EECS 4314

Assignment 3: Discrepancy Analysis Report

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

Contents

Abstract

This report analyzes the evolution of the OpenPilot autonomous driving system from its conceptual framework to concrete implementation. It contrasts the high-level architecture with the detailed construction of software components, emphasizing the integration of design patterns and the cereal messaging framework for efficient inter-process communication. Reflexion analysis uncovers the alignment and discrepancies between planned and actual architectures, highlighting the significance of subsystem dependencies and dynamic safety assessments through updated state diagrams. Challenges such as navigating a complex codebase and gaps in documentation are addressed, underscoring the importance of clear documentation and the adaptability of software architecture. The findings contribute insights into the development of large-scale software systems, emphasizing the need for effective communication and documentation in facilitating system evolution and team collaboration.

Introduction

The evolution of autonomous driving technologies has been marked by significant advancements, with OpenPilot by Comma.ai standing out for its open-source approach to democratization of this field. This report explores the development of OpenPilot, tracing its journey from an idealized conceptual architecture to its practical, concrete implementation. It highlights the transition from a high-level design, emphasizing scalability and modularity, to the detailed execution involving software design patterns, the cereal messaging framework for inter-process communication, and the integration of subsystems like camerad and modeld.

Through reflexion analysis, the report compares the envisioned and actual architectures, focusing on subsystem dependencies and the application of dynamic safety measures. It addresses challenges such as the complex codebase and gaps in documentation, which underscore the necessity of clear documentation and adaptability in software architecture. By examining these aspects, the report aims to shed light on the intricacies of developing open-source autonomous driving systems and the critical role of effective communication and documentation in facilitating their evolution.

Conceptual Architecture

The Conceptual Architecture of OpenPilot, as outlined from the Assignment 1 Report, presents an idealized vision of the system’s structure. It is layered to support separation of concerns and is built to facilitate an extensible, modular design that can

evolve over time. The architecture is conceptualized to handle various aspects of autonomous driving systems including hardware interfacing, sensor data processing, core decision-making logic, and user interaction. It is meant to capture the system's intended behaviors and interactions but does not delve into the minutiae of service interactions or the specifics of messaging frameworks.

Subsystems within this conceptual framework are interconnected in a manner that supports a publish-subscribe model, ensuring that components such as the Camerad and Sensord subsystems can operate with a degree of autonomy, publishing data that other parts of the system can subscribe to and act upon. This high-level design focuses on creating a scalable and maintainable system that can adapt to new requirements and technologies.

Concrete Architecture

In contrast, the concrete architecture depicted in the Assignment 2 Report describes the actual software components and their interactions within the implemented OpenPilot system. This architecture translates the conceptual layers into practical modules and classes, integrating design patterns such as Singleton and Factory to manage memory and sensor information more effectively. It reveals the essential role of the "cereal" module as the messaging specification framework that underpins inter-process communication across the system. This framework ensures that subsystems like "camerad" and "modeld" can publish and subscribe to messages, orchestrating tasks such as sensor input processing and vehicle control in a highly efficient and reliable manner.

The report details specific subsystems like the Camerad Subsystem, breaking it down into components that handle various camera operations, and shows how they interface with both internal modules and external systems. This concrete architecture demonstrates how the system's event-driven nature is realized through message passing and inter-module communication, with a focus on real-time responsiveness.

Summary of our Investigation Steps

1. **Conceptual Architecture Understanding:** We reviewed OpenPilot's architectural documentation and designs, gaining insights into its intended styles, patterns, components, and interactions.

2. **Key Component Identification:** We pinpointed crucial components, modules, and layers, such as the user interface, control algorithms, and sensor integration, and analyzed their expected interactions.

3. **Concrete Architecture Analysis:** We delved into OpenPilot's codebase, examining repository structures and documentation. We aligned actual code elements with conceptual components and scrutinized key modules and dependencies to ensure they matched the planned architecture. Special attention was given to core functionality files and those with significant integration roles.

4. **Monitoring Modifications to Files:** We focused on recently and frequently modified files to detect potential architectural deviations or inconsistencies.

5. **Discrepancy Identification:** We identified any missing or additional components in the implementation that diverged from the conceptual design, ensuring a comprehensive understanding of any architectural discrepancies.

Conceptual vs Concrete Architecture Differences

Absences

Messaging Framework: While the conceptual architecture indicates the need for a communication system between components, it doesn't specify the heavy reliance on a particular messaging framework like cereal. The concrete architecture reveals cereal's critical role in enabling inter-process communication.

Dynamic Dependencies: The conceptual architecture does not fully anticipate the dynamic and runtime-determined dependencies that arise from the publish-subscribe system implemented in the concrete architecture.

Divergences

Modularity and Interdependencies: The conceptual architecture emphasizes modularity without detailing how tightly coupled the components can become in practice. The concrete architecture shows the depth of these interdependencies, particularly how various components rely on the cereal messaging system.

Hardware Integration: The conceptual design does not fully articulate the extent of integration between software and hardware components (like the panda device and various sensors). The concrete architecture details these integrations, showing how hardware inputs are essential to the system's operations.

Reflexion Analysis

We were able to find some dependencies by looking through Openpilots github commit history and performing the 4Ws analyzation on them.

1. Camera → Sensor [8]

Which?	Camera (Openpilot/System/camerd/cameras/camera_qcom2.cc) depends on sensors(Openpilot/System/camerad/sensors/ar0231_registers.h)
Who?	Third-party developer(GitHub user: deanlee)
When?	Dec 7, 2023
Why?	This dependency was introduced to help improve code organization and clarity within the camerad module. The content of sensor2_i2c.h was moved to ar0231_registers.h reflecting the specific purpose of the file, which is to define register settings for the ar0231 sensor. Moving the file to the system/camerad/sensors directory aligns with a more logical and structured organization of files within the module. It separates sensor-specific configurations from other camera-related functionalities, making the codebase more understandable and maintainable.

2. Camera → python VisionBuf [9]

Which?	Camera (Openpilot/System/camerd/snapshot/snapshot.py)

	depends on Python VisionBuf class(Openpilot/cereal/visionipc)
Who?	Third-party developer(GitHub user: mitchellgoffpc)
When?	Jul 25, 2023
Why?	This dependency was introduced to help improve code readability and maintainability, and to align with best practices by utilizing attributes of the buf object instead of passing individual parameters. This change reduces the number of arguments the function requires and enhances its flexibility.

3. Car → carcontroller and carstate [10]

Which?	Car (Openpilot/selfdrive/car) depends on carcontroller and carstate (Openpilot/selfdrive/car/mock)
Who?	Third-party developer(GitHub user: adeebshihadeh)
When?	March 17th , 2024
Why?	This dependency was introduced to simplify and optimize the loading process. It aims to enhance code maintainability, optimize initialization processes, and facilitate testing by introducing mock implementations of car state and controller interfaces.

State Diagram 1: Assisted Lane Change

In the original Assisted Lane Change state diagram, the process flows directly from a blind spot check to the resulting actions, a binary choice of alerting the driver or proceeding with the lane change. The updated diagram introduces safety confirmation states prior to the lane change.

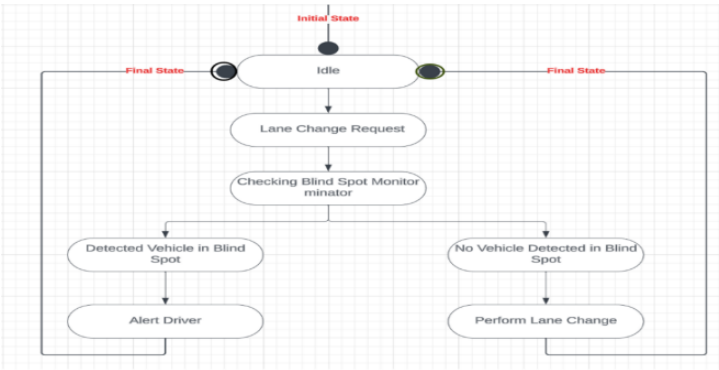

Figure : State Diagram from Conceptual Architecture

Unlike the initial version, the updated diagram shows an ongoing safety assessment, even after the lane change has started, demonstrating the system's commitment to continuous safety precautions. It also provides clarity on the aftermath of driver alerts, showing that the system's reaction is relied upon the driver's acknowledgement, thus forming a responsive feedback loop. Moreover, the revised diagram integrates additional system components, specifically camerad and sensord, indicating their critical roles in ongoing safety checks.

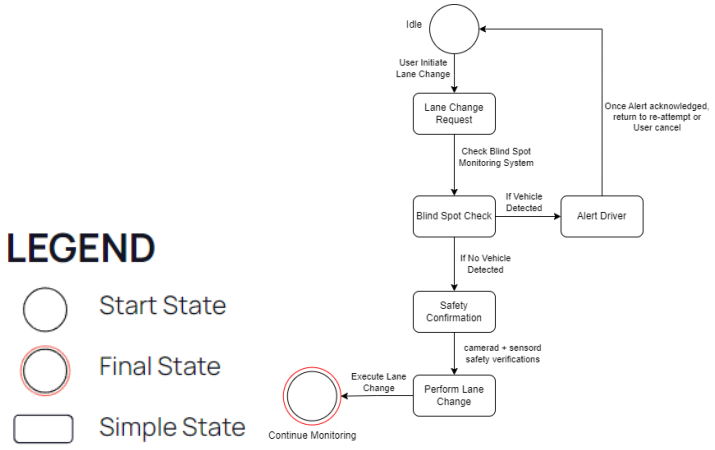

Figure : State Diagram Updated

State Diagram 2: Navigation on OpenPilot

The first version of the Navigation features' state diagram shows a basic path from start to finish. It's straightforward but skips over the system's complex behaviors. The new diagram provides a more detailed view, showing a thorough journey with added elements like sensors for detecting paths and cameras for watching the route. This matches the detailed and safe design of the system. It also shows that after reaching the destination, the system resets to "Idle," ready for another trip. This update captures the system's ongoing, repetitive functionality, a point the original diagram didn't cover.

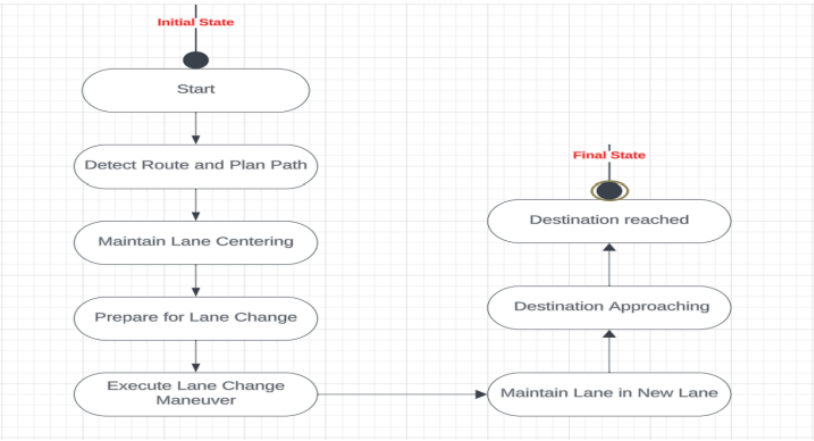

Figure : State Diagram from Conceptual Architecture

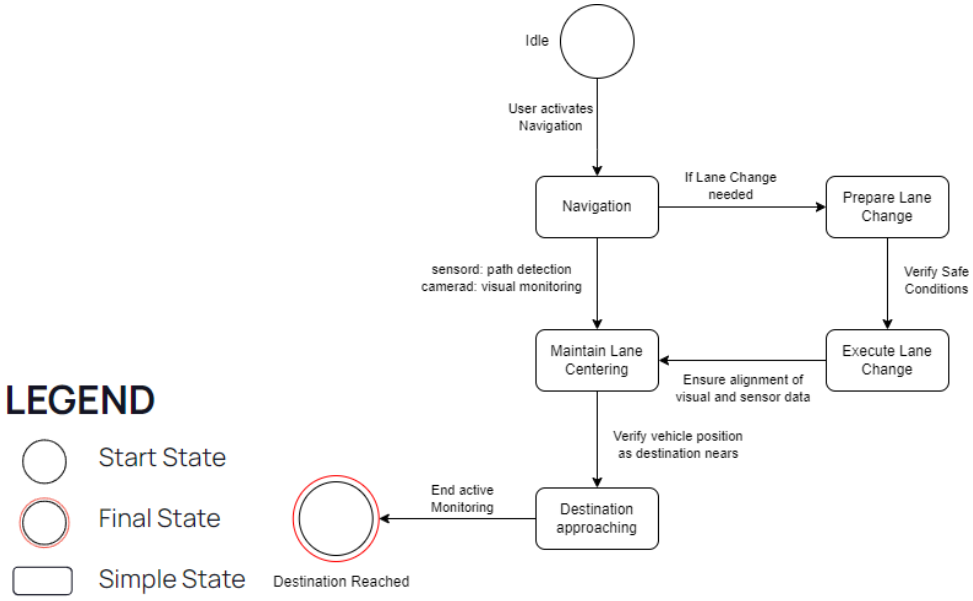

Figure : Updated State Diagram

Sequence Diagrams

The early diagrams gave a simple overview of how the system was supposed to work and interact, but they didn't show the detailed parts of the actual system that were later revealed in the concrete architecture. Essential components such as modeld, controlsd and panda weren’t included, leading to a simplified representation of the system. Moving to the concrete stage, these missing components became evident, highlighting the early diagrams needed to be updated to fully and accurately show how OpenPilot works.

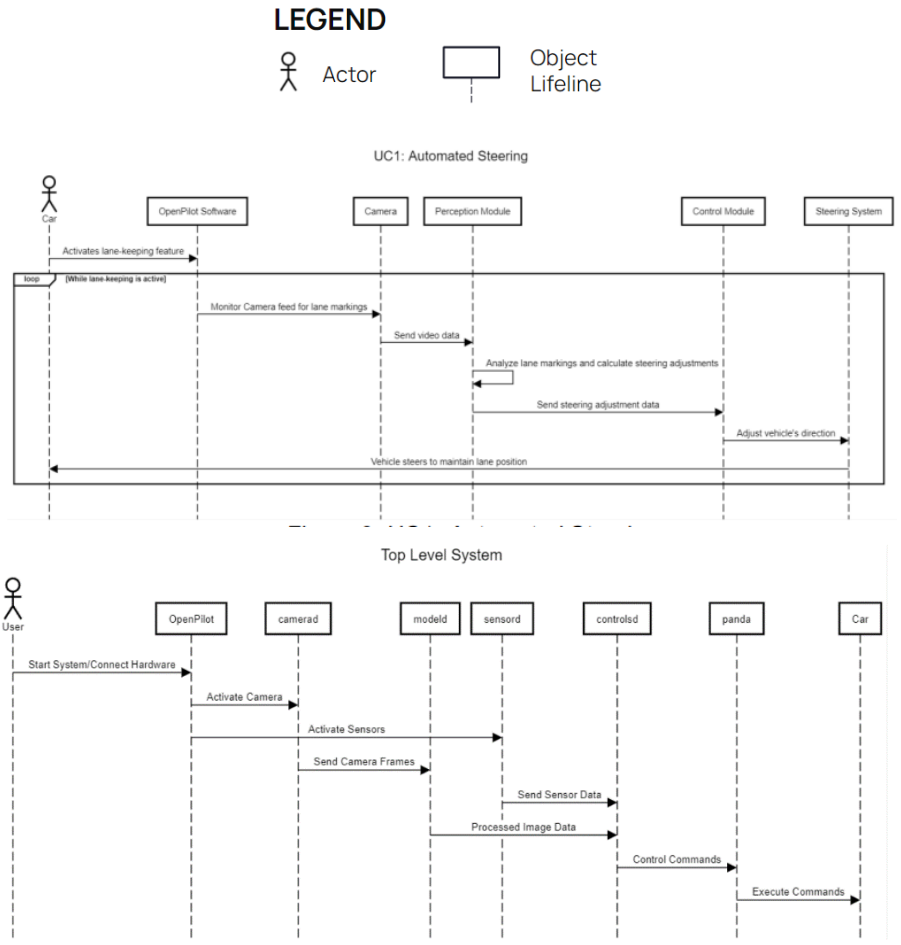

Figure : Top: Conceptual Architecture Sequence Diagram

Bottom: Updated Sequence Diagram

Use Cases

The use case diagrams derived from the concrete architecture of OpenPilot provide a high-level overview of the interactions between the users (actors) and the system. These diagrams focus on the intended functionalities and user goals without detailing the specific implementation mechanisms. These diagrams bridge the user requirements and the system's functional design. They remain unchanged even as the system undergoes architectural evolution from conceptual design to concrete implementation. The core use cases remain valid as they capture the essence of what the system is meant to do for its users.

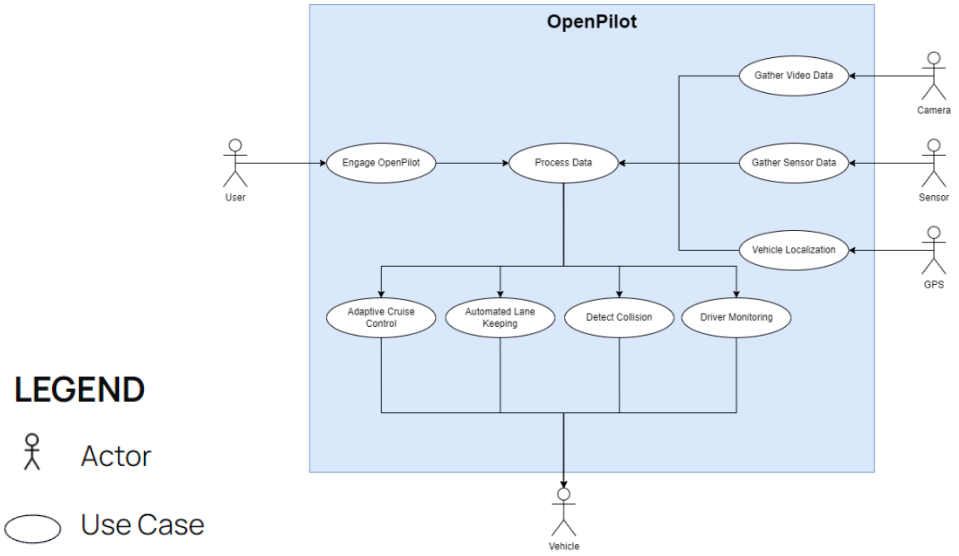

Figure : High-Level Use Case Diagram of OpenPilot

Concurrency in Camerad & Sensord:

Concurrency is present in our subsystem as both are essential to Openpilot. Camerad component deals with processing data from cameras on the vehicle. Concurrency in camerad allows it to handle multiple streams of camera data simultaneously. This means that camerad can process data from multiple cameras in real-time, which is essential for tasks such as object detection, lane detection, and obstacle avoidance. Sensord is responsible for managing data from various sensors installed in the vehicle, such as LiDAR, radar, GPS, and IMU (Inertial Measurement Unit). Concurrency in sensord enables it to handle data from these sensors concurrently. This allows

Openpilot to gather and process information from multiple sensors simultaneously, enhancing the system's perception capabilities and providing more accurate situational awareness. Here are some effects of Concurrency:

1. **Improved Performance**: Concurrency allows camerad and sensord to process data in parallel, which can significantly improve the overall performance of Openpilot. This is particularly important in real-time applications like autonomous driving, where timely processing of sensor data is critical for safe and efficient operation.
2. **Enhanced Robustness**: Concurrency can make Openpilot more robust against fluctuations and variations in sensor data. By processing data from multiple sensors concurrently, the system can cross-validate information and reduce the impact of sensor failures or inconsistencies.
3. **Increased Scalability**: Concurrency enables Openpilot to scale more effectively, as it can efficiently utilize computing resources to handle increasing amounts of sensor data. This scalability is crucial for supporting advanced features and accommodating future enhancements in autonomous driving technology.
4. **Real-time Responsiveness**: Concurrency ensures that Openpilot can respond to changes in the environment promptly. By processing data concurrently, the system can maintain real-time awareness of its surroundings and make timely decisions to navigate safely.

In summary, concurrency in camerad and sensord components of Openpilot plays a vital role in enabling real-time processing of sensor data, improving performance, robustness, scalability, and responsiveness of the autonomous driving system.

Limitations

During our analysis, we encountered several limitations that impacted the depth and clarity of our findings. One of the main issues was just how complex OpenPilot is. Its codebase is huge, filled with many layers of directories that were tough to get through. This made it hard to figure things out at times and we worried about possibly getting some of our findings wrong because of it.

Another big hurdle was the gaps in documentation. We often found that pull requests and commits weren’t documented as clearly as we needed. Sometimes, a new dependency would be added without any mention in the commit messages, or we'd

run into commit details that were missing or conversations between developers and reviewers that were hard to follow. This made it tricky to track the system's changes accurately.

The rapid changes in OpenPilot's system architecture also added to our challenges. To really understand the modifications over time, we had to look closely at a series of commits. Doing this was harder than expected, especially with the documentation issues we were already facing. It showed us how fast the system was changing, but also how tough it can be to keep a clear picture of the architecture’s evolution over time.

Lessons Learned

From all this project, we gained some valuable insight into large scale software developments and the importance of understanding the architecture of a system. We saw that there was a bigger gap between what we planned in the conceptual architecture and what actually happened in the concrete implementation than we expected. Our analysis revealed that divergences were more common than absences of planned components. This indicates that while the foundational aspects of the conceptual architecture were generally incorporated into the concrete architecture, their practical implementations often took different forms to accommodate real-world complexities.

Diving into the commit logs, pull request conversations, and manually checking the changes gave us a better view of why certain decisions were made. It also gave us insights into the practical challenges of large scale software development and the evolution of these systems over time. One of the biggest takeaways was just how important good documentation is. It's essential for keeping the team aligned and making it easier for new members to understand the architecture and contribute effectively. We also learned the value of detailed commit logs and the need for clear documentation of pull requests. These practices are key for understanding how the system evolves, why new dependencies are introduced, and keeping a clear record of the project's development.

Looking back at these challenges and what we learned from them, it’s clear that analyzing the differences between OpenPilot's conceptual and concrete architectures gave us a detailed look into the practical side of software development, offering

lessons that are valuable for planning, documenting, and managing projects in the future.

Conclusion

In summary, this report's discrepancy analysis between OpenPilot's conceptual and concrete architectures illuminates the evolution from design to deployment. By focusing on both a top subsystem level and a detailed subsystem design, we've pinpointed and rationalized discrepancies, suggesting modifications to both architectures to improve alignment and efficiency. These recommendations aim to optimize OpenPilot's functionality, reflecting on the dynamic nature of software development where adaptation is key.

The analysis underscores the importance of flexibility, clear documentation, and the iterative refinement process in software architecture. Lessons learned from navigating the discrepancies highlight the value of an integrated design approach that accommodates evolving requirements and emerging technologies.

This report not only meets the assignment's objectives by detailing discrepancy analysis and mitigation strategies but also contributes valuable insights into software development practices. Embracing these lessons can enhance the development of sophisticated systems like OpenPilot, advancing the capabilities of autonomous driving technology.

References:

[1] “Openpilot documentation,” docs.comma.ai, https://docs.comma.ai/LIMITATIONS.html

[2] “How openpilot works in 2021,” comma.ai blog, https://blog.comma.ai/openpilot-in-2021/ .

[3] “Openpilot documentation,” docs.comma.ai, https://docs.comma.ai/SAFETY.html .

[4] “How openpilot works in 2021,” comma.ai blog, https://blog.comma.ai/openpilot-in-2021/ .

[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.

[6] An introduction to software architecture, https://userweb.cs.txstate.edu/~rp31/papers/intro_softarch.pdf

[7]https://en.wikipedia.org/wiki/Openpilot

[8] https://github.com/commaai/openpilot/commit/bdf868ddc276e2cf8a644376afe1a9634bdf06fe
[9] https://github.com/commaai/openpilot/commit/f0ae0c34cda2d032bc270efe6360a1dd1bbd77f2
[10] https://github.com/commaai/openpilot/commit/afc96972c80743378ad0e091d9dca8c216b09298