 2024).
[6] What is adaptive cruise control?, https://www.caranddriver.com/research/a32813983/adaptive-cruise-control/ (accessed Feb. 13, 2024).
[7] “ArXiv.org e-print archive,” arXiv.org e-Print archive, https://arxiv.org/ (accessed Feb. 13, 2024).
[8] DOI name 10.1007 values, https://doi.org/10.1007 (accessed Feb. 13, 2024).
[9] “Jobs,” comma, https://comma.ai/jobs#open-positions (accessed Feb. 13, 2024).
[10] “Openpilot 0.9.5,” comma.ai blog, https://blog.comma.ai/095release/ (accessed Feb. 13, 2024).
[11] “Openpilot documentation,” docs.comma.ai, https://docs.comma.ai/index.html (accessed Feb. 13, 2024).
[12] Lightweight self-adaptive configuration using machine ..., https://www.cs.ubc.ca/~rtholmes/papers/cascon_2021_araujo.pdf (accessed Feb. 13, 2024).
[13] “Openpilot 0.8.15,”comma.ai blog, https://blog.comma.ai/0815release/ (accessed Feb. 13, 2024).
[14] ZeroMQ, https://zeromq.org/ (accessed Feb. 13, 2024).
[15] “Openpilot documentation,” docs.comma.ai, https://docs.comma.ai/cereal/README.html (accessed Feb. 13, 2024).
[16] D. P. Losada, J. L. Fernández, E. Paz, and R. Sanz, “Distributed and modular can-based architecture for hardware control and Sensor Data Integration,” MDPI, https://www.mdpi.com/1424-8220/17/5/1013 (accessed Feb. 13, 2024).
[17] “Openpilot documentation,” docs.comma.ai, https://docs.comma.ai/panda/README.html (accessed Feb. 13, 2024).
[18] “Comma 3X – installation guide,” comma 3X –, https://comma.ai/setup/comma-3x (accessed Feb. 13, 2024).
[19] “Openpilot - Open source advanced driver assistance system,” comma.ai - make driving chill, https://www.comma.ai/openpilot (accessed Feb. 13, 2024).
[20] “Comma 3x - make driving chill,” - make driving chill, https://comma.ai/shop/comma-3x (accessed Feb. 13, 2024).
[21] “How openpilot works in 2021,” comma.ai blog, https://blog.comma.ai/openpilot-in-2021/ (accessed Feb. 13, 2024).
[22] “Lane Departure warning: Innovation,” Innovation |, https://www.nissan-global.com/EN/INNOVATION/TECHNOLOGY/ARCHIVE/LDW/#:~:text=Lane%20Departure%20Warning%20provides%20an,the%20driver%20to%20take%20action (accessed Feb. 13, 2024).
[23] “Forward collision warning,” My Car Does What, https://mycardoeswhat.org/deeper-learning/forward-collision-warning/#:~:text=Forward%20collision%20warning%20systems%20warn,road%20ahead%20while%20you%20drive (accessed Feb. 13, 2024).