Discrepancies in OpenPilot's Conceptual and Concrete Architectures

Jerry Wu (jerrywu0@my.yorku.ca), **Joseph Spagnuolo** (joe13@my.yorku.ca), **Tarun Bhardwaj** (tarunb4@my.yorku.ca), **Krishna Raju** (krishnar@my.yorku.ca), **Irsa Nasir** (inasi022@my.yorku.ca), **Stanley Ihesiulo** (ihesiulo@my.yorku.ca)

Due April 10 2024

Contents

Abstract

1 Introduction and Overview

This report embarks on an analytical journey to dissect and understand the nuances between the conceptual and concrete architecture of OpenPilot, a leading-edge autonomous driving system. It aims to provide details on the discrepancies that have emerged as the project transitioned from conceptual to concrete, offering a detailed examination of both the overarching framework and the intricacies of specific subsystems. By undertaking a reflexion analysis, this report not only identifies the divergences at the top level but also delves deeper into the detailed design level for localization and calibration, providing a granular perspective on the differences encountered.

The structure of this report is organized to facilitate a clear and comprehensive understanding of our findings. It begins with a comparison of the top-level architecture, breaking down the conceptual subsystems and their interactions. It compares these to the concrete architecture. This report also discuses the file architecture and the overall style of OpenPilot's architecture, laying the groundwork for a thorough investigation of the discrepancies uncovered.

Subsequently, the report narrows its focus to a critical analysis of localization and calibration, comparing the second-level subsystems and interactions as envisioned versus as executed. Through specific use cases, we explore the practical challenges and deviations that arose during the implementation phase.

To continue, the report addresses these discrepancies and proposes thoughtful modifications to both the conceptual and concrete architectures. It articulates the rationale behind these suggestions, aiming to align OpenPilot more closely with its original vision while also enhancing its functionality and efficiency. The proposed changes are discussed with a forward-looking perspective, considering their implications for the system's future development.

The report discusses the mitigation of identified discrepancies and the lessons learned through this reflective process. It provides valuable insights into the importance of adaptability and continous refinement in the architeural design of complex systems like OpenPilot. Through its structured analysis and proposed solutions, it contributes to the ongoing dialogue on best practices in software architecture, inviting readers to engage with the findings and participate in the collaborative process of innovation and improvement.

2 Top Level Architecture Comparison

2.1 Conceptual Architecture Subsystems/Interactions

In general, OpenPilot can be depicted as a layered style architecture which contains 8 layers.

Layer 1: Vehicle Interface

Layer 2: Communication and Data Interpretation

Layer 3: Sensing and Actuation

Layer 4: Core Neural Network Runners

Layer 5: Localization and Calibration

Layer 6: Control Algorithms

Layer 7: System Management and Logging

Layer 8: User Interface and Experience

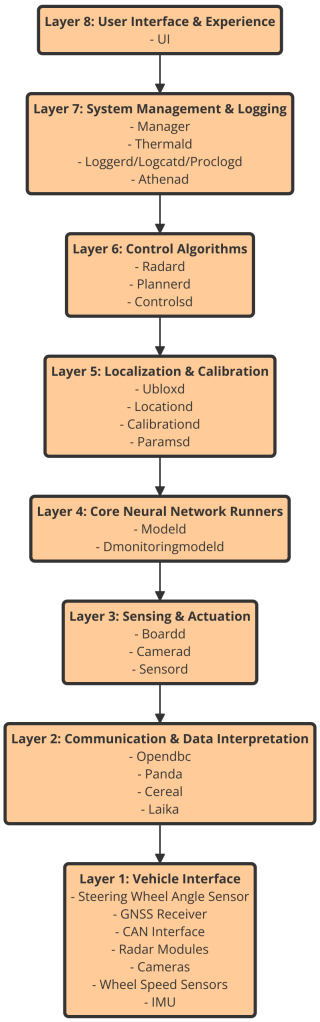

Figure 1: A bottom up diagram detailing the conceptual architecture of openpilot

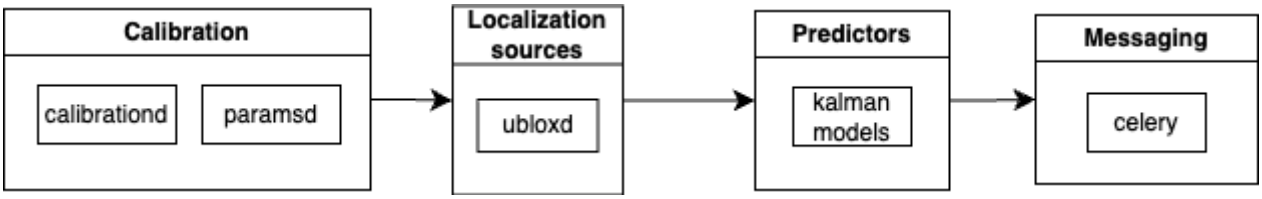

Figure 2: The top level conceptual architecture

- The **vehicle interface layer** includes the car's hardware and peripherals that interface with OpenPilot. This includes **CAN** interface which enables communication with the car's internal network, **steering wheel angle** sensor measures angle of steering wheel, **camera** provides visual input around the vehicle, **IMU** also provides movement and orientation information, **GNSS Receiver** provides global positioning data, and along with other vehicle sensors that are used to ensure the vehicle is functioning properly [1].

- The **communication and data interpretation layer** consists of libraries like opendbc which interprets CAN bus data based on **DBC** files, **panda** which is hardware that is used to read/write from and to the CAN bus, **laika** which processes **GPS** data for more precise positioning, and **cereal** which is the message protocol for inter-process communication [1].

- The **sensing and actuation layer** includes **boardd** which manages communication between the panda and the openpilot software, **camerad** which processes image data from the car's cameras, and **sensord** which handles IMU and additional sensor data reading. All together this layer manages the direct interaction with the car's hardware and the data from the sensors [1].
- The **core neural network runners layer** includes **modeld** which runs the driving policy in order for the neural network to make driving decisions, **dmonitoringmodeld** which runs the driver monitoring neural network to ensure that the driver is paying attention, **dmonitoringd** which contains logic to see whether the driver can take over the wheel if necessary. Overall, this layer runs the neural networks which are responsible for driving and driver monitoring [1].
- The **localization and calibration layer** includes **ubloxd** which parses GNSS data for localization, **locationd** which uses a Kalman filter service to provide precise vehicle localization, **calibrationd** which calibrates the camera to align it with the vehicle's frame, and **paramsd** which adjusts the vehicle parameters for more accurate control [1].
- The **control algorithms layer** uses **radard** which processes radar data for detecting objects and tracking them, **plannerd** which laterally and longitudinally calculates the path planning, and **controlsd** which generates and sends the control commands to the actuators of the vehicle [1].
- The **system management and logging layer** contains **manager** which oversees the lifecycle of openpilot services, **thermald** which monitors the thermal state of the device and ensures it does not overheat, **loggerd** which records driving and analyses that data for improvements, **logcatd** which logs system messages and errors, **proclogd** which logs detailed process data, and **athenad** which manages the connection to common.ai services for updates and any remote assistance [1].
- Lastly, **the user interface and experience layer** has a **UI** which is in charge of displaying the interface for the user, including a live camera feed, system alerts and status [1].

2.2 Concrete Architecture Subsystems/Interactions

2.2.1 File Architecture

In order to find out the architecture of openpilot, it is necessary to look at the file architecture and dependencies within Understand.

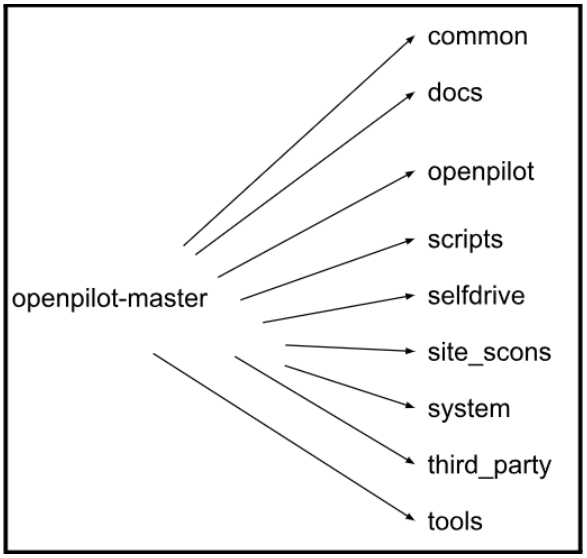

Figure 3: A recreation of the Understand graph detailing the subfolders within the openpilot project

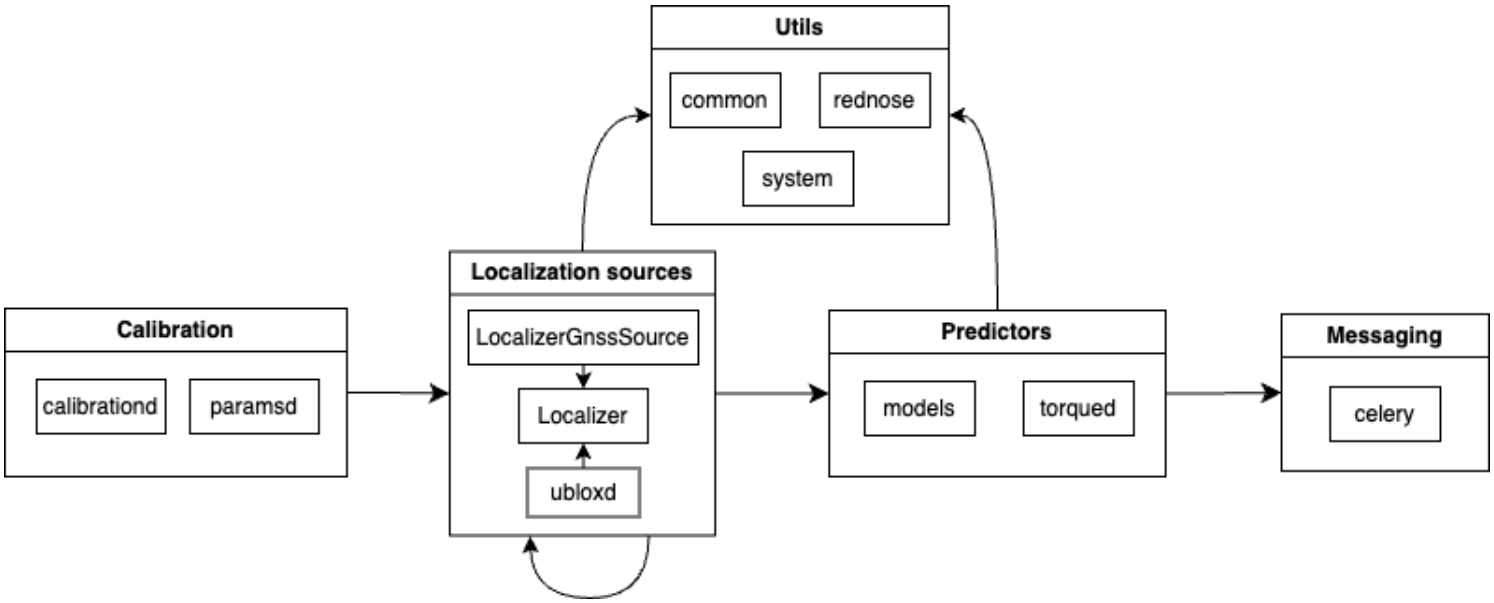

Figure 4: The top level concrete architecture

After dissecting the architecture of the overall software system, the group figured out that the main subsystems that made up openpilot's functionalities were: common, third_party, tools, system, and selfdrive.

Their individual functions include:

- common - A common place where many of the subsystems call upon its functionalities. We can see in the dependency diagram for common, selfdrive, system, and tools all depend on functions within common.
- third_party - External third party tools such as JSON and Kaitai parsing tools.
- tools - Contains shell scripts for setting up openpilot on different operating systems. As well as functionalities for running the comma device and other peripheries.
- system - Provides many services for setting up system attributes such as: reading audio measurements, setting the time and timezone, publishing log messages. Also contains subfolders that perform important driving functionality including:
  - Camerad – Captures video [5]
  - Loggerd – Records driving and analyzes data [5]
  - Sensord – Sensor reading [5]
  - Ubloxd – Handles GNSS data (parses for location services) [5]
- selfdrive - Handles self driving functionality, such as transforming the video with the neural network, provide localization based on data from the System subfolder, and facilitate communication between the panda and the software. Also contains subfolders that perform important driving functionality including:
  - Locationd – Provide localization (using data from ubloxd) [5]
  - Boardd – Communication between panda and openpilot software. Acts as publisher for messages [5]

– Modeld – Transforms the video with the neural network [5]

– Thermald – Monitor thermal state of the device running openpilot. Additionally, monitors other properties such as CPU and RAM usage, as well as power status [5]

2.2.2 Architecture Style of OpenPilot

The group came to the conclusion that the architectural style that best suited the entire openpilot system is a **layered** architectural style. This conclusion was reached after looking at dependency diagrams in Understand.

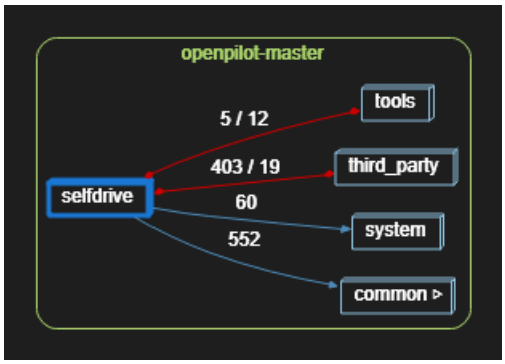

Figure 5: Selfdrive's dependency graph from Understand. Here it is apparent that there are two way dependencies between selfdrive and third_party, as well as a one way dependency towards common

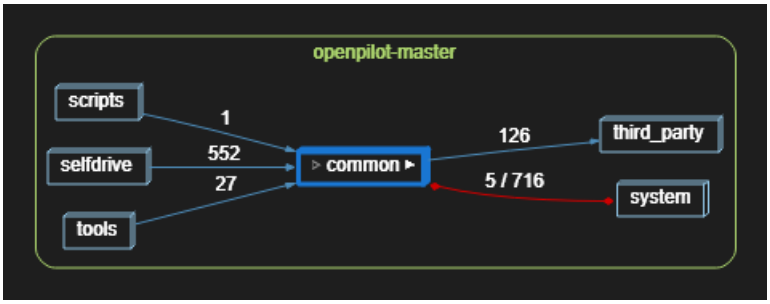

Figure 6: Common's dependency graph from Understand. There is a one way dependency between common and third_party.

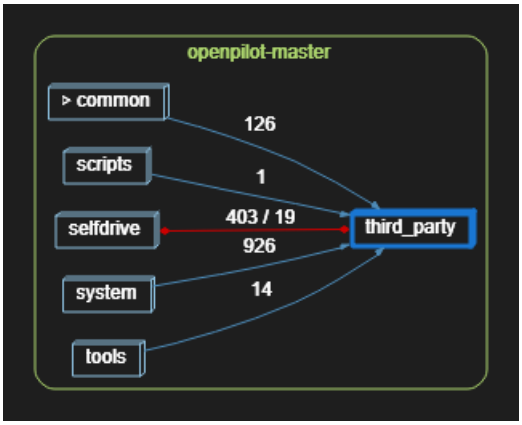

Figure 7: third_party's dependency graph from Understand. Here the dependencies between common and selfdrive

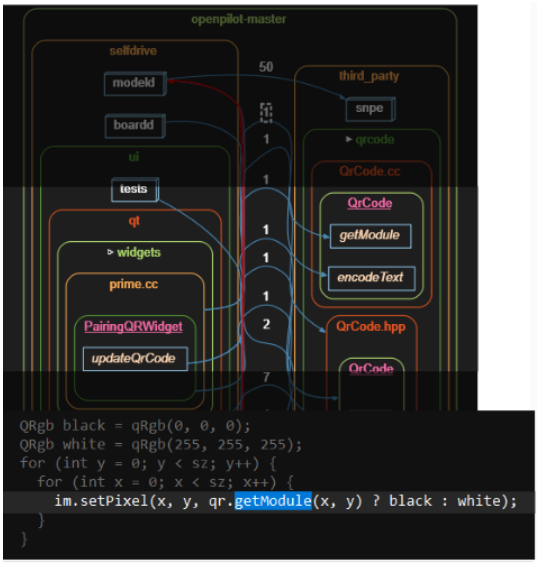

Figure 8: One of the dependencies between selfdrive and third_party. Within selfdrive, the updateQrCode calls on the getModule method within third_party

Within Figure 2, 3, and 4, as well as the file architecture of the whole system, there is clear segmentation between subsystems based on functional purpose. Additionally, looking at the dependencies, certain subfolders, such as selfdrive have dependencies on folders such as common and third_party. From here it is possible to model selfdrive as a higher layer than third_party with common as an abstraction layer in between. However, as seen with selfdrive's direct dependency on third_party, the system allows for non-adjacent communication, as seen in Figure 5.

It is important to note, that the layered architecture the group came up for the conceptual design does not match the concrete architecture seen in the Understand file. It is an important lesson that conceptual models of software often do not capture the complexities and nuances of an actual software architecture, and while they remain important frames of reference, it is often the case that the needs of the software, whether they be for performance or any other reason, will stray away from these architectural models.

In this case, although it was stated that selfdrive was a higher layer separated from third_party by common, there is still two way communication between the two that is non adjacent, meaning the previous conceptual model did not capture that relationship, and should be changed to accommodate it.

2.3 Discrepancies Between Concrete and Conceptual Architecture

Between the conceptual and concrete architectures for the top level system there are many discrepancies that can be found.

1. **Layering vs. Modularity:**

   The conceptual architecture suggests a layered approach where each layer has a clear function, and upper layers depend on lower layers. However, the dependency graphs and directory structure indicate a more modular approach where certain directories (like common) have widespread dependencies, suggesting that the layering may not be as strict in practice.

2. **Centralization of Common Code:** The common directory in the concrete architecture seems to be a central hub that other components depend on, which was not outlined explicitly in the conceptual architecture. This could be an indication of shared utilities or core functions that many parts of the system use.

3. **Presence of Third-party Components:** The conceptual architecture may not detail how third-party components (reflected in the third_party directory) integrate with the rest of the system. In the concrete architecture, these dependencies are explicitly shown, which is important for understanding how external code affects the system.

4. **Detail of Inter-component Communications:** The dependency graphs show the specific interactions between components, such as the number of connections, which are often not depicted in conceptual diagrams including ours. This level of detail can expose potential areas of high coupling that might require refactoring.

5. **Direct vs. Transitive Dependencies:** In the concrete architecture, some components depend on others indirectly through transitive dependencies. This can obscure the understanding of the system's structure if the conceptual model only shows direct dependencies.

2.4 Reflexion analysis

which?	LocalizationSources -> LocalizationSources `Localizer::locationd_thread` (in `selfdrive/locationd/locationd.cc`) depends on `ubloxd.main` (in `system/ubloxd/ubloxd.cc`)
who?	haraschax
when?	August 28, 2023 PR: #29687 PR title: Locationd: enable laikad
why?	Using `ubloxd` to provide a `LocalizerGnssSource`. `ubloxd` is not directly depended on by `locationd`, but `ubloxd` publishes its data to `celery`, and `locationd` subscribes to this data stream.

Figure 9: Reflexion analysis of top level using the 4 Ws model

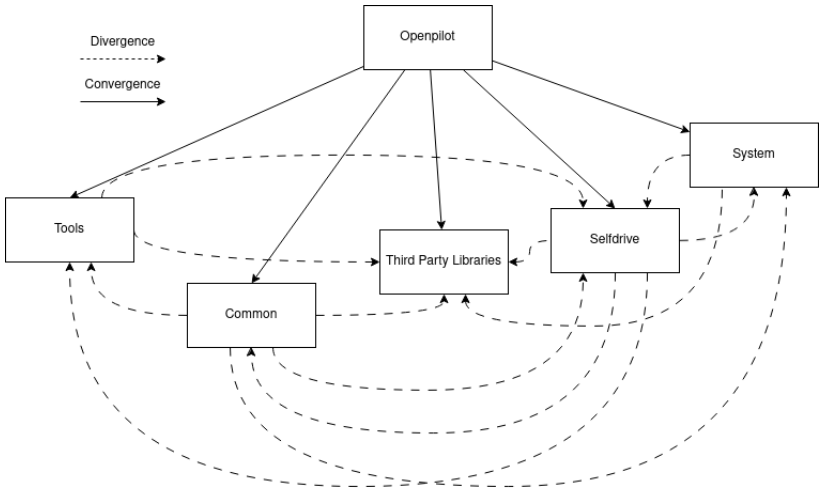

Figure 10: Reflexion analysis diagram of top level

2.5 Differences of Absences and Divergences of Top Level

Module	Diverges to
Tools	Third Party Libraries, Self Drive
Common	Tools, Third Party Libraries, Self Drive, System
Third Party Libraries	None
Selfdrive	Tools, Third Party Libraries, Self Drive, System
System	Third Party Libraries, Self Drive

Figure 11: A table summarizing the differences of absences in the top level subsystems

2.6 Proposed Modifications and Rationales

A possible modification to minimize discrepancies is to change the conceptual architecture to match the concrete architecture. This should be done in the interest of eliminating unnecessary layers from usages between modules. By doing this modules can directly access other modules without having to go through the Openpilot parent module. Doing this will ensure that communication happens quicker and more directly which is beneficial especially in the context of this system which is extremely time sensitive. The objective should be to process data and return results as quickly as possible. Therefore keeping these divergences and changing the conceptual architecture will improve the system.

3 Localization and Calibration Comparison

3.1 Conceptual Architecture Second Level Subsystems/Interactions

- **Repository Style Access**: Within the conceptual architecture, the principle of Repository Style Access is manifested through the presence of a central module Locationd. This module serves as a central repository responsible for managing and orchestrating all the other modules within the subsystem. Acting as a central point for modules to access and interact, the module plays a crucial role in facilitating seamless communication and coordination between various system components. Through this centralized approach, the system gains enhanced organization and control over module deployment and utilization. Moreover, the Repository Style Access model promotes consistency and standardization in accessing system resources, ensuring that modules can efficiently locate and utilize the functionalities they require. By consolidating module management under a single entity, the system architecture achieves a higher level of cohesion and modularity, thereby enhancing its scalability and maintainability. Ultimately, the adoption of Repository Style Access contributes to the overall robustness and efficiency of the system, enabling smooth operation and effective utilization of system resources.

- **Modularity**: In the system's conceptual architecture, Modularity is heavily utilized. It assists in structuring the system into independent modules, each encompassing a set of functionalities. This approach facilitates the organization and management of the system by breaking it down into manageable units, thus reducing complexity and enhancing development, maintenance, and scalability. Moreover, in the conceptual architecture each module has a well-defined interface with the central Module Locatond, ensuring strong communication and interaction between modules through Locationd.

3.2 Concrete Architecture Second Level Subsystems/Interactions

- **Publisher-subscriber**: In the context of the system, the Publisher Subscriber model is implemented and serves as a pivotal mechanism for both receiving and distributing various location and sensor data. This model operates on the principle of decoupling, where publishers are responsible for broadcasting data while subscribers selectively receive the information they require. Publishers within the system are tasked with generating and transmitting data streams originating from multiple sources, including location services and various sensors. Conversely, subscribers possess the capability to subscribe to specific data streams relevant to their respective functionalities or processes. Through this model, the system achieves flexibility and efficiency, allowing different components to interact seamlessly while accommodating diverse data requirements.

- **Direct Module Communication**: In the system Direct Module Communication describes the capability of specific modules to interact directly with one another, bypassing the need for an intermediary module. This approach enables streamlined communication pathways, where modules can access and utilize the features or functionalities of other modules as required, without introducing unnecessary layers of abstraction. By eliminating the reliance on intermediary modules, the system benefits from enhanced efficiency and responsiveness. This direct interaction fosters a more agile and adaptable system architecture, where modules can seamlessly collaborate and exchange information, ultimately contributing to improved performance and flexibility. Moreover, Direct Module Communication facilitates quicker decision-making processes and facilitates rapid responses to dynamic system conditions, aligning with the overarching goal of optimizing system functionality and responsiveness.

3.3 Use cases

Because use cases are a higher level concept that is relatively disjoint from the architecture of a system, they pretty much remain unchanged when transitioning from the conceptual architecture to the concrete architecture. Below is an example use case illustrating the user requesting navigation assistance from `locationd`. The `locationd` module will receive data from the GPS, whereby it will process the data and send the location data back to the user via the software's user interface.

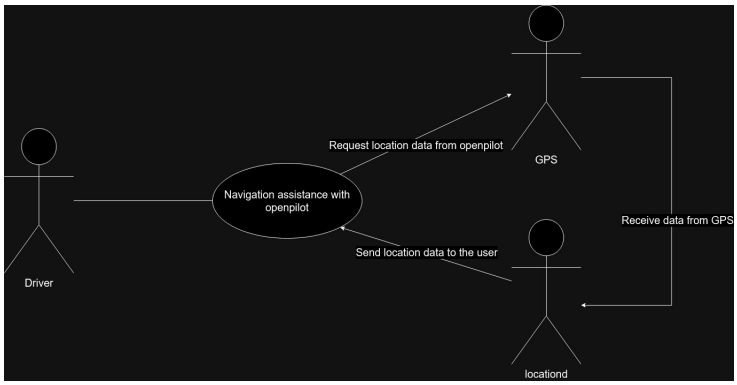

Figure 12: Navigation assistance use case for `locationd`

3.4 Discrepancies Between Concrete and Conceptual Architecture

Key points of discrepancy that can be inferred may include:

- The repository structure may not neatly organize files and directories according to the conceptual layers.

- Some software modules may perform functions that span multiple layers in the conceptual architecture.
- There may be additional utility or helper modules in the repository that are not represented in the conceptual model.
- The naming conventions in the repository might not directly reflect the names used in the conceptual architecture.

3.5 Reflexion analysis

which?	Models -> Utils: `predict_and_observe` (in `selfdrive/locationd/models/live_kf.cc`) depends on `Estimate` (in `rednose/helpers/ekf_sym.h`)
who?	deanlee
when?	August 24, 2023. PR: #28719 PR title: locationd: passing eigen objects by reference
why?	`Estimate` is a class defined in the `rednose` module. It is being used statically by the `models` module of `locationd` as a return type - it is used as a data structure to store the return value. No data passing between modules takes place.

Figure 13: Reflexion analysis of models using the 4 Ws model

which?	Models -> Utils `generate_code` (in `selfdrive/locationd/models/car_kf.py`) depends on `gen_code` (in `rednose/helpers/ekf_sym.py`)
who?	pd0wm
when?	Feb 26, 2020 PR: #1123 PR title: Kalman filter to identify vehicle parameters
why?	`gen_code` is a helper function defined in the `rednose` module. It is used in the `models` module of `locationd` to generate some code, as a string, and write it to the specified folder provided in its input parameter. This was done as part of the implementation of kalman filter to identify vehicle parameters. This utility function `gen_code` persists no internal state, but only operates on its input - no data passing between the two modulesis involved, just computation

Figure 14: Reflexion analysis of rednose using the 4 Ws model

which?	LocalizationSources -> Utils: `get_position_geodetic` (in `selfdrive/locationd/locationd.cc`) depends on `ecef2geodetic` (in `common/transformations/coordinates.cc`)
who?	jwooning
when?	April 20, 2021 PR: #20622 PR title: convert locationd to c++
why?	This change was made when converting this file to c++. `ecef2geodetic` is a converter function that converts from ECEF to geodetic. it is essentially a helper function.

Figure 15: Reflexion analysis of ecef2geodetic using the 4 Ws model

which?	LocalizationSources -> Utils: `handle_sensor` (in `selfdrive/locationd/locationd.cc`) depends on `SENSOR_TYPE_ACCELEROMETER` (in `system/sensord/sensors/constants.h`)
who?	jwooning
when?	April 20, 2021 PR: #20622 PR title: convert locationd to c++
why?	`SENSOR_TYPE_ACCELEROMETER` is just a constant defined in the `system` submodule, and is being used in the `locationd` submodule.

Figure 16: Reflexion analysis of ubloxd using the 4 Ws model

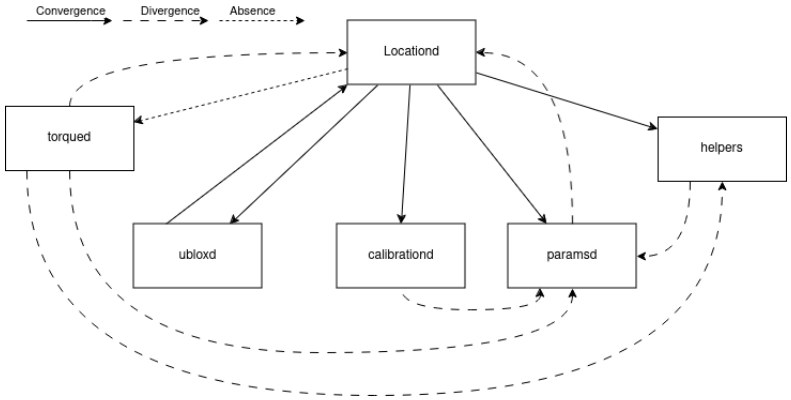

Figure 17: Reflexion analysis diagram of locationd

3.6 Differences of Absences and Divergences of Localization/Calibration

Module	Diverges to
Torqued	Locationd, Paramsd, helpers
Ubloxd	None
Calibrationd	Paramsd
Paramsd	Locationd
Helpers	Paramsd

Module	Absent From
Torqued	Locationd
Ubloxd	None
Calibrationd	None
Paramsd	None
Helpers	None

Figure 18: Tables summarizing differences of absences in locationd and ubloxd

3.7 Proposed Modifications and Rationales

A way we can minimize discrepancies here is to change the conceptual architecture to match the divergences in the concrete architecture, and accept the absence and modify the conceptual architecture. This change

should be taken to streamline the communication process between modules by eliminating unnecessary layers. By enabling direct access between modules, rather than routing through the Openpilot parent module, communication will occur more swiftly and efficiently. This is particularly advantageous given the time-sensitive nature of the system. The primary goal is to expedite data processing and result delivery. Thus, addressing these discrepancies and adjusting the conceptual architecture will enhance the overall system performance. Regarding the access it is unnecessary for Locationd to access torqued because torqued is only used by paramsd and helpers module. This requirement is satisfied if the divergences in the reflexion analysis are accepted. So this absence can be safely accepted.

4 Mitigation of Discrepancies

4.1 Locationd

In general there are a number of ways in which the number of discrepancies between the conceptual and concrete architecture for locationd can be mitigated. Some of these include, but are not limited to:

- **Model Driven Development**: When designing a new feature, it is important to establish patterns and conventions for how the feature or component should function and what its purpose is. Some of these conventions include:
  - **Design patterns**: Developers should use commonly known design patterns among other developers so as to maintain consistency throughout the system and minimize dependencies.
  - **Coding conventions**: Coding style and documentation should follow a consistent style in order to ease maintainability and enhance readability
  - **Architectural guidelines**: Documentation should not gloss over any of the important details for the underlying architecture of a system and should include explanations for all top level systems as well as lower level ones.
  - **Code refactoring**: People have different coding styles, especially in an open source project where anyone around the world can contribute to it. This may lead to inconsistencies in coding styles, thus making it harder to analyze the source code effectively. In order to minimize these inconsistencies, we can do the following:

5 Discussion and Lessons Learned

While analyzing the similarities and discrepancies between the conceptual and concrete architectures of the system as a whole, and in depth for `locationd` and `ubloxd`, we were able to deduce a number of key learning points that we took away from the analysis.

- **Iterative Development Helps Identify Discrepancies Early**: Developers should aim to adopt an iterative development approach to allow for early detection of discrepancies and timely adjustments to the concrete architecture.
- **Prototyping Facilitates Validation**: Use prototyping to validate critical aspects of the conceptual architecture early in the development process, helping to identify and address discrepancies before significant resources are invested.
- **Continuous Integration and Deployment Promotes Alignment**: Implement CI/CD pipelines to automate the integration and deployment of changes, facilitating frequent validation of the concrete architecture against the conceptual architecture.

6 Conclusion

In summarizing our exploration of the differences between OpenPilot's planned and actual architectures, we've uncovered valuable lessons on the importance of constant reevaluation in design processes. Our detailed analysis at both broad and specific levels revealed where and why gaps emerged between the conceptual and concrete stages. We proposed strategic adjustments to both architectures, underlining the necessity for adaptability in the face of evolving technological requirements. These proposals aim to reconcile the initial vision with practical outcomes, focusing on enhancing functionality and efficiency. The logic behind these adjustments is deeply informed by an understanding of OpenPilot's goals and the tech landscape, emphasizing our dedication to aligning theory with practice effectively. This report not only suggests ways to improve OpenPilot but also offers broader insights into the critical role of ongoing refinement in software architecture. By sharing our findings online, we hope to spark community discussion and collaborative improvement.

Looking ahead, we recommend prioritizing the assessment of these changes and staying open to further refinements, ensuring OpenPilot remains responsive to future challenges and opportunities. This includes establishing mechanisms for continuous feedback from users and developers, which can provide real-world insights into the system's performance and areas for enhancement. Additionally, fostering a culture of innovation within the development team will be crucial, encouraging the exploration of new ideas and technologies that can contribute to OpenPilot's evolution.

References

[1] "How openpilot works in 2021," comma.ai, https://blog.comma.ai/openpilot-in-2021/ (accessed Feb. 13, 2024).

[2] S. Anumakonda, "The state of comma.ai," Medium.com, https://srianumakonda.medium.com/the-state-of-comma-ai-2140aabc6f52 (accessed Feb. 13, 2024).

[3] "openpilot 0.9.5," comma.ai, https://blog.comma.ai/095release/ (accessed Feb. 13, 2024).

[4] "Openpilot - Open source advanced driver assistance system," comma.ai, https://comma.ai/openpilot (accessed Feb. 13, 2024).

[5] "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).

[6] "Comma 3x - make driving chill," comma.ai, https://comma.ai/shop/comma-3x (accessed Feb. 13, 2024).

[7] G. M. Smith, "What Is CAN Bus (Controller Area Network) and How It Compares to Other Vehicle Bus Networks," dewesoft.com, https://dewesoft.com/blog/what-is-can-bus (accessed Feb. 14, 2024).

[8] "Other Global Navigation Satellite Systems (GNSS)," Gps.gov, Oct. 19, 2021. https://www.gps.gov/systems/gnss/#:~:text=Global%20navigation%20satellite%20system%20(GNSS,a%20global%20or%20regional%20basis (accessed Feb. 14, 2024).