



Mémoire de fin d'étude

Pour l'obtention du diplôme de Master

Filière: Electronique
Spécialité: Informatique Industrielle

Présenté par : Ahmed Khodhir REZIG et Sid Ahmed LAKEHAL

Thème

Développement d'une plateforme
d'analyse et de détection pour
le diagnostic avancé des véhicules

Soutenu publiquement, le 20/ 06 / 2026 , devant le jury composé de:

Dr BELLOUFA Fayssal	MCB ESSAT	Président
Dr MESSIRDI Mohammed	MCB Université Mascara	Directeur de mémoire
Dr OUDJAMA Farid	MCB ESSAT	Co- Directeur de mémoire
Dr KARAOUZENE Zoheir	MCB ESSAT	Examineur 1
Dr MERAD Fayza	MRB CDS Oran	Examineur 2
M ABDELLAOUI Abdou	Conseillé Diagnostic Automobil	Invité
Dr OUHOUD Amina	MCA ESSAT	CDE
Dr MIMOUNI Faiza	MCA ESSAT	CATI

Année universitaire : 2025 /2026.

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

Dédicace

Je dédie d'abord ce travail à moi-même, pour la patience, la persévérance et la détermination qui m'ont permis de surmonter les difficultés de ce parcours. Pour chaque défi affronté, chaque moment de doute dépassé et chaque effort qui m'a conduit jusqu'ici, je suis fier du chemin accompli.

À **ma chère mère, K. BENZINA**, mon refuge et ma plus belle force. Ton amour inconditionnel, tes prières silencieuses, tes sacrifices et ta tendresse ont été ma lumière dans les moments les plus sombres. Je ne trouverai jamais les mots justes pour exprimer tout ce que je te dois. Si j'ai pu arriver jusqu'ici, c'est avant tout grâce à toi.

À **mon père, Ahmed LAKEHAL**, dont la présence discrète mais constante m'a toujours guidé. Ta force, ton courage et ton engagement pour notre famille m'ont appris la valeur du travail et de la dignité. Cette réussite est autant la tienne que la mienne.

À **mon frère, Ilyes**, et à **mes sœurs, Amel et Kaouther**, pour leur amour sincère, leur soutien et leur présence qui ont rendu chaque étape plus légère.

À tous les membres de ma famille, pour leur affection, leurs prières et leur soutien, même à distance.

À mes camarades de promotion, avec qui j'ai partagé des années riches en émotions, en efforts et en souvenirs inoubliables. Ces années resteront gravées en moi. Une pensée particulière à **Ahmed Khodhir**, mon binôme, mon ami et mon frère de parcours, ainsi qu'à **Aboubaker, Yacine, Youcef, Amine, Aissam, Habib, Salah, Mohammed et Abderahmane**, qui ont rendu cette aventure plus humaine et plus belle.

Enfin, je dédie ce travail à toutes les personnes qui ont cru en moi, parfois plus que je ne croyais en moi-même, et qui m'ont soutenu sans attendre en retour. Leur présence, même silencieuse, a compté plus que tout.

Ce travail est bien plus qu'un simple mémoire. C'est le fruit de nuits de doute, de jours d'effort, de sacrifices invisibles et de rêves tenaces. Qu'il reste comme une trace de mon passage, la preuve qu'un jour, Sid Ahmed a essayé, a persisté, et a construit quelque chose de lui-même.

Sid Ahmed LAKEHAL

Dédicace

Je dédie d'abord ce travail à moi-même, pour la patience, la persévérance et la détermination qui m'ont permis de surmonter les difficultés de ce parcours. Pour chaque défi affronté, chaque moment de doute dépassé et chaque effort qui m'a conduit jusqu'ici, je suis fier du chemin accompli.

À ma chère mère, dont l'amour inconditionnel, les prières, les sacrifices et l'attention constante ont toujours été ma plus grande source de force. Aucun mot ne saurait exprimer pleinement ma gratitude pour tout ce que tu as fait pour moi et pour le soutien que tu m'as apporté tout au long de ma vie.

À mon père, **Ammar**, qui a toujours répondu à mes besoins, m'a soutenu et a veillé à ce que je ne manque de rien. Ta générosité, ta force et ta présence constante m'ont donné la confiance et la stabilité nécessaires pour avancer. Cette réussite est aussi la tienne.

À mon frère, **Aness**, et à mes sœurs, **Rihab, Manar et Loudjaine**, pour leur affection et leurs encouragements, ainsi qu'à la petite **Amira**, qui apporte de la joie à notre famille.

À mes grand-mères, dont les prières et l'amour m'ont accompagné tout au long de ma vie, et à la mémoire de mes deux grands-pères, qu'Allah leur fasse miséricorde. Leurs valeurs et leur héritage resteront toujours vivants au sein de notre famille.

À tous mes oncles et toutes mes tantes, pour leur affection, leurs conseils et leur soutien, et tout particulièrement à mon oncle, **Mohammed Lamine**, qui a toujours été pour moi bien plus qu'un oncle : un grand frère, un guide et une personne sur laquelle je peux compter.

À tous les membres de ma famille, où qu'ils se trouvent, pour leur amour et leurs encouragements constants.

À tous mes camarades de promotion, avec qui j'ai partagé cinq années inoubliables d'efforts, de défis, de rires, de réussites et de souvenirs. Plus particulièrement à **Sid Ahmed**, mon binôme, mon ami et mon frère tout au long de ce parcours, ainsi qu'à **Aboubaker, Yacine, Youcef, Amine, Aissam, Habib, Salah, Mohammed et Abderahmane**, dont l'amitié a rendu cette aventure plus précieuse et inoubliable.

À mes amis de la résidence universitaire, pour les moments de partage et les souvenirs vécus durant ces années.

À mes amis d'enfance, **Yacine HADDAD, Abdelmoez MOULAY, Wassim ZOUAI et Mohammed El Habib REZGANE**, dont l'amitié sincère a traversé les années.

Enfin, je dédie ce travail à toutes les personnes qui ont cru en moi, m'ont soutenu et ont contribué, de près ou de loin, à la personne que je suis devenu.

J'ai consacré à ce projet le meilleur de mes efforts, de mon savoir et de ma détermination. Puisse-t-il demeurer comme la trace de mon passage, le témoignage qu'un jour, Ahmed était ici, qu'il a imaginé, construit et laissé derrière lui quelque chose qui lui survivra.

Ahmed Khodhir REZIG

Remerciements

Au nom de Dieu, le Tout-Puissant, le Clément, le Miséricordieux, qui nous a encouragés à rechercher le savoir et la science, et qui nous a dotés des facultés et des moyens nécessaires pour y parvenir. Nous Lui rendons grâce pour la volonté, le courage, la patience et la force qu'Il nous a accordés tout au long de ces années d'études.

Nous exprimons notre profonde gratitude et nos sincères remerciements à notre encadrant, **Monsieur Mohammed MESSIRDI**, pour sa confiance, son soutien constant et sa grande patience tout au long de ce projet. Vous avez su nous guider avec bienveillance et générosité, et avez été bien plus qu'un enseignant : un véritable mentor. Grâce à votre expérience dans la recherche et l'enseignement, ainsi qu'à vos précieux conseils, nous avons pu découvrir et approfondir le monde de la recherche scientifique dans le domaine du diagnostic automobile.

Nous remercions également chaleureusement Monsieur **Farid OUDJAMA**, notre co-encadrant, pour sa disponibilité, sa patience et ses précieux conseils tout au long de ce travail.

Nous tenons à remercier l'ensemble des enseignants du département d'Électronique de l'École Supérieure en Sciences Appliquées de Tlemcen pour les connaissances et les compétences qu'ils nous ont transmises au fil des années.

Nos pensées reconnaissantes vont également à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce projet.

Enfin, nous adressons nos sincères remerciements aux membres du jury pour l'intérêt qu'ils ont porté à notre travail, ainsi que pour leurs remarques et suggestions constructives qui contribueront à enrichir notre réflexion et à améliorer ce travail.

Résumé

Ce mémoire s'inscrit dans les domaines des systèmes d'information et du traitement de données appliqués au diagnostic automobile. Il décrit la chaîne de transmission et de valorisation des codes DTC collectés par le dispositif de lecture embarqué vers un serveur distant. Les codes sont acheminés à une application web développée avec Django, qui interagit avec une base de données PostgreSQL renfermant la signification détaillée de l'intégralité des DTC Renault/Dacia. La fiabilité du référentiel ainsi constitué est ensuite éprouvée par une confrontation systématique avec les diagnostics fournis par des outils professionnels de référence, tels que la gamme Launch. L'utilité de ce travail est de doter l'outil de diagnostic d'une base d'interprétation centralisée, fiable et validée.

Mots-clés : DTC, serveur distant, Django, PostgreSQL, Launch, base d'interprétation.

Abstract

This thesis falls within the fields of Information Systems and Data Processing applied to automotive diagnostics. It describes the transmission and value-creation chain of Diagnostic Trouble Codes (DTCs) collected by an on-board diagnostic device and sent to a remote server. The codes are routed to a web application developed with Django, which interacts with a PostgreSQL database containing detailed definitions for the entire range of Renault/Dacia DTCs. The reliability of the resulting repository is then tested through systematic comparison with diagnostics provided by industry-standard professional tools, such as the Launch product line. The primary objective of this work is to provide the diagnostic tool with a centralized, reliable, and validated interpretation base.

Keywords: DTC, remote server, Django, PostgreSQL, professional tools, Launch, interpretation base.

ملخص

تتمحور هذه المذكرة حول نظم المعلومات ومعالجة البيانات المطبقة في مجال تشخيص المركبات. وتستعرض الدراسة مسار نقل واثمين أكواد الأعطال، بدءاً من تجميعها عبر أجهزة القراءة المدججة، وصولاً إلى معالجتها في خادم مركزي. يتم توجيه هذه البيانات إلى تطبيق ويب متطور يعتمد على إطار العمل دجانغو، والذي يتكامل بدوره مع قاعدة بيانات بوستغراس شاملة، تحتوي على التفسيرات الدقيقة والتفصيلية لكافة رموز الأعطال الخاصة بمجموعة رونو - داسيا. ولضمان أعلى معايير الدقة والموثوقية، أُخضع هذا المرجع التقني لاختبارات صارمة عبر مقارنة مخرجاته بصفة منهجية مع نتائج أدوات التشخيص الاحترافية الرائدة عالمياً، وفي مقدمتها مجموعة لانش. وتكمن القيمة الجوهرية لهذا العمل في تزويد منظومة التشخيص بقاعدة بيانات تفسيرية موحدة، موثقة، ومصادق عليها، مما يعزز من كفاءة عمليات الصيانة والإصلاح.

الكلمات المفتاحية: أكواد الأعطال، خادم مركزي، دجانغو، بوستغراس، أدوات التشخيص الاحترافية، لانش، قاعدة بيانات تفسيرية.

Table des matières

Remerciements

Liste des figures

Liste des tableaux

Liste des abréviations

Introduction générale

1	État de l'art	1
1.1	Introduction	1
1.2	L'automobile moderne : de la mécanique à l'électronique embarquée	1
1.2.1	Évolution de l'architecture interne des véhicules	1
1.2.2	L'émergence des systèmes électroniques et logiciels dans les automobiles	2
1.2.3	La nécessité du diagnostic électronique dans les véhicules modernes	3
1.3	Le diagnostic embarqué (OBD) : historique et évolution	4
1.3.1	Avant la standardisation : le diagnostic pré-OBD (avant 1988)	4
1.3.2	L'avènement de l'OBD-I : première tentative de standardisation	5
1.3.3	L'OBD-II : vers une normalisation universelle	6
1.3.4	La lecture des codes de défaut à l'ère pré-numérique : la méthode par clignotement	6
1.4	Les codes de défaut (DTC) : structure, normes et classification	7
1.4.1	Naissance et standardisation des DTC modernes	7
1.4.2	Les normes de référence : SAE J2012 et ISO 15031-6	7
1.4.3	Classification des DTC par domaine fonctionnel (P, B, C, U)	8
1.4.4	DTC génériques et DTC constructeurs	8
1.5	Les bases de correspondance DTC et les outils de diagnostic	9
1.5.1	L'apparition des bases de correspondance et leur rôle dans l'interprétation des DTC	9
1.5.2	Les outils de diagnostic : panorama et classification	10
1.5.2.1	Les outils standards et les interfaces génériques (ELM327, ...)	10
1.5.2.2	Les outils multimarques (Launch, ThinkDiag, ...)	11
1.5.2.3	Les outils constructeurs et spécialisés (CAN CLIP, Renolink, PyRen, ...)	12
1.6	Conclusion	14
2	L'infrastructure logicielle du système de diagnostic à distance	15
2.1	Introduction	15

2.2	Les serveurs web : concepts fondamentaux	15
2.2.1	Définition et rôle d'un serveur web	15
2.2.2	Architecture client-serveur	16
2.3	Django : un cadre de développement web en Python	17
2.3.1	Présentation de Django	17
2.3.2	L'architecture MVT (Modèle-Vue-Gabarit)	18
2.3.3	Cas d'utilisation et domaines d'application de Django	19
2.4	Architecture du serveur Django	19
2.4.1	Les composants principaux du serveur Django	19
2.4.2	Le mécanisme de routage des requêtes	20
2.4.3	Relation entre les points de terminaison et les vues	20
2.5	Architecture et fonctionnement des fichiers XML dans la base de données DDT2000 de Renault/Dacia	20
2.5.1	Présentation du logiciel DDT2000	20
2.5.2	Rôle des fichiers XML dans DDT2000	20
2.5.3	Structure générale d'un fichier XML DDT2000	20
2.5.4	Les sections importantes d'un fichier XML	21
2.5.5	Fonctionnement global du système	21
2.6	PostgreSQL : le système de gestion de base de données du serveur	21
2.6.1	Présentation de PostgreSQL et ses caractéristiques	21
2.7	Conclusion	22
3	Implémentation et méthodologie	23
3.1	Introduction	23
3.2	Solution proposée	23
3.3	Création du serveur Django	24
3.3.1	Création de l'environnement de travail	24
3.3.2	Création du projet Django	24
3.3.3	Création d'une application Django	25
3.3.4	Lancement du serveur sur le port 8000	25
3.4	Création de la base de données PostgreSQL	25
3.4.1	Installation de PostgreSQL	26
3.5	Préparation de la base de données	26
3.5.1	Table DTC_standard	26
3.5.2	Table DTC_XML	27
3.6	Remplissage de la table DTC_standard par extraction automatisée de données web	27
3.6.1	Principe de l'extraction automatisée de données web	27
3.6.2	Insertion des données dans PostgreSQL	28
3.7	Analyse des fichiers XML pour remplir la table DTC_XML	29
3.7.1	Principe de l'analyse XML	29
3.7.2	Extraction des informations utiles	29
3.7.3	Insertion dans la table DTC_XML	30
3.8	Connexion entre Django et PostgreSQL avec psycopg2	31
3.9	Définition des points de terminaison du serveur	32
3.10	Accès au serveur via Internet avec Cloudflared	32
3.11	Application mobile développée avec MIT App Inventor	33
3.11.1	Rôle de l'application mobile	33
3.11.2	Composants principaux de l'application	33
3.11.3	Fonctionnement de l'application	34
3.12	Test et validation de l'interprétation	36

3.12.1	Objectif de la validation	36
3.12.2	Méthodologie de test	37
3.12.3	Analyse des résultats	37
3.13	Conclusion	37
Conclusion générale et perspectives		39
Bibliographie		41

Liste des figures

1.1	Évolution de l'architecture interne des véhicules	2
1.2	Exemples d'unités de contrôle électroniques (ECU)	3
1.3	Panorama des outils de diagnostic automobile	10
1.4	Interface générique OBD-II ELM327	11
1.5	Outil de diagnostic multimarque Launch	11
1.6	Outil de diagnostic ThinkDiag	12
1.7	Outil constructeur Renault CAN CLIP	13
1.8	Interface spécialisée Renolink	13
1.9	Outil de diagnostic à code source ouvert PyRen	14
2.1	Architecture client-serveur	17
2.2	Logo du cadre de développement Django	17
2.3	Architecture MVT de Django	18
2.4	Logo de PostgreSQL	21
3.1	Interface d'accueil de l'application mobile VEYA	34
3.2	Interface de visualisation des données du véhicule en temps réel	35
3.3	Interfaces de lecture et d'interprétation des codes de défaut (DTC)	36

Liste des tableaux

3.1 Principaux composants de l’application mobile et leurs rôles 33

Liste des abréviations

Acronyme	Signification
AUTOSAR	AUTomotive Open System ARchitecture
CAN	Controller Area Network
CARB	California Air Resources Board
DLC	Data Link Connector
DTC	Diagnostic Trouble Code
ECU	Electronic Control Unit
EOBD	European On-Board Diagnostics
ESSAT	École Supérieure en Sciences Appliquées de Tlemcen
LIN	Local Interconnect Network
MIL	Malfunction Indicator Lamp
MOST	Media Oriented Systems Transport
OBD	On-Board Diagnostics
VEYA	Vehicle Eye Analysis

Introduction générale

L'automobile contemporaine est l'une des réalisations technologiques les plus complexes de notre époque. Loin de n'être qu'un simple assemblage de pièces mécaniques, elle constitue aujourd'hui un véritable système embarqué intégrant des dizaines de calculateurs électroniques qui communiquent en temps réel au travers de réseaux de communication internes tels que le CAN. Cette sophistication a rendu le diagnostic des pannes inaccessible aux seuls sens du technicien et a imposé l'intégration, dès la conception, de fonctions de diagnostic embarqué (OBD). Standardisé à l'échelle mondiale, le protocole OBD-II permet aujourd'hui à tout outil conforme d'accéder à une liste normalisée de codes de défaut (DTC) et de paramètres de fonctionnement. Parallèlement, la généralisation des communications sans fil et de l'Internet des objets offre l'opportunité de déporter le diagnostic hors de l'atelier, en transmettant ces codes à des serveurs distants capables de les interpréter en temps réel.

C'est dans ce contexte que s'inscrit le présent mémoire de master, qui s'intègre dans un projet plus large articulant deux volets complémentaires. Le volet ingénierie se concentre sur la conception d'un dispositif de diagnostic embarqué, capable de dialoguer avec le port OBD-II d'un véhicule, de collecter les codes de défaut et de les acheminer vers une infrastructure serveur via une liaison sans fil. Le volet master, objet de ce manuscrit, porte quant à lui sur la réalisation de cette infrastructure serveur : une plateforme web centralisée qui reçoit les DTC, interroge une base de connaissances structurée et retourne aux clients distants la signification précise des anomalies détectées, qu'il s'agisse de codes génériques ou de codes spécifiques aux constructeurs.

L'objectif central de ce travail est de doter le système de diagnostic d'une base d'interprétation centralisée, fiable et validée, accessible à distance, indépendante du dispositif embarqué, et couvrant l'intégralité des DTC Renault/Dacia. Cette architecture répond aux lacunes identifiées dans les solutions existantes et ouvre la voie à une nouvelle génération d'outils de diagnostic connectés, évolutifs et accessibles.

Le présent mémoire est organisé en trois chapitres complémentaires, dont la progression suit une logique allant du général au particulier, de la théorie à la réalisation.

Le chapitre 1 retrace l'évolution de l'automobile, depuis son architecture mécanique jusqu'aux systèmes embarqués complexes actuels. Il présente l'historique de

la standardisation du diagnostic (OBD-I, OBD-II), la structure et la classification des codes DTC, ainsi qu'un panorama complet des outils de diagnostic existants, des interfaces génériques aux solutions constructeurs spécialisées.

Le chapitre 2 pose les fondations théoriques et technologiques du système développé. Il introduit les concepts de serveur web et d'architecture client-serveur, présente le framework Django et son architecture MVT, détaille la structure des fichiers XML de DDT2000 exploités comme référentiel DTC, et décrit PostgreSQL comme moteur de persistance des données.

Le chapitre 3 décrit concrètement la mise en œuvre du système, en montrant comment les technologies présentées au chapitre 2 ont été assemblées et configurées pour aboutir à une plateforme fonctionnelle.

État de l'art

1.1 Introduction

L'automobile contemporaine n'est plus un simple assemblage de pièces mécaniques; elle est devenue un système embarqué complexe, intégrant des dizaines de calculateurs électroniques communiquant en temps réel [34, 7]. Cette évolution profonde a rendu le diagnostic traditionnel obsolète, et a imposé l'émergence de systèmes de diagnostic électronique standardisés. Au cœur de ces systèmes se trouvent les Codes de Défaut (DTC), apparus progressivement avec les normes OBD-I puis OBD-II, et dont la lecture et l'interprétation ont considérablement évolué au fil des décennies.

Ce chapitre établit l'état de l'art du diagnostic automobile en retraçant le passage d'une architecture mécanique à une plateforme électronique et logicielle complexe. Il expose l'historique de la normalisation, depuis les premiers développements jusqu'à la généralisation de l'OBD-II et ses protocoles standardisés. L'analyse détaille ensuite la structure normalisée des codes de défaut, leur classification par domaine fonctionnel (P, B, C, U) et la distinction entre codes génériques et constructeurs. Le chapitre recense également les bases de correspondance informatisées et propose une typologie des outils de diagnostic, des interfaces génériques aux solutions constructeurs spécialisées.

1.2 L'automobile moderne : de la mécanique à l'électronique embarquée

1.2.1 Évolution de l'architecture interne des véhicules

Historiquement, les véhicules automobiles étaient constitués de systèmes mécaniques, hydrauliques et pneumatiques isolés, où chaque composant fonctionnait de manière quasi autonome. Les liaisons entre les commandes du conducteur et les organes du véhicule étaient directes (câbles, tringlerie). Avec l'évolution des normes antipollution et des exigences de sécurité et de confort, cette architecture a été fondamentalement repensée [39, 28]. Les systèmes mécaniques ont été progressivement couplés à des capteurs et des actionneurs, centralisant ainsi le contrôle vers des unités de traitement locales. Cette évolution est également illustrée par le passage des carburateurs traditionnels aux systèmes d'injection électronique (EFI) [26].

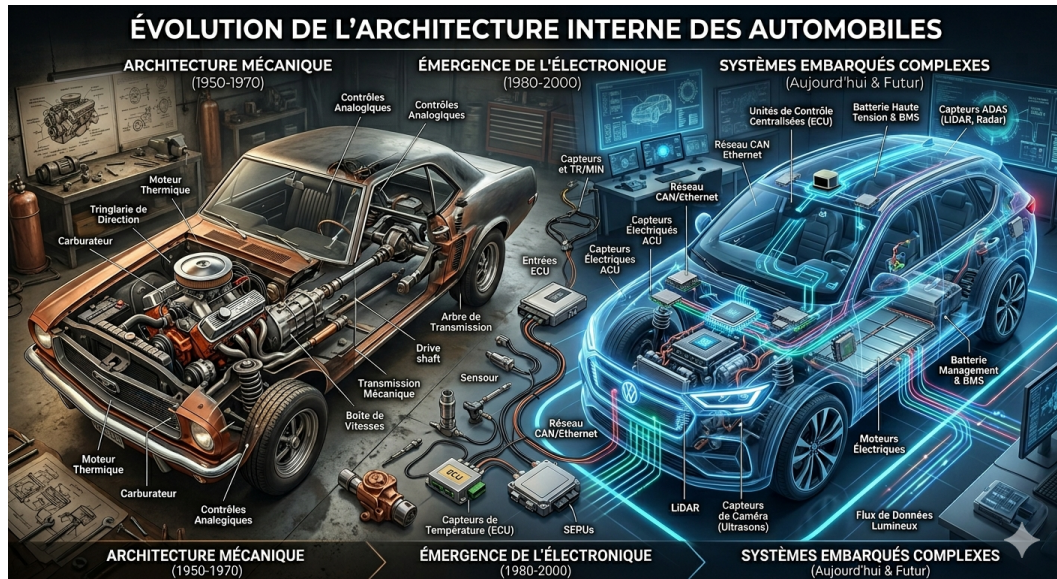


Figure 1.1: Évolution de l'architecture interne des véhicules

1.2.2 L'émergence des systèmes électroniques et logiciels dans les automobiles

À partir des années 1980, le nombre d'unités de contrôle électronique (ECU) n'a cessé de croître dans les véhicules. Chaque ECU est un calculateur dédié à la supervision et au contrôle d'un sous-système spécifique du véhicule. Parmi les ECU les plus répandus, on distingue :

- L'**ECM** (Engine Control Module) : gère l'injection, l'allumage, le ralenti et les émissions du moteur.
- Le **TCM** (Transmission Control Module) : contrôle les rapports de boîte de vitesses automatique.
- L'**ABS/ESP** : supervise le système de freinage antiblocage et la stabilité dynamique.
- L'**Airbag Control Unit** : gère le déclenchement des coussins gonflables.
- Le **BCM** (Body Control Module) : contrôle les équipements de confort (vitres, verrouillage, éclairage).

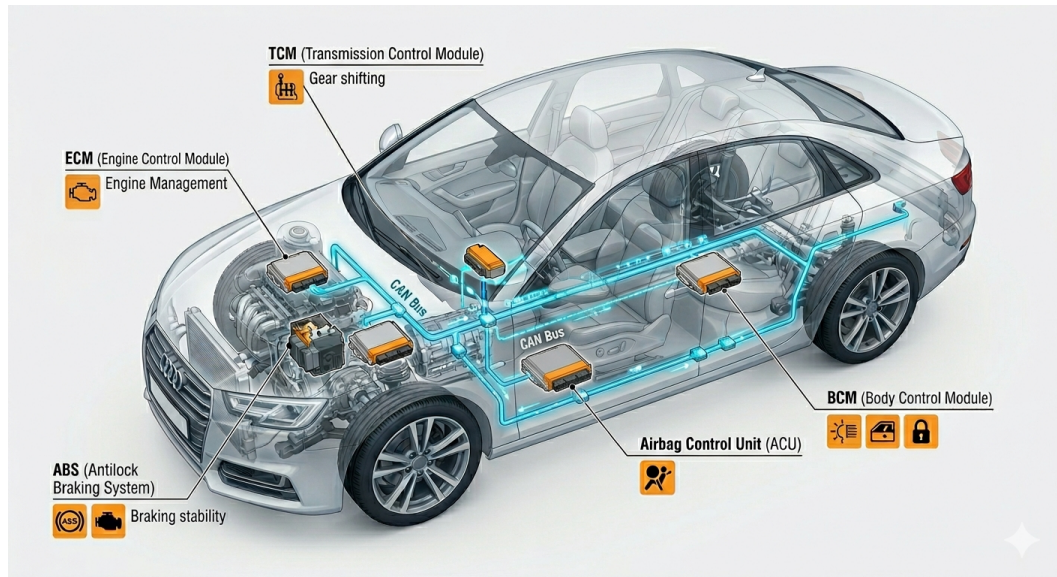


Figure 1.2: Exemples d'unités de contrôle électronique (ECU)

Un véhicule moderne haut de gamme peut embarquer jusqu'à 80 à 100 ECU interconnectés, échangeant en permanence des données via des réseaux de communication internes [17, 35]. Le nombre de microcontrôleurs embarqués est passé de 3–5 unités à plus de 30–40 par véhicule au cours des dernières années [38]. Les principaux bus de communication utilisés sont le CAN, développé par Bosch en 1986 et devenu le standard dominant, ainsi que le LIN, le MOST pour le multimédia, et plus récemment le FlexRay et l'Ethernet automobile pour les applications à haute bande passante (véhicules autonomes, ADAS) [35, 39].

Cette multiplication des ECU a transformé l'automobile en un véritable réseau informatique embarqué. Le logiciel embarqué, appelé *firmware*, occupe une place centrale : un véhicule moderne contient plusieurs centaines de millions de lignes de code, surpassant en complexité logicielle de nombreux systèmes aéronautiques [17]. Face à cette explosion de la complexité logicielle, l'industrie a développé le standard AUTOSAR afin d'harmoniser les interfaces entre les couches matérielles et logicielles des ECU [16].

1.2.3 La nécessité du diagnostic électronique dans les véhicules modernes

L'omniprésence des systèmes électroniques dans les véhicules modernes a profondément modifié la nature des pannes et, par conséquent, les méthodes de diagnostic. Contrairement à une défaillance mécanique, souvent perceptible par un bruit, une vibration ou une fuite visible, une défaillance électronique peut être silencieuse, intermittente, ou se manifester par des symptômes indirects qui rendent son identification difficile sans outillage adapté [24].

Face à cette complexité, le diagnostic électronique s'est imposé comme une nécessité absolue, pour plusieurs raisons fondamentales :

- **Sur le plan technique**, les ECU surveillent en permanence des centaines de paramètres (températures, pressions, débits, tensions, vitesses de rotation). Lorsqu'une valeur sort d'une plage acceptable, l'ECU enregistre un code de défaut et peut adapter le comportement du véhicule (mode dégradé). Sans outil de diagnostic permettant d'interroger ces ECU, il est impossible pour un technicien d'accéder à ces informations [34, 14].
- **Sur le plan réglementaire**, les législations antipollution (CARB en Californie, EURO en Europe) imposent aux constructeurs l'intégration de systèmes de surveillance des émissions capables de détecter toute défaillance susceptible d'augmenter les rejets polluants. Le non-respect de ces normes expose le conducteur à des sanctions lors des contrôles techniques [20].
- **Sur le plan économique**, un diagnostic rapide et précis réduit les temps d'immobilisation du véhicule, évite les remplacements de pièces par défaut, et optimise le travail du technicien. Des études sectorielles estiment que le recours au diagnostic électronique réduit le temps de réparation de 30 à 50 % par rapport à une approche purement empirique [1].

Le diagnostic électronique n'est donc plus une option, mais une composante essentielle de la maintenance automobile moderne.

1.3 Le diagnostic embarqué (OBD) : historique et évolution

1.3.1 Avant la standardisation : le diagnostic pré-OBD (avant 1988)

Avant l'apparition des systèmes OBD standardisés, chaque constructeur automobile développait ses propres mécanismes de détection et de signalement des défauts, sans aucune interopérabilité entre marques. Dès les années 1960 et 1970, quelques systèmes rudimentaires de surveillance électronique existaient déjà sur certains véhicules haut de gamme, mais ils étaient propriétaires, peu documentés, et inaccessibles aux techniciens indépendants [21, 24].

Le constructeur Volkswagen est souvent cité comme l'un des pionniers, ayant introduit dès 1968 un système de gestion électronique du moteur (Bosch D-Jetronic) équipé de fonctions de détection d'anomalies. General Motors, de son côté, intégrait dès le début des années 1980 des fonctions de diagnostic dans ses calculateurs moteur (système ALDL — Assembly Line Diagnostic Link), avec un connecteur physique accessible permettant la lecture de codes de défaut via un outil propriétaire [26].

Ces approches présentaient des limitations majeures :

- Les connecteurs de diagnostic variaient d'un constructeur à l'autre, voire d'un modèle à l'autre.
- Les protocoles de communication étaient propriétaires et non documentés publiquement.
- Les codes de défaut n'étaient pas standardisés : un même code pouvait désigner des défauts différents selon la marque.
- L'accès aux outils de diagnostic était réservé aux réseaux officiels des constructeurs.

Cette fragmentation du diagnostic automobile a constitué le principal moteur de la démarche de standardisation qui allait aboutir à l'OBD-I puis à l'OBD-II.

1.3.2 L'avènement de l'OBD-I : première tentative de standardisation

La première réglementation imposant un système de diagnostic embarqué standardisé est issue de la CARB, l'agence californienne de contrôle de la qualité de l'air. En 1988, la CARB impose à tous les véhicules neufs vendus en Californie l'intégration d'un système OBD-I (*On-Board Diagnostics*, première génération), avec obligation d'entrée en vigueur progressive entre 1988 et 1991 [31, 7].

Les exigences de l'OBD-I portaient principalement sur la surveillance des composants directement liés aux émissions polluantes comme le capteur d'oxygène (sonde lambda), le système de recirculation des gaz d'échappement (EGR) et la commande du carburant et le système d'injection

L'OBD-I imposait également l'allumage d'un voyant au tableau de bord (le MIL (*Malfunction Indicator Lamp*), communément appelé "*Check Engine*") en cas de détection d'une anomalie affectant les émissions.

Cependant, l'OBD-I souffrait de limitations importantes qui en réduisaient la portée pratique [14, 26] :

- Absence de standardisation du connecteur : chaque constructeur utilisait son propre connecteur physique.
- Absence de protocole de communication unifié : les échanges de données restaient propriétaires.
- Codes de défaut non harmonisés : chaque constructeur définissait ses propres codes.
- Couverture diagnostique limitée : seuls les composants liés aux émissions étaient concernés.

L'OBD-I représentait néanmoins une avancée conceptuelle majeure : pour la première fois, la loi imposait aux constructeurs l'intégration d'un système de diagnostic accessible et la signalisation obligatoire des défauts via un voyant normalisé. Ces principes allaient être repris et considérablement étendus par l'OBD-II.

1.3.3 L'OBD-II : vers une normalisation universelle

Face aux limites de l'OBD-I, la CARB publie en 1994 la réglementation OBD-II, rendue obligatoire pour tous les véhicules à essence vendus aux États-Unis à partir de 1996, puis étendue aux véhicules diesel à partir de 2005 [29]. En Europe, une réglementation équivalente (l'EOBD (*European On-Board Diagnostics*)) est imposée par la directive européenne, avec application obligatoire dès 2001 pour les véhicules à essence et 2004 pour les diesels.

L'OBD-II constitue une rupture fondamentale par rapport à son prédécesseur, en imposant une standardisation complète sur plusieurs plans [8, 19] :

- **Le connecteur physique** : l'OBD-II impose un connecteur unique et normalisé, le DLC (*Data Link Connector*), de type J1962, à 16 broches, positionné obligatoirement dans l'habitacle, à moins de 60 cm du volant [29].
- **Les protocoles de communication** : l'OBD-II autorise initialement plusieurs protocoles physiques (ISO 9141-2, ISO 14230 KWP2000, SAE J1850 VPW, SAE J1850 PWM), tous convergeant depuis 2008 vers le protocole unique ISO 15765-4 CAN, désormais obligatoire sur tous les véhicules OBD-II [19].
- **Les paramètres surveillés (PIDs)** : l'OBD-II définit un ensemble de paramètres standardisés accessibles via des requêtes normalisées, incluant la vitesse du véhicule, le régime moteur, la température du liquide de refroidissement, la pression d'admission, l'avance à l'allumage, et bien d'autres [11].
- **Les codes de défaut** : l'OBD-II impose un format universel de codage des défauts (les DTC), défini par les normes SAE J2012 et ISO 15031-6, garantissant qu'un même code possède la même signification quel que soit le constructeur pour les codes dits *génériques* [8, 28].

L'OBD-II a ainsi posé les fondations d'un écosystème ouvert du diagnostic automobile, permettant l'émergence d'outils de diagnostic tiers, indépendants des réseaux constructeurs.

1.3.4 La lecture des codes de défaut à l'ère pré-numérique : la méthode par clignotement

Avant la généralisation des interfaces numériques et des logiciels de diagnostic, une méthode simple permettait d'extraire les codes de défaut sans outil sophistiqué : la lecture par clignotement. Sur de nombreux véhicules des années 1980 et 1990, le technicien pouvait, à l'aide d'un simple cavalier ou d'une manipulation de la clé de contact, activer le mode de diagnostic. Le calculateur faisait alors clignoter un voyant du tableau de bord (souvent le témoin "*Check Engine*") selon un motif répétitif : un nombre de clignotements pour les dizaines, une pause, puis un nombre de clignotements pour les unités.

Par exemple, un défaut de capteur de température pouvait se manifester par trois clignotements suivis de deux clignotements, correspondant au code 32 [26]. Cette méthode présentait de nombreuses contraintes :

- Nécessité de connaître la procédure spécifique à chaque constructeur et modèle.
- Décodage manuel des séquences de clignotements à l'aide d'un tableau de correspondance papier.
- Impossibilité d'accéder aux données en temps réel ou de mesurer des paramètres dynamiques.

- Risque d'erreur de comptage des clignotements, source de mauvais diagnostic.

Malgré ses limitations, cette méthode a constitué pendant une décennie le principal moyen d'accès au diagnostic embarqué pour les techniciens non équipés d'outillage constructeur. Elle illustre parfaitement la nécessité de la standardisation qui allait s'imposer avec l'OBD-II [26].

1.4 Les codes de défaut (DTC) : structure, normes et classification

1.4.1 Naissance et standardisation des DTC modernes

Un DTC (*Diagnostic Trouble Code*) est un code alphanumérique généré et stocké par un ECU lorsqu'il détecte une anomalie dans le fonctionnement d'un système ou d'un composant qu'il supervise [11]. La notion de code de défaut existe depuis les premiers systèmes de diagnostic embarqué des années 1980, mais sa standardisation moderne est indissociable de l'avènement de l'OBD-II.

Avant la standardisation, les codes de défaut étaient purement numériques, propriétaires, et leur signification dépendait entièrement du constructeur. L'introduction de l'OBD-II et des normes associées a imposé un format universel : une lettre suivie de quatre chiffres (ex. : P0301, B1234, C0035, U0100) [37]. Ce format encode à la fois le domaine fonctionnel concerné, le caractère générique ou constructeur du code, et la nature précise du défaut.

Un DTC est enregistré lorsqu'un paramètre surveillé dépasse un seuil de tolérance défini, pendant une durée ou sur un nombre de cycles de conduite suffisant pour confirmer la réalité de l'anomalie (mécanisme dit de confirmation du défaut ou *trip counter*). Certains défauts déclenchent immédiatement le MIL, d'autres nécessitent plusieurs occurrences confirmées avant signalisation [11]. Les DTC sont stockés dans une mémoire non volatile de l'ECU et persistent après l'extinction du moteur. Ils peuvent être effacés manuellement via un outil de diagnostic ou automatiquement par l'ECU après un nombre défini de cycles de conduite sans réapparition du défaut [37].

1.4.2 Les normes de référence : SAE J2012 et ISO 15031-6

La standardisation des DTC repose sur deux normes techniques de référence, complémentaires et largement harmonisées entre elles :

- La **norme SAE J2012**, publiée par la *Society of Automotive Engineers*, définit le format, la structure et la signification des codes de défaut pour les véhicules routiers, dans le contexte réglementaire nord-américain. Elle spécifie notamment la liste des DTC génériques (*standardized* ou *generic*), la signification de chaque position du code, et les règles de création de codes constructeurs [31, 19].
- La **norme ISO 15031-6**, publiée par l'*International Organization for Standardization*, est l'équivalent international de la SAE J2012. Elle s'inscrit dans la famille de normes ISO 15031, dédiée à la communication entre véhicule et outil de diagnostic externe.

L'ISO 15031-6 définit les mêmes éléments que la SAE J2012 dans un cadre normatif international, applicable notamment dans le contexte de l'EOBD européen.

Ces deux normes sont aujourd'hui très largement harmonisées, de sorte qu'un DTC générique défini par la SAE J2012 possède la même signification que son équivalent dans l'ISO 15031-6. Cette harmonisation est essentielle pour garantir l'interopérabilité des outils de diagnostic à l'échelle mondiale [2].

1.4.3 Classification des DTC par domaine fonctionnel (P, B, C, U)

La première lettre d'un DTC identifie le domaine fonctionnel du système concerné par le défaut. Quatre domaines sont définis [21, 31] :

- **P — Powertrain** (Groupe motopropulseur) : regroupe tous les défauts relatifs au moteur, à la transmission, et aux systèmes directement liés aux émissions (injection, allumage, catalyseur, sonde lambda, EGR, turbo, boîte de vitesses automatique). C'est la catégorie la plus étendue, couvrant la majorité des codes définis dans les normes.
- **B — Body** (Carrosserie / équipements habitacle) : concerne les systèmes électroniques de l'habitacle et de la carrosserie : airbags et prétensionneurs, climatisation, sièges à mémoire, rétroviseurs électriques, lève-vitres, éclairage intérieur.
- **C — Chassis** (Châssis) : regroupe les systèmes mécaniques et électroniques liés à la dynamique du véhicule : ABS, ESP, direction assistée électrique (EPS), suspension active, freinage électromécanique.
- **U — Network** (Réseaux de communication) : identifie les défauts de communication entre ECU sur les bus de données internes du véhicule (CAN, LIN, FlexRay). Un code U indique généralement qu'un calculateur ne reçoit plus les messages attendus d'un autre calculateur, ou qu'une communication est perturbée ou absente [35].

La structure complète d'un DTC est la suivante :

P	0	3	0	1
Domaine	Générique/Const.	Sous-système	N° de défaut	spécifique
(P/B/C/U)	(0 = générique)			

1.4.4 DTC génériques et DTC constructeurs

La norme OBD-II distingue deux grandes familles de DTC selon leur origine et leur portée [21, 8] :

- Les **DTC génériques** (*standardized* ou *SAE codes*) sont définis et imposés par les normes SAE J2012 et ISO 15031-6. Ils sont identifiés par un 0 en deuxième position (ex. : P0301 — Raté d'allumage détecté, cylindre 1). Leur signification est universelle et identique pour tous les constructeurs et tous les véhicules OBD-II. Tout outil de diagnostic compatible OBD-II peut lire et interpréter ces codes [8].

- Les **DTC constructeurs** (*manufacturer-specific codes*) sont définis librement par chaque constructeur pour couvrir des défauts non couverts par les codes génériques. Ils sont identifiés par un 1, 2 ou 3 en deuxième position (ex. : P1326 — Détection de cognement moteur (Hyundai/Kia), P1351 — Circuit de commande bobine d'allumage (General Motors)). Leur signification est propre au constructeur et peut varier entre deux marques [21].

Cette dualité a des implications pratiques importantes pour les outils de diagnostic :

- Un outil générique (type ELM327) peut lire et interpréter correctement tous les DTC génériques sur n'importe quel véhicule OBD-II.
- Pour les DTC constructeurs, l'outil doit disposer d'une base de données spécifique à chaque constructeur, ou être conçu pour communiquer avec les protocoles propriétaires de la marque concernée.
- Certains constructeurs chiffrent ou protègent l'accès à leurs DTC constructeurs, limitant leur lecture aux seuls outils officiels ou agréés.

La connaissance de cette distinction est fondamentale pour comprendre les différences entre les catégories d'outils de diagnostic présentées dans la section suivante.

1.5 Les bases de correspondance DTC et les outils de diagnostic

1.5.1 L'apparition des bases de correspondance et leur rôle dans l'interprétation des DTC

Un code de défaut brut, tel que P0301, ne représente en lui-même qu'un identifiant numérique. Pour devenir exploitable par un technicien, il doit être mis en relation avec une description textuelle détaillant la nature de l'anomalie, les systèmes impliqués et, dans l'idéal, les causes probables ainsi que les procédures de diagnostic préconisées. Cette mise en correspondance est réalisée par les bases de données DTC, également appelées bases de correspondance [13, 34].

L'émergence de ces bases est contemporaine de la diffusion à grande échelle de l'OBD-II au cours des années 1990. Initialement, chaque constructeur diffusait ses propres tables de correspondance sous la forme de manuels techniques imprimés (*service manuals*), réservés aux réseaux de concessionnaires agréés. L'essor d'Internet, conjugué au renforcement progressif du cadre légal, a favorisé une ouverture graduelle de ces ressources, avec l'émergence de plateformes en ligne d'abord communautaires puis professionnelles [9].

Sur le plan technique, ces bases de correspondance adoptent des formats variés : fichiers structurés XML ou bases de données relationnelles. Elles fonctionnent comme de véritables dictionnaires, reliant l'identifiant normalisé du défaut à une description textuelle claire de la panne, et parfois à des procédures de réparation détaillées. La gestion centralisée de ces données sur un serveur distant (précisément l'objet du présent mémoire) offre

un double avantage : elle garantit une mise à jour constante du référentiel et permet une interprétation rapide des codes sans alourdir la mémoire du dispositif embarqué [13, 9].

La qualité et l'exhaustivité de la base de correspondance DTC conditionnent directement la valeur d'un outil de diagnostic. Une base qui se limite aux seuls codes génériques s'avère insuffisante pour prendre en charge les défauts spécifiques aux constructeurs. En revanche, une base intégrant les codes constructeurs et régulièrement actualisée autorise un diagnostic d'une grande précision sur un large éventail de véhicules [34].

1.5.2 Les outils de diagnostic : panorama et classification

L'écosystème des outils de diagnostic automobile s'est considérablement développé depuis la standardisation OBD-II. On distingue trois grandes catégories d'outils, différenciées par leur niveau de standardisation, leur couverture de véhicules, et leurs fonctionnalités.

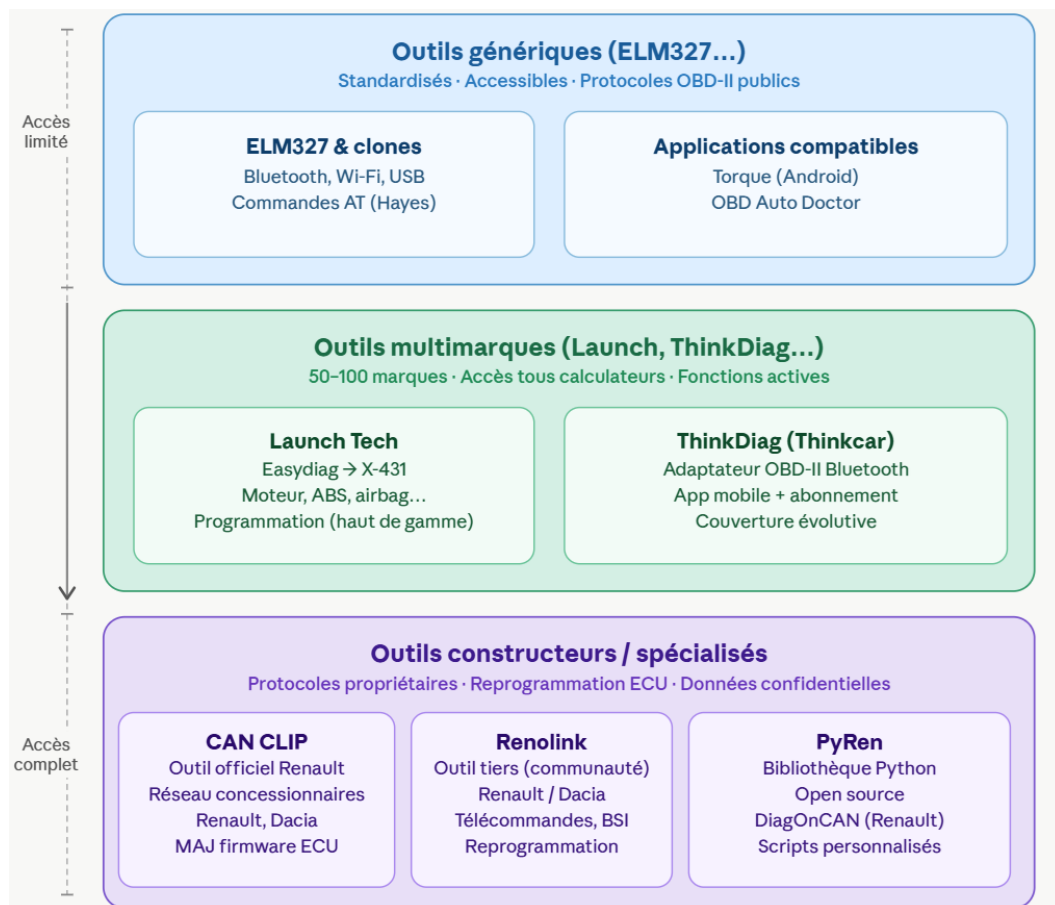


Figure 1.3: Panorama des outils de diagnostic automobile

1.5.2.1 Les outils standards et les interfaces génériques (ELM327, ...)

Les interfaces génériques OBD-II constituent la catégorie la plus accessible et la plus répandue. Elles reposent sur des circuits intégrés standardisés dont le plus emblématique est l'ELM327, développé par la société canadienne ELM Electronics. Ce microcontrôleur implémente l'ensemble des protocoles OBD-II (ISO 9141-2, KWP2000, J1850, CAN ISO 15765) et expose une interface de commande via un jeu d'instructions AT (*Hayes command set*), accessible par liaison série, Bluetooth, Wi-Fi ou USB [8, 40].

L'ELM327 et ses clones permettent à n'importe quelle application (sur smartphone, PC ou tablette) de communiquer avec le port OBD-II d'un véhicule via des commandes simples et documentées. Des applications open source comme Torque (Android) ou OBD Auto Doctor exploitent cette interface pour afficher les DTC et les paramètres temps réel [36].



Figure 1.4: Interface générique OBD-II ELM327

1.5.2.2 Les outils multimarques (Launch, ThinkDiag, ...)

Les outils de diagnostic multimarques représentent un niveau supérieur, combinant compatibilité étendue et accès aux systèmes propriétaires d'un grand nombre de constructeurs. Ces outils intègrent des bases de données constructeurs régulièrement mises à jour, et implémentent les protocoles de communication propriétaires de plusieurs dizaines de marques [24, 14].

Parmi les représentants les plus connus de cette catégorie :

- **Launch Tech** (marque chinoise fondée en 1992) propose une gamme étendue allant de l'adaptateur Bluetooth Easydiag aux stations de diagnostic professionnelles X-431. Ces outils permettent d'accéder à tous les calculateurs d'un véhicule (moteur, boîte, ABS, airbag, climatisation, tableau de bord), de lire et effacer les DTC constructeurs, et pour les modèles haut de gamme, d'effectuer des fonctions de programmation et de calibration [24].



Figure 1.5: Outil de diagnostic multimarque Launch

- **ThinkDiag** (marque de Thinkcar Tech) propose une solution hybride : un adaptateur OBD-II Bluetooth de faible encombrement couplé à une application mobile évolutive par abonnement, permettant d'accéder à une couverture multimarque progressivement enrichie.



Figure 1.6: Outil de diagnostic ThinkDiag

Ces outils se distinguent par une couverture de 50 à 100 marques automobiles selon les modèles, l'accès à la totalité des calculateurs du véhicule, des fonctions actives (activation de composants pour test, réinitialisation de services), et des mises à jour régulières de la base de données constructeurs [14].

1.5.2.3 Les outils constructeurs et spécialisés (CAN CLIP, Renolink, PyRen, ...)

Au sommet de la hiérarchie des outils de diagnostic se trouvent les outils constructeurs et spécialisés, conçus pour offrir un accès complet et profond à l'ensemble des systèmes d'une marque spécifique. Ces outils implémentent les protocoles propriétaires du constructeur dans leur intégralité, et permettent des opérations impossibles avec des outils génériques : reprogrammation d'ECU, appairage de calculateurs, adaptation de composants, accès aux données constructeur confidentielles [1].

- **CAN CLIP** est l'outil de diagnostic officiel du groupe Renault, utilisé dans les réseaux de concessionnaires agréés. Il offre un accès exhaustif à tous les calculateurs des véhicules Renault, Dacia et anciennement Samsung Motors, incluant les fonctions de programmation, de mise à jour de firmware d'ECU, et d'accès aux bases de données techniques Renault [24].



Figure 1.7: Outil constructeur Renault CAN CLIP

- **Renolink** est un outil tiers spécialisé sur les véhicules Renault/Dacia, développé par une communauté indépendante. Il permet notamment la programmation des télécommandes, le remplacement et l'appairage des calculateurs BSI (*Body and Security Interface*), et certaines fonctions de reprogrammation non accessibles via les outils génériques.



Figure 1.8: Interface spécialisée Renolink

- **PyRen** est une bibliothèque Python open source dédiée à la communication avec les véhicules Renault via le protocole DiagOnCAN propriétaire de Renault. Elle permet d'écrire des scripts de diagnostic personnalisés, d'accéder aux paramètres internes des ECU Renault, et de développer des outils de diagnostic sur mesure. PyRen illustre l'émergence d'un écosystème open source autour du diagnostic automobile constructeur.

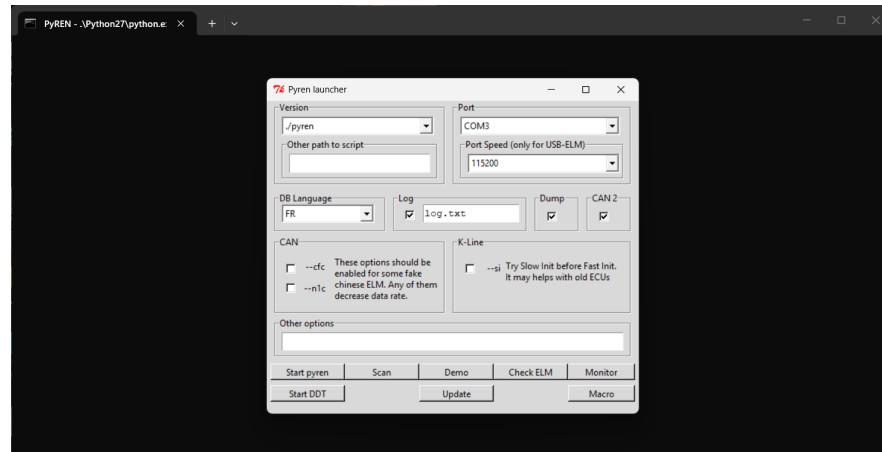


Figure 1.9: Outil de diagnostic à code source ouvert PyRen

1.6 Conclusion

Ce chapitre a permis de retracer le chemin parcouru par l'automobile, depuis ses origines mécaniques jusqu'à son architecture électronique et logicielle actuelle, en mettant en lumière la place centrale qu'occupent les DTC dans le diagnostic moderne [34, 7]. L'analyse des normes SAE J2012 et ISO 15031-6, ainsi que la classification des codes de défaut, a révélé l'existence d'un cadre technique rigoureux sur lequel s'appuient les outils de diagnostic existants.

Ces outils, qu'ils soient génériques, multimarques ou constructeurs, présentent néanmoins des limites en termes d'accessibilité, d'autonomie et d'intégration avec des systèmes distants. C'est précisément dans cette lacune que s'inscrit notre projet, qui vise à concevoir une solution complète alliant acquisition locale des DTC via le port OBD et traitement intelligent côté serveur. Le chapitre suivant présentera les technologies et choix techniques retenus pour la mise en œuvre de cette solution.

L'infrastructure logicielle du système de diagnostic à distance

2.1 Introduction

Le diagnostic automobile à distance repose sur une infrastructure logicielle robuste, capable de recevoir des données issues de véhicules, de les traiter en temps réel et de restituer des résultats exploitables. Cette infrastructure ne se limite pas à un simple programme : elle constitue un assemblage cohérent de technologies complémentaires, chacune jouant un rôle précis dans la chaîne de traitement.

Ce chapitre présente les fondations logicielles sur lesquelles repose notre système. Après avoir défini le concept de serveur web et l'architecture client-serveur qui gouverne les échanges dans notre système, nous introduisons Django, le framework Python retenu pour le développement du serveur applicatif, en détaillant son architecture interne et ses composants. Nous exposons ensuite la structure des fichiers XML utilisés comme référentiel de données DTC Renault/Dacia, avant de présenter PostgreSQL, le système de gestion de base de données choisi pour la persistance et l'interrogation de ces données.

2.2 Les serveurs web : concepts fondamentaux

2.2.1 Définition et rôle d'un serveur web

Un serveur web est un système informatique (désignant à la fois le matériel et le logiciel qui y est exécuté) dont la fonction principale est de recevoir des requêtes provenant de clients distants, de les traiter, et de leur retourner une réponse appropriée. Dans sa forme la plus élémentaire, un serveur web reçoit des requêtes formulées selon le protocole HTTP (HyperText Transfer Protocol) ou sa variante sécurisée HTTPS, et y répond en fournissant des ressources : pages HTML, données JSON, fichiers, ou résultats de traitements [32, 23, 25]. Sur le plan logiciel, un serveur web remplit plusieurs fonctions fondamentales :

- La réception et le routage des requêtes : le serveur écoute en permanence sur un port réseau défini (généralement le port 80 pour HTTP et 443 pour HTTPS) et dirige chaque requête entrante vers le module de traitement approprié [23].
- L'exécution de la logique applicative : au-delà de la simple fourniture de fichiers statiques, les serveurs web modernes exécutent du code applicatif capable d'interagir avec des bases de données, de valider des données, et de produire des réponses dynamiques [25].
- La gestion des sessions et de l'authentification : le serveur maintient l'état des connexions utilisateurs et contrôle l'accès aux ressources protégées [32].
- La journalisation et la supervision : tout événement significatif (requête reçue, erreur, temps de réponse) est enregistré pour permettre la supervision et le débogage [23].

Dans le contexte du diagnostic automobile à distance, le serveur web constitue le point central de l'architecture : il reçoit les codes DTC transmis par le dispositif embarqué, interroge la base de données pour en extraire la signification, et retourne au client une réponse structurée contenant l'interprétation du défaut.

2.2.2 Architecture client-serveur

L'architecture client-serveur est un modèle d'organisation des systèmes informatiques distribués dans lequel les tâches et les responsabilités sont réparties entre deux catégories d'entités : les clients, qui initient des demandes de service, et les serveurs, qui traitent ces demandes et fournissent les ressources ou les résultats correspondants [32, 23, 25].

Le client est l'entité qui formule une requête. Il peut s'agir d'un navigateur web, d'une application mobile, d'un script automatisé, ou comme dans notre cas un dispositif embarqué OBD transmettant des codes de défaut. Le client n'a généralement pas connaissance de la manière dont le serveur traite sa requête ; il se contente d'envoyer une demande et d'attendre une réponse [32].

Le serveur est l'entité qui reçoit la requête, exécute le traitement nécessaire (consultation de base de données, calcul, transformation de données), et retourne une réponse au client. Le serveur est conçu pour traiter simultanément les requêtes de multiples clients, ce qui implique une gestion efficace de la concurrence et des ressources [25, 18].

Les échanges entre client et serveur sont régis par des protocoles standardisés. Dans le contexte des applications web modernes, le protocole HTTP/HTTPS structure ces échanges autour de méthodes sémantiques [32, 23].

Cette architecture présente plusieurs avantages déterminants pour notre projet : la centralisation de la logique de traitement sur le serveur facilite la maintenance et les mises à jour du référentiel DTC sans intervention sur les dispositifs embarqués ; la scalabilité permet d'étendre le service à un nombre croissant de véhicules connectés ; et l'indépendance entre client et serveur autorise l'évolution de chacun sans impacter l'autre [23, 25].

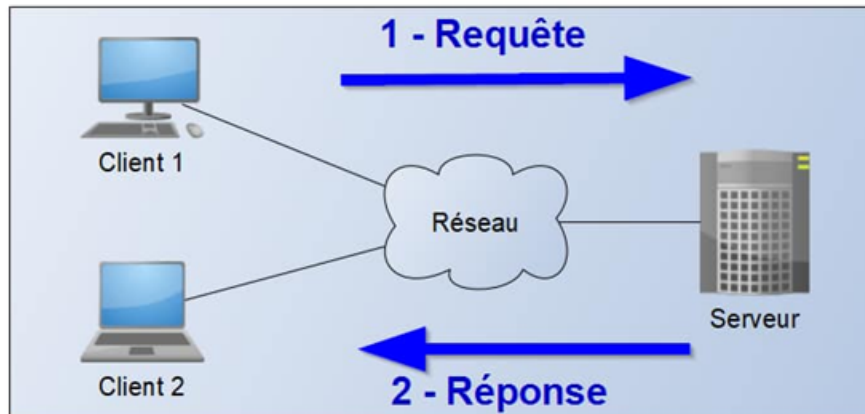


Figure 2.1: Architecture client-serveur

2.3 Django : un cadre de développement web en Python

2.3.1 Présentation de Django

Django est un framework web open source écrit en langage Python, publié initialement en 2005. Il est aujourd'hui maintenu par la Django Software Foundation et constitue l'un des frameworks web les plus utilisés au monde, aux côtés de Ruby on Rails et de Spring Boot [3, 15].



Figure 2.2: Logo du cadre de développement Django

La philosophie de Django repose sur deux principes directeurs explicitement formulés dans sa documentation officielle [3] :

Django se distingue par son approche *"batteries included"* : le framework embarque nativement un grand nombre de fonctionnalités prêtes à l'emploi [15], notamment :

- Un ORM (Object-Relational Mapper) permettant d'interagir avec la base de données via des objets Python sans écrire de SQL brut [3, 15].
- Un système d'authentification complet (gestion des utilisateurs, groupes, permissions) [3].
- Un panneau d'administration automatiquement généré à partir des modèles de données [15].
- Un système de routage d'URL flexible et expressif [3].
- Un moteur de templates pour la génération de réponses HTML dynamiques [3].
- Des mécanismes de sécurité intégrés contre les attaques CSRF, XSS, et les injections SQL [6].

Sa maturité, sa documentation abondante et sa large communauté en font un choix pertinent pour des projets nécessitant fiabilité, évolutivité et rapidité de développement [3, 15, 5]. Dans le cadre de ce mémoire, Django constitue le socle logiciel du serveur de diagnostic, assurant l'interface entre les requêtes des clients distants, la logique d'interprétation des DTC et l'accès aux données persistantes.

2.3.2 L'architecture MVT (Modèle-Vue-Gabarit)

Django adopte une architecture dérivée du modèle MVC (Model-View-Controller) traditionnel, désignée sous l'acronyme MVT pour Model-View-Template [3, 15]. Cette architecture établit une séparation claire des responsabilités au sein de l'application:

Le **Modèle (Model)** représente la couche d'accès aux données. Il définit la structure de la base de données sous forme de classes Python, que l'ORM de Django traduit automatiquement en tables relationnelles. Le modèle encapsule également les règles métier relatives aux données (validations, relations, contraintes) et expose une interface cohérente pour les manipulations (création, lecture, mise à jour, suppression) [15, 6].

La **Vue (View)** constitue le cœur de la logique applicative. Elle reçoit une requête HTTP, interagit avec les modèles pour récupérer ou modifier les données nécessaires, applique les traitements requis, puis sélectionne et alimente un template ou retourne directement une réponse structurée (par exemple du JSON dans le cas d'une API REST). Dans l'interprétation MVT de Django, la vue joue le rôle du contrôleur : elle orchestre le flux de traitement sans gérer elle-même le rendu final [3, 15].

Le **Template (Template)** est responsable de la présentation. Il s'agit d'un fichier texte contenant du code HTML mêlé à un langage de gabarit simple (Django Template Language), permettant d'afficher dynamiquement les données transmises par la vue. Dans une API, cette couche peut être absente ou remplacée par un sérialiseur qui transforme les données en format JSON ou XML [3, 6].

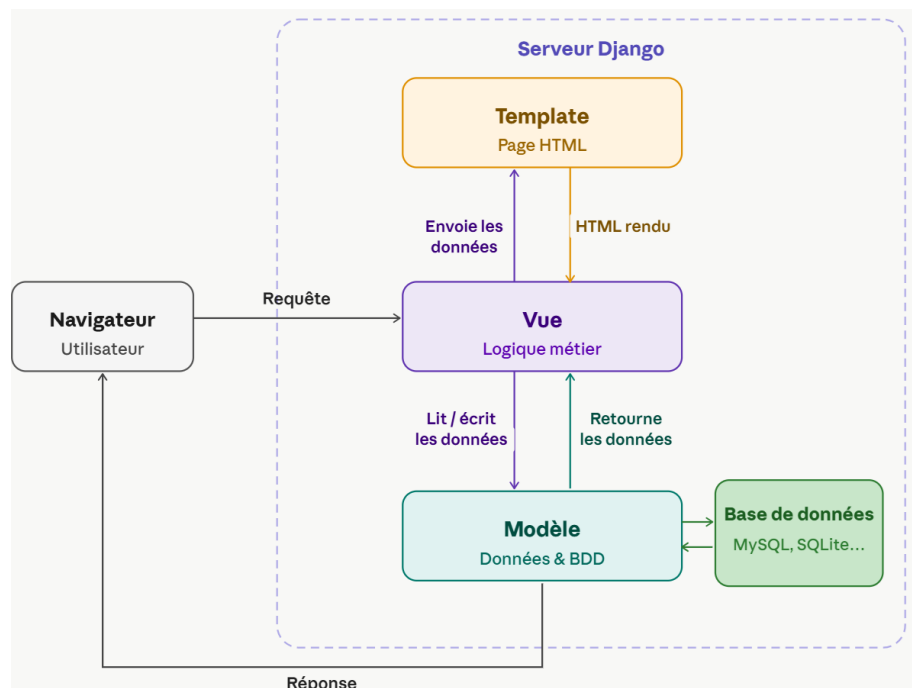


Figure 2.3: Architecture MVT de Django

2.3.3 Cas d'utilisation et domaines d'application de Django

Django est employé dans des contextes extrêmement variés, allant des sites de presse à fort trafic (The Washington Post, The Guardian) aux plateformes de médias sociaux (Instagram, Pinterest) [3, 15], en passant par des applications scientifiques, des solutions de e-commerce et des portails d'entreprise. Ses atouts tels que le développement rapide, la robustesse, la sécurité et un écosystème riche, le rendent particulièrement adapté aux projets où la fiabilité des données est critique et où le temps de développement doit être optimisé [5].

2.4 Architecture du serveur Django

2.4.1 Les composants principaux du serveur Django

Un projet Django est structuré en composants organisés hiérarchiquement [3, 15]. Au niveau supérieur se trouve le projet (*project*), qui représente l'ensemble de l'application web et contient la configuration globale. À l'intérieur du projet, on distingue des applications (*apps*), qui sont des modules autonomes et réutilisables regroupant chacun une fonctionnalité cohérente.

Les composants principaux d'un projet Django sont les suivants :

- `settings.py` : fichier de configuration central du projet. Il définit les paramètres de connexion à la base de données, la liste des applications installées, les middlewares actifs, la configuration des fichiers statiques et médias, les clés secrètes de sécurité, et les paramètres du serveur de messagerie le cas échéant [3].
- `urls.py` (project-level) : fichier de routage principal. Il définit la correspondance entre les URLs entrantes et les vues chargées de les traiter. Il peut déléguer à des fichiers `urls.py` d'applications spécifiques via la directive `include()` [15].
- `models.py` : définit la structure des données de l'application sous forme de classes Python. Chaque modèle est traduit en table de base de données par l'ORM de Django [3, 15].
- `views.py` : contient la logique de traitement des requêtes. Chaque vue reçoit un objet `HttpRequest`, effectue le traitement nécessaire, et retourne un objet `HttpResponse` ou `Render` (dans le cas des pages HTML) [15].
- `admin.py` : enregistre les modèles auprès du panneau d'administration Django, permettant leur gestion via une interface graphique automatiquement générée [3].

L'assemblage de ces composants, orchestré par le cycle requête-réponse de Django, constitue une architecture modulaire et extensible, dans laquelle chaque brique peut être adaptée ou remplacée en fonction des besoins du projet [3, 15].

2.4.2 Le mécanisme de routage des requêtes

Le routage est le processus par lequel le serveur Django détermine quelle vue doit être exécutée pour une requête HTTP donnée, en fonction de l'URL demandée [3]. Ce mécanisme repose sur un ou plusieurs fichiers de configuration d'URL (`urls.py`) qui contiennent des motifs d'URL associés à des fonctions ou des classes de vue. Lorsqu'une requête parvient au serveur, Django parcourt ces motifs dans l'ordre de déclaration et s'arrête au premier qui correspond. Cette approche, simple et déclarative, permet une gestion claire des points d'entrée de l'application [15].

2.4.3 Relation entre les points de terminaison et les vues

Dans un serveur fondé sur le framework Django, la relation entre les *endpoints* (points d'accès URL) et les vues constitue un élément fondamental de l'architecture applicative [3, 15]. Un *endpoint* représente une interface d'accès aux ressources ou aux fonctionnalités du système à travers une URL spécifique. Il définit notamment les méthodes HTTP autorisées, telles que GET ou POST, ainsi que, le cas échéant, le format et la structure des données échangées entre le client et le serveur [18].

La vue associée à cet *endpoint* assure le traitement de la requête reçue. Elle prend en charge l'exécution de la logique métier nécessaire, l'interaction éventuelle avec la base de données, ainsi que la génération de la réponse appropriée destinée au client. Ainsi, chaque *endpoint* agit comme un point d'entrée vers une vue chargée de réaliser les opérations correspondant à la requête formulée [3, 15].

2.5 Architecture et fonctionnement des fichiers XML dans la base de données DDT2000 de Renault/Dacia

2.5.1 Présentation du logiciel DDT2000

DDT2000 (Diagnostic Data Tool 2000) est un logiciel de diagnostic constructeur utilisé par Renault et Dacia pour communiquer avec les calculateurs électroniques des véhicules. Il permet d'effectuer des opérations de diagnostic, de lecture de défauts, de configuration et de tests d'actionneurs. Son fonctionnement repose principalement sur une base de données technique contenant des fichiers XML décrivant les différents calculateurs automobiles [22].

2.5.2 Rôle des fichiers XML dans DDT2000

Les fichiers XML sont utilisés dans DDT2000 afin de structurer et d'organiser les informations nécessaires à la communication avec les calculateurs électroniques. Le langage XML offre une représentation hiérarchique claire, lisible et facilement exploitable par le logiciel [27, 41]. Ces fichiers permettent ainsi de définir les protocoles de communication, les commandes de diagnostic et l'interprétation des données reçues depuis le véhicule [41, 22].

2.5.3 Structure générale d'un fichier XML DDT2000

Un fichier XML DDT2000 est généralement organisé en plusieurs sections décrivant les caractéristiques d'un calculateur automobile. Parmi ces sections figurent les informations générales du calculateur, les protocoles de communication, les requêtes de diagnostic, les variables de données et les commandes de contrôle [41, 22]. Cette structure hiérarchique facilite l'interprétation des informations techniques par le logiciel de diagnostic [27].

2.5.4 Les sections importantes d'un fichier XML

Les sections principales d'un fichier XML DDT2000 remplissent des fonctions spécifiques dans le processus de diagnostic. La section « Protocoles » définit le mode de communication utilisé, tandis que les sections « Requêtes » et « Variables » décrivent respectivement les commandes envoyées et les méthodes d'interprétation des réponses du calculateur [41]. D'autres sections permettent également de gérer les tests d'actionneurs et l'affichage graphique des données [22].

2.5.5 Fonctionnement global du système

Le fonctionnement de DDT2000 repose sur une interaction entre le logiciel, l'interface de communication et le calculateur électronique du véhicule. Le logiciel charge d'abord le fichier XML correspondant au calculateur ciblé, puis utilise les informations qu'il contient pour envoyer des requêtes via les protocoles CAN ou K-Line [22]. Les réponses reçues sont ensuite décodées et affichées sous une forme compréhensible pour l'utilisateur [41].

2.6 PostgreSQL : le système de gestion de base de données du serveur

2.6.1 Présentation de PostgreSQL et ses caractéristiques

PostgreSQL est un système de gestion de base de données relationnelle (SGBD) open source, reconnu pour sa robustesse, sa fiabilité et sa conformité aux standards SQL. Développé initialement à l'Université de Californie à Berkeley, PostgreSQL est aujourd'hui maintenu par une communauté internationale de développeurs [10, 30].

Il permet de stocker, organiser, manipuler et sécuriser de grandes quantités de données. PostgreSQL est utilisé dans de nombreux domaines tels que les applications web, les systèmes d'information, l'analyse de données, les applications financières et les plateformes à grande échelle [4, 10].



Figure 2.4: Logo de PostgreSQL

PostgreSQL se distingue des autres SGBDR par un ensemble de caractéristiques techniques :

- **Open Source** : PostgreSQL est un système de gestion de base de données libre et gratuit. Son code source est accessible et peut être modifié ou distribué librement [4].

- **Compatibilité SQL** : Il respecte les standards du langage SQL, ce qui facilite la création, la manipulation et l'interrogation des bases de données [12].
- **Fiabilité des transactions** : PostgreSQL garantit l'intégrité des données grâce au respect des propriétés ACID, assurant des transactions sécurisées et cohérentes [4, 10].
- **Haute performance** : Il dispose d'un moteur d'optimisation performant permettant d'exécuter rapidement des requêtes simples ou complexes [30, 33].
- **Extensibilité** : Le système permet d'ajouter des extensions, des types de données personnalisés et des fonctions spécifiques selon les besoins [4].
- **Sécurité** : PostgreSQL propose des mécanismes avancés d'authentification, de gestion des rôles et de contrôle des accès aux données [4].
- **Gestion des données complexes** : Il prend en charge des formats modernes comme JSON, XML et les données géographiques via des extensions spécialisées [30].
- **Multi-utilisateur** : Plusieurs utilisateurs peuvent accéder simultanément à la base de données sans conflit grâce à la gestion de concurrence [4].
- **Sauvegarde et réplication** : PostgreSQL offre des outils de sauvegarde et de réplication assurant la disponibilité et la récupération des données [33].
- **Portabilité** : Il fonctionne sur plusieurs systèmes d'exploitation comme Linux, Windows et macOS, ce qui facilite son déploiement [12].
- **Communauté active** : PostgreSQL bénéficie d'une grande communauté internationale qui contribue continuellement à son amélioration et à sa documentation [30].
- **Stabilité** : Le système est reconnu pour sa robustesse et sa capacité à gérer efficacement de grandes quantités de données sur le long terme [4, 10].

2.7 Conclusion

Ce chapitre a posé les fondations conceptuelles et techniques de l'infrastructure logicielle de notre système de diagnostic à distance. Partant du modèle client-serveur qui gouverne les échanges entre le dispositif embarqué et le serveur, nous avons présenté Django comme le framework structurant le serveur applicatif, en détaillant son architecture MVT et la relation entre ses *endpoints* et ses vues. Nous avons ensuite décrit les fichiers XML comme vecteur de structuration du référentiel DTC Renault/Dacia, et PostgreSQL comme moteur de persistance et d'interrogation des données.

L'ensemble de ces éléments constitue le socle conceptuel et technologique du serveur. Le chapitre suivant détaillera la méthodologie d'implémentation : il s'agira de décrire comment ces composants ont été assemblés, configurés et programmés pour aboutir à un système fonctionnel de diagnostic à distance.

Implémentation et méthodologie

3.1 Introduction

Après avoir présenté, dans le chapitre précédent, les fondements théoriques de l'infrastructure logicielle du système de diagnostic à distance, ce chapitre décrit la mise en œuvre pratique de la solution proposée. L'objectif n'est plus seulement d'expliquer les technologies utilisées, mais de montrer comment elles ont été assemblées pour former une plateforme fonctionnelle capable de recevoir des données issues du diagnostic automobile, de les stocker, de les interpréter et de les rendre accessibles à distance.

Dans un système de diagnostic classique, l'interprétation des codes défauts dépend souvent d'un outil local, d'une base de données intégrée ou d'une documentation consultée manuellement par le technicien. Cette approche présente plusieurs limites : la mise à jour des données est difficile, les outils professionnels restent coûteux, et l'accès à l'information est limité à la personne présente devant le véhicule. Dans notre cas, le besoin est différent : il s'agit de centraliser l'intelligence du diagnostic dans un serveur distant, capable de dialoguer avec une base de données structurée et d'offrir une réponse exploitable à un utilisateur ou à un expert, même lorsque celui-ci n'est pas physiquement présent devant le véhicule.

La plateforme développée doit donc assurer plusieurs fonctions complémentaires : recevoir les requêtes envoyées depuis une application ou un dispositif connecté au port OBD, rechercher l'interprétation correspondante dans une base de données PostgreSQL, retourner une réponse claire sous forme de données structurées, et permettre une communication à distance en temps réel avec le véhicule. Cette logique permet de transformer le port OBD, normalement accessible uniquement localement, en un point d'accès distant supervisé par un serveur robuste, capable de fournir des interprétations fiables et de faciliter l'intervention d'un expert à distance.

3.2 Solution proposée

La solution proposée repose sur une architecture client-serveur. Le serveur constitue le centre du système. Il est développé avec le framework Django, choisi pour sa robustesse, sa clarté et sa capacité à organiser proprement les différentes parties d'une application web. Ce serveur reçoit les requêtes HTTP envoyées par les clients, appelle les fonctions de traitement nécessaires, interroge la base de données PostgreSQL, puis renvoie les résultats sous forme de réponses exploitables.

La base de données PostgreSQL joue le rôle de référentiel central. Elle contient deux types d'informations. Le premier type concerne les codes DTC standards, c'est-à-dire les codes génériques associés à leurs causes probables, symptômes et réparations possibles.

Le deuxième type concerne les données extraites des fichiers XML Renault/Dacia, utilisées pour interpréter des informations plus spécifiques aux calculateurs, comme les requêtes à transmettre, les réponses attendues, les positions des données utiles dans les trames, ainsi que les méthodes d'interprétation.

L'accès distant au serveur est assuré par Cloudflared, qui permet de créer un tunnel sécurisé entre la machine locale exécutant le serveur Django et Internet. Ainsi, même si le serveur tourne localement sur le port 8000, il devient accessible depuis l'extérieur à travers une adresse HTTPS générée par Cloudflare.

Enfin, une application mobile développée avec **MIT App Inventor** permet d'ouvrir une session de diagnostic en direct à distance. Elle peut également transmettre les requêtes d'un expert distant au véhicule et renvoyer en temps réel les réponses provenant du port OBD.

3.3 Création du serveur Django

3.3.1 Création de l'environnement de travail

La première étape consiste à créer un environnement de travail isolé pour le projet. En programmation Python, un environnement virtuel permet de regrouper les bibliothèques nécessaires à un projet sans modifier l'installation principale de Python sur l'ordinateur. Cela évite les conflits entre plusieurs projets utilisant des versions différentes de bibliothèques.

La création de l'environnement virtuel se fait avec la commande suivante :

```
python -m venv venv
```

Après sa création, l'environnement doit être activé. Sous Windows, la commande utilisée est :

```
venv\Scripts\activate
```

Une fois l'environnement activé, les bibliothèques nécessaires peuvent être installées. Dans notre cas, la bibliothèque principale est Django, utilisé pour créer le serveur web.

```
pip install django
```

3.3.2 Création du projet Django

Un projet Django représente la structure générale du serveur. Il contient les fichiers de configuration, les paramètres de connexion à la base de données, les routes principales et les modules de lancement.

La création du projet s'effectue avec la commande suivante :

```
django-admin startproject diagnostic_server
```

Cette commande génère un dossier contenant plusieurs fichiers importants, notamment `settings.py`, `urls.py` et `manage.py`.

Le fichier `manage.py` permet de lancer et d'administrer le projet. Le fichier `settings.py` regroupe les paramètres généraux du serveur, comme la configuration de la base de données, les applications installées et les paramètres de sécurité. Le fichier `urls.py` définit les chemins d'accès, aussi appelés endpoints, qui permettent aux clients d'appeler les différentes fonctions du serveur.

3.3.3 Création d'une application Django

Dans Django, une application est un module interne qui regroupe une fonctionnalité précise. Dans notre projet, nous avons créé une application dédiée au diagnostic automobile.

```
python manage.py startapp diagnostic
```

Cette application contient notamment les fichiers suivants :

- `views.py` : contient les fonctions exécutées lorsqu'un client appelle un endpoint ;
- `urls.py` : contient les URLs propres à l'application ;
- `models.py` : peut contenir les modèles de données si l'ORM Django est utilisé ;
- `apps.py` : décrit la configuration de l'application.

L'application doit ensuite être ajoutée dans le fichier `settings.py`, dans la liste `INSTALLED_APPS`, afin que Django la reconnaisse comme partie intégrante du projet.

```
INSTALLED_APPS = [  
    ...  
    'diagnostic',  
]
```

3.3.4 Lancement du serveur sur le port 8000

Une fois la structure du projet créée, le serveur peut être lancé localement à l'aide de la commande suivante :

```
python manage.py runserver 8000
```

Par défaut, Django démarre un serveur de développement accessible à l'adresse :

```
http://127.0.0.1:8000/
```

Cette adresse signifie que le serveur est accessible uniquement depuis la machine locale. Pour autoriser l'accès depuis d'autres appareils du même réseau, on peut utiliser :

```
python manage.py runserver 0.0.0.0:8000
```

Dans ce cas, Django écoute sur toutes les interfaces réseau de la machine. Cette configuration est utile pour tester le serveur depuis un autre appareil, comme un smartphone connecté au même réseau.

3.4 Création de la base de données PostgreSQL

PostgreSQL a été choisi comme système de gestion de base de données pour stocker les informations nécessaires au diagnostic. Il permet de gérer des données structurées, volumineuses et fiables. Dans notre cas, il est utilisé pour stocker les codes DTC standards ainsi que les données extraites des fichiers XML Renault/Dacia.

3.4.1 Installation de PostgreSQL

Sous Windows, PostgreSQL peut être installé à partir de son installateur officiel. Pendant l'installation, plusieurs éléments sont configurés, notamment le mot de passe de l'utilisateur principal postgres, le port d'écoute du serveur PostgreSQL et l'outil d'administration pgAdmin.

Le port par défaut de PostgreSQL est : 5432.

Ce port permet aux applications externes, comme le serveur Django, de se connecter au moteur PostgreSQL.

Après l'installation, il est possible d'utiliser pgAdmin ou l'outil en ligne de commande psql pour créer une base de données dédiée au projet. La création de la base peut être effectuée avec les commandes suivantes :

```
CREATE DATABASE obd_database;  
CREATE USER obd_user WITH PASSWORD 'mot_de_passe';  
GRANT ALL PRIVILEGES ON DATABASE obd_database TO obd_user;
```

À ce stade, la base de données est créée, mais les tables ne sont pas encore définies. Leur création fait partie de l'étape suivante, appelée préparation de la base de données.

3.5 Préparation de la base de données

La préparation de la base de données consiste à définir les structures nécessaires au stockage des informations de diagnostic. Dans notre système, deux tables principales ont été créées : une table pour les DTC standards et une table pour les données issues des fichiers XML.

3.5.1 Table DTC_standard

La table DTC_standard contient les informations générales associées aux codes DTC standards. Elle permet d'obtenir rapidement une première interprétation d'un code défaut, avec ses causes probables, ses symptômes et les réparations possibles.

La structure proposée est la suivante :

```
CREATE TABLE DTC_standard (  
    id SERIAL PRIMARY KEY,  
    dtc VARCHAR(10) NOT NULL,  
    causes TEXT,  
    symptomes TEXT,  
    reparations TEXT  
);
```

Le champ id représente l'identifiant unique de chaque ligne. Le champ dtc contient le code défaut, par exemple P0301 ou P0135. Le champ causes contient les causes probables du défaut. Le champ symptomes décrit les manifestations possibles observées sur le véhicule. Enfin, le champ réparations contient les actions de réparation ou de vérification proposées.

Cette table joue le rôle d'un dictionnaire de base. Lorsqu'un code DTC est reçu par le serveur, celui-ci peut rechercher ce code dans la table et retourner à l'utilisateur une explication claire.

3.5.2 Table DTC_XML

La table DTC_XML contient les données extraites des fichiers XML Renault/Dacia. Ces fichiers sont plus riches que les simples bases de correspondance DTC, car ils contiennent des informations techniques sur les calculateurs, les requêtes à envoyer, les réponses attendues et les méthodes d'interprétation des trames.

La structure proposée est la suivante :

```
CREATE TABLE DTC_XML (  
    id SERIAL PRIMARY KEY,  
    Adresse VARCHAR(20),  
    AutoIdents JSONB,  
    Date_fichier DATE,  
    Nom_fichier TEXT,  
    Datas XML  
);
```

Le champ `id` est l'identifiant unique de chaque fichier XML. Le champ `Adresse` indique l'adresse du calculateur (une adresse peut correspondre à un calculateur moteur, airbag ou tableau de bord). Le champ `AutoIdents` contient une liste de dictionnaires sous format JSON. Ces informations servent à identifier automatiquement le calculateur et le fichier XML correspondant. Le champ `Date_fichier` indique la date associée au fichier XML. Le champ `Nom_fichier` contient le nom du fichier XML source, ce qui permet de retrouver l'origine de chaque donnée. Le champ `Datas` contient le contenu XML utile. Cette colonne est importante, car elle regroupe les informations nécessaires au dialogue avec le calculateur : les requêtes à transmettre, l'interprétation des réponses, ainsi que la position des données utiles dans la trame reçue.

3.6 Remplissage de la table DTC_standard par extraction automatisée de données web

3.6.1 Principe de l'extraction automatisée de données web

Le web scraping est une technique qui consiste à récupérer automatiquement des informations depuis des pages web. Au lieu de copier manuellement les données affichées sur un site, un programme lit le contenu de la page, extrait les parties utiles, puis les enregistre dans une base de données.

Dans notre cas, cette méthode a été utilisée pour récupérer une base de données open source contenant des codes DTC standards avec leurs descriptions, causes, symptômes et réparations. Les données extraites ont ensuite été insérées dans la table `DTC_standard`.

Un exemple simple permet d'illustrer le principe. Supposons qu'une page web contienne une liste de codes DTC sous forme de tableau. Un script Python peut ouvrir cette page, lire les lignes du tableau, extraire chaque code et sa description, puis les stocker dans PostgreSQL.

Exemple simplifié :

```
import requests  
from bs4 import BeautifulSoup  
  
url = "https://exemple.com/dtc"  
page = requests.get(url)
```

```

soup = BeautifulSoup(page.text, "html.parser")

lignes = soup.find_all("tr")

for ligne in lignes:
    colonnes = ligne.find_all("td")
    if len(colonnes) >= 4:
        dtc = colonnes[0].text.strip()
        causes = colonnes[1].text.strip()
        symptomes = colonnes[2].text.strip()
        reparations = colonnes[3].text.strip()

        print(dtc, causes, symptomes, reparations)

```

Ce script montre le fonctionnement général : le programme télécharge une page, analyse son contenu HTML, cherche les lignes du tableau, puis récupère les colonnes utiles. Dans l'implémentation réelle, ces données sont ensuite insérées dans la table DTC_standard.

3.6.2 Insertion des données dans PostgreSQL

Après extraction, les données sont insérées dans PostgreSQL. Cette étape permet de transformer des informations disponibles sous forme de pages web en données structurées, directement exploitables par le serveur Django.

L'insertion peut être réalisée avec la bibliothèque `psycopg2` :

```

import psycopg2

connexion = psycopg2.connect(
    dbname="obd_database",
    user="obd_user",
    password="mot_de_passe",
    host="localhost",
    port="5432"
)

curseur = connexion.cursor()

requete = """
INSERT INTO DTC_standard (dtc, causes, symptomes, reparations)
VALUES (%s, %s, %s, %s)
"""

curseur.execute(requete, (dtc, causes, symptomes, reparations))

connexion.commit()
curseur.close()
connexion.close()

```

Cette opération est répétée pour chaque code DTC récupéré. À la fin du processus, la table contient une base d'interprétation exploitable par le serveur.

3.7 Analyse des fichiers XML pour remplir la table DTC_XML

3.7.1 Principe de l'analyse XML

Le parsing XML consiste à lire un fichier XML et à extraire automatiquement les informations importantes qu'il contient. Contrairement à un fichier texte simple, un fichier XML est structuré avec des balises. Chaque balise peut contenir une information précise, par exemple le nom d'un calculateur, une adresse de communication, une requête de diagnostic ou une méthode d'interprétation.

Dans notre système, les fichiers XML Renault/Dacia sont utilisés comme source technique pour enrichir la base de données. Le serveur peut ensuite retrouver le fichier correspondant à un calculateur donné et utiliser les données qu'il contient pour interpréter les réponses reçues depuis le véhicule.

3.7.2 Extraction des informations utiles

Le parser parcourt chaque fichier XML et extrait les informations nécessaires au remplissage de la table DTC_XML.

Les informations extraites sont les suivantes :

- l'adresse du calculateur ou du contexte de communication ;
- les informations d'identification automatique du calculateur ;
- la date du fichier ;
- le nom du fichier XML ;
- le contenu XML utile, conservé dans la colonne Datas.

La colonne Datas est volontairement conservée sous forme XML afin de garder la structure originale des informations. Elle contient notamment les requêtes qui doivent être envoyées au calculateur, les réponses attendues, les règles de décodage, et la position des données utiles dans la trame. Par exemple, une réponse peut contenir plusieurs octets, mais seulement certains octets représentent la valeur recherchée. Le fichier XML permet alors d'indiquer où se trouve cette valeur et comment elle doit être convertie pour devenir compréhensible.

Un exemple simplifié de parsing peut être présenté comme suit :

```
import xml.etree.ElementTree as ET

chemin_fichier = "exemple.xml"
arbre = ET.parse(chemin_fichier)
racine = arbre.getroot()

nom_fichier = "exemple.xml"
adresse = racine.findtext("./Address")
date_fichier = racine.findtext("./Date")

autoidents = []
for autoident in racine.findall("./AutoIdent"):
    autoidents.append(autoident.attrib)
```

```
datas = ET.tostring(racine, encoding="unicode")
```

Dans cet exemple, le programme lit le fichier XML, cherche certaines balises, extrait leurs valeurs, puis conserve le contenu XML sous forme de texte structuré. Ces données sont ensuite insérées dans PostgreSQL.

3.7.3 Insertion dans la table DTC_XML

L'insertion des données extraites dans la table DTC_XML peut être réalisée avec psycopg2 :

```
import json
import psycopg2

connexion = psycopg2.connect(
    dbname="obd_database",
    user="obd_user",
    password="mot_de_passe",
    host="localhost",
    port="5432"
)

curseur = connexion.cursor()

requete = """
INSERT INTO DTC_XML (Adresse, AutoIdents, Date_fichier, Nom_fichier, Datas)
VALUES (%s, %s, %s, %s, %s)
"""

curseur.execute(
    requete,
    (
        adresse,
        json.dumps(autoidents),
        date_fichier,
        nom_fichier,
        datas
    )
)

connexion.commit()
curseur.close()
connexion.close()
```

Cette méthode permet de transformer un ensemble de fichiers XML en une base de données interrogeable. Au lieu de chercher manuellement dans des fichiers, le serveur peut effectuer une recherche directe dans PostgreSQL.

3.8 Connexion entre Django et PostgreSQL avec psycopg2

Pour permettre au serveur Django d'interroger la base de données, nous avons utilisé la bibliothèque psycopg2. Cette bibliothèque assure la communication entre Python et PostgreSQL. Elle permet d'ouvrir une connexion, d'exécuter des requêtes SQL, de récupérer les résultats, puis de fermer proprement la connexion.

Le principe général est le suivant :

```
import psycopg2

def get_connection():
    return psycopg2.connect(
        dbname="obd_database",
        user="obd_user",
        password="mot_de_passe",
        host="localhost",
        port="5432"
    )
```

Cette fonction peut être appelée depuis les vues Django. Par exemple, lorsqu'un client envoie un code DTC au serveur, une vue Django peut récupérer ce code, ouvrir une connexion à PostgreSQL, rechercher l'interprétation correspondante, puis retourner le résultat.

Exemple simplifié :

```
from django.http import JsonResponse
import psycopg2

def rechercher_dtc(request):
    code = request.GET.get("dtc")
    connexion = get_connection()
    curseur = connexion.cursor()
    curseur.execute(
        "SELECT dtc, causes, symptomes, reparations "
        "FROM DTC_standard WHERE dtc = %s",
        (code,)
    )
    resultat = curseur.fetchone()

    curseur.close()
    connexion.close()

    if resultat:
        return JsonResponse({
            "dtc": resultat[0],
            "causes": resultat[1],
            "symptomes": resultat[2],
            "reparations": resultat[3]
        })
    else:
        return JsonResponse({
```

```

        "erreur": "Code DTC non trouvé"
    })

```

Cette vue transforme une simple requête HTTP en une recherche dans la base de données. Le résultat est renvoyé au client sous forme JSON, un format léger et facile à utiliser par une application mobile ou web.

3.9 Définition des points de terminaison du serveur

Les endpoints sont les adresses qui permettent aux clients d'accéder aux fonctions du serveur. Chaque endpoint est associé à une vue Django. Lorsqu'un client appelle une URL, Django cherche la vue correspondante et exécute le traitement associé.

Exemple de configuration dans le fichier `diagnostic/urls.py` :

```

from django.urls import path
from . import views

urlpatterns = [
    path('ver',          views.ver,          name='ver'),
    path('diag',        views.diag,         name='diag'),
    path('report',       views.report,       name='report'),
    path('signification',views.signification,name='signification'),
    path('causes',       views.causes,       name='causes'),
    path('symptomes',    views.symptomes,    name='symptomes'),
    path('reparation',   views.reparation,    name='reparation'),
    path('live',         views.live,         name='live'),
]

```

Cette organisation permet de séparer clairement les différentes fonctions du serveur: recherche DTC, recherche XML et communication en direct.

3.10 Accès au serveur via Internet avec Cloudflared

Pendant le développement, le serveur Django fonctionne localement sur la machine du développeur, généralement à l'adresse :

```
http://localhost:8000
```

Cependant, cette adresse n'est pas accessible depuis Internet. Pour permettre à un téléphone, à un expert distant ou à une autre machine externe d'accéder au serveur, nous avons utilisé Cloudflared.

Cloudflared permet de créer un tunnel entre le serveur local et le réseau Cloudflare. Le principe est simple : le serveur continue de fonctionner localement, mais Cloudflared génère une adresse HTTPS publique qui redirige les requêtes vers le port local 8000.

La commande utilisée est :

```
cloudflared tunnel --url http://localhost:8000
```

Après l'exécution de cette commande, Cloudflared fournit une URL publique sous la forme :

`https://nom-temporaire.trycloudflare.com`

Cette adresse peut être utilisée par l'application mobile ou par un expert distant pour accéder au serveur Django. Ainsi, le serveur local devient accessible à distance sans nécessiter de configuration complexe du routeur ou d'ouverture manuelle de ports.

Cette solution est particulièrement adaptée aux tests et à la validation du prototype, car elle permet de rendre rapidement le serveur accessible à travers Internet. Dans une version finale destinée à un usage permanent, il serait préférable d'utiliser un serveur hébergé, un nom de domaine stable et des mécanismes d'authentification renforcés.

3.11 Application mobile développée avec MIT App Inventor

3.11.1 Rôle de l'application mobile

Une application mobile a été développée avec MIT App Inventor afin de fournir une interface simple au mécanicien ou à l'expert. Cette application joue un rôle important dans l'architecture, car elle relie le monde physique du véhicule au serveur distant.

Le port OBD du véhicule est connecté à notre dispositif, lequel est relié au serveur. Ainsi, grâce à l'application mobile, également connectée au serveur, le mécanicien peut accéder à distance au port OBD du véhicule et effectuer différentes opérations de diagnostic.

3.11.2 Composants principaux de l'application

Dans MIT App Inventor, l'application est construite à partir de composants graphiques et de blocs logiques. Les composants principaux utilisés sont :

Composant	Rôle
Button	Déclencher des actions (pour choisir l'une des fonctionnalités de l'application)
WebView	Affichage d'une page HTML pour la visualisation des données en temps réel, dans une interface moderne.

Tableau 3.1: Principaux composants de l'application mobile et leurs rôles

L'application a été conçue de manière simple afin qu'un mécanicien puisse l'utiliser sans connaissance informatique avancée.

3.11.3 Fonctionnement de l'application

L'application offre plusieurs fonctionnalités, permettant à l'expert de choisir l'action qu'il souhaite exécuter. Il peut, par exemple, consulter en temps réel les données du véhicule, lire les codes défaut (DTC), les effacer ou encore tester certains actionneurs (perspective). Toutes ces opérations peuvent être réalisées à distance via Internet.

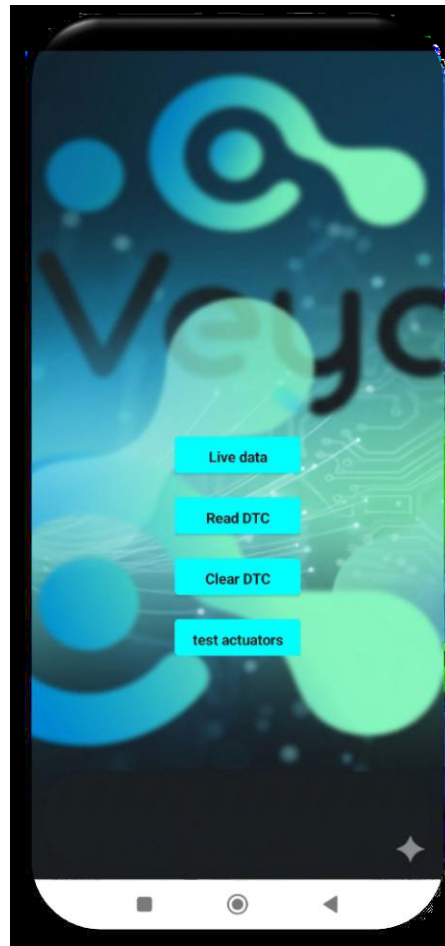


Figure 3.1: Interface d'accueil de l'application mobile VEYA

Le bouton « **Live Data** » permet au mécanicien de visualiser en temps réel les valeurs des différents capteurs du véhicule.

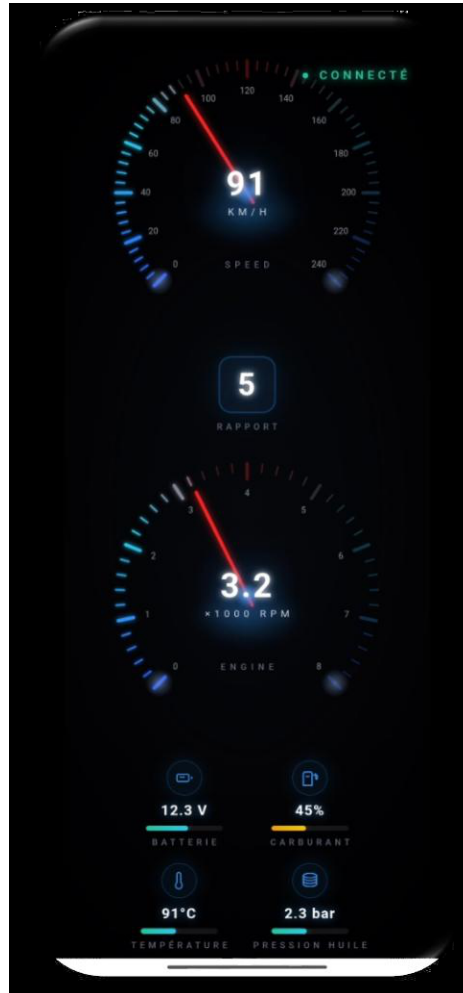


Figure 3.2: Interface de visualisation des données du véhicule en temps réel

Le bouton « **Read DTC** » permet de récupérer les codes défaut (DTC) présents dans le véhicule ainsi que l'état de chacun d'eux (actif ou historique). En sélectionnant un code défaut, sa description détaillée s'affiche.

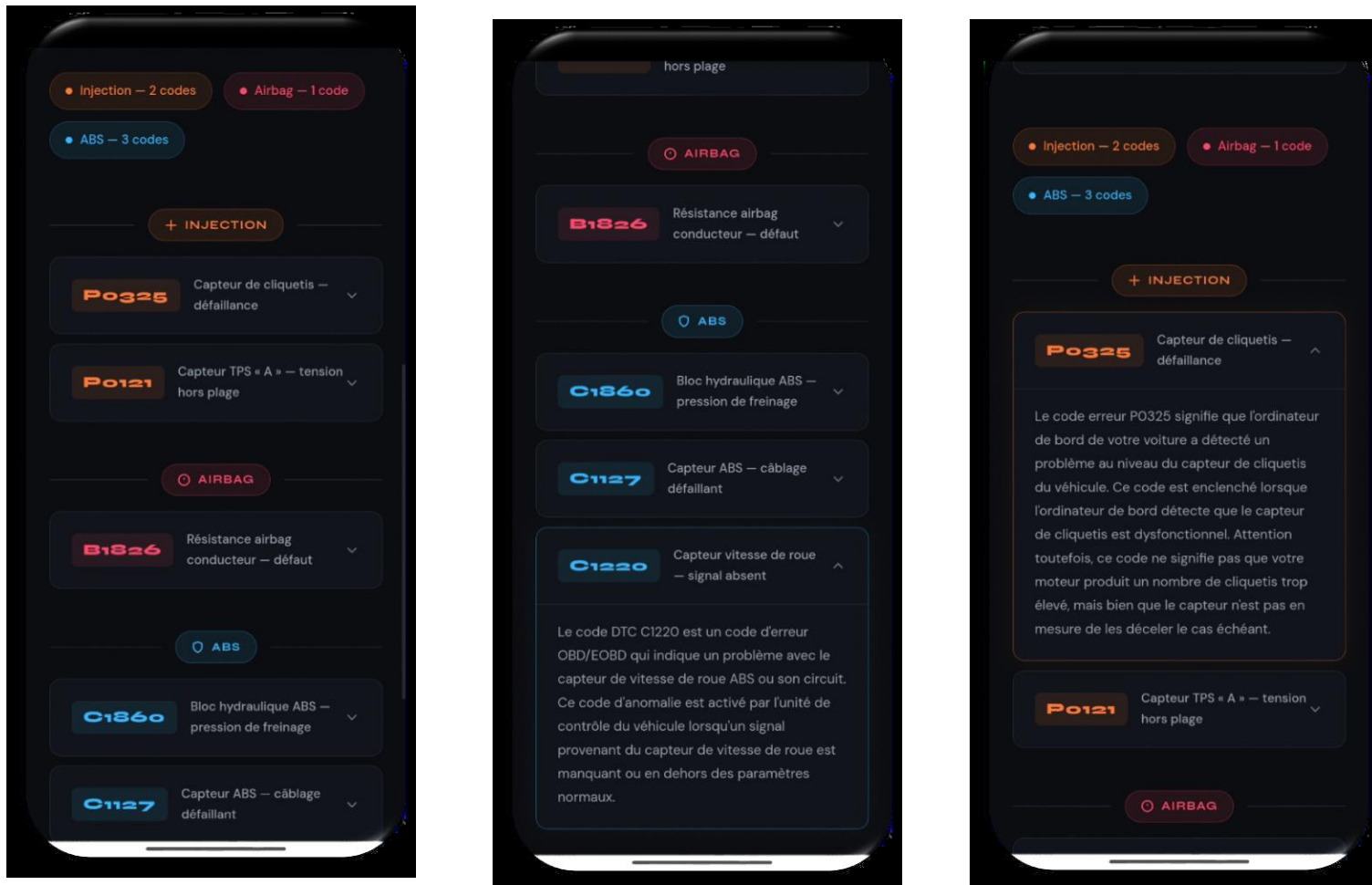


Figure 3.3: Interfaces de lecture et d'interprétation des codes de défaut (DTC)

Le bouton « **Clear DTC** » permet d'effacer les codes défaut (DTC). Toutefois, les codes correspondant à des défauts toujours présents réapparaîtront si la panne n'a pas été réparée. Le bouton « **Test Actuators** » permet d'envoyer des commandes aux actionneurs afin de vérifier leur fonctionnement.

3.12 Test et validation de l'interprétation

3.12.1 Objectif de la validation

La validation a pour objectif de vérifier que les interprétations fournies par notre plateforme sont cohérentes avec celles d'un outil professionnel de diagnostic. Pour cela, les résultats obtenus par le serveur ont été comparés avec ceux fournis par un outil Launch, utilisé comme référence pratique.

L'outil Launch permet de lire les codes défauts présents dans les calculateurs du véhicule et d'afficher leur interprétation. En comparant ces résultats avec ceux retournés par notre serveur, il est possible d'évaluer la fiabilité de la base de données construite et la pertinence du traitement effectué.

3.12.2 Méthodologie de test

La procédure de test suivie est la suivante :

- Connecter l’outil Launch au port OBD du véhicule.
- Lire les codes défauts présents dans les calculateurs.
- Noter les codes DTC affichés, leurs descriptions et les calculateurs concernés.
- Connecter notre système au même véhicule.
- Lire les mêmes codes DTC (et donc la fiabilité des fichiers XML).
- Transmettre les résultats au serveur Django.
- Comparer les interprétations retournées par le serveur avec celles de l’outil Launch.

3.12.3 Analyse des résultats

Les premiers tests permettent de vérifier que le serveur est capable de recevoir un code DTC, d’interroger la base de données et de retourner une interprétation exploitable. Lorsque le code existe dans la table `DTC_standard`, le serveur fournit directement les causes, symptômes et réparations. Lorsque l’information est liée à un calculateur ou à un contexte Renault/Dacia spécifique, la table `DTC_XML` permet de rechercher les informations techniques nécessaires à une interprétation plus détaillée.

La comparaison avec l’outil Launch constitue une étape importante, car elle permet de situer notre solution par rapport à un outil professionnel déjà utilisé dans les ateliers. Une correspondance entre les deux résultats renforce la confiance dans la base d’interprétation construite. En cas de différence, l’écart peut provenir d’une variation dans la base de données utilisée, d’un manque d’information dans le référentiel open source, ou d’une interprétation constructive plus spécifique disponible uniquement dans l’outil professionnel.

Donc que la fiabilité du système ne dépend pas uniquement du serveur lui-même, mais aussi de la qualité des données intégrées dans la base. Plus le référentiel est complet, plus les réponses retournées par la plateforme sont précises.

3.13 Conclusion

Ce chapitre a présenté la réalisation pratique de la plateforme de diagnostic à distance. La solution développée repose sur un serveur Django, une base de données PostgreSQL et une application mobile App Inventor assurant l’accès à distance au port OBD du véhicule. Le serveur joue le rôle de centre de traitement : il reçoit les requêtes, interroge les tables de données, interprète les informations disponibles et retourne des résultats structurés aux clients.

La base de données a été organisée autour de deux tables principales. La table `DTC_standard` permet de stocker les codes défauts standards avec leurs causes, symptômes et réparations. La table `DTC_XML` permet de conserver les informations issues des fichiers XML Renault/Dacia, notamment les adresses de calculateurs, les identifications automatiques, les requêtes de diagnostic et les règles d’interprétation des réponses.

L’utilisation de `psycopg2` a permis d’établir la communication entre Django et PostgreSQL, tandis que Cloudflared a rendu le serveur local accessible depuis Internet à travers un tunnel sécurisé. L’application App Inventor a complété l’architecture en permettant au mécanicien de connecter le système au port OBD et de participer à une session live de diagnostic à distance.

Enfin, la validation par comparaison avec un outil professionnel Launch a permis de vérifier la cohérence des interprétations retournées par la plateforme. Cette étape confirme l'intérêt de centraliser l'interprétation des données de diagnostic dans un serveur distant, capable d'être enrichi, mis à jour et consulté en temps réel. Le système développé constitue ainsi une base fonctionnelle pour un outil de diagnostic connecté, évolutif et accessible.

Conclusion générale et perspectives

L'évolution rapide de l'automobile vers des systèmes embarqués de plus en plus complexes a profondément transformé les méthodes de maintenance et de diagnostic. Dans ce contexte, l'accès aux informations produites par les calculateurs électroniques est devenu indispensable pour identifier rapidement et efficacement les anomalies des véhicules modernes.

L'objectif de ce mémoire était de développer une infrastructure serveur capable de recevoir, stocker et interpréter à distance les codes de défaut (DTC) issus d'un dispositif de diagnostic embarqué. Pour répondre à cet objectif, une plateforme reposant sur Django et PostgreSQL a été conçue et mise en œuvre. Cette architecture permet de centraliser les données de diagnostic, de les traiter automatiquement et de restituer aux utilisateurs des informations exploitables à distance.

L'une des principales contributions de ce travail réside dans la création d'une base d'interprétation regroupant les codes DTC standardisés ainsi que les codes Renault/Dacia extraits des fichiers XML du référentiel DDT2000. Grâce à l'automatisation de l'extraction, de l'intégration et de l'interrogation des données, le système fournit rapidement des descriptions précises des défauts détectés.

La fiabilité de cette base a été validée par une comparaison systématique avec les résultats obtenus à l'aide d'outils professionnels de référence, notamment ceux de la gamme Launch. Les résultats obtenus ont montré une forte concordance entre les interprétations fournies par le serveur et celles délivrées par les équipements professionnels, confirmant ainsi la pertinence et la validité du référentiel développé.

Perspectives de développement

Plusieurs améliorations sont envisagées pour faire évoluer le prototype vers une plateforme complète et utilisable à grande échelle. Parmi celles-ci figurent l'intégration d'un système d'authentification et de gestion des rôles, l'utilisation de modèles de langage pour vulgariser les descriptions techniques des DTC, ainsi que l'ajout de fonctionnalités avancées de diagnostic et d'intervention à distance sous réserve de mécanismes de sécurité adaptés.

L'extension de la base de données à d'autres constructeurs automobiles constitue également une perspective importante afin d'évoluer vers une solution multimarque.

Enfin, le projet pourrait évoluer vers un écosystème automobile complet connectant les différents acteurs du secteur. Les conducteurs pourraient transmettre les défauts détectés et demander une assistance directement depuis l'application, qui intégrerait également une carte affichant les services de dépannage et les ateliers de réparation les plus proches. Les magasins de pièces détachées pourraient, quant à eux, proposer les composants adaptés aux réparations identifiées. Une telle solution permettrait de réduire le temps de diagnostic, d'améliorer la coordination entre les intervenants et d'assurer une prise en charge efficace

du véhicule jusqu'à sa réparation.

En définitive, le travail réalisé et les perspectives identifiées constituent une contribution concrète au domaine du diagnostic automobile et montrent que cette architecture peut évoluer vers une solution plus intelligente, plus sécurisée, multimarque et intégrée à un véritable écosystème de services automobiles.

Bibliographie

- [1] Bedretchuk, João Paulo, Sergio Arribas García, Thiago Nogiri Igarashi, Rafael Canal, Anderson Wedderhoff Spengler, and Giovani Gracioli. Low-cost data acquisition system for automotive electronic control units, 2023.
- [2] Chatterjee, Rik, Carson Green, and Jeremy Daily. Exploiting diagnostic protocol vulnerabilities on embedded networks in commercial vehicles, 2024.
- [3] Chen, Songtao, Shahed Ahmmed, Karu Lal, and Chunhua Deming. Django web development framework: Powering the modern web, 2020.
- [4] Choina, Marcin and Maria Skublewska-Paszkowska. Performance analysis of relational databases MySQL, PostgreSQL and oracle using doctrine libraries, 2022.
- [5] Divyashree N. and Nandini Prasad K. S. Design and development of we-CDSS using django framework: Conducing predictive and prescriptive analytics for coronary artery disease, 2022.
- [6] Duisebekova, Kulanda Seitbekovna, Roman Khabirov, and Azamat Zholzhan. DJANGO as secure web-framework in practice, 2021.
- [7] Guardiola, Carlos, Christian Vigild, Frederik De Smet, and Klaus Schusteritz. From OBD to connected diagnostics: A game changer at fleet, vehicle and component level, 2021.
- [8] Iskandar, Karto, Alfred Tambayong, Muhammad Rafif Fawwaz Mulya, Steven Cendra Elfanlie, and Maria Grace Herlina. Mobile-based car diagnostic application using onboard diagnostic-II scanner, 2023.
- [9] Kaiwartya, Omprakash, Abdul Hanan Abdullah, Yue Cao, Ayman Altameem, Mukesh Prasad, Chin-Teng Lin, and Xiulei Liu. Internet of vehicles: Motivation, layered architecture, network model, challenges, and future aspects, 2016.
- [10] Kavuluri, Harsha Vardhan Reddy. PostgreSQL vs. oracle: A comparative study of performance, scalability, and enterprise adoption, 2024.
- [11] Kim, Hwa-seon, Seong-jin Jang, and Jong-wook Jang. A study on development of engine fault diagnostic system, 2015.
- [12] Klimek, Bartłomiej Mirosław and Maria Skublewska-Paszkowska. Comparison of the performance of relational databases PostgreSQL and MySQL for desktop application, 2021.

- [13] Kriebe, Stefan, Kusmenko, Rumpe, and Shumeiko. Learning error patterns from diagnosis trouble codes, 2019.
- [14] Kulkarni, Prajakta, P. K. Rajani, and Kanupriya Varma. Development of on board diagnostics (OBD) testing tool to scan emission control system, 2016.
- [15] Kumar, Manoj and Rainu Nandal. Python's role in accelerating web application development with django, 2024.
- [16] Lee, Joo-Chul and Tae-Man Han. ECU configuration framework based on AUTOSAR ECU configuration metamodel, 2009.
- [17] Li, Yufeng, Wenqi Liu, Qi Liu, Xiangyu Zheng, Ke Sun, and Chengjian Huang. Complying with ISO 26262 and ISO/SAE 21434: A safety and security co-analysis method for intelligent connected vehicle, 2024.
- [18] Lim, Mingyu. Directly and indirectly synchronous communication mechanisms for client-server systems using event-based asynchronous communication framework, 2019.
- [19] Malekian, Reza, Ntefeng Ruth Moloisane, Lakshmi Nair, B. T. Maharaj, and Uche A. K. Chude-Okonkwo. Design and implementation of a wireless OBD II fleet management system, 2017.
- [20] Mohammadpour, Javad, Matthew Franchek, and Karolos Grigoriadis. A survey on diagnostics methods for automotive engines, 2011.
- [21] Moniaga, J. V., S. R. Manalu, D. A. Hadipurnawan, and F. Sahidi. Diagnostics vehicle's condition using OBD-II and raspberry pi technology: Study literature, 2018.
- [22] Nagaveni, K. S. and S. Gayathri. ECU extract analyzer and merger tool, 2021.
- [23] Nyabuto, Geoffrey. Client-server architecture, a review, 2023.
- [24] Oluwaseyi, Medashe Michael and Abolarin Matthew Sunday. Specifications and analysis of digitized diagnostics of automobiles: A case study of on board diagnostic (OBD II), 2020.
- [25] Oluwatosin, Haroon Shakirat. Client-server model, 2014.
- [26] Putra, Dwi Sudarno, Joe Vane N. Antifuesto, Budi Utomo Wisesa, Zainul Hamidi, and Mutiara Vinta. Design and development of a portable diagnostic tester for fault detection in EFI vehicle control systems, 2025.
- [27] Qi, Yang, Wang Zhulin, and Wang Hongyun. Standard description of common element model data type based on XML schema, 2018.
- [28] Rimpas, Dimitrios, Andreas Papadakis, and Maria Samarakou. OBD-II sensor diagnostics for monitoring vehicle operation and consumption, 2020.
- [29] Rodríguez, Armando Rodríguez, José Raúl Vento Álvarez, and Ricardo Inouye Rodríguez. Implementation of an OBD-II diagnostics tool over CAN-BUS with arduino, 2018.
- [30] Salunke, Sanket Vilas and Abdelkader Ouda. A performance benchmark for the PostgreSQL and MySQL databases, 2024.

- [31] Sawant, Pooja Rajendra and Yashwant B. Mane. Design and development of on-board diagnostic (OBD) device for cars, 2018.
- [32] Sinha, Alok. Client-server computing, 1992.
- [33] Solarz, Arkadiusz and Tomasz Szymczyk. Oracle 19c, SQL server 2019, PostgreSQL 12 and MySQL 8 database systems comparison, 2020.
- [34] Soldani, Siegfried. Vers le diagnostic embarqué de défaillances dans les systèmes à événements discrets : application au domaine automobile, 2008.
- [35] Suwatthikul, Jittiwut. Fault detection and diagnosis for in-vehicle networks, 2010.
- [36] Thomas, Anita Maria, Jyothis Joseph, and Reshma Mathew. Mobile application for automobile management system, 2009.
- [37] Varshney, Akanksha, Sanjay Joshi, and K. Namrata. Automated testing of faults of an automotive system, 2019.
- [38] Vrachkov, Dimitar Georgiev and Dimitar Georgiev Todorov. Automotive diagnostic trouble code (DTC) handling over the internet, 2018.
- [39] Wang, Wenwei, Kaidi Guo, Wanke Cao, Hailong Zhu, Jinrui Nan, and Lei Yu. Review of electrical and electronic architectures for autonomous vehicles: Topologies, networking and simulators, 2024.
- [40] Zhang, Hong and Ding Li. Diagnostic communication and visual system based on vehicle UDS protocol, 2019.
- [41] Zheng, Zhiliang and Bailin Liu. Research on the application of XML in fault diagnosis IETM, 2022.