EECS 4314

Assignment 2: Concrete Architecture of OpenPilot

AUTHORS:

JOSHUA GENAT joshgena@my.yorku.ca
NOBAIHA ZAMAN RAYTA nobaiha@my.yorku.ca
ABHIRAMI VENUGOPAL abhirami@my.yorku.ca
SAYED MOHAMMED sayedmohammed1965@gmail.com
AHMED EL-DIB eldib@my.yorku.ca
JETHRO EMILRAJ jethro.emilraj@gmail.com
DEXTER EROMOSELE dexter25@my.yorku.ca
VIVAAN RAJPUT vivaanxr@protonmail.com

Table of Contents

Abstract

This report provides a detailed analysis of the Concrete Architecture of OpenPilot, a driver assistance system by comma.ai, exploring its architecture, functionality, design patterns, and underlying framework. It highlights OpenPilot's role in autonomous driving technology, outlines its concrete architecture and subsystem interactions, and discusses the Architecture Derivation Process and its influence on system performance and scalability. The report examines the use of design patterns, particularly the Singleton Pattern in the Camera System's Memory Manager Class and the Factory Pattern for Sensor Information Management. It also compares the Conceptual and Concrete Architecture to show OpenPilot's design evolution. The study emphasizes the need for developer collaboration, a modular architecture, and clear external interfaces for OpenPilot's development. Concluding, the report offers insights into OpenPilot's architecture, contributing to advancements in autonomous driving technology.

Introduction

This report offers an introduction into how OpenPilot actually works. OpenPilot is a smart driving assistant created by comma.ai. It helps drivers stay safe and enjoy their time on the road by doing things like keeping the car in the lane and adjusting its speed. OpenPilot uses advanced technology, like cameras and sensors, to make smart decisions while driving. In this report, we will explore how OpenPilot's technology is put together. We will break down its various parts and how they work together. By the end, we will have a better understanding of how OpenPilot keeps you safe and makes driving easier.

High Level Concrete Architecture

Open Pilot is designed with modularity and real-time processing at its core, enabling it to interact seamlessly with a vehicle's systems and sensors. The architecture employs various architectural styles, including **layered**, **event-driven (Implicit Invocation)**, and **process control**, to facilitate a responsive and adaptable autonomous driving solution.

Layered Architecture in Open Pilot

Open Pilot's layered architecture organizes the system into hierarchical strata, where each layer fulfills a unique role and mainly interfaces with the adjacent layers, fostering system robustness and flexibility through functional abstraction, encapsulation, and modularization. At the foundation, the Hardware Layer connects directly with the vehicle's hardware, such as cameras, GPS modules, and the CAN bus, enabling direct control system communication. Above this, the Sensor Data Processing Layer processes raw sensor and hardware data into usable information for decision-making. The Core Functionality Layer, at the heart of Open Pilot, houses the principal logic and decision-making mechanisms, including driving policy and control algorithms. The Communication and Interface Layer, at the top, oversees user and external system interactions through user interfaces and communication protocols. Additionally, the Logging and

Monitoring Layer, transcending traditional layering, spans the entire architecture to provide essential operational insights and performance optimization through system logging and health monitoring.

Event-Driven Architecture (Implicit Invocation) in Open Pilot

In Open Pilot, the EDA (Implicit Invocation) plays a pivotal role in orchestrating interactions among different system components. This architecture enables the system to respond dynamically to changes in the driving environment or internal state, fostering a reactive and modular design.

Components and Their Roles

1. **Event Publishers**: These are components that generate events based on specific triggers, such as changes in sensor data or internal state updates.
   Example Modules: camerad: Publishes events related to image data processing or changes detected in the visual field, **sensord**: Generates events based on sensor data variations or anomalies.

2. **Event Listeners (Subscribers)**: These components listen for specific events and execute their functionality in response, without needing to know which component published the event.
   Example Modules: Control algorithms: React to sensor events by adjusting vehicle controls, **Logging systems (proclogd, logcatd)**: Subscribe to system events for logging and monitoring purposes.

Event-Driven Interactions in Open Pilot

- When ***camerad*** detects a notable change in the visual field, it publishes an event that the control algorithms subscribe to. These algorithms then process the event and adjust the vehicle's steering or speed accordingly.
- ***sensord*** might publish an event when an unexpected object is detected near the vehicle. The navigation or obstacle avoidance modules, as subscribers, would respond to this event by recalculating the vehicle's path.

Process Control Architecture in Open Pilot

Open Pilot's Process Control Architecture establishes a perpetual loop of input, processing, and output, thereby ensuring that the vehicle's actions remain consistent with desired outcomes and aptly respond to environmental variations. At the outset, Sensor Input Management components gather and preprocess data from the vehicle's sensors, laying the groundwork for decision-making. This includes camerad, which handles visual data, and sensord, which deals with data from physical sensors like accelerometers and gyroscopes. Following this, Control Algorithms, including the path planning and control module, use the sensor data to decide how to maintain the vehicle's intended state by calculating adjustments to steering, throttle, and braking. Actuator Control, via boardd, then translates these decisions into physical commands to the

vehicle's control systems, effectuating steering, acceleration, or braking. Concurrently, Monitoring and Feedback mechanisms, such as the health monitoring system, continuously evaluate the vehicle's state and environmental conditions. This evaluation provides essential feedback to the control algorithms, enabling a closed-loop system that facilitates real-time adjustments. This interplay between sensor input management, decision-making algorithms, actuator control, and ongoing monitoring and feedback ensures the vehicle adheres to its intended path and adjusts dynamically for safety and performance optimization.

Subsystems and their Interactions

Given the interactions we found using Understand, we could visualize the subsystem architecture as follows:

Camerad Subsystem:

Cameras Subsystem:

Camera Utility Module: Handles utility functions and interfaces with third-party libraries.

Camera QCOM2 Module: Adapts or enhances camera functionalities for Qualcomm-specific hardware.

Camera Common Module: Central hub for camera operations, interfacing with selfdrive, tools, and other camera modules.

This organization allows for a clear separation of concerns, with each module handling specific aspects of camera management and operation, while also detailing the interactions between modules and external systems.

1. Camerd subsystem: The general subsystem that houses all camera-related features is called the Camerad Subsystem. It includes the camera module and all of its features.
2. Cameras Subsystem: This subsystem of camerad is in charge of the hardware and software for the cameras. It is made up of various parts, each playing a different role:

a. The camera_util.h and camera_util.cc files are part of the camera utility module. Utility functions and communication with external libraries or modules are handled by these files. This might be viewed as an interfacing layer or utility within the Cameras Subsystem due to their substantial interaction with external components (24 inputs from.h and.cc combined).
b. Camera QCOM2 Module: This module, which is exclusive to Qualcomm hardware (probably, QCOM2 refers to a Qualcomm chipset or camera specification), is composed of camera_qcom2.cc and camera_qcom2.h. It interacts with the common camera functionalities and has a sizable number of third-party inputs (181 from.cc), suggesting that it plays a part in customizing or improving the common functionality for hardware that is special to Qualcomm.
c. Camera Common Module, which includes camera_common.h and camera_common.cc, is the central component of the Cameras Subsystem. It receives 282 + 1 (from selfdrive) + 8 (from tools) + inputs from camera_util.cc, camera_qcom2.cc, and camera_qcom2.h. It interfaces

directly with other components of the subsystem. It's likely that the basic camera functions and utilities provided by this module are improved upon or used by other modules.

3. External Interactions:

a. Tools and selfdrive inputs: According to these inputs into camera_common.h, the Camera Common Module interacts with both driving features and development tools to function as a hub for camera-related operations in the larger OpenPilot system.

Interactions with third parties: A number of modules have interactions with libraries or modules from third parties, which suggests external dependencies or integrations with the OpenPilot project. Codecs, drivers, and other libraries for multimedia processing may be examples of this.

Architecture Derivation Process

Conceptual Architecture Planning: Initially, the necessity of a modular and layered architecture would guide the conceptual framework. This involves defining high-level modules and layers, such as hardware interaction, data processing, decision-making, and user interface layers, envisioning how these would interact and support the system's scalability and adaptability. The conceptual architecture would focus on the abstract principles of system design, like modularity, to ensure that the framework could accommodate future changes and features without significant overhauls.

Design Patterns Integration: The integration of design patterns like the Singleton and Factory patterns would be planned at this stage. Conceptually, this involves identifying areas where resource management and code reusability are critical, such as memory management and sensor information processing. These patterns would be envisioned as solutions to prevent redundancy, facilitate maintenance, and enable the easy integration of new sensor types, laying a foundation for the concrete architecture.

Event-Driven Architecture Adoption: The conceptual architecture would incorporate the idea of an event-driven system to enhance responsiveness and modifiability. This would involve planning for components to react to events rather than polling for changes, enabling real-time processing and decision-making. The architecture would outline how different parts of the system could communicate through events, ensuring system dynamism and adaptability.

Planning for External Dependencies and Feedback Loops: At the conceptual level, the architecture would consider the integration of third-party libraries and specialized modules, acknowledging the challenges and planning for the integration process. It would also incorporate the idea of continuous feedback loops for real-time monitoring and decision-making, acknowledging the importance of adaptability and agile responses to changing conditions.

Concrete Architecture Development: With the conceptual framework in place, the development of the concrete architecture would begin. This phase involves detailed design and implementation planning, translating high-level concepts into specific technologies, libraries, and coding practices. For instance, deciding on the specific implementation of the Singleton and

Factory patterns, selecting third-party libraries, and designing the structure of feedback loops and event-handling mechanisms.

Emphasis on Documentation, Testing, and Community Feedback: Finally, the architecture would incorporate strategies for comprehensive documentation to support onboarding and collaboration, thorough testing and validation protocols for ensuring safety and reliability, and mechanisms for gathering and integrating user and community feedback into ongoing development cycles.

Architecture Styles

Event-Driven Architecture (Implicit Invocation): OpenPilot leverages an event-driven architecture, where different components communicate via events rather than direct interactions. This approach enhances modularity and responsiveness, crucial for real-time systems like OpenPilot. Components like ***camerad*** generate events based on sensor inputs, which are then processed by other parts of the system in a loosely coupled manner. **Layered Architecture**: The system employs a layered architectural style, organizing its components into hierarchical layers that interact with each other in a well-defined manner. From the hardware interaction layer up through sensor data processing, core functionality, and user interface layers, OpenPilot ensures ***separation of concerns*** and organized interaction between components. **Process Control Architecture**: Given OpenPilot's need for real-time monitoring and control, it aligns with the process control architecture. This style is evident in how OpenPilot continuously processes input from various sensors, makes decisions, and controls the ***vehicle's actuators*** to achieve desired driving behaviors, all within tight feedback loops.

Design Patterns

By skillfully using the patterns, OpenAI developers create an intuitive and flexible framework to support the development of artificial intelligence. Below are the two essential design patterns that are used in the OpenAI's architecture.

Singleton Pattern Implementation in Camera System's MemoryManager Class

The Singleton pattern is used within the camera system's MemoryManager class to ensure the existence of only one instance throughout the application's lifecycle. This pattern promotes efficient resource utilization and simplifies memory management tasks within OpenAI's architecture. The camera system within OpenAI's architecture leverages the Singleton pattern to guarantee the existence of a single instance of the MemoryManager class. By restricting instantiation to within the class itself and providing global access to this instance, the system ensures consistency in memory management operations and prevents unnecessary resource duplication. The MemoryManager class serves as a central component, managing memory-related operations efficiently.

This encapsulation reinforces the Singleton pattern's intent, emphasizing the importance of having only one instance globally accessible. Other parts of the code interact with the MemoryManager instance through a global variable, promoting seamless communication and coordination between different components of the system. By enforcing a single instance of the MemoryManager class, OpenAI's architecture maintains consistency, simplifies debugging and maintenance efforts, and enhances the overall robustness and reliability of the system.

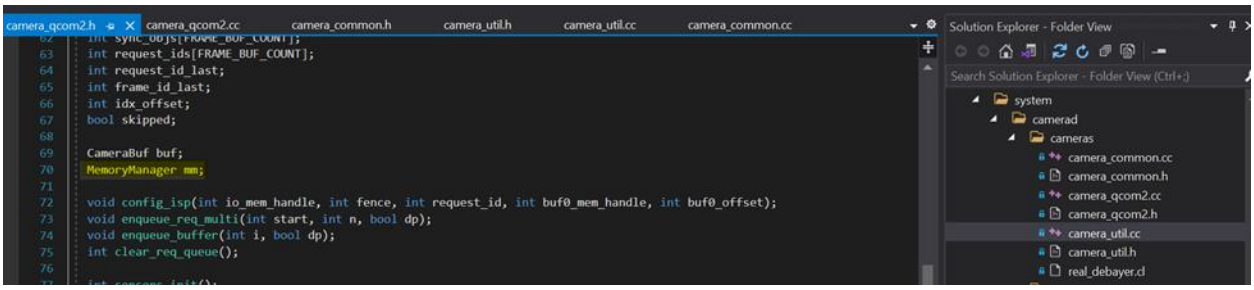

Figure 1: Singleton Pattern in the MemoryManager Class

Factory Pattern Implementation for Sensor Information Management

The Factory pattern plays an important role in managing sensor information within OpenAI's architecture, facilitating the creation of concrete implementations tailored to specific sensor models. The Factory pattern promotes modularity, reusability, and scalability in handling sensor data. OpenAI's architecture employs the Factory pattern to manage sensor information effectively. The SensorInfo class encapsulates common functionality for various sensor information objects, providing a clear and concise interface. Concrete subclasses such as AR0231, OX03C10, and OS04C10 offer specific implementations tailored to different sensor models, ensuring modularity and reusability of code components.

The factory aspect of the design pattern comes into play when instances of SensorInfo or its subclasses are created based on runtime conditions or configurations. This abstraction of the object creation process enhances flexibility and scalability, enabling the addition of new sensor types without requiring modifications to existing client code.Client code interacts with the SensorInfo interface without needing awareness of the specific subclass being used, enhancing code maintainability and simplifying future enhancements or modifications. The architecture allows for seamless extension and modification to accommodate new sensor types, ensuring minimal disruption to existing functionality.

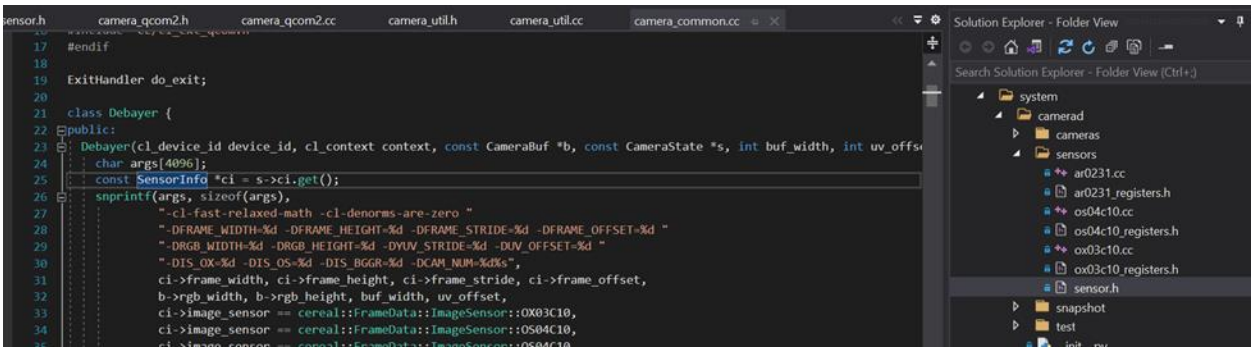

Figure 2: Factory Pattern for Sensor Information Management

Conceptual vs. Concrete Architecture

Event-Driven Architecture (Conceptual) vs. Message Passing and Inter-Module Communication (Concrete):

Conceptually, OpenPilot uses an event-driven architecture, where different components communicate and coordinate actions through events. This allows the system to be responsive and modular. In the concrete implementation, this is visible in how different modules in OpenPilot communicate through ***event signaling/message passing***. While the GitHub code shows a running system and the presence of various independent modules that interact.

Process Control Architecture (Conceptual) vs. Real-Time Monitoring and Control Loops (Concrete):

At the conceptual level, OpenPilot follows a process control architecture, continuously monitoring inputs and adjusting outputs (vehicle controls) in a feedback loop. Looking at the repository, the ***concrete implementations of this in the form of real-time data processing***, control algorithms, and feedback mechanisms. Specific files and modules are dedicated to processing sensor inputs, making decisions, and sending commands to the vehicle's control systems, reflecting this continuous loop of monitoring and control in real time.

Lessons Learned

In dissecting Open Pilot's architecture, valuable insights can be extracted about software design, particularly in the realm of ***autonomous driving systems***. By examining the architecture as derived from the directory structure, understanding the design patterns employed, and analyzing the interactions within the ***camerad*** subsystem, several lessons can be learned that are applicable to a broad range of software development endeavors: **Modularity Enhances Flexibility**: The system's modular design, illustrated by its clear separation into distinct directories and components, facilitates scalability, ease of testing, and integration of new features. **Directory Structure Reflects Architecture**: Open Pilot's directory organization mirrors its architectural design, highlighting the importance of a ***reflective and well-organized file system*** for maintainability and understandability.**The need for more**

Event-Driven and Layered Architectures: The use of event-driven and layered architectures in Open Pilot showcases their effectiveness in systems needing ***rapid responsiveness*** and clear separation of concerns, respectively.**Real-Time Processing is Crucial**: The process control architecture within Open Pilot underscores the necessity of real-time data processing and feedback mechanisms, especially in ***safety-critical applications***. **Subsystem Integration and Autonomy**: The ***camerad*** subsystem demonstrates how complex components can be seamlessly integrated yet maintain a level of autonomy, emphasizing the balance between ***interdependence and independence*** in subsystem design.

Understand Screenshots

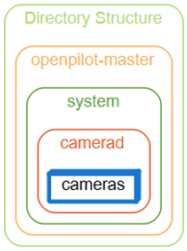

Figure 3: Directory structure

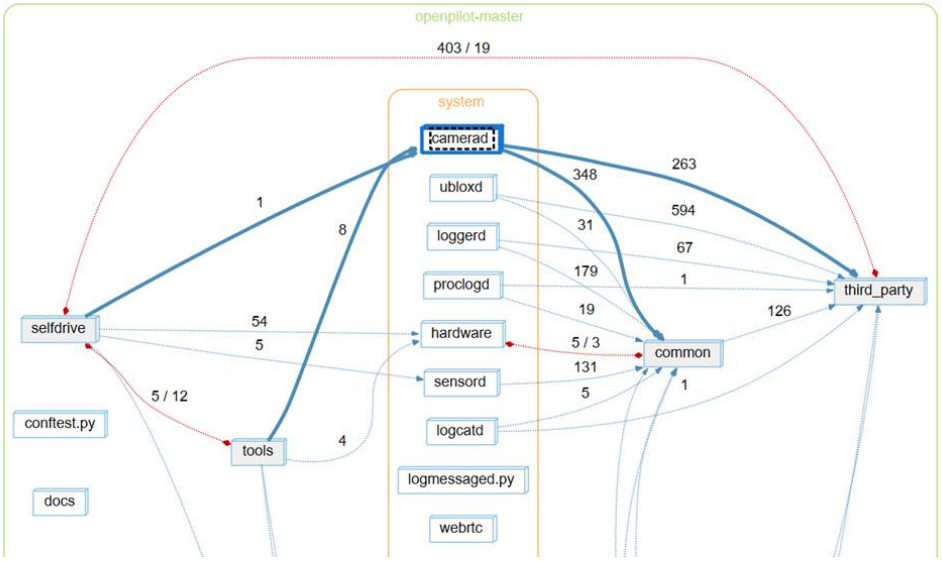

Figure 4: camerad structure

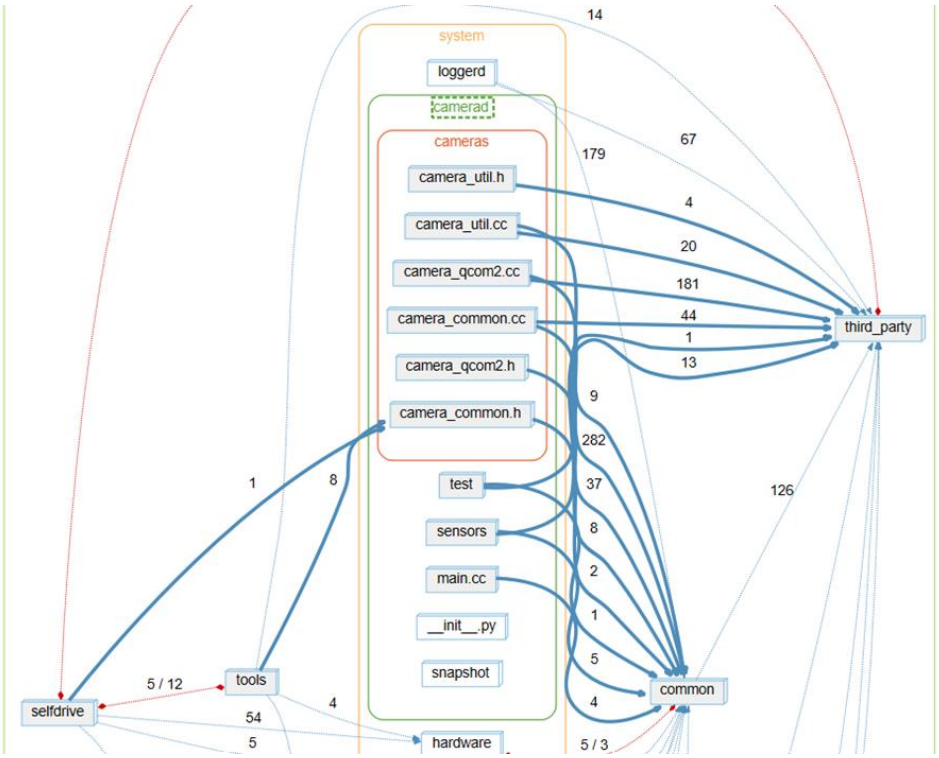

Figure 5: cameras structure

Limitations

When we examine how OpenPilot works, both in theory and in practice, we gain valuable insights into its design and how it operates. However, even though this report explores OpenPilot's concrete Architecture, there are some things we need to think about.

Top Level Architecture: Conceptual vs. Concrete Architecture

OpenPilot's structure has two sides: the conceptual architecture and how it actually works (concrete architecture). The conceptual ideas tell us about things like how it responds to events, controls processes, and is organized in layers. But turning these ideas into something real means making practical decisions to make it work in the real world. So, while the big ideas give us a good overview, the actual design of OpenPilot must balance these ideas with real-world needs.

Concurrency in OpenPilot

Parallel Processing of Sensor Data: Open Pilot receives data from various sensors, including cameras, GPS, and inertial measurement units (IMUs), all of which must be processed in parallel. This concurrent processing ensures that the system can make timely decisions based on a comprehensive understanding of the vehicle's current state and environment.

Asynchronous Event Handling: Through its event-driven architecture, Open Pilot can handle multiple events asynchronously. This means that the system doesn't need to wait for one operation to complete before starting another, enhancing its ability to respond to real-time changes in the driving environment.

Multithreaded Execution: Open Pilot employs multithreading, where different threads manage various tasks like sensor data processing, decision-making, and actuator control. This multithreaded approach allows Open Pilot to perform complex computations and actions concurrently, maximizing the system's efficiency and responsiveness.
Open Pilot is managed through a combination of parallel processing, asynchronous event handling, multithreading, effective inter-process communication, concurrency control mechanisms, and real-time scheduling.

Diagrams

Use Case Diagram

This diagram shows the interactions between the users and the camerad and sensord subsystems. Each use case within the OpenPilot system is a function that the system performs automatically after being engaged by the user. "Process Data" is a pivotal step necessary for the proper execution of the driving functions like "Adaptive Cruise Control" and "Automated Lane Keeping." The "Vehicle" at the bottom is the actor that ultimately carries out the commands resulting from these use cases.

Drive Vehicle: Takes control of the driving.
Adaptive Cruise Control: Adjusts car's speed to maintain a safe distance from other vehicles.
Automated Lane Keeping: Detects lane lines and keeps the car centered in its lane.
Driver Monitoring: Ensures the driver is attentive and ready to take control if needed.

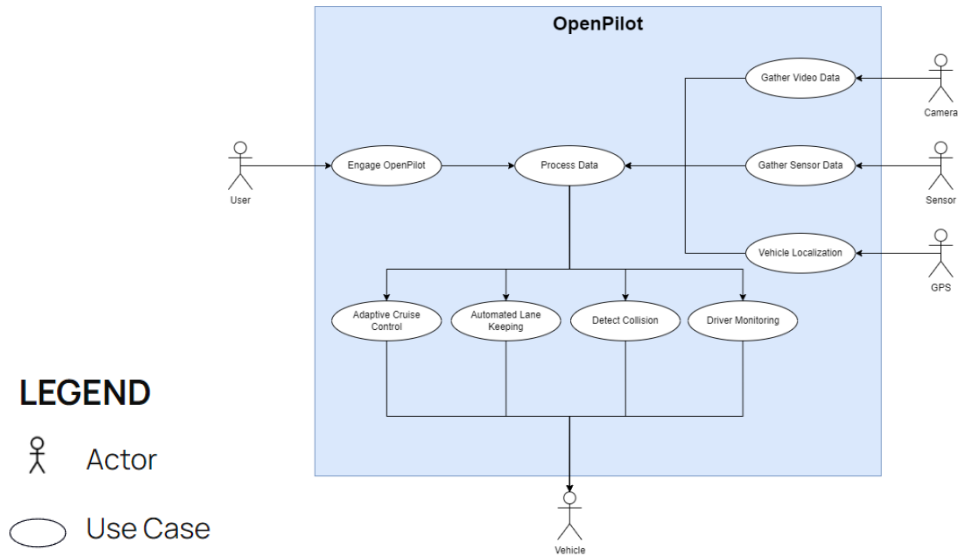

Figure 6: Use Cases Diagram

Sequence Diagram

This diagram illustrates the order of interactions between camerad, sensord, and other components within the OpenPilot system, outlining the data flow from the moment a sensor captures data to when a decision is made, and a command is executed.

User: The sequence begins with the user starting the OpenPilot system
OpenPilot: The central software system
camerad: Starts capturing video frames. This data is crucial to perform tasks like Automated Lane-Keeping.
modeld: Receives the camera frames and processes them. Applies machine learning to interpret road conditions, detect lanes, obstacles, and other relevant features.
sensord: In parallel, the sensord is activated to gather data from the vehicle's sensors. This includes data like speed and acceleration.
controlsd: Both the processed data from modeld and sensord are sent to controlsd. Here, the data is used to make driving decisions and generate control commands.
panda: This hardware interface takes the control commands and communicates to the car's internal systems.
Car: Executes the commands, which can include steering, braking, or accelerating.

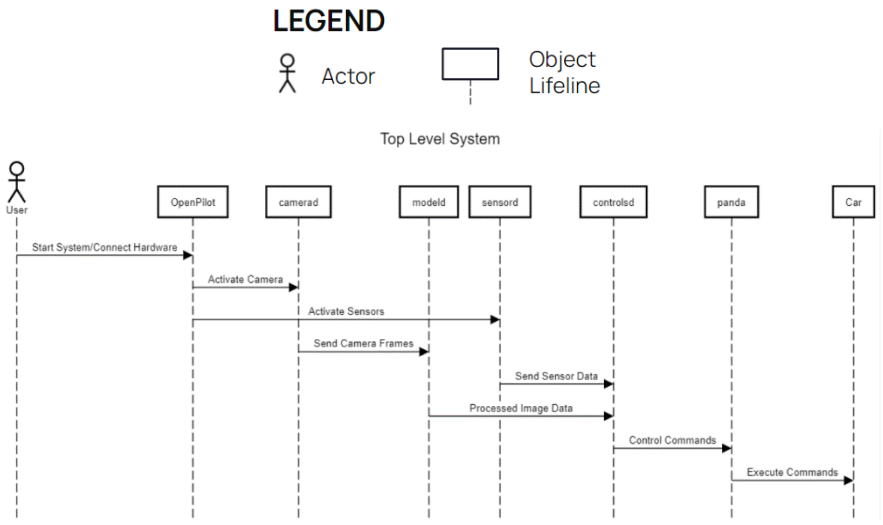

Figure 7: Sequence Diagram

State Diagrams

Camera

- The camerad subsystem of OpenPilot operates in a cycle to manage the video frames.
- Activates with user engagement, transitioning to configure the cameras.
- Once ready, it shifts to Capturing, recording the road, essential for real-time responses.

- The captured footage is then processed, utilizing machine vision algorithms and deep learning for object detection and classification.
- Post-processing, the subsystem enters a Relay state to transmit the analyzed data to OpenPilot's planning and control modules, influencing driving decisions.
- In the event of an error during any of these stages, an Error state enters.

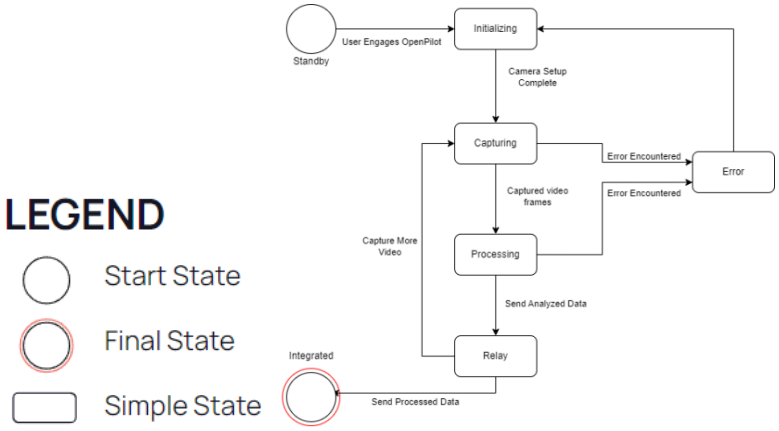

Figure 8: State *Diagram for Camerad*

Sensor

- The state diagram for the sensord subsystem within OpenPilot is like Camerad, only difference is including an additional step of combining the data from various sensors including Rader, GPS, Accelerometer, and more depending on the car make and model. Piling together all sorts of collected data into a coherent format and sent over
- Upon user engagement, transitions to Initializing, where sensors are calibrated for accurate data collection.
- Moving to Capturing, this actively collects data from various vehicle sensors.
- This data is combined into a coherent format before being sent out.
- In the Evaluate state, assess this fused data to determine the vehicle's current speed, position, and heading, which are critical metrics for driving assistance.

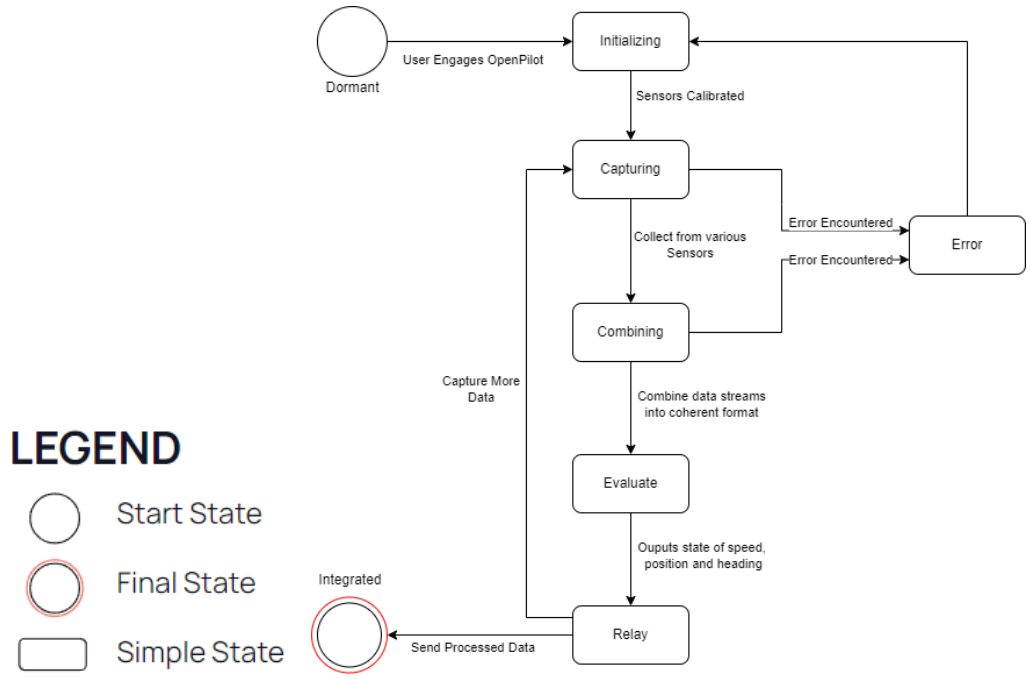

Figure 9: State Diagram for sensord

Data Dictionary

Key Words	Definition
Event	Represents an occurrence or action in the system
Sensor Data	Data collected from various sensors
Camerad Subsystem	Manages camera inputs, processing images for further analysis by the system
Camera QCOM2 Module	Adapts or enhances camera functionalities for Qualcomm-specific hardware
Camera Common Module	Central hub for camera operations, interfacing with selfdrive, tools, and other camera modules
Log	Recorded activity or events for debugging

Conclusion

This report provides an insightful exploration into OpenPilot's concrete architecture, looking at its significance in advancing driving technology. Using Understand, we extracted the architecture, organizing top-level entities into a coherent subsystem structure. By analyzing the application of design patterns and the architectural style, it reveals the system's robust framework and scalability. We have aimed to enhance the

readability and comprehension of OpenPilot's architectural style, contributing to a comprehensive understanding of its design and functionality.

References:

https://en.wikipedia.org/wiki/Openpilot
[1] “Openpilot documentation,” docs.comma.ai, https://docs.comma.ai/LIMITATIONS.html (accessed Feb. 13, 2024).
[2] “How openpilot works in 2021,” comma.ai blog, https://blog.comma.ai/openpilot-in-2021/ (accessed Feb. 13, 2024).
[3] “Openpilot documentation,” docs.comma.ai, https://docs.comma.ai/SAFETY.html (accessed Feb. 13, 2024).
[4] “How openpilot works in 2021,” comma.ai blog, https://blog.comma.ai/openpilot-in-2021/ (accessed Feb. 13, 2024).
[5] “Management layer,” Documentation:FR:Dep:MgmtLayer:Current - Genesys Documentation, https://docs.genesys.com/Documentation/FR/latest/Dep/MgmtLayer#:~:text=The%20Management%20Layer%20provides%3A,that%20records%20applications%20maintenance%20events. (accessed Feb. 13, 2024).
[6] An introduction to software architecture, https://userweb.cs.txstate.edu/~rp31/papers/intro_softarch.pdf (accessed Feb. 13, 2024).