Documentation Openpilot

Version 1.0

Lucas Pichon & Louis Leroy

janv. 12, 2025

Contents

Essentiellement, OpenPilot lit les données provenant de divers capteurs (caméra, IMU, capteur d'angle du volant, récepteur GNSS, etc.), traite leurs sorties et envoie des entrées pertinentes à un grand réseau de neurones. Ce dernier transforme les sorties en commandes exploitables pour les actionneurs du véhicule. Un autre aspect essentiel est de s'assurer que les conducteurs restent attentifs lorsque le système est activé.

Pour qu" OpenPilot fonctionne, plusieurs bibliothèques open source ont également été développées, notamment :

- **opendbc** : bibliothèque pour interpréter le trafic sur le bus CAN.
- **panda** : interface pour les véhicules.
- **cereal** : spécification de messagerie éditeur/abonné pour les systèmes robotiques.
- **rednose** : bibliothèque de filtres de Kalman pour l'odométrie visuelle, le SLAM, etc.

OpenPilot est composé de différents services qui communiquent entre eux via un système de messagerie inter-processus à un éditeur et plusieurs abonnés (spécifié dans cereal). Ces services peuvent être classés comme suit :

- Capteurs et actionneurs
- Réseaux neuronaux
- Localisation et calibration
- Contrôle
- Système, journalisation et services divers

CHAPTER 1

Services

Cette partie traite des différents scripts présents dans OpenPilot. Ces scripts communiquent grâce à un système de publication/abonnement. La liste des services est située dans le fichier /cereal/service.h Capteurs et actionneurs

1.1 Capteurs

- **camerad** *(/system/camerad)* : Gère à la fois la caméra de la route et la caméra du conducteur (ainsi qu'une caméra supplémentaire grand angle pour la route sur le comma three), et prend en charge l'autofocus et l'autoexposition. La gestion des flux vidéo est assurée par les scripts *camera_common.cc*, *camera_common.h* et *camera_qcom2.cc*.
- **annotated_camera** *(/selfdrive/ui/qt/onroad)* : Responsable de la mise à jour et du rendu de la fenêtre « onroad » (sur la route). Il sert de conteneur pour la mise à jour et le rendu de cette fenêtre, intégrant des éléments tels que les alertes sur la route, les informations du conducteur et les autres éléments de l'interface utilisateur liés à la conduite.
- **sensod** *(/system/sensord)* : Configure et lit les capteurs (gyroscope, accéléromètre, magnétomètre et capteurs de lumière).
- **micd** *(/system)*: Gestion d'un microphone pour mesurer la pression acoustique et calculer les niveaux de pression acoustique (SPL) dans le domaine sonore. Il applique également un filtrage A-weighting pour obtenir une approximation du niveau de bruit perçu par l'oreille humaine.

1.2 Réseaux de neurones

- **modeld** *(/selfdrive/modeld/modeld.py)* : Lit le flux d'images de visionipc (système de communication inter-processus pour échanger des images) dans le réseau de neurones principal de conduite. En utilisant ces données ainsi que les entrées de désir (changement de voie, etc.), le réseau de neurones tente de prédire où et comment conduire, dans les conditions données. *modeld* exécute le modèle supercombo, qui génère le chemin de conduite souhaité ainsi que d'autres métadonnées, y compris les lignes de voie, les voitures en tête, les bords de la route, et plus encore.
- **dmonitoringmodeld** *(/selfdrive/modeld/dmonitoringmodeld.py)* : Exécute le dmonitoring_model en utilisant le flux d'images de la caméra orientée vers le conducteur et prédit la posture de la tête du conducteur, si ses yeux

sont ouverts ou fermés, et s'il porte des lunettes de soleil. Cela permet de savoir que le conducteur est toujours attentif et qu'il est capable de reprendre la main à tout moment.

- **dmonitoringd** *(/selfdrive/monitoring/monitoring.py)* : Contient la logique permettant d'évaluer si le conducteur peut reprendre le contrôle si nécessaire. Sinon, il alerte le conducteur. Cette décision est basée sur les sorties du modèle de surveillance du conducteur, les informations sur la scène provenant du modèle de conduite, et quelques autres paramètres. Les conducteurs sont empêchés d'engager OpenPilot s'ils ont été distraits pendant de longues périodes.

1.3 Localisation et calibration

- **ubloxd** *(/system/ubloxd)*: Analyse les données GNSS, qui sont ensuite utilisées pour la localisation.
- **locationd** *(/selfdrive/locationd/locationd.py)*: Responsable de la localisation de la voiture dans le monde. La localisation est une étape essentielle pour décrire avec précision l'état de la voiture et son interaction avec le monde. C'est ainsi qu'OpenPilot connaît la vitesse, la présence d'un virage incliné ou si d'une pente. Ces données provenant de plusieurs capteurs sont combinées avec un filtre de Kalman (live_kf). Ce localisateur fournit la position, l'orientation, la vitesse, la vitesse angulaire et l'accélération de la voiture.
- **calibrationd** *(/selfdrive/locationd/calibrationd.py)*: L'entrée du modèle de conduite neuronal est transformée dans le cadre calibré, qui est aligné avec l'inclinaison et le lacet du véhicule. Cela normalise le flux d'images pour tenir compte des différentes façons dont les utilisateurs montent leurs appareils sur leurs pare-brises.
- **paramsd** *(/selfdrive/locationd/paramds.py)*: Il existe de nombreux paramètres spécifiques aux modèles de voitures (masse, rapport de direction, rigidité des pneus, décalage de direction, distance du centre de masse à l'axe des roues, etc.) qui sont nécessaires pour convertir un trajet de conduite en signaux que la voiture comprend, comme l'angle de direction. Une grande partie de ces paramètres sont codés en dur et sont considérés comme des constantes. Cependant, certains de ces paramètres sont soit inexactes, changent au cours de la vie d'une voiture, ou même pendant une conduite en fonction des conditions de la route. Il est donc nécessaire d'estimer précisément ces paramètres pour mieux contrôler la façon dont la voiture réagit aux entrées. Nous utilisons un filtre de Kalman (car_kf) pour cela, en supposant un modèle de véhicule à voie unique.
- **pigeond** *(/system/ubloxd/)*: Gère des interactions avec un récepteur GNSS Ublox via un port série. Ce récepteur est utilisé pour recevoir et envoyer des messages de configuration, de commande et d'initialisation. Il y a également des fonctionnalités pour gérer l'alimentation du récepteur, envoyer des messages de type UBX, configurer la vitesse de transmission, et récupérer ou envoyer des données liées à l'ublox, telles que les informations de position, le temps, ou les messages d'assistance.
- **qcomgpsd** *(/system/)*: Gère la configuration, la collecte et la gestion des données GPS à partir d'un modem Qualcomm.
- **torqued** *(/selfdrive/locationd)*: Destiné à estimer et ajuster les paramètres du système de contrôle latéral du véhicule (torque), en utilisant des données en temps réel et en ajustant dynamiquement les calculs de friction et d'accélération latérale en fonction des conditions de conduite.
- **ubloxd** *(/system/ubloxd)*: Traite les données GPS provenant d'un appareil u-blox. Il écoute les messages bruts u-blox entrants, les analyse et envoie les informations traitées.
- **ugpsd** *(/system)*: Récupère les données GPS d'un module Unicore, les traite, puis les publie.

1.4 Contrôle

- **radard** *(/selfdrive/controls/radar.py)*: Interprète les données brutes provenant des différents radars présents sur les modèles de voitures et les convertit en un format canonique.
- **plannerd** *(/selfdrive/controls/plannerd.py)*: La planification est séparée en deux parties : la planification latérale (direction) et la planification longitudinale (accélérateur/frein). Les deux utilisent un résolveur MPC (Contrôle Prédictif Modélisé) pour garantir que les plans soient fluides et optimisent des coûts raisonnables. Le réseau

neuronal de conduite prédit où la voiture devrait être, mais le planificateur latéral détermine comment y parvenir. En utilisant la prédiction de trajectoire du réseau neuronal (et parfois la prédiction des lignes de voie), et un résolveur MPC, le planificateur latéral estime combien la voiture doit tourner (courbure) dans les prochaines secondes.La planification longitudinale dépend encore principalement des voitures de tête. Elle prend les estimations fusionnées (réseau neuronal + radar) des voitures de tête et la vitesse cible souhaitée, les insère dans un résolveur MPC, et calcule un bon profil d'accélération pour les prochaines secondes.

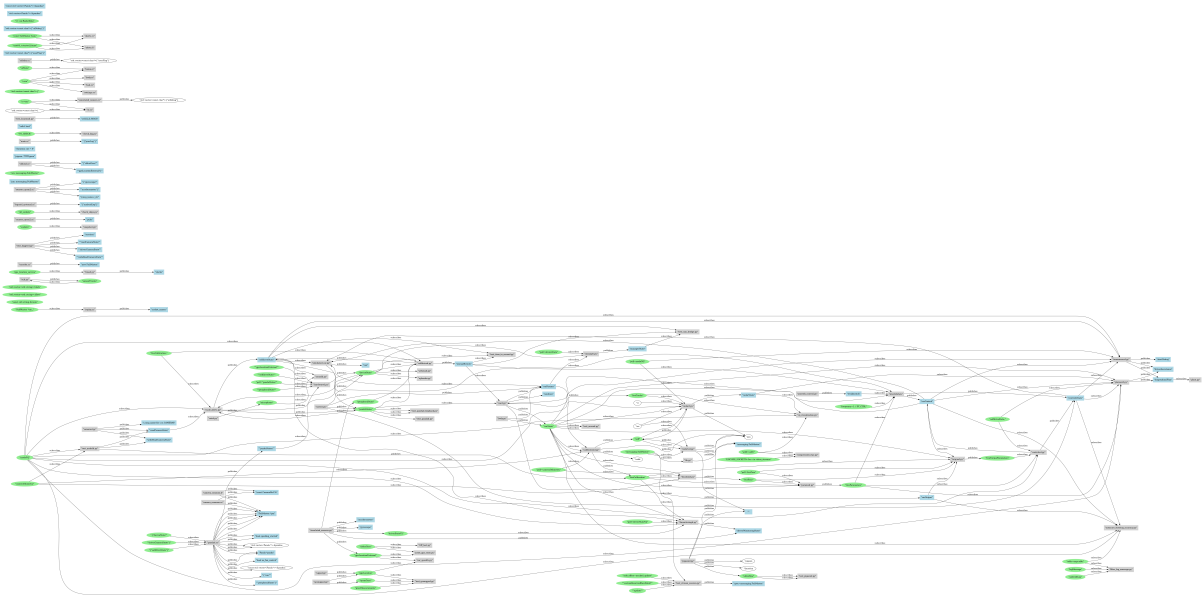
- **controlsd** *(/selfdrive/controls/controlsd.py)*: Contrôle réellement la voiture. Il reçoit le plan du service *plannerd*, sous forme de courbures et de vitesses/accélérations, et le convertit en signaux de contrôle. Ces cibles agnostiques au véhicule (accélération et angle de direction) sont ensuite converties en commandes CAN spécifiques au véhicule, qui fonctionnent avec l'API de cette voiture, à travers un système de contrôle en boucle fermée qui fonctionne à 100Hz. Controlsd analyse également les données brutes du CAN de la voiture et les publie dans un format canonique.
- **car_specific** *(/selfdrive/car/)*: Responsable de l'interface et du contrôle des systèmes spécifiques à chaque modèle de véhicule dans OpenPilot, notamment pour la gestion des commandes de conduite (direction, accélération, freinage) et la communication avec les systèmes du véhicule via le bus CAN.
- **card** *(/selfdrive/car)*: Gère la communication avec le véhicule via le bus CAN.
- **maneuversd** *(/tools/longitudinale_maneuvers)*: gérer des manœuvres longitudinales pour un véhicule, en envoyant des commandes de vitesse et d'accélération en fonction de l'état du véhicule.
- **measure_steering_accuracy** *(/tools/tuning)* : Mesure la précision de la direction. Il compare l'angle de direction réel avec l'angle de direction souhaité pour calculer l'erreur.

1.5 Sevices systèmes, journalisation et divers

- **manager** *(/system/manager)*: Gère le démarrage et l'arrêt de tous les processus décrits ci-dessus.
- **loggerd/ logcatd/ proclogd** *(/system/)*: Gèrent l'enregistrement de tous les logs d'openpilot. Les données vidéo compressées et les données des capteurs sont enregistrées comme données d'entraînement, afin de continuer à améliorer les réseaux neuronaux. Tous les messages Cereal, rapports de plantage système, etc. sont également enregistrés afin de comprendre et de résoudre les pannes.
- **athenad** *(/system/athenad)*: Etablit une connexion websocket avec les serveurs comma.ai et gère toutes les demandes liées au dispositif venant de connect.comma.ai. Le dispositif peut être atteint en envoyant des appels API REST à athena.comma.ai. Exemples d'appels API possibles : demander la tension de la batterie, définir une destination de navigation, obtenir la localisation de la voiture ou des demandes pour télécharger des fichiers.
- **ui** *(/selfdrive/ui)*: Gère tout ce qui est affiché à l'utilisateur. Lorsque la voiture est éteinte, il contient un guide de formation pour intégrer les nouveaux utilisateurs, affiche l'état du système et expose certains paramètres. Lorsque la voiture est allumée, le flux de la caméra orientée vers la route est affiché avec des visualisations superposées du chemin de conduite, des lignes de voie et des voitures de tête.
- **alerts** *(/selfdrive/ui/qt/onroad)*: Définit et gèrent les différents types d'alertes que le système peut générer pour informer le conducteur de diverses situations ou anomalies.
- **hardwared** *(/system/hardware)*: Gère divers aspect matériels du système (surveillance, gestion thermique, collecte et gestion des données, gestion du périphérique, des alertes et des évènements tactiles, contrôle du démarrage système)
- **pandad** *(/selfdrive.pandad)*: Gère la gestion des cartes Panda. Son rôle principal est de maintenir les cartes Panda à jour, de vérifier leur état de fonctionnement et de gérer les processus de récupération en cas de défaillance.
- **selfdrived** *(/selfdrive/selfdrived)*: Surveille en permanence le véhicule, déclenche des événements en fonction de la situation, et assure la gestion des alertes pour les éventuels dysfonctionnements.
- **timed** *(/system)*: Synchronise l'heure système avec celle obtenue à partir d'un service GPS.

- **soundd** *(/selfdrive/ui/soundd)*: Module responsable de la gestion des alertes sonores.

1.6 Intération entre les scripts



CHAPTER 2

Hiérarchie

2.1 cereal

Le système de messagerie **cereal** est un élément essentiel d'openpilot. Il s'appuie sur msgq comme backend de publication/abonnement (pub/sub) et utilise Cap'n Proto pour la sérialisation des structures de données. Ce système permet une communication efficace entre les différents modules du système.

cereal/service.py

- Définit les services utilisés pour la communication entre les différents modules d'openpilot.
- Les services représentent des flux de données ou des types de messages échangés, comme les données des capteurs du véhicule (caméra, radar) et les modules de contrôle de conduite.
- Utilise une architecture pub/sub implémentée via msgq pour la transmission des données interprocessus.

cereal/*.capnp

- Contient les définitions des types de messages via Cap'n Proto.
- Les messages utilisent des unions pour représenter différents types de paquets (données GPS, capteurs, états système, etc.).

Répertoires spécifiques

- **cereal/gen/C++** : Fichiers générés automatiquement pour supporter l'infrastructure du projet.
- **cereal/include/c++.capnp** : Définitions pour la sérialisation/désérialisation via Cap'n Proto.

cereal/messaging

- Gère le fonctionnement du système de messagerie d'openpilot.
- Utilise des sockets et des files de messages pour la transmission efficace des données.

2.2 common

Contient des outils et des composants partagés utilisés dans l'ensemble du projet. Ce répertoire regroupe des scripts et des bibliothèques génériques qui sont essentiels pour le fonctionnement des différents modules.

2.3 doc

Contient un ensemble de fichiers .md.

2.4 msgq_repo

MSGQ est un système de communication interprocessus (IPC) basé sur une architecture publication/abonnement (pub/sub) avec un producteur unique et plusieurs consommateurs. Il utilise un buffer circulaire en mémoire partagée pour transmettre les messages efficacement. Les messages sont préfixés par une taille et les positions d'écriture et de lecture sont gérées avec des pointeurs et des flags de validité. En cas de dépassement ou de retard des lecteurs, des mécanismes de réinitialisation assurent la continuité. MSGQ est conçu pour remplacer des systèmes comme ZMQ et inclut également VisionIPC pour gérer les grands buffers (images/vidéos).

2.5 opendbc_repo

opendbc est une API Python conçue pour interagir avec les systèmes électroniques des véhicules, en particulier pour contrôler et lire des données liées à la direction, l'accélérateur, le freinage, la vitesse, et l'angle de direction. Ce projet s'appuie sur des technologies comme l'Assistance au Maintien de Voie (LKAS) et le Contrôle Automatique de Vitesse Adaptatif (ACC), qui permettent d'interfacer avec le bus CAN des voitures.

- **opendbc_repo/opendebc/dbc** : Répertoire de fichiers DBC.
- **opendbc_repo/opendebc/can** : Bibliothèque pour analyser et constuire des messages CAN à partir de fichier DBC.
- **opendbc_repo/opendebc/car** : Bibliothèque de haut niveau pour interfacer avec des voitures en utilisant Python.

2.6 panda

Gère le firmware du dongle Panda et assure une communication CAN sécurisée entre openpilot et le véhicule, la validation du code pour éviter tout dysfonctionnement et la possibilité de personnaliser les fonctionnalités en compilant son propre firmware.

- **board** : Code pour la carte STM32
- **drivers** : Drivers
- **python** : Bibliothèque utilisateur Python pour interagir avec le Panda
- **tests** : Tests et programmes d'aide pour le Panda

2.7 rednose_repo

Bibliothèque de filtre de Kalman pour l'odométrie visuelle, le SLAM, etc.

2.8 selfdrive

Contient divers scripts pour la conduite autonome. Les différents répertoires ont déjà été évoqués dans la partie script.

2.9 system

Contient des éléments qui gèrent l'infrastructure de base du système. Là aussi les différents répertoires ont déjà été évoqués dans la partie script.

2.10 teleoprtc_repo

Fournit un ensemble d'abstractions pour la communication via WebRTC avec openpilot.

2.11 third_party

Contient des bibliothèques et des dépendances externes qui sont utilisées dans le projet.

2.12 tinygrad_repo

C'est un framework de deep learning qui permet ici de prédire le comportement de conduite.

2.13 tools

Contient divers outils comme des scripts d'installation ou encore de simulation.

- **bodyteleop** : Permet l'interaction avec le comma via une interface web
- **ubuntu_setup.sh** : Script pour l'installation sur Ubuntu
- **mac_setup.sh** : Script pour l'installation sur Mac
- **cabana** : Afficher et tracer les messages CAN provenant des entraînements ou en temps réel
- **camerastream** : Streaming des caméras sur le réseau
- **car_porting** : Améliore l'intégration des véhicules à openpilot
- **joystick** : Contrôler la voiture avec un joystick
- **latencylogger** : Analyse la latence d'openpilot
- **lib** : Des bibliothèques pour prendre en charge les outils et lire les journaux d'openpilot
- **longitudinale_maneuvers** : Implémentation d'un système de contrôle longitudinale basé sur des actions prédéfinies
- **plotjuggler** : Outil pour les logs d'openpilot
- **replay** : Rejouer des trajets et simuler des services openpilot
- **rerun** : Permet de visualiser et analyser les logs
- **scripts** : Drivers
- **sim** : Lancer openpilot dans un simulateur
- **tuning** : Outil pour mesurer la précision de la direction du véhicule
- **webcam** : Lancer openpilot sur un pc avec une webcam