Université de Strasbourg

Master d'informatique
Parcours SIRIS
Science et Ingénierie des Réseaux, de l'Internet et des Systèmes

---

Projet master

Système de monitoring pour une flotte de véhicules

Membres du projet :

Éloi Colin — eloi.colin@etu.unistra.fr
Maxime Friess — maxime.friess@etu.unistra.fr
Florian Halm — florian.halm@etu.unistra.fr
Florent Hardy — florent.hardy@etu.unistra.fr
Kevin Hentz — kevin.hentz@etu.unistra.fr
Jessica Klotz — jessica.klotz@etu.unistra.fr

Chef de projet :

Nicolas Elfering — nicolas.elfering@etu.unistra.fr

Table des matières

Chapitre 1

Introduction

Ce mémoire s'inscrit dans le cadre de l'Unité d'Enseignement "Projet Master", enseignée lors de la deuxième année du cursus Master SIRIS [1] à l'université de Strasbourg. Dans ce contexte, les étudiants ont été séparés en deux équipes de sept membres. L'objectif de cette UE est de réaliser en équipe un projet d'envergure similaire à celui d'une entreprise. Le sujet du projet est libre et c'est aux étudiants de le choisir. Il doit néanmoins satisfaire certaines contraintes. La liste de ces contraintes est disponible en annexe A. Les points clefs à satisfaire sont un scénario innovant et l'utilisation d'objets connectés. Afin de contrôler le bon déroulement du projet, les professeurs suivants ont été choisis pour suivre l'avancé du projet, Mme Anissa LAMANI et M. Julien MONTAVONT. Ces derniers représenteront un client inventé par les étudiants.

Le déroulement de cette UE est composé de quatre étapes : Dans une première partie, les étudiants devront inventer et contextualiser l'existence de ce client à travers un document représentant un appel d'offres. Dans une seconde partie, les étudiants font des recherches et organisent les tâches essentielles afin de répondre aux attentes de l'appel d'offres. Le résultat de cette étape qui doit être soumis au plus tard le 20 octobre 2024, est concrétisé par un dossier soumissionnaire et la présentation de la maquette au client. Cet ensemble de pièces permet de montrer que l'entreprise fictive des étudiants est en mesure de réaliser le projet. Dans une troisième partie, les étudiants devront mettre en place le projet. Cette partie contient un jalon intermédiaire durant lequel le chef de projet présentera à mi-parcours les avancées de son équipe au client. Et enfin, la quatrième partie correspond à la présentation du projet dans sa globalité à l'écrit, à travers ce mémoire, et à l'oral, lors d'une soutenance du projet en présence de toute l'équipe.

Dans un monde où la voiture est de plus en plus connectée, et où les constructeurs automobiles sont capables de détecter des anomalies et de réparer le véhicule de façon préventive, nous avons pensé qu'il serait intéressant de proposer une solution aux assurances dans le but de superviser leurs véhicules assurés et de réduire les coûts d'assurance, de réparations, etc... des coûts qui sont supportés aussi bien par l'assuré que par l'assurance. Le problème principal faisant obstacle à la mise en place d'une telle solution est la compétitivité des constructeurs automobiles qui produisent leurs propres systèmes de monitoring. Étant donné le nombre de systèmes ainsi que leur confidentialité, le fait de créer un système compatible à toutes les propositions existantes relèvent d'un défi financier et politique. Cependant, depuis 2002, il existe une norme sur tous les véhicules appelé OBD-2 [2]. Cette dernière vise à récupérer les informations des différents capteurs des véhicules à partir d'une prise installée dans chaque véhicule. Le sujet que nous avons choisi est donc l'élaboration d'un système de monitoring pour une flotte de véhicules à partir de la prise OBD-2, d'un capteur GPS et d'une caméra embarquée.

Le mémoire est constitué de six chapitres : Le premier chapitre présentera les différentes équipes, les différentes entreprises fictives, ainsi que leur rôle principal. Dans le second chapitre, nous présenterons la gestion du projet. Les trois chapitres suivants présenteront les aspects techniques de notre solution, en commençant par la partie développement applicatif, puis le système embarqué et enfin l'infrastructure. Finalement, dans un sixième et dernier chapitre, nous apporterons en conclusion nos retours sur l'expérience d'un tel projet.

1. Science et Ingénierie des Réseaux, de l'Internet et des Systèmes
2. Prise permettant de communiquer directement avec le ECU (Engine Control Unit) [SS14]

Chapitre 2

Présentation des acteurs

Afin de contextualiser le projet, nous allons expliquer dans cette partie l'organisation des acteurs dans ce projet.

2.1 NAIF

NAIF - National Auto Insurance France est une assurance française fictive que nous avons inventée. Cette dernière a comme objectif de créer un système de monitoring pour superviser sa flotte automobile. Son but est de proposer un tarif en fonction de la conduite et de la gestion du véhicule du client. Cette stratégie vise à réduire les coûts globaux d'assurance et à promouvoir une conduite responsable auprès de leurs clients. Compte tenu du grand nombre de clients de NAIF, la solution doit également être capable de gérer une montée en charge significative, garantissant ainsi la fiabilité et la performance du système même en cas d'utilisation intensive. Grâce à ce projet, NAIF espère également se démarquer de la concurrence en offrant des services personnalisés, basés sur des analyses précises et fiables des données des véhicules.

2.2 BugBrothers

L'entreprise fictive répondant à l'appel d'offres est décrite comme une entreprise spécialisée dans les systèmes embarqués. Afin d'appuyer sa notoriété, nous avons défini que les créateurs de l'entreprise étaient d'anciens ingénieurs travaillant chez Airbus et ayant aidé à construire l'A380. Dans ce contexte, nous avons défini trois équipes pour résoudre cette solution : une équipe de systèmes embarqués, une équipe pour l'infrastructure et une pour le développement applicatif ainsi qu'un chef de projet. La répartition des membres dans les équipes a été effectuée au tout début du projet. Néanmoins, l'objectif principal était que les membres de toutes les équipes aient à minima travaillé dans une autre équipe. Pour plus de détails, la répartition du travail est définie dans le prochain chapitre.

Les membres ont été répartis comme suit :

— infrastructure : Éloi Colin et Florent Hardy
— développement applicatif : Kevin Hentz et Jessica Klotz
— systèmes embarqués : Maxime Friess et Florian Halm

Chapitre 3

Gestion de projet

3.1 Méthode de travail

Notre projet a suivi une méthodologie de travail agile, avec des sprints d'une semaine permettant une adaptation rapide face aux imprévus. Une réunion hebdomadaire regroupant toutes les équipes était organisée, durant laquelle chaque membre présentait ce qui avait été accompli pendant cette dernière semaine et les tâches à venir. Cela offrait l'avantage de fournir à tous les membres une vue d'ensemble du projet, permettant ainsi de mieux comprendre les défis rencontrés et pourquoi certaines tâches pouvaient prendre plus de temps que prévu. De plus, ces réunions favorisaient l'échange d'idées entre les groupes, où des membres d'autres équipes pouvaient proposer des solutions aux problèmes rencontrés.

La gestion de projet s'appuyait sur des outils tels que YouTrack [1] pour la répartition et le suivi des tâches, et Git [2] pour la gestion des versions de code. Ces outils ont permis de structurer efficacement le travail et de maintenir une vision globale des objectifs. De plus, pour faire part de l'avancement du projet, une réunion hebdomadaire (dans la mesure du possible) a été prévue avec les clients. Ces réunions avaient pour but de montrer les avancées réalisées, de discuter des points bloquants et d'ajuster les priorités si nécessaire.

3.2 Organisation des tâches

La réalisation du projet s'est articulée autour de trois grandes composantes : l'infrastructure, les systèmes embarqués et le développement web. Ces éléments ont permis de structurer et de prioriser les travaux nécessaires pour atteindre nos objectifs.

Après avoir rédigé l'appel d'offre et dans le but d'y répondre, la première réunion d'équipe a été consacrée à la répartition des membres en sous-groupes. Chaque équipe a alors été chargée de réaliser des recherches pour la semaine suivante afin d'identifier des tâches principales et de proposer des premières pistes sur leur mise en œuvre. Lors de la seconde réunion, toutes ces tâches ont été compilées, mais certains éléments sont restés flous, notamment le load balancing et les IP flottantes intra-site. Malgré cela, une répartition des tâches a été effectuée de manière à ce qu'aucune équipe ne se retrouve isolée pour les tâches complexes, et qu'aucune tâche principale ne repose sur un seul membre de l'équipe, garantissant ainsi une collaboration efficace.

Les temps pour les différentes tâches ont été estimés lors d'une réunion, mais certaines tâches, pour lesquelles nous n'avions aucune expérience préalable, avaient des durées incertaines. Ainsi, certaines se sont avérées plus longues et d'autres plus courtes que prévu, nécessitant des ajustements en cours de projet. Par ailleurs, le projet nécessitait un minimum de 150 heures de travail par étudiant, ce qui a été réparti en un minimum de 8 heures de travail par semaine pour chaque membre. Cette organisation a permis de maintenir un suivi régulier malgré les imprévus rencontrés.

Les diagrammes de Gantt présentés dans cette section sont des versions regroupées, offrant une vue synthétique des étapes principales pour chaque composante. Les versions complètes et détaillées de ces

1. https://m2projg1.youtrack.cloud/
2. https://git.unistra.fr/nelfering/projetmaster

diagrammes sont disponibles en annexe : B.1, B.2, B.3.

3.2.1 Infrastructure

La planification des tâches pour l'infrastructure s'est concentrée sur les éléments essentiels comme la mise en place de ProxMox, la configuration des IP flottantes, la gestion des pare-feux, et l'installation des outils de monitoring. Le diagramme de Gantt illustre l'organisation de ces étapes et leur répartition dans le temps.

Un point clé de cette organisation est l'intégration de tâches spécifiques pour chaque site (Site 1 et Site 2), ce qui a permis de répartir efficacement les responsabilités entre les équipes. En effet, le groupe de deux personnes a été divisé, avec un spécialiste pour la C315 et un autre pour OpenStack. Cependant, afin d'éviter tout problème en cas d'absence ou de maladie, chaque étape a été documentée de manière rigoureuse. La planification inclut également des périodes pour tester et valider les performances, comme la synchronisation inter-site et l'évaluation des performances globales.

Un problème majeur a été rencontré lors de la mise en place du load balancer. Initialement, nous avions prévu d'utiliser un seul nom de domaine avec des entrées pour deux adresses IPv4, une pour OpenStack et une pour la C315. Un système de load balancing avec Nginx devait également rediriger les requêtes en fonction de la charge sur chaque site. Cela a conduit à la coexistence de deux systèmes de load balancing concurrents, ce qui compliquait considérablement la gestion. Finalement, nous avons opté pour un site principal et un site secondaire en cas de panne, chaque site ayant son propre nom de domaine. Cette solution, bien que moins élégante, s'est révélée fonctionnelle, notamment grâce à l'utilisation de la PWA pour basculer facilement entre les sites.

De plus, une tâche initiale prévoyait un système d'IP flottantes inter-site, mais nous n'avons pas trouvé de solution viable dans les délais impartis. Cette tâche a donc été supprimée au profit de travaux sur la supervision avec Nomad et Consul.

Sur la figure 3.1 ci-dessous, les tâches en noir et bleu correspondent aux étapes prévues initialement, alors qu'en rouge, on a représenté les modifications effectuées. Ainsi nous pouvons constater que PRO-44 a pris beaucoup plus de temps car elle dépendait de trois autres tâches(PRO-70, 73, 43). De plus, le contenu de PRO-43 a été modifié.

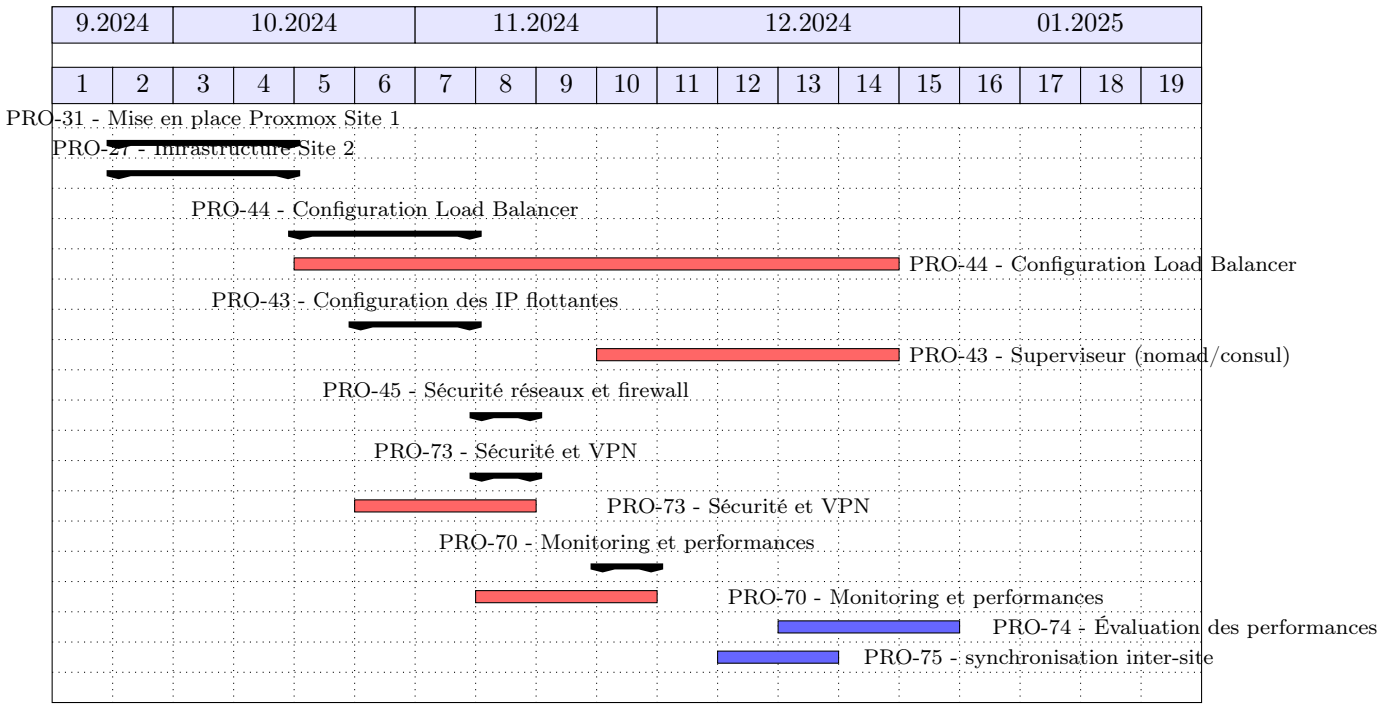


FIGURE 3.1 – Diagramme de Gantt de la partie infrastructure

3.2.2 Applicatif

Le développement applicatif a été divisé en deux grandes catégories : le frontend et le backend. En suivant l'évolution des tâches du système embarqué, l'objectif principal est de créer le backend et le frontend pour une fonctionnalité du système embarqué qui a été entamée et prête à être testée.

Le diagramme de Gantt (figure 3.2) présente une structure claire où les dépendances entre les différentes tâches sont bien identifiées. Cette hiérarchisation a permis d'éviter les blocages entre équipes.

Un avantage supplémentaire de ce diagramme est la mise en avant de la redondance et de la sécurité en fin de projet, comme l'intégration de l'authentification et la communication HTTPS sécurisée, garantissant une solution robuste.

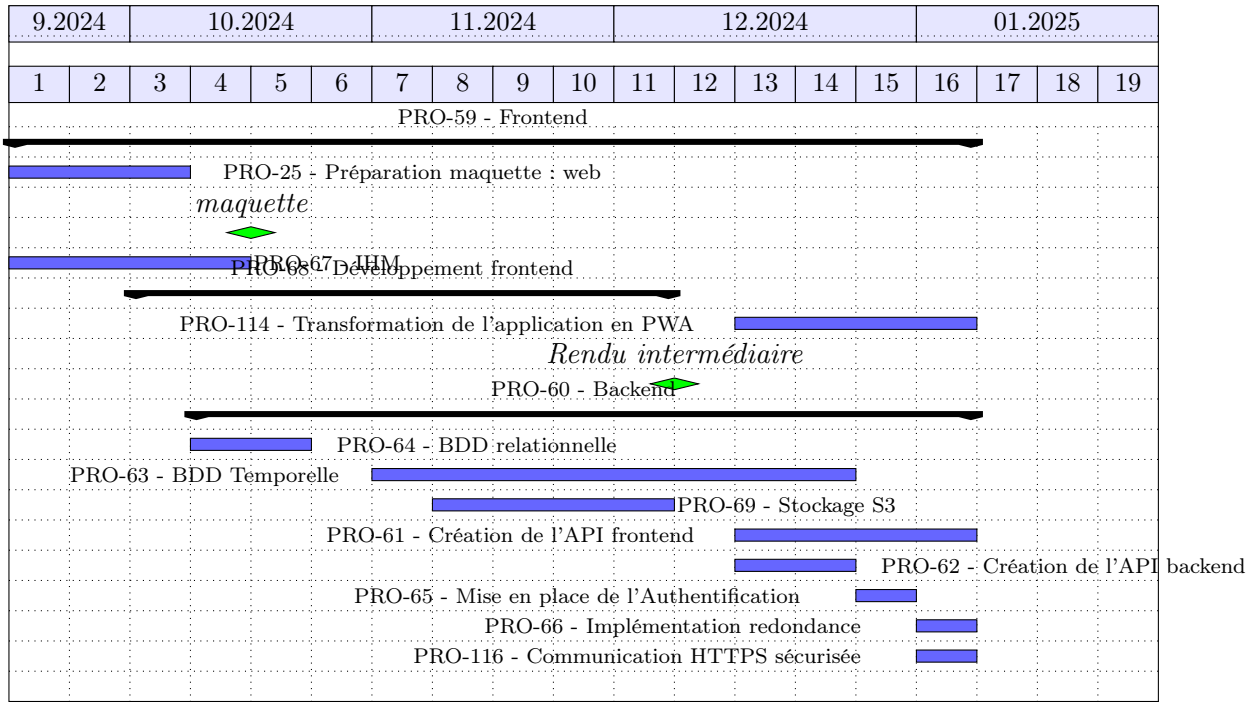


FIGURE 3.2 – Diagramme de Gantt de la partie applicative

3.2.3 Systèmes embarqués

La partie systèmes embarqués s'est concentrée sur l'intégration des capteurs (GPS, caméra, OBD-2) et la communication entre les microcontrôleurs (Raspberry Pi et Arduino). Le diagramme de Gantt (figure 3.3) détaille les étapes principales, depuis la préparation de la maquette jusqu'à l'automatisation des firmwares.

Un aspect notable est l'organisation des tâches en modules, comme les données GPS, le flux vidéo de la caméra, et les données OBD-2. Ces modules ont été planifiés indépendamment, tout en gardant une intégration progressive avec le backend. Cela a permis aux équipes de travailler en parallèle, tout en assurant une compatibilité entre les différents composants. Pour le module OBD-2, son inclusion était dépendante de la livraison du matériel. Ce module était donc critique et dépendait d'un élément hors de notre contrôle, ce qui en a fait une tâche particulièrement sensible à gérer.

Des retards ont également été rencontrés, notamment pour les capteurs GPS, où nous avons perdu deux semaines à cause d'un faux contact dans le matériel. Après réparation, la tâche a pu être terminée rapidement. Cependant, ces retards ont également affecté la vidéo et l'OBD-2, ce qui nous a contraints à abandonner une fonctionnalité initialement prévue.

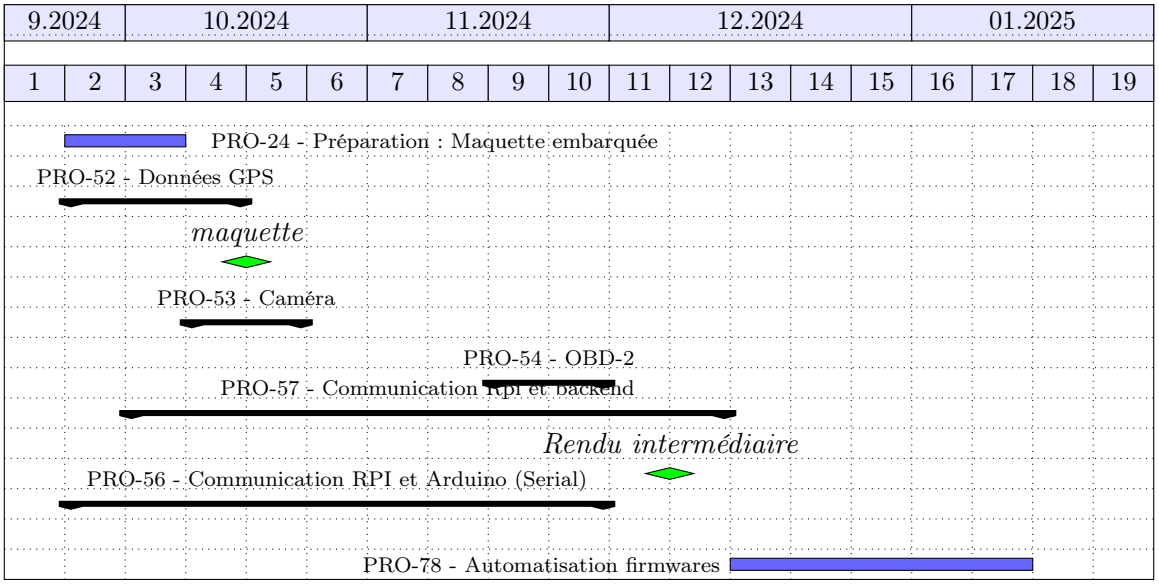


FIGURE 3.3 – Diagramme de Gantt de la partie systèmes embarqués

Chapitre 4

Applicatif

L'objectif de cette solution applicative est de développer un système innovant dédié à la gestion et au suivi en temps réel d'une flotte de véhicules. Conçue pour répondre aux besoins spécifiques d'une assurance, l'application offre une vision centralisée, interactive et dynamique des véhicules. Elle se distingue par une interface utilisateur intuitive et des fonctionnalités avancées permettant la collecte, le traitement et la présentation des données.

Développée en s'appuyant sur des technologies modernes et éprouvées, la solution s'articule autour d'une architecture robuste et évolutive, d'un design centré sur l'utilisateur, et d'une interconnexion fluide entre ses différents composants. L'application intègre des outils de cartographie interactive, des fonctionnalités de gestion des données techniques et des notifications en temps réel, tout en garantissant une expérience utilisateur optimisée sur tous les supports.

Le développement de ce projet a mobilisé des compétences variées dans les domaines du développement frontend et backend, de la gestion des bases de données ou encore de la sécurité des échanges.

Les sections suivantes détaillent les différents aspects techniques et fonctionnels de l'application, notamment son interface utilisateur, son architecture backend, ses choix en matière de stockage et de gestion des données, ainsi que l'interconnexion de ses composants. Ces éléments illustrent les choix technologiques effectués, les solutions adoptées et les défis surmontés pour atteindre les objectifs fixés.

4.1 Interface utilisateur

L'interface utilisateur de l'application a été conçue pour offrir une expérience fluide, intuitive et agréable. Elle repose sur un tableau de bord central accessible dès la page d'accueil, qui affiche tous les véhicules sur une carte interactive et leurs notifications associées. Cette carte, construite avec Leaflet, permet de visualiser en temps réel la localisation des véhicules à l'aide de marqueurs dynamiques. D'autres pages complètent l'application : une pour l'ajout de clients, de véhicules et de contrats via des formulaires ergonomiques, une pour la recherche et la visualisation des données, et une dédiée à la consultation des détails d'un véhicule. Cette dernière propose des onglets thématiques, regroupant des informations générales, tels que les trajets affichés sur une carte interactive grâce à Leaflet, un flux vidéo, et des données techniques OBD-2.

L'interface a été développée avec React TypeScript pour garantir une structure robuste, réutilisable et facile à maintenir. **React** a été choisi pour sa simplicité d'utilisation et son approche modulaire, permettant de créer des composants réutilisables qui simplifient le développement et la maintenance. **TypeScript**, grâce à son typage statique, renforce la fiabilité et réduit les erreurs dès la phase de développement, tout en améliorant la lisibilité et la documentation du code.

La bibliothèque **Axios** gère les appels API, assurant une communication rapide et fiable avec le backend, facilitant ainsi l'intégration des données et leur mise à jour en temps réel. Le choix de **Leaflet** pour la cartographie interactive enrichit l'expérience utilisateur en offrant des fonctionnalités avancées, telles que le zoom fluide, la visualisation claire des trajets, et l'utilisation de marqueur sur la carte pour représenter les véhicules et leurs positions. Cela permet aux utilisateurs de suivre visuellement et efficacement les déplacements de la flotte.

L'application inclut également un **thème sombre**, réduisant la fatigue visuelle lors d'une utilisation prolongée, notamment dans des environnements à faible luminosité. Par ailleurs, le **design réactif** (responsive design) garantit une adaptabilité parfaite à tous les types d'appareils, qu'il s'agisse d'ordinateurs, de tablettes ou de smartphones. Les utilisateurs peuvent ainsi bénéficier d'une expérience homogène, intuitive et fluide, quel que soit le support utilisé, rendant l'application accessible à un large éventail d'utilisateurs et de contextes.

Ces choix techniques et esthétiques visent à optimiser l'interaction avec l'application tout en répondant aux attentes des utilisateurs en termes de performance, de convivialité et de modernité.

Malheureusement, le manque de données disponibles via le système OBD-2 a considérablement limité la mise en œuvre de certaines fonctionnalités initialement prévues dans le projet. Par exemple, nous n'avons pas pu calculer un score basé sur la conduite du conducteur ou sur l'entretien du véhicule, faute d'informations suffisantes. Ce déficit de données a également empêché la création d'un résumé mensuel détaillé des informations liées au véhicule, notamment en ce qui concerne sa consommation de carburant, laissant cette partie de l'application incomplète et incapable de générer une éventuelle réduction pour le mois.

Initialement, nous avions également prévu d'attribuer à chaque véhicule un niveau de criticité en fonction de son état et de ses performances, mais l'absence de données pertinentes n'a pas permis de mettre en place cet indicateur.

4.2 Architecture Backend

Le backend de notre application a été développé en **Python**, en utilisant le framework Flask. Ce choix s'est révélé stratégique pour plusieurs raisons. En tirant parti des compétences existantes de l'équipe en Python, nous avons accéléré le développement tout en favorisant une collaboration fluide. De plus, la vaste communauté autour de cette technologie nous assure un support constant et facilite la résolution des problèmes techniques.

Flask, un framework minimaliste, s'est imposé comme un choix évident. Il fournit les fonctionnalités essentielles pour le développement d'API sans alourdir le projet avec des composants superflus. Sa légèreté le rend particulièrement adapté à des projets de petite envergure comme le nôtre, où un framework plus complet, tel que Django, aurait introduit une complexité et une charge supplémentaires inutiles.

La structure du code a été conçue pour assurer une séparation claire des responsabilités et faciliter la maintenance. Les *endpoints* sont organisés par fonctionnalité dans deux fichiers distincts :

— `frontend_controller.py` : dédié aux interactions avec le frontend.

— `embedded_controller.py` : destiné aux communications avec le système embarqué.

Ces *endpoints* s'appuient sur des services spécialisés pour accéder aux bases de données PostgreSQL et InfluxDB. Par ailleurs, les fonctions utilitaires sont centralisées dans un fichier `helper.py`, ce qui améliore la réutilisabilité du code en regroupant des tâches communes, telles que la vérification du format d'une adresse e-mail ou le hachage de chaînes de caractères.

Durant le développement du backend, plusieurs défis ont émergé. Garantir la sécurité des endpoints, notamment en ce qui concerne l'authentification et la validation des entrées, a exigé une attention particulière pour éviter les vulnérabilités courantes. En outre, le défi de maintenir une architecture claire tout en intégrant de nouvelles fonctionnalités et endpoints a nécessité une vigilance accrue. Nous avons tout de même réussi à implémenter la majorité des endpoints nécessaires pour le frontend et le système embarqué.

En conclusion, cette architecture backend, appuyée par une organisation modulaire et des choix technologiques adaptés, répond efficacement aux besoins du projet tout en garantissant une évolutivité et une maintenabilité optimales.

4.3 Stockage et gestion des données

Afin de répondre aux exigences spécifiques de notre application en termes de structure, de stockage et de gestion des données, nous avons choisi une combinaison de technologies de bases de données. Cette

approche nous permet d'assurer une haute disponibilité des informations, une redondance maximale et une capacité d'adaptation aux évolutions futures. Voici une liste des technologies intégrées :

— **PostgreSQL** : Choisi pour le stockage des données structurées (employés, clients, véhicules, contrats), PostgreSQL est une base relationnelle robuste et scalable, offrant une gestion efficace des relations entre entités et une intégrité des données via des transactions ACID.

— **InfluxDB** : Utilisé pour gérer les séries temporelles, comme les données OBD-2 ou les notifications, InfluxDB est optimisé pour manipuler les données horodatées et les requêtes basées sur le temps.

— **MinIO** : Pour le stockage de fichiers volumineux, comme les images, MinIO est une solution de stockage d'objets compatible avec S3. Elle est adaptée à la gestion de grands volumes de données non structurées, tout en offrant une haute disponibilité, durabilité et scalabilité.

Un schéma détaillé (voir Figure 4.1) illustre les dépendances entre les différents systèmes de stockage de données et leur rôle au sein de l'application. Ces technologies permettent de répondre aux besoins spécifiques du projet tout en assurant la performance et la flexibilité du système.

Au départ, il était prévu de stocker des vidéos de 30 secondes ainsi que des images. Toutefois, le stockage des vidéos a été abandonné afin de privilégier l'implémentation d'autres fonctionnalités. En pratique, MinIO est utilisé pour stocker une seule image par véhicule, cette image étant remplacée à chaque réception d'une nouvelle.

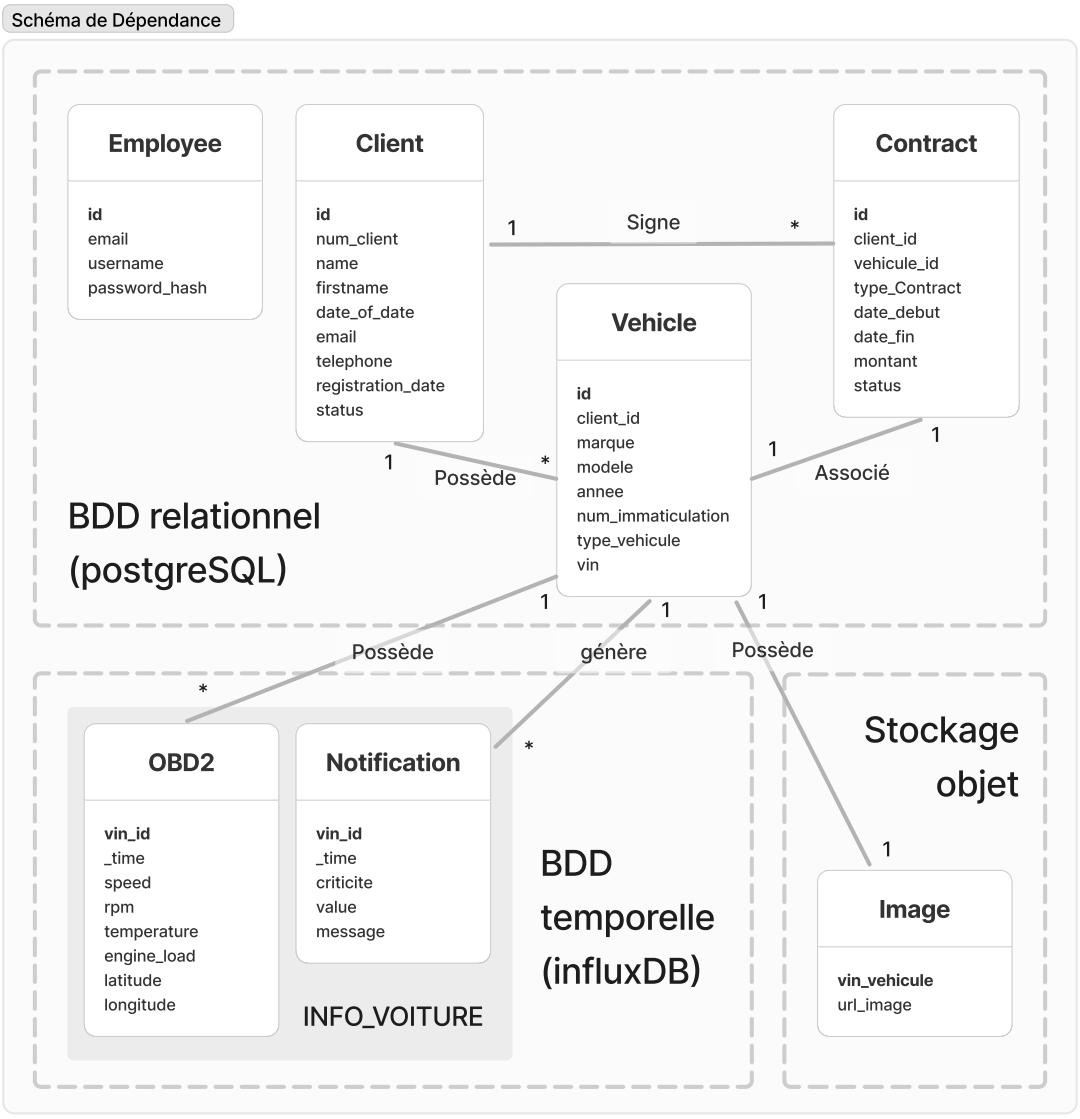


FIGURE 4.1 – Schéma de dépendance

4.4 Interconnexion des composants

L'architecture de notre application repose sur une interaction fluide et efficace entre le dispositif embarqué, le serveur backend, les bases de données, et le frontend. Chaque composant joue un rôle essentiel dans le traitement, le stockage, et la présentation des données. Nos différents composants interagissent grâce à des technologies adaptées telles que :

- Les **API RESTful** (Representational State Transfer) sont au cœur des interactions entre les composants. Elles offrent une interface standardisée et fiable pour les échanges de données, utilisant les méthodes HTTP (GET, POST, PUT, DELETE) pour interagir avec les ressources. Ce choix assure une communication intuitive et cohérente, facilitant le développement et la maintenance de l'application.
- Les **Websockets** qui, contrairement aux API RESTful qui reposent sur des requêtes client, permettent au serveur de pousser directement les mises à jour vers le client, assurant une communication dynamique en temps réel.
- La **sécurité des interactions** est assurée par un système d'authentification basé sur les JWT (JSON Web Tokens). Lors de la connexion, un JWT est généré et vérifié à chaque requête API, garantissant que seules les requêtes authentifiées peuvent accéder aux données. De plus, toutes les communications sont sécurisées via HTTPS, assurant la confidentialité des échanges.
- Les **échanges de données** se font au format JSON, choisi pour sa légèreté et sa lisibilité. Ce format facilite la sérialisation et la désérialisation des données, rendant les transferts rapides et efficaces entre les composants.

L'intégration de ces composants a posé quelques défis, notamment liés à la sécurisation des communications via HTTPS, indispensable pour le bon fonctionnement des cookies JWT. Cette configuration dépendait de l'équipe infrastructure, ce qui a temporairement retardé la mise en place. Une fois cette dépendance résolue, l'intégration s'est effectuée sans encombre.

4.4.1 Interactions dispositif embarqué - backend

Le dispositif embarqué, identifié par un VIN (Vehicle Identification Number), transmet les données télémétriques au serveur backend via une API RESTful. Ces données sont immédiatement stockées dans InfluxDB, une base de données optimisée pour les séries temporelles. En parallèle, le backend publie ces données dans une WebSocket, chaque dispositif ayant une "room" dédiée identifiée par son VIN, permettant une diffusion en temps réel.

4.4.2 Interactions frontend - backend

Le frontend de l'application, développé en React Typescript, interagit avec le backend de deux manières principales :

- **Requêtes API RESTful** : Ces requêtes permettent aux utilisateurs de consulter l'historique des données du véhicule dans InfluxDB et de gérer les données relationnelles stockées dans PostgreSQL (clients, véhicules, contrats, etc...). Le backend interroge les bases de données et renvoie les résultats au frontend.
- **Connexion WebSocket** : Pour les utilisateurs qui souhaitent suivre en temps réel les données d'un véhicule particulier, le frontend se connecte à la "room" de la WebSocket correspondant au VIN du véhicule afin de recevoir instantanément les nouvelles données émises par le dispositif embarqué.

L'architecture de l'application intègre également un reverse proxy Nginx, placé entre le frontend et le backend. Ce composant joue un rôle crucial en redirigeant correctement les différentes requêtes vers leurs destinations respectives. Nginx assure non seulement la gestion efficace du trafic, mais contribue également à la sécurisation des communications et à l'équilibrage de la charge. Les détails techniques de cette mise en œuvre seront approfondis dans la partie infrastructure du rapport 6.

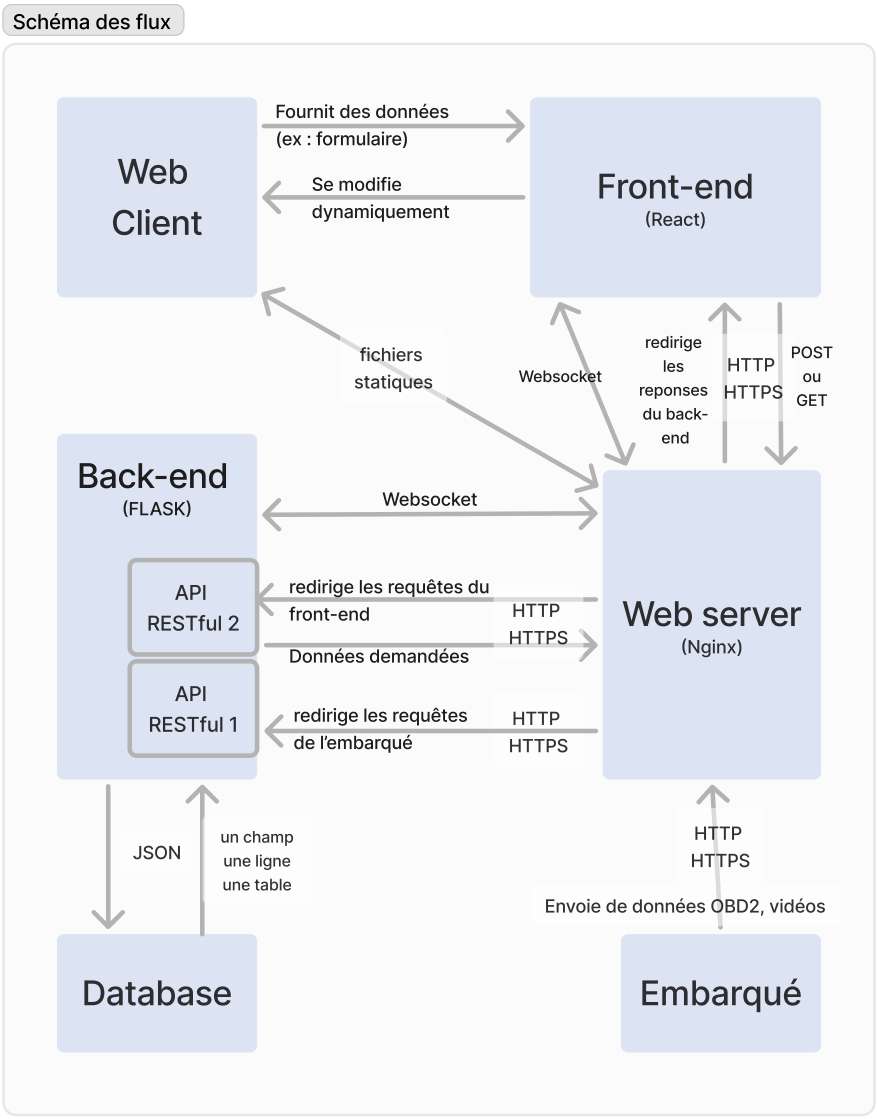


FIGURE 4.2 – Schéma des flux

4.4.3 Schéma des interactions

Un schéma détaillé (voir Figure 4.2) illustre les flux de données entre les différents composants, clarifiant les communications entre le frontend, le backend, les bases de données et le dispositif embarqué.

Cette architecture, bien que confrontée à quelques défis d'intégration, a permis de répondre efficacement aux exigences du projet, assurant une communication robuste et sécurisée entre les différentes parties de l'application.

4.5 Perspectives d'amélioration

Bien que l'application soit pleinement fonctionnelle, plusieurs améliorations restent envisageables pour enrichir l'expérience utilisateur et renforcer la sécurité du système. Ces perspectives, non mises en œuvre principalement par manque de temps, sont détaillées ci-dessous.

— 4.5.1 Mise en place d'une authentification à deux facteurs (2FA)

Pour renforcer la sécurité des utilisateurs, l'intégration d'une authentification à deux facteurs est une priorité future. Ce mécanisme ajouterait une couche supplémentaire de protection en

demandant une vérification additionnelle (comme un code envoyé par email) lors de la connexion. Cette amélioration garantirait une meilleure protection contre les accès non autorisés.

— 4.5.2 Gestion des rôles utilisateurs

Actuellement, l'application propose uniquement une visualisation employeur. L'ajout de différents rôles utilisateurs, tels que les clients et les administrateurs, permettrait une gestion plus fine des permissions et des accès. Chaque type d'utilisateur pourrait ainsi bénéficier d'une interface adaptée à ses besoins spécifiques, renforçant l'ergonomie et l'efficacité de l'application.

— 4.5.3 Analyse approfondie des données de conduite

Avec davantage de données fournies par le dispositif embarqué, l'application pourrait offrir des analyses plus détaillées et utiles aux utilisateurs. Par exemple, des scores mensuels de conduite, des statistiques annuelles, et des évaluations des économies réalisées grâce à une conduite douce pourraient être générés. Ces fonctionnalités permettraient aux utilisateurs de mieux comprendre et améliorer leurs habitudes de conduite.

— 4.5.4 Personnalisation de l'interface utilisateur

Une autre perspective d'amélioration est l'introduction d'une multitude de thèmes personnalisables pour le site. Permettre aux utilisateurs de choisir entre différents thèmes de couleurs améliorerait l'expérience utilisateur en rendant l'interface plus attrayante et adaptée aux préférences individuelles.

— 4.5.5 Notifications enrichies

Les notifications pourraient être optimisées pour assurer un suivi plus précis de l'entretien des véhicules. Une fonctionnalité d'alertes informerait les utilisateurs des besoins essentiels, tels que la réalisation d'une vidange, le remplacement des pneus, ou encore les rappels automatisés pour le contrôle technique. Ces améliorations contribueraient à une gestion efficace des échéances réglementaires et à la prévention de problèmes tels que la perte d'adhérence due à l'usure des pneus.

En conclusion, ces axes d'améliorations visent à accroître la sécurité, la convivialité, et la richesse fonctionnelle de l'application. La mise en œuvre future de ces fonctionnalités contribuera à offrir une solution encore plus complète et satisfaisante pour les utilisateurs finaux.

Chapitre 5

Systèmes embarqués

5.1 Introduction

Notre équipe de deux personnes était chargée de la conception et de la mise en œuvre de systèmes embarqués (Arduino Uno et Raspberry Pi 4 Model B). L'objectif de notre équipe était de mettre en place une solution pouvant être branchée sur le port OBD-2 de chaque véhicule. Celle-ci doit permettre de récupérer la position GPS du véhicule ainsi que des informations précises sur celui-ci telles que le nombre de tours par minute, le niveau d'essence ou la charge du moteur. Elle doit également offrir la possibilité de prendre périodiquement des photos du véhicule, de diffuser un flux vidéo à la demande, et de conserver une vidéo des 30 dernières secondes du véhicule, ce qui sera utile dans le cas où un problème serait observé sur celui-ci pour en identifier la cause.

5.2 Données du véhicule

Nous utilisons plusieurs types de capteurs afin de récupérer les données liées au véhicule et les transmettre au service.

5.2.1 GPS

Nous avons utilisé un module GPS DFRobot TEL0094 afin de récupérer la position du véhicule de manière périodique. Ce module est connecté, via un lien série, à un Arduino Uno lui-même connecté à un Raspberry Pi. Nous utilisons la librairie TinyGPSPlus sur l'Arduino pour récupérer différentes informations :

- Le nombre de satellites GPS connectés
- La position du véhicule (longitude, latitude, altitude)
- La vitesse du véhicule
- L'orientation du véhicule
- La date et l'heure obtenues depuis les satellites GPS

Ces informations nous permettent de localiser précisément le véhicule, dans l'espace et dans le temps. Une fois récupérées par l'Arduino, ces informations sont encodées dans une structure JSON (voir annexe C) puis transmises au Raspberry Pi via une liaison série sur port USB.

Une amélioration possible serait de connecter directement le module GPS au Raspberry Pi afin de se passer de l'Arduino Uno et de réduire le coût de production du système embarqué. Nous n'avons malheureusement pas eu la possibilité de le faire, par manque de librairie adaptée à notre module sur le Raspberry Pi et par manque de temps.

5.2.2 OBD-2

Nous utilisons un adaptateur OBD-2 vers port série basé sur l'ELM327. Cet adaptateur nous permet de récupérer en temps réel des données depuis le véhicule, comme le nombre de tours par minute, la

vitesse, etc. Cet adaptateur est directement connecté au Raspberry Pi via une liaison série sur port USB. Nous utilisons la librairie python-obd afin de récupérer les données suivantes :

— Charge du moteur
— Température du liquide de refroidissement
— Pression de l'entrée d'air
— Tours par minute du moteur
— Vitesse du véhicule
— Niveau d'essence
— Température de l'air ambiant

5.2.3 Photos

Selon le cahier des charges, l'infrastructure embarquée doit être capable de capturer régulièrement des images du véhicule, de les stocker dans le cas d'une panne de connexion Internet, de capturer un flux vidéo à la demande, et enfin d'enregistrer en continu un flux vidéo contenant au moins les 30 dernières secondes, afin de pouvoir identifier la cause d'une anomalie sur le véhicule.

Pour que le Raspberry Pi puisse prendre des photos et enregistrer des vidéos, nous utilisons la caméra Raspberry Pi Camera Module 3 Wide et la librairie Python Picamera2.

La caméra branchée sur le Raspberry Pi a trois objectifs :

— envoyer périodiquement des photos prises de l'intérieur du véhicule à Flask (backend) et les rendre visibles sur le site web (naif-control.u-strasbg.fr ou naif-backup.u-strasbg.fr, voir 6.1.6 pour plus d'informations)
— diffuser un flux vidéo dans le cas où l'utilisateur le décide sur le site web
— enregistrer un flux vidéo continu des 30 dernières secondes du véhicule

Cette section détaille comment le premier objectif est accompli. Nous verrons ensuite dans les sections 5.3 et 5.4 comment les deux autres objectifs sont accomplis.

Le mode de fonctionnement "normal" (aucun flux vidéo n'a été demandé par un utilisateur du site web) consiste à envoyer périodiquement des photos (et des données GPS et OBD-2) à Flask, le frontend récupère ensuite la photo la plus récente et l'affiche sur le site web. La figure 5.1 illustre ce mode de fonctionnement.

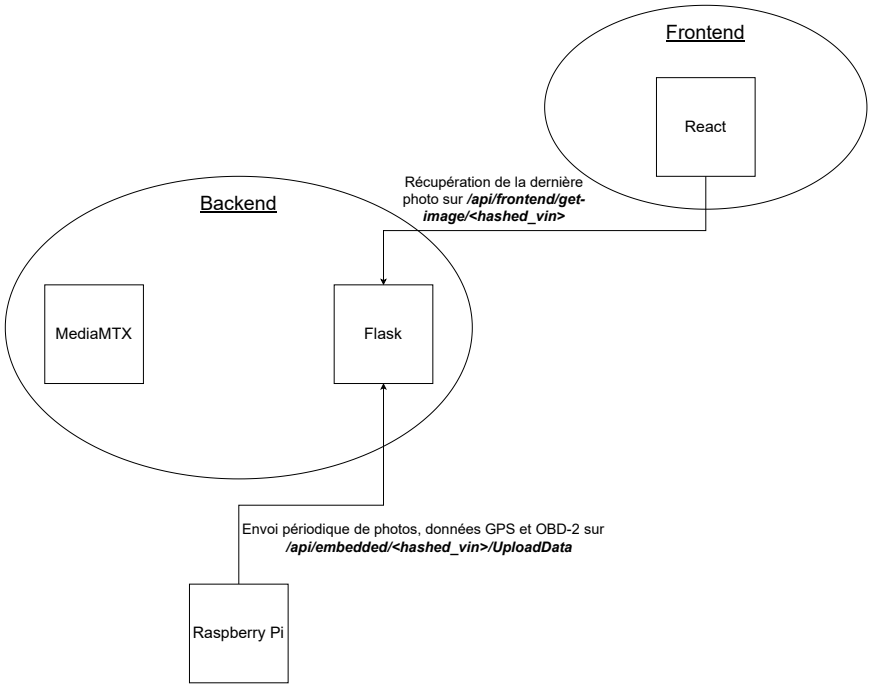


FIGURE 5.1 – Déroulement des communications lorsqu'il n'y a pas de flux vidéo

1. Le Raspberry Pi envoie périodiquement des photos sur la route `/api/embedded/<hashed_vin>-/UploadData` de Flask.
2. Les photos reçues par Flask sont enregistrées dans un bucket S3.
3. Le frontend React va périodiquement (toutes les 30 secondes) récupérer la dernière photo stockée dans le bucket S3 sur Flask via une requête HTTP GET sur la route `/api/frontend/get-image/-<hashed_vin>`

5.2.4 Transmission des données

Les données récoltées sont transmises périodiquement au backend, sur le framework web **Flask**, via des requêtes HTTP POST sur l'endpoint `/api/embedded/<hashed_vin>/UploadData`. Ces données sont encodées dans une structure JSON (voir annexe C) stockant plusieurs entrées GPS, OBD-2 et photos récoltées à différents instants. Lorsque le Raspberry Pi récolte assez d'entrées (5 par défaut), elles sont automatiquement envoyées au backend.

À titre informatif, les endpoints spécifiques aux systèmes embarqués débutent par `/api/embedded` et ceux spécifiques au frontend par `/api/frontend`. Une difficulté rencontrée était de faire en sorte que Flask sache à quel véhicule appartient une requête reçue d'un Raspberry Pi. Pour cela, nous avons décidé d'intégrer dans chacun des endpoints le *hashed_vin*[1] (aussi appelé de manière équivalente *car_id* dans ce mémoire) du véhicule auquel appartient le Raspberry Pi.

5.3 Diffusion d'un flux vidéo

5.3.1 Technologies utilisées

Le serveur de streaming temps réel utilisé est **MediaMTX**, cela pour plusieurs raisons :
— MediaMTX est léger et sans dépendance, ce qui en fait un choix parfaitement adapté pour un système aux ressources limitées comme un Raspberry Pi
— beaucoup de protocoles de streaming sont supportés (RTSP, HLS, WebRTC...)
— bien que MediaMTX soit léger, il peut gérer de multiples flux vidéo concurrents
— il dispose d'une compatibilité directe avec de nombreuses caméras compatibles avec le Raspberry Pi
— ce serveur est encore activement maintenu quotidiennement et mis à jour presque tous les mois

MediaMTX est, tout comme Flask, hébergé sur le backend de notre infrastructure.

Pour le streaming vidéo, nous avons choisi les deux protocoles suivants :
— **Real Time Streaming Protocol** (RTSP) est utilisé pour la signalisation, Real-time Transport Protocol (RTP) pour le transport média du flux vidéo diffusé par le Raspberry Pi et Real-time Transport Control Protocol (RTCP) pour obtenir les statistiques associées aux flux RTP.
Pour information, RTSP supporte les protocoles de transport UDP (latence plus faible mais risque de pertes de trames) et TCP (latence plus forte mais protocole plus fiable). Nous avons opté pour TCP car nous voulons nous assurer que l'intégralité de la vidéo soit lisible pour détecter le moindre problème sur un véhicule, même si cela peut impliquer une latence plus élevée.
— **HTTP Live Streaming** (HLS), après conversion du flux vidéo RTSP par MediaMTX, pour l'afficher sur le site web.

Nous avons choisi RTSP car ce protocole a déjà fait ses preuves dans l'industrie pour les caméras IP. Du fait que RTSP ne soit pas pris en charge nativement par les navigateurs web, il nous fallait convertir le flux vidéo RTSP dans un autre protocole : nous avons choisi HLS, un protocole de streaming basé sur HTTP qui est pris en charge par de nombreux navigateurs web et qui est le protocole le plus utilisé pour le streaming live[2].

1. Le Vehicle Identification Number (VIN) est un identifiant unique pour chaque véhicule.
2. `https://bitmovin.com/wp-content/uploads/2022/12/bitmovin-6th-video-developer-report-2022-2023.pdf#page=15`

5.3.1.1 Comparaison du schéma de communication choisi

Le schéma de communication choisi (flux vidéo RTSP diffusé par le Raspberry Pi puis conversion en HLS pour l'affichage web) nous paraît être le plus efficace pour notre architecture parmi les autres schémas suivants.

Motion JPEG (MJPEG) : MJPEG est un format de compression vidéo où chaque image JPEG est envoyée individuellement dans un message HTTP, le rendant donc nativement pris en charge par les navigateurs web. Cependant, MJPEG souffre de plusieurs défauts par rapport à la solution proposée :

- l'envoi d'une seule image non compressée par message HTTP est très inefficace et consomme énormément de bande passante dans le cas où le flux vidéo est de haute qualité ou à haut débit d'images
- il n'y a aucun moyen de récupérer l'audio ou de mettre en pause un flux vidéo
- MJPEG ne supporte pas l'adaptive bitrate streaming (technique permettant d'ajuster la qualité d'un flux vidéo selon les capacités du récepteur)

Utiliser uniquement HLS : cette solution reviendrait à utiliser HLS sur le Raspberry Pi à la place de RTSP. Cela permettrait de réduire la charge sur MediaMTX car ce dernier n'aurait plus besoin de convertir chaque flux vidéo RTSP en HLS. Cependant, ce léger gain de performances sur MediaMTX n'en est pas vraiment un car la charge sera déplacée sur le Raspberry Pi. En effet, le protocole HLS, qui fonctionne avec des fichiers de segments .ts et des fichiers de playlist .m3u8, est bien plus lourd à traiter que les paquets envoyés par RTP, ce qui en fait un choix moins judicieux que RTSP pour le Raspberry Pi.

Utiliser WebRTC à la place de HLS : ces deux protocoles ont chacun leurs avantages et inconvénients. WebRTC possède une latence plus faible [3] (ordre de la seconde) que HLS (ordre de la dizaine de secondes) bien que cet écart s'est resserré avec l'extension Low-Latency HLS utilisée par MediaMTX qui réduit la latence à environ 3 secondes. Toutefois, WebRTC est plus complexe à mettre en œuvre (voir ce lien pour plus d'informations) et supporte moins bien la montée en charge que HLS. Il reste à noter que la modularité de l'architecture du projet permet de passer facilement d'un protocole à l'autre.

Utiliser RTMP à la place de RTSP : RTMP (Real-Time Messaging Protocol) est un protocole de streaming qui était initialement propriétaire (utilisé par Flash) et qui a été rendu libre par Adobe. Cependant, ce protocole est moins efficace que RTSP pour l'envoi d'un flux vidéo depuis une caméra du fait de sa latence plus élevée.

5.3.2 Diffusion d'un flux vidéo sur une période donnée

Pour qu'un flux vidéo associé à un véhicule qui possède un certain `hashed_vin` soit diffusé par le Raspberry Pi, deux étapes sont nécessaires :

1. l'utilisateur démarre un flux vidéo sur le site web : le frontend envoie une notification à Flask, puis ce dernier envoie une notification au Raspberry Pi lui indiquant de démarrer un flux vidéo
2. le Raspberry Pi diffuse ensuite le flux vidéo à MediaMTX qui notifie Flask et rend l'URL du flux vidéo visible au frontend pour que l'utilisateur du site web puisse y accéder

Ces deux prochaines sections expliquent en détail le déroulement de ces deux étapes.

5.3.2.1 Étape 1 : demande d'un flux vidéo

Comme expliqué dans la section 5.2.3, le Raspberry Pi, dans le cas normal, envoie périodiquement des photos à Flask. Pour indiquer au Raspberry Pi de diffuser un flux vidéo, le cheminement suivant, indiqué par la figure 5.2, est nécessaire.

3. `https://www.digitalsamba.com/blog/webrtc-vs-hls`

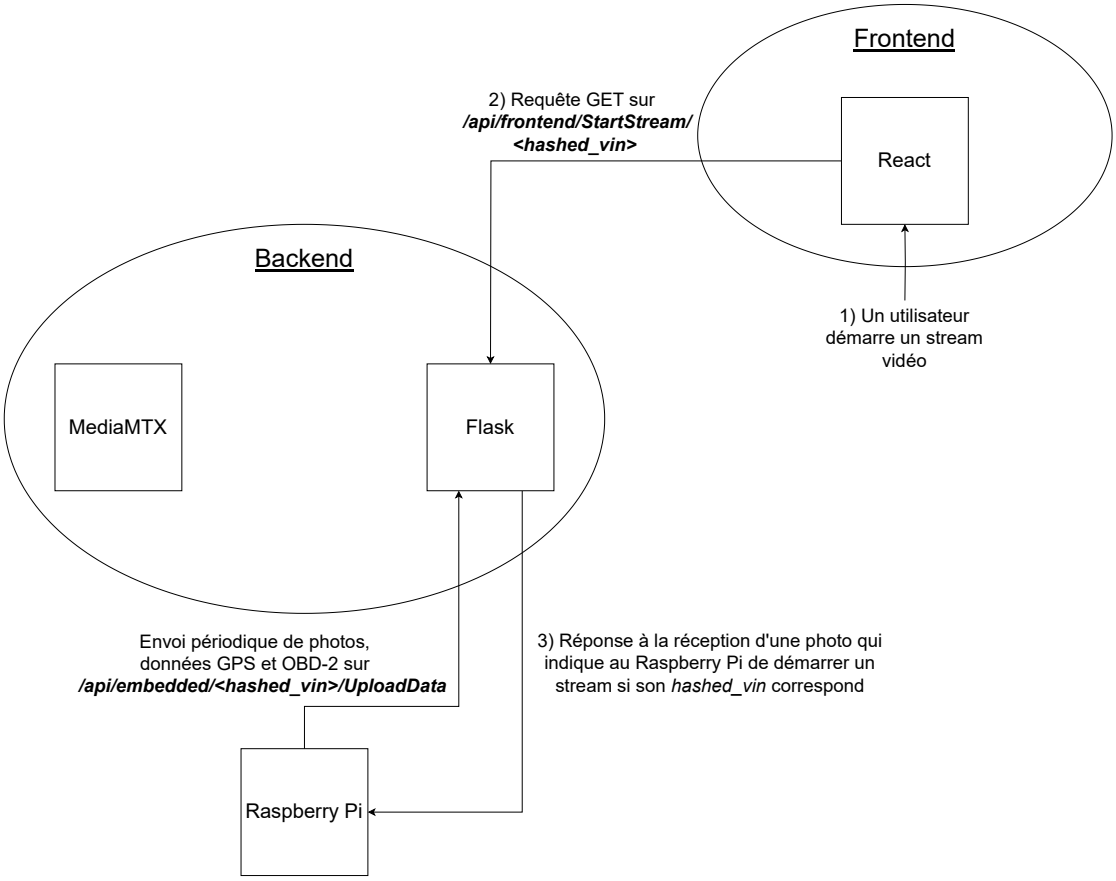


FIGURE 5.2 – Étape 1 : demande d'un flux vidéo

1. L'utilisateur du site web décide de lancer un flux vidéo sur un véhicule possédant un certain `hashed_vin`.
2. Une requête HTTP GET est envoyée sur la route `/api/frontend/StartStream/<hashed_vin>` vers Flask afin d'indiquer à ce dernier que le Raspberry Pi associé à ce `hashed_vin` devra prochainement démarrer un flux vidéo. Pour cela, nous disposons d'un dictionnaire Python nommé `PyCamDict` dont chaque entrée (clef, valeur) est de la forme (`hashed_vin, streamAsked`), avec `streamAsked` un booléen valant True quand un flux vidéo a été demandé par le frontend pour ce `hashed_vin`.
3. Lorsque Flask reçoit une photo du Raspberry Pi, il récupère le `hashed_vin` associé à ce dernier et vérifie si l'entrée correspondante dans le dictionnaire Python `PyCamDict` indique la valeur True. Si c'est le cas, Flask répond non pas avec un message HTTP 200 classique, mais avec un message HTTP 200 contenant une chaîne indiquant au Raspberry Pi qu'il devra diffuser un flux vidéo (la durée du flux vidéo étant fixée à 30 secondes) et qu'il doit donc momentanément cesser l'envoi de photos.

5.3.2.2 Étape 2 : diffusion et affichage du flux vidéo

Pour que le flux vidéo diffusé par le Raspberry Pi soit visible sur le site web, le cheminement suivant, indiqué par la figure 5.3, est nécessaire.

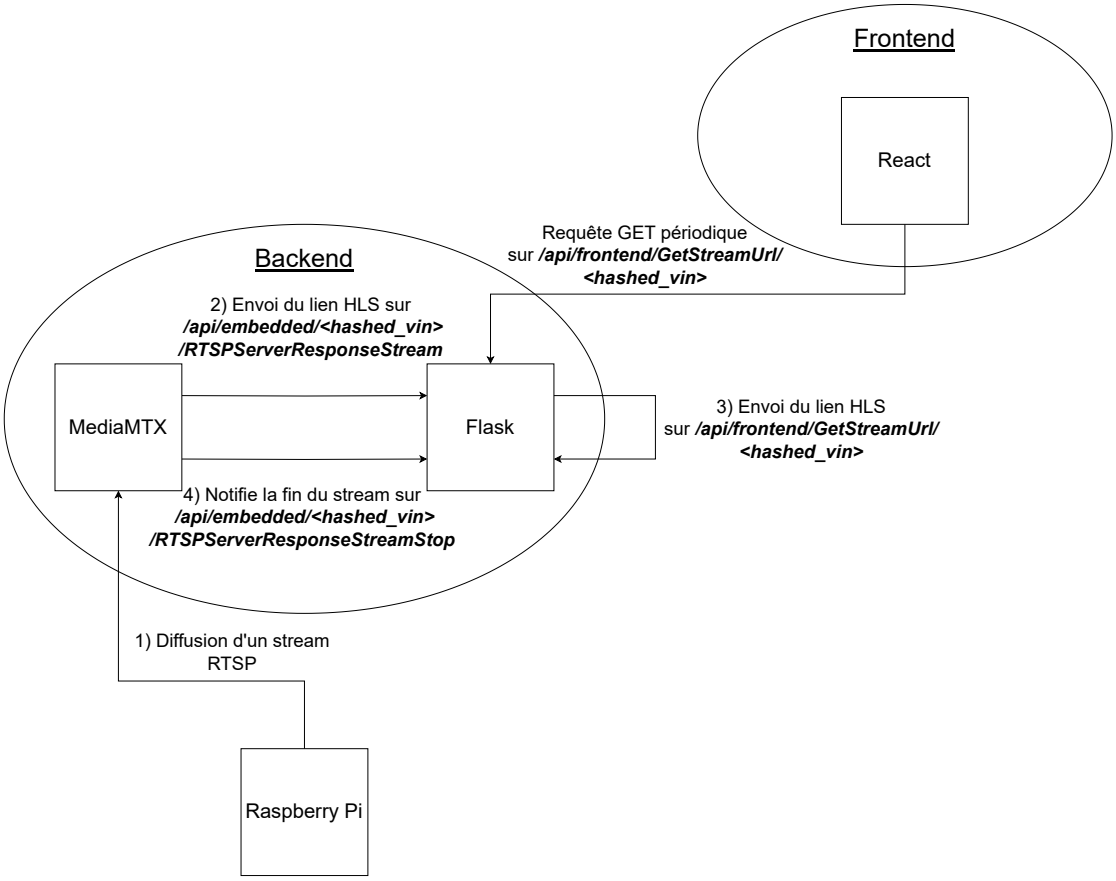


FIGURE 5.3 – Étape 2 : diffusion et affichage du flux vidéo

1. Le Raspberry Pi reçoit la réponse de Flask lui indiquant qu'il doit démarrer un flux vidéo. Il lance donc la diffusion d'un flux vidéo RTSP avec Picamera2/FFmpeg (`rtsp://IP_BACKEND:8554/api/-hls/<hashed_vin>`) vers MediaMTX.
2. Une fois la session TCP établie avec le Raspberry Pi, MediaMTX va convertir le flux vidéo RTSP en HLS (voir 5.3.1) et envoyer l'URL du flux vidéo HLS à Flask sur la route `/api/embedded/<hashed_vin>/RTSPServerResponseStream`.
3. L'URL HLS (`https://IP_BACKEND:8888/api/hls/<hashed_vin>`) du flux vidéo reçu par Flask est envoyé sur la route `/api/frontend/GetStreamUrl/<hashed_vin>`, route utilisée par le frontend pour récupérer l'URL du flux vidéo. Une fois l'URL récupéré, le flux vidéo sera visible sur le site web.
4. Quand le flux vidéo diffusé par le Raspberry Pi est terminé, MediaMTX notifie Flask en lui envoyant un message sur la route `/api/embedded/<hashed_vin>/RTSPServerResponseStreamStop`. En réponse, Flask modifie l'entrée associée à ce `hashed_vin` dans le dictionnaire Python `PyCamDict` en lui assignant la valeur False.

5.4 Enregistrement vidéo continu des 30 dernières secondes

La dernière fonctionnalité requise pour la caméra est d'enregistrer en continu un flux vidéo contenant au moins les 30 dernières secondes, afin de pouvoir identifier la cause d'une anomalie sur le véhicule.

Pour ce faire, nous avons choisi de filmer en continu des séquences vidéo d'une durée de 5 secondes, puis de rassembler les 6 segments les plus récents grâce à FFmpeg pour former une vidéo de 30 secondes au format .mp4, enregistrée directement sur le dispositif embarqué. À chaque fois qu'un nouveau segment est filmé, la vidéo est modifiée : le segment le plus ancien est retiré pour laisser place au nouveau.

Chapitre 6

Infrastructure

L'infrastructure doit répondre à des contraintes de disponibilité, d'adaptabilité et de sécurité. La manière avec laquelle nous répondons à ces contraintes est détaillée ci-dessous.

6.1 Vision globale de l'infrastructure

L'infrastructure est divisée en deux sites, le premier : le site C315, est situé dans la salle C315 du campus d'Illkirch et le second : le site OpenStack est hébergé dans le datacenter du campus de l'Esplanade.

6.1.1 Architecture du site OpenStack

Le site OpenStack est géré par le gestionnaire de site RedHat OpenStack. Un outil facilitant la création et gestion de machines virtuelles, réseaux, adresses IP flottantes et pare-feux.

Le site OpenStack est divisé en 3 parties :

— L'environnement de test et de développement
— L'environnement de secours
— L'environnement utilitaire

Le schéma ci-dessous (Figure 6.1) représente le site OpenStack. On y voit les différentes machines virtuelles (VM) ainsi que leur répartition dans les différents groupes. On voit par exemple les VM de développement assignées en fonction du développement qui y est effectué, la "VM Frontend" sert d'environnement de travail pour l'équipe développant la partie frontale de l'application web. Les VM "backup" de l'environnement de secours quant à elles ne sont pas assignées à une tâche en particulier, c'est l'orchestrateur de l'environnement de secours qui décide des tâches qui les encombrera. Finalement la machine "utils" sert principalement à héberger le serveur VPN et la machine bastion sert à héberger le bastion ssh.

6.1.2 Architecture du site C315

Le site C315 est formé de 3 machines NUC et d'un RaspberryPi connectés par un switch. La plateforme de virtualisation ProxMox est installée sur les machines NUC.

Les NUCs sont tous identiques et contiennent chacun un processeur i5-9500T avec 6 cœurs physiques, 32 Go de RAM et 256 Go de disque SSD.

Le RaspberryPi est un RaspberryPi 4 possédant 4 Go de RAM.

La connectivité réseau est assurée par un switch Cisco Catalyst 3560 qui dispose de 24 ports Ethernet 10/100 et de 2 ports Gigabit Ethernet SFP.

Nous avons d'abord réfléchi à installer Openstack comme plateforme de virtualisation sur le site C315 afin d'avoir deux sites les plus proches possibles. Cependant OpenStack étant un service hautement configurable (et de façon "obligatoire", il n'existe pas d'Openstack standard il faut obligatoirement choisir et configurer ses composants), l'installer considère une très grande charge de travail et la parité entre un site privé et notre site est presque inatteignable.

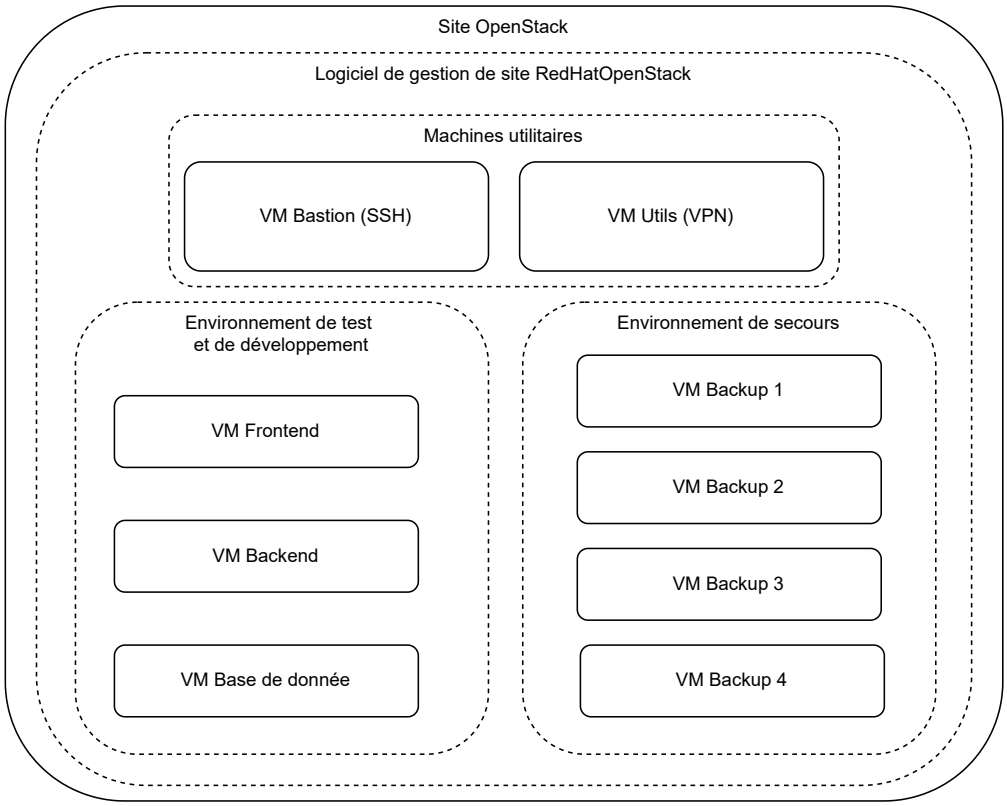


FIGURE 6.1 – Schéma de l'architecture du site OpenStack

Après avoir donc écarté cette option, nous avons évalué les solutions Proxmox et XCP-NG. Notre choix s'est porté sur la solution Proxmox pour plusieurs raisons :

— le support natif de la haute disponibilité
— pour ce faire le support natif (via l'installation d'un paquet) de CEPH, une technologie de stockage distribué
— une documentation fournie, complète et bien écrite [1]
— des modules Ansible permettant l'automatisation de certaines tâches [2]
— une bonne expérience passée avec la plateforme

Proxmox offre notamment les fonctionnalités suivantes :

6.1.2.1 CEPH

CEPH [3] est une solution de stockage distribué hautement performante. CEPH permet de distribuer et de répliquer notre stockage, ce qui vital en cas notamment de défaillance matérielle.

Nous pouvons en effet perdre un tiers de nos disques sans perdre une seule donnée. Le coût est donc logiquement d'un tiers de notre stockage qui est alors utilisé pour la redondance des données.

6.1.2.2 Haute disponibilité

La haute disponibilité permet de migrer automatiquement les VM en cas de défaillance d'une machine hôte. Cela permet d'assurer la continuité du service même en cas de problème matériel.

1. `https://pve.proxmox.com/wiki/Main_Page`
2. `https://docs.ansible.com/ansible/latest/collections/community/general/proxmox_module.html`
3. `https://ceph.io/en/`

6.1.2.3 Ansible

Les VM peuvent intégralement être créées par Ansible. Cela permet d'avoir des fichiers de définition et une reproduction propre de notre architecture. Nous n'avons malheureusement pas eu le temps d'implémenter la création de VM par Ansible.

6.1.2.4 Cloud Init

Cloud Init [4] permet à la création de VM de configurer de nombreux paramètres comme le réseau ou le hostname et d'y ajouter sa clé SSH.

6.1.2.5 Montée en charge

Le fait d'avoir choisi une plateforme cloud telle que Proxmox permet de facilement rajouter des machines physiques, il suffit d'y installer la solution et de les ajouter à notre cluster.

6.1.3 Réseau OpenStack

Les VM du site OpenStack possèdent chacune une adresse dans le sous-réseau `192.168.0.0/24`. C'est via ces adresses que les VM communiquent entre elles, ce sont des communications intra-site. On compte parmi ces communications par exemple :

- Redirection de requêtes : un proxy inverse va rediriger les requêtes à une machine capable d'y répondre. Les requêtes pour l'interface web iront en direction des serveurs web tandis que les requêtes d'écriture dans la base de données iront sur un serveur dorsal.
- État de santé : la VM utils effectue régulièrement des demandes de diagnostic auprès de l'ensemble des VM afin de connaître leur état de santé.
- Communications inter-services : certains logiciels utilisés (par exemple Consul et Nomad) nécessitent que les VM communiquent entre elles. Ils utilisent l'adresse dans le sous-réseau `192.168.0.0/24` pour communiquer.

La VM utils est également dans deux autres espaces d'adressages : `10.7.0.0/24` et `10.8.0.0/24` qui sont les deux sous-réseaux attribués aux connexions VPN. Le serveur VPN côté OpenStack possède l'adresse `10.7.0.1` et le client VPN `10.8.0.12`.

Certaines VM se voient attribuer une adresse IP flottante, en particulier le bastion et la VM qui héberge le proxy inverse dans l'environnement de secours.

Ces informations sont schématisées dans la Figure 6.2 ci-dessous. Les parties concernant le trafic entrant/sortant du site C315 et en provenance d'Internet sont précisées dans les parties suivantes : Interconnexion des sites et Accessibilité depuis Internet.

Comme précisé dans la partie précédente concernant le site OpenStack, le logiciel de gestion de site se charge de créer le sous-réseau `192.168.0.0/24`, d'allouer et d'assigner les adresses IP. Il assure que les VM aient un accès au réseau Internet, via l'allocation de routeurs, afin qu'elles puissent se mettre à jour et installer des logiciels par exemple.

Si nous avions eu plus de temps, nous aurions assigné un sous-réseau à chaque groupe de VM en fonction de leur groupe. C'est-à-dire que les machines de l'environnement de test et de développement ne partageraient pas leur espace d'adressage avec les machines de l'environnement de secours. Cela entraverait la cartographie du site si un intrus venait à s'introduire dans le réseau.

6.1.4 Réseau C315

Les machines physiques sont connectées à un switch CISCO Catalyst 3560. Elles possèdent toutes une adresse IP privée dans le sous-réseau `10.0.0.0/24`. Le Raspberry Pi a lui en plus une adresse IP publique dans le sous-réseau `130.79.255.33/27` qui lui permet d'accéder à Internet via la `gateway` `130.79.255.62` (elle nous est fournie et n'est pas gérée par nous). Ces informations sont résumées dans la figure 6.3.

4. `https://cloud-init.io/`

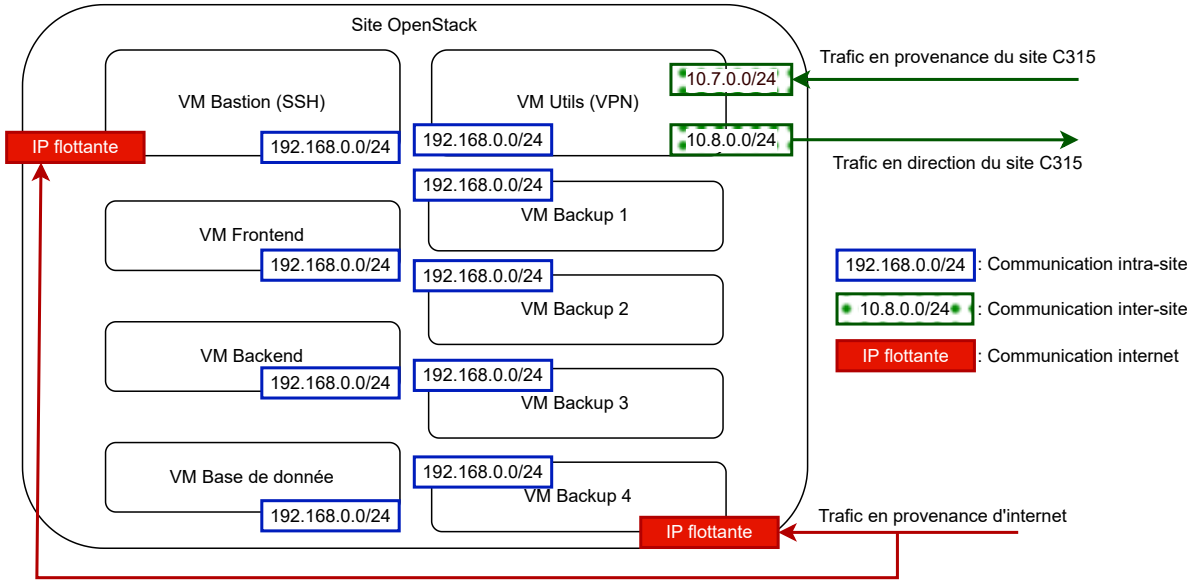


FIGURE 6.2 – Schéma de l'architecture du réseau du site OpenStack

Le Raspberry Pi fait office de NAT pour que les machines n'étant pas accessibles depuis Internet puissent quand même y accéder, notamment pour effectuer des mises à jour (nécessaire pour installer CEPH sur les machines Proxmox). Ce NAT est effectué via des règles `iptables`.

Les machines hôtes Proxmox n'utilisent le réseau que pour s'y connecter, via rebond sur le Raspberry Pi, installer des paquets au besoin et faire de l'observation.

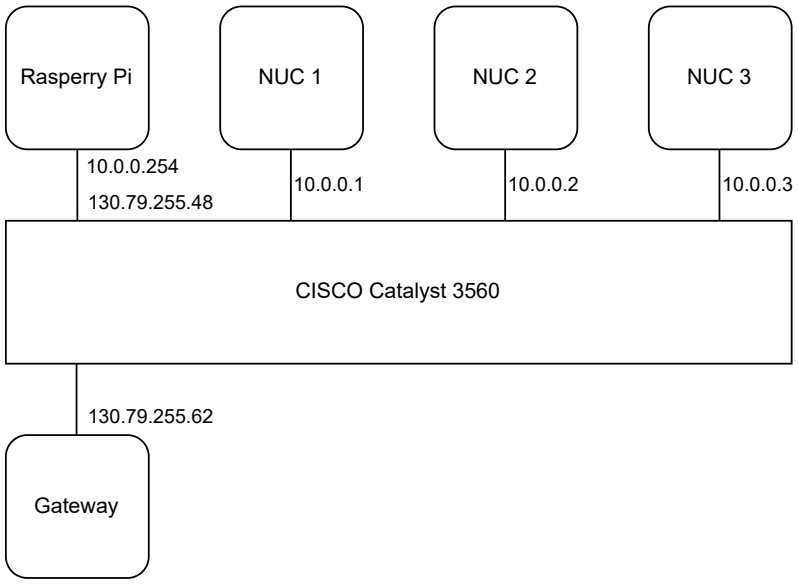


FIGURE 6.3 – Schéma de l'architecture du réseau du site C315

Les VM ont toutes une IP locale dans les sous-réseaux `10.0.0.0/24` et `fc00::/7`. Le Raspberry Pi leur fournit un accès réseau par NAT, de la même façon que pour les machines physiques Proxmox. Les trafics réseaux sont similaires à ceux du site OpenStack :

— Trafic via le proxy (redirection de requêtes)
— État de santé
— Communications inter-services, par exemple `Nomad` et `Consul`

Pour obtenir la connectivité complète, notamment pour communiquer d'un site à l'autre, les VM sont configurées avec des routes, installées via `Netplan` [5] (l'OS utilisé étant Ubuntu), lui même installé via `Ansible`, qui leur indique par exemple pour le VPN de passer par l'interface cliente VPN du Raspberry Pi. Leurs adresses sont elles configurées via Netplan et via Proxmox par le biais du mécanisme Cloud-Init [6].

La connexion à distance aux VM se fait donc via un rebond par le Raspberry Pi si elles n'ont pas d'adresse publique.

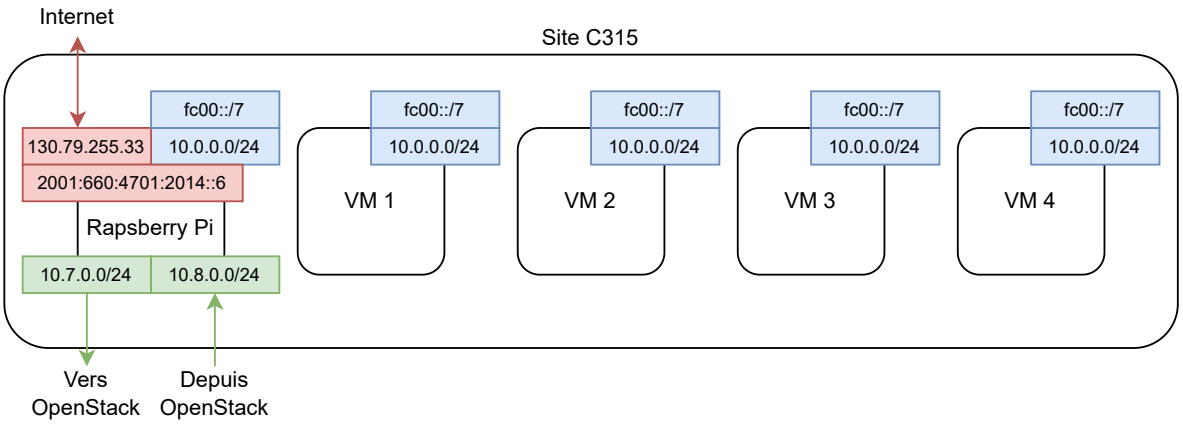


FIGURE 6.4 – Schéma de l'architecture du réseau des VM du site C315

6.1.5 Interconnexion des sites

L'interconnexion des sites sert principalement à s'échanger l'état de santé des sites. Elle permet également la copie des informations stockées dans les bases de données afin d'avoir plusieurs versions à jour des données en cas de problème sur l'une des bases de données.

L'interconnexion a été rendue possible grâce à la création de 2 clients et 2 serveurs VPN WireGuard [7]. Wireguard a été choisi car c'est un outil plus facile d'accès qu'openVPN [8] qui nécessite une gestion poussée des certificats et des clés de chiffrement, ce qui n'est pas propice à l'automatisation. Chaque site possède 1 client et 1 serveur VPN chacun. Comme schématisé dans la Figure 6.2, les requêtes qui sortent d'un site utilisent le client VPN dudit site pour atteindre le serveur VPN de l'autre site. Cette encapsulation permet de propager le sous-réseau d'un site à l'autre. Les machines dans le sous-réseau `192.168.0.0/24` sont capables de ping une machine dans le sous-réseau `10.0.0.0/24` et vice versa. Ces communications sont également chiffrées assurant une confidentialité lorsqu'elles sont en transit.

6.1.6 Accessibilité depuis Internet

L'accessibilité depuis Internet a été assurée via des adresses IP flottantes assignées à des VM.

Nous avons également demandé à avoir deux noms de domaines :

— naif-control.u-strasbg.fr pointe vers `130.79.255.40` et `2001:660:4701:2014::6` qui sont des adresses IP flottantes du site C315 assurant une connectivité IPv4 et IPv6.

— naif-backup.u-strasbg.fr pointe vers `185.155.92.173`, une adresse du site OpenStack. Malheureusement notre allocation de ressource ne nous permettait pas d'assigner une adresse IPv6 flottante au site OpenStack.

Ces noms de domaines facilitent l'accès aux applications qui sont accessibles via du texte plutôt qu'une adresse IP. Les noms de domaine ont également été certifiés par l'autorité de confiance Let's

5. `https://netplan.io/`
6. `https://pve.proxmox.com/wiki/Cloud-Init_Support`
7. `https://www.wireguard.com/`
8. `https://openvpn.net/`

Encrypt [9] et leur outil Certbot. Let's Encrypt a été utilisé car c'est la seule autorité de confiance qui soit gratuite et jugée digne de confiance par les navigateurs les plus utilisés [10]. La certification permet le chiffrement des communications au travers du réseau Internet, assurant la confidentialité des échanges.

6.2 DevOps

Il est nécessaire d'introduire une méthode afin d'assurer que les logiciels écrits par les développeur·euse·s de la couche frontale et dorsale soient facilement déployables et accessibles par les utilisateur·ice·s finaux·les. La partie dite "DevOps" contient la méthode avec laquelle nous assurons que le passage de la partie développement à la partie opératoire se fasse sans accroc.

6.2.1 Environnement de développement et de test

L'environnement de développement est hébergé sur le site OpenStack. Chaque VM a sa propre utilité (Frontend pour la partie frontale, Backend pour la partie dorsale et BDD pour la gestion des bases de données). Chacune de ces VM héberge un ou plusieurs conteneurs Docker [11] modifiables à la volée afin que le développement se fasse efficacement.

Le principal point négatif de cette architecture est qu'elle est instable si plusieurs équipes travaillent sur des parties différentes en même temps. Il aurait fallu créer autant d'environnements que d'équipes et faire en sorte que chaque environnement soit à jour lorsqu'une nouvelle fonctionnalité était publiée.

Le point positif de cette architecture est qu'il suffit de se connecter à l'environnement pour pouvoir développer, il n'est pas nécessaire d'initialiser un serveur web, des conteneurs Docker etc. sur sa machine.

6.2.2 Conteneurisation et déploiement des applications

Comme précisé dans la partie précédente, le développement se faisait au sein d'un conteneur Docker. Il suffit donc d'en faire une image à héberger sur un registre d'images Docker. Puis cette image peut être utilisée telle quelle dans l'environnement de production.

Pour ce faire, nous avons développé une pipeline CI/CD sur GitLab. Cette pipeline se lance à chaque fois que du code est poussé sur le dépôt. La pipeline recrée l'image Docker à partir du nouvel état du code et la publie sur un registre d'images Docker sur GitLab.

En réalité, deux images similaires sont créées : une archive, nommée après le hash du commit auquel elle appartient et une nommée "latest" qui écrase la précédente image qui était nommée "latest". Par convention, les images nommées "latest" sont celles utilisées dans l'environnement de production. Il suffit de dire aux logiciels responsables d'aller prendre l'image "latest" plutôt que de modifier le nom de l'image à chaque fois. Si cette image "latest" venait à contenir des bugs ou à dysfonctionner, nous pouvons rapidement aller chercher une précédente image de type archive à réutiliser pour revenir à une version stable.

6.2.3 Environnement de production

6.2.3.1 Orchestration

Pour gérer nos conteneurs nous avons étudié deux possibilités : kubernetes et la stack Consul/Nomad [12] de Hashicorp.

Kubernetes est un orchestrateur de conteneurs open source originellement développé par Google, aujourd'hui géré par la Cloud Native Foundation. C'est une plateforme complète et complexe, qui permet de gérer de très nombreux types de conteneurs et de charges de travail.

9. `https://letsencrypt.org/`
10. `https://letsencrypt.org/docs/certificate-compatibility/` : Firefox, Chrome, iOS...
11. `https://www.docker.com/`
12. `https://www.nomadproject.io/`

Nomad est un orchestrateur de conteneurs développé par Hashicorp. Il permet via des "jobs" de définir des charges de travail qu'il va ensuite déployer sur plusieurs nœuds. Consul quant à lui est un système de gestion d'application, notamment utilisé par Nomad afin de découvrir les services.

Nous avons choisi la deuxième option car :

— Consul permet une découverte des services en leur associant un nom de domaine local, ce qui est nécessaire pour le reverse proxy ainsi qu'une détection de panne
— Nomad et Consul s'intègrent très bien ensemble
— kubernetes est un outil plus lourd et complexe
— c'est un outil que les équipes techniques connaissaient et appréciaient

6.2.3.2 Jobs

Un job Nomad est une définition d'une charge de travail. Il spécifie qu'une ressource (un conteneur pour nous) doit fonctionner quelque part et ses paramètres mais pas sur quel hôte. Une fois le job spécifié, Nomad se charge d'allouer les ressources et de déployer le service correspondant.

Un exemple (raccourci) est notre job qui gère React se situe (en annexe).

On peut y voir notamment un datacenter (ensemble d'hôtes dans Nomad) sur lequel déployer le service, les paramètres de montée en charge (notamment on voit vient bien que Prometheus est notre source de données) ou les ressources allouées par défaut.

Les autres jobs sont assez similaires, avec surtout la configuration spécifique au service qui change (réseau, mounts...). Celui qui diffère profondément est le job pour le reverse proxy : en premier au déploiement sur un hôte, il déploie automatiquement l'adresse publique sur la machine en question et surtout, il met à jour dynamiquement la configuration du reverse proxy afin de répartir la charge.

6.2.3.3 Montée en charge dynamique

Il existe un job nommé `autoscaler-prod/backup` (en fonction du site) qui, en fonction de la charge des machines récupérée depuis Prometheus et de la valeur `target-value` définie dans chaque job, va créer et détruire des conteneurs pour répondre à la charge.

Le fonctionnement est : si l'utilisation récupérée depuis Prometheus est supérieure à celle définie dans le job alors on crée un nouveau conteneur, et après un petit de temps si la charge est redescendue, on peut en supprimer un. Cela nous permet donc de répondre dynamiquement à la demande.

6.2.3.4 Reverse proxy

Nous avons choisi Nginx comme reverse proxy pour ses performances, sa relative facilité de configuration et car contrairement aux autres options similaires, nous avions déjà utilisé cet outil.

Le reverse proxy permet de diriger le trafic vers les bons services et de faire de la répartition de charge (selon les paramètres qui lui ont été spécifiés par Nomad).

6.2.4 Environnement de secours

L'environnement de secours est hébergé sur le site OpenStack et sert d'alternative à l'environnement de production en cas de dysfonctionnement de ce dernier. Il est capable de répondre aux requêtes de la même manière que le ferait l'environnement de production, à l'exception des requêtes IPv6, qui ne peuvent être traitées dû à l'absence d'un espace d'adressage public IPv6 sur le site OpenStack qui nous a été attribué. Le passage d'un site à l'autre en cas de dysfonctionnement se fait comme suit :

1. Les requêtes provenant de la flotte de véhicule et de la PWA en direction du site principal échouent. Le logiciel émetteur détecte l'échec et modifie automatiquement la destination pour les envoyer vers le site secondaire. L'URL `naif-control.u-strasbg.fr` ne répond plus alors il essaye automatiquement l'URL de secours : `naif-backup.u-strasbg.fr`.
2. Si nous avions eu plus de temps, nous aurions créé une alerte à l'aide de Prometheus (par mail ou téléphone) qui alerte les responsables du site principal.

3. Pendant que le site principal dysfonctionne, l'URL principale ne fonctionne plus mais les client·e·s du service web ont été averti·e·s qu'une URL de secours est disponible et utilisent cette URL.
4. Les responsables de l'infrastructure modifient l'entrée DNS pour rediriger les requêtes en direction de l'URL principale vers le site de secours en attendant de réparer le site principal. Si nous avions eu un contrôle sur les règles BGP, nous aurions utilisé l'anycast qui permet d'assigner la même adresse IP à plusieurs machines, ce qui aurait retiré la nécessité de modifier l'entrée DNS étant donné que le basculement aurait été automatique. L'anycast aurait également permis de faire de l'équilibrage de charge entre sites.
5. Une fois le site principal réparé, l'entrée DNS est rétablie et les données qui ont été insérées dans les bases de données du site secondaire sont copiées sur le site principal, que ce soit manuellement pour InfluxDB [13] ou automatiquement pour PostgreSQL et MinIO.

L'environnement de secours est disponible de manière permanente, c'est-à-dire qu'à tout moment les client·e·s peuvent aller sur l'interface web servie par le site OpenStack alors que l'environnement de production est toujours disponible. Si nous avions eu plus de temps, nous aurions automatisé la mise à disposition de l'environnement de secours seulement si l'environnement de production dysfonctionnait.

6.2.5 Observabilité

L'observabilité est essentielle à la gestion d'un site. Parmi les avantages qu'elle procure se trouvent : la surveillance de l'état des systèmes, la compréhension des états de santé en temps réel ; collecter, analyser et corréler des données provenant de différentes sources ; comprendre ce qu'il se passe lorsque des problèmes apparaissent et pourquoi ils apparaissent ; anticiper les périodes de fort trafic (par exemple le jour plutôt que la nuit, période de promotion sur une offre).

L'observabilité sur nos sites est faite par les logiciels Prometheus [14] (qui inclut une base de données et des logiciels utilitaires) et Grafana [15] (qui est une suite d'outils de visualisation de données). Ces logiciels ont été choisis car ils sont très simples à installer ainsi qu'à automatiser et les outils de collecte de données le sont tout autant. Les autres solutions type DataDog [16] possèdent des avantages que Prometheus/Grafana n'ont pas, en particulier la gestion du stockage des logs : le stockage est centralisé, redondé, indexé et archivé nativement contrairement à Prometheus. Mais DataDog ne propose pas la même granularité, en termes de métriques collectées que Prometheus, en effet les informations remontées sont définies dans des "exporteurs" qui peuvent être écrits par la fondation responsable de Prometheus ou la communauté de développeur·euse·s.

Dans notre infrastructure nous avons décidé d'utiliser les exporteurs cAdvisor (écrit par Google) et Node Exporter Full (écrit par la fondation Cloud Native Computing Foundation). cAdvisor exporte les données relatives aux conteneurs Docker qui tournent sur la VM (%CPU, %RAM, entrées/sorties réseau) tandis que Node Exporter Full exporte les données relatives à la VM en elle-même (%CPU, %RAM, entrées/sorties réseau, % utilisation disque entre autres).

6.3 Stockage des données

Les données envoyées par les équipements embarqués dans la flotte de véhicule sont le point central du modèle commercial de ce projet. Il est donc nécessaire de rendre ces données redondantes, de créer des copies de secours et de les rendre disponibles tout en étant scalable.

Les données OBD-2 et GPS envoyées par la flotte sont stockées dans une base de données temporelle InfluxDB. Les données concernant les client·e·s et le personnel de Naif sont stockées dans une base de données relationnelle PostgreSQL. Les images envoyées par la flotte sont stockées dans un stockage de type S3 MinIO, afin d'avoir accès à ces images : l'index est stocké dans la base de données temporelle.

13. Nous n'avons malheureusement pas eu le temps d'intégrer les outils de synchronisation comme Telegraf/Kafka à l'architecture et seule la version Enterprise (payante) d'InfluxDB permet la synchronisation native après un temps d'arrêt.
14. `https://prometheus.io/`
15. `https://grafana.com/`
16. `https://www.datadoghq.com/`

6.3.1 Données temporelles

Les bases de données temporelles indexent leurs données en fonction d'une date, toutes les données doivent donc être horodatées. L'indexation par date réduit considérablement l'espace mémoire occupé, (1/4 de ce qu'utiliserait une base de données relationnelle PostgreSQL [Par]) sans grandement affecter la rapidité des requêtes sur un nombre raisonnable de données (entre 0% de ralentissements jusqu'à 500% par rapport à PostgreSQL en fonction du nombre de données demandées : de 1 donnée à 500000 soit 1% plus lent toutes les 1000 données en moyenne). Typiquement, si l'on souhaite savoir la vitesse maximale d'un véhicule sur une certaine période ou alors l'évolution de la pression des pneus au fil du temps alors stocker les données en fonction de leur date d'apparition est clairement le bon choix. C'est précisément pour répondre à ce besoin d'analyse au fil du temps que nous avons choisi une base de données temporelle InfluxDB [17].

Nous avons choisi InfluxDB de manière arbitraire car toutes les bases de données temporelles que nous avons étudiées étaient capables de redondance et scalabilité. C'est malheureusement trop tard que nous avons remarqué que la version gratuite d'InfluxDB ne permet pas la réplication rétroactive des données de manière native. C'est-à-dire que si une base de données tombe en panne puis est réparée, il n'existe pas de solution ne sollicitant que la version gratuite d'InfluxDB qui permette de récupérer automatiquement les données qui n'ont pas été reçues dans une autre bases de données qui elle aurait reçu les données.

Il n'y a que la synchronisation en temps réel, que nous utilisons pour mettre à jour l'environnement de secours, qui est disponible et non la synchronisation rétro-active. La synchronisation temps réel permet à l'environnement de secours d'avoir (quasiment) les mêmes données que l'environnement de production dès lors qu'il est utilisé.

Si nous avions su qu'InfluxDB nécessitait une licence pour synchroniser rétro-activement, nous aurions plutôt choisi TimescaleDB [18].

6.3.2 Données structurées

Les données concernant les client·e·s et employé·e·s sont des données structurées (voir Figure 4.1) qui seront bien moins volumineuses que les données envoyées par la flotte de véhicules donc nous avons décidé de les stocker dans une base de données relationnelle PostgreSQL.

PostgreSQL [19] a été choisi arbitrairement parmi les systèmes de gestion de bases de données relationnelles qui proposaient de la scalabilité et de la redondance nativement. Nous aurions également pu choisir MySQL [20] ou MariaDB [21] qui proposent également ces avantages.

6.3.3 Stockage de type S3

Le stockage de type S3 est optimisé pour conserver des objets larges et non structurés (BLOB : binary large object), comme des images par exemple. Similairement à PostgreQSL, MinIO possède des capacités de redondance et de synchronisation entre plusieurs nœuds.

6.4 Sécurité

6.4.1 Pare-feux

La plateforme Openstack offre nativement une gestion fine des accès réseaux via le mécanisme des `security groups` [22]. Ce mécanisme permet de créer des groupes avec les ports autorisés en entrée et sortie ainsi qu'un préfixe IP distant sur lequel appliquer ces règles. La figure 6.5 montre une capture d'écran d'un groupe de sécurité dans Openstack.

17. `https://www.influxdata.com/`
18. `https://www.timescale.com/`
19. `https://www.postgresql.org/`
20. `https://www.mysql.com/fr/`
21. `https://mariadb.org/`
22. `https://docs.openstack.org/nova/latest/user/security-groups.html`

Affichage de 4 éléments

Direction	Type de protocole (EtherType)	Protocole IP	Plage de ports	Préfixe IP distante	Groupe de sécurité distant	Description	Actions
Entrée	IPv4	TCP	8086	10.0.0.0/8	-	-	Supprimer une Règle
Entrée	IPv4	TCP	8086	172.16.0.0/12	-	-	Supprimer une Règle
Entrée	IPv4	TCP	8086	130.79.0.0/16	-	-	Supprimer une Règle
Entrée	IPv4	TCP	8086	192.168.0.0/24	-	-	Supprimer une Règle

FIGURE 6.5 – Exemple d'un groupe de sécurité dans l'interface web d'OpenStack

Sur la plateforme Proxmox, nous utilisons un pare-feu pour obtenir le même effet.

Les connexions sont globalement permises sur les réseaux locaux (`10.0.0.0/8, 192.168.0.0/24 et 172.16.0.0/12`) et fermées vers l'extérieur, mises à part celles nécessaires au fonctionnement de l'application (ports 80 et 443 par exemple). Les connexions SSH sont elles uniquement autorisées sur une machine bastion par site (voir la partie accès aux machines).

6.4.2 Accès aux machines et services

Chaque site possède une machine bastion qui sert de point d'entrée unique : il est nécessaire d'avoir la bonne clé privée pour y accéder et même si un utilisateur mal intentionné réussissait à se connecter à cette machine, elle-même n'a pas accès aux autres machines sans clé privée.

Cela nous permet à la fois d'accéder à des machines qui ne sont pas accessibles depuis Internet et de grandement réduire la surface d'attaque.

Le fait d'avoir une machine unique permet aussi de très facilement couper l'accès SSH en cas de problème (il sera ensuite possible de le retrouver en réactivant la machine bastion depuis les interfaces de contrôle des sites) et permet aussi en cas de faille (comme par exemple la CVE 2024-6387 [23]) d'avoir un nombre très réduit de machines à mettre à jour.

6.4.3 Sécurité des logiciels installés

Afin de garantir la sécurité des systèmes, nous avons fait attention à n'utiliser que des logiciels récents et à jour, si possible en version LTS (Long Term Support, versions supportées plus longtemps, pour Ubuntu par exemple). Cela permet à la fois de disposer de tous les patchs de sécurité et de faciliter leur suivi et les mises à jour.

Les logiciels installés sur les machines sont séparables en deux groupes :

— ceux installés via docker (par exemple React), si jamais un conteneur venait à être compromis, l'attaquant ne pourrait en sortir limitant donc grandement l'impact

— ceux installés sur la machine hôte (par exemple Consul), dans ce cas on crée un utilisateur par service afin de limiter de la même façon l'attaque à uniquement le service compromis et non pas toute la machine

6.5 Évaluation de performances

Nous avions promis, dans notre réponse à l'appel d'offre, de répondre à des contraintes de délai, de temps de réponse à une montée en charge et de forte quantité de client·e·s.

Le temps mesuré entre la mise en panne d'un conteneur et sa ré-exécution par Nomad est de 18 secondes, le temps d'exécution du conteneur est aux alentours de 7 secondes peu importe le conteneur, la plupart du temps est utilisée pour récupérer l'image dans le registre distant. Soit un total de 25 secondes de temps d'arrêt si tous les conteneurs venaient à dysfonctionner d'un coup.

L'intervalle de temps entre la détection d'une montée en charge est entre 5 et 10 secondes (en fonction de si la hausse apparaît au début ou à la fin d'un période de sondage), ajoutée aux 7 secondes

23. `https://www.cert.ssi.gouv.fr/alerte/CERTFR-2024-ALE-009/`

de déploiement d'un conteneur cela donne 12 secondes de temps de réaction à une hausse de la montée en charge

Nous avons utilisé l'outil `ab`[24] de la fondation Apache qui envoie des requêtes HTTP à un serveur ainsi que l'outil `iperf` qui permet non seulement d'avoir une estimation de la bande passante maximale que peut supporter un réseau mais également de saturer le réseau afin de simuler un réseau saturé. Les résultats obtenus sont présents sur la Figure 6.6

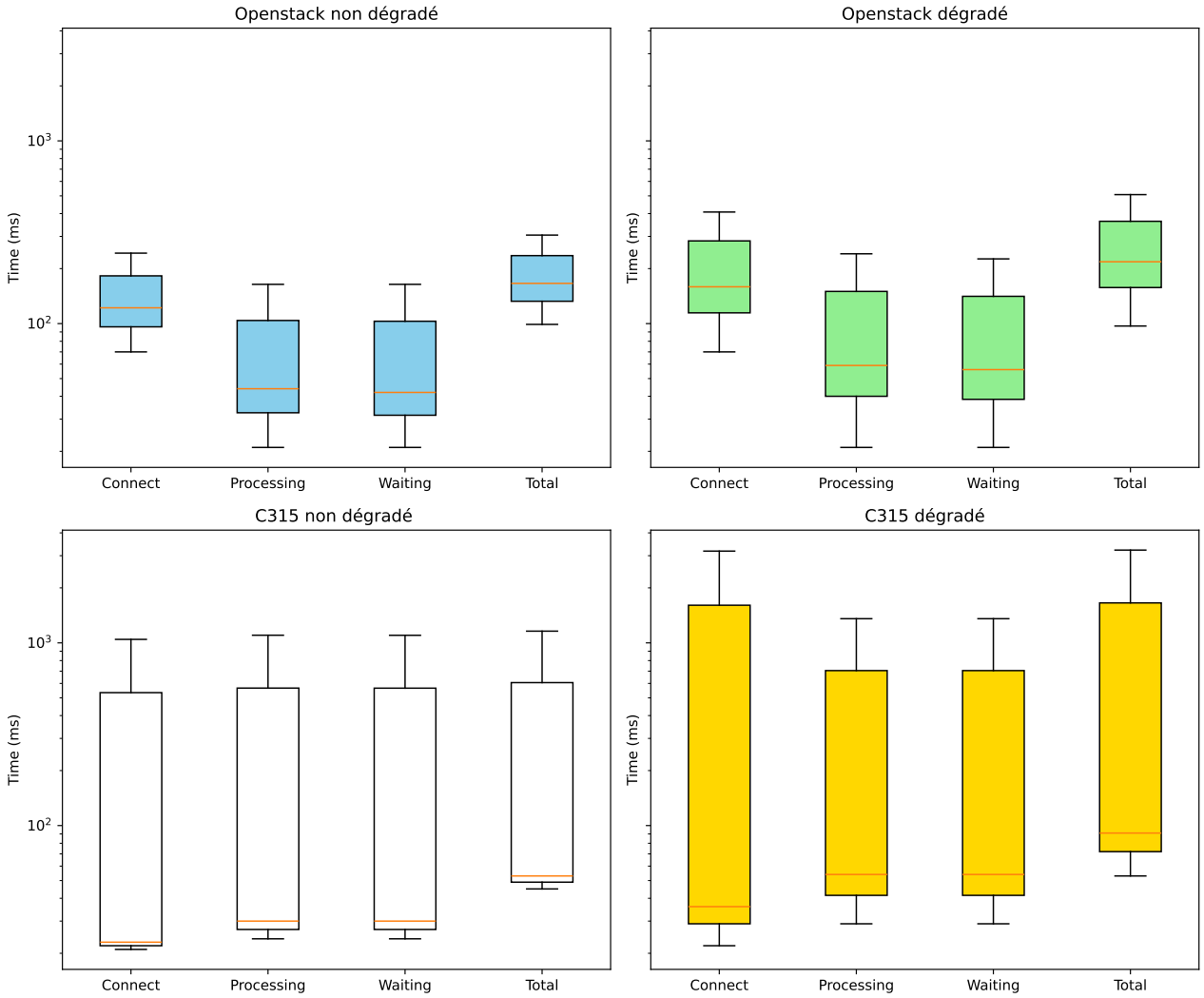


FIGURE 6.6 – Résultats des évaluations de performances sur les différents sites

6.6 Documentation et cartographie

La documentation de l'infrastructure se découpe en une partie destinée aux développeur·euse·s et une autre pour les personnes responsables de l'infrastructure.

La documentation pour les développeur·euse·s définit, via des pages wiki hébergées sur le dépôt GitLab, les différents points nécessaires pour développer en particulier : la connexion à l'environnement de développement.

La documentation pour les personnes responsables de l'infrastructure se décompose en :

1. plusieurs cartographies de l'infrastructure qui énumèrent les machines, leurs adresses, les services et logiciels qui tournent dessus (voir annexe E).
2. des playbooks Ansible et scripts commentés qui contiennent les fichiers de configuration et la liste des commandes effectuées pour atteindre l'état actuel de l'infrastructure.

24. `https://httpd.apache.org/docs/2.4/programs/ab.html`

Chapitre 7

Conclusion

La réalisation de ce projet a représenté un défi complexe, mais particulièrement enrichissant sur les plans technique, organisationnel et humain. À travers la mise en place des trois composantes principales — infrastructure, systèmes embarqués et développement applicatif — nous avons pu appliquer nos connaissances théoriques tout en développant de nouvelles compétences.

En perspective, les bases posées par ce projet pourraient être améliorées en explorant la collaboration avec des constructeurs automobiles pour améliorer le système existant. Une autre amélioration sera d'ajouter des modules pour connecter des capteurs externes tels que des éthylotests ou d'autres dispositifs pertinents afin d'élargir les fonctionnalités et d'adresser les besoins spécifiques des utilisateurs.

Enfin, nous aimerions inclure quelques retours personnels sur cette expérience.

— Nicolas Elfering : en tant que chef de projet, bien que le rôle soit de s'occuper de la gestion de projet, j'aurais aimé consacrer davantage de temps aux aspects techniques bien que j'aie pu contribuer à certaines tâches, notamment dans le domaine de l'applicatif et des systèmes embarqués.

— Florian Halm : Ce projet a été une expérience vraiment enrichissante pour ma part. En effet, travailler sur des systèmes embarqués, domaine qui m'était complètement inconnu avant ce projet, m'a permis d'apprendre beaucoup de nouvelles choses, notamment sur le fonctionnement de quelques protocoles de streaming et sur l'utilisation d'un Raspberry Pi ou d'un Arduino.

— Kévin Hentz : Participer à ce projet en tant que membre de l'équipe applicative a été une expérience à la fois stimulante et formatrice. Ce projet m'a permis d'approfondir mes compétences en développement full stack, tout en explorant de nouvelles technologies et méthodologies. Travailler en étroite collaboration avec mes collègues sur les différentes facettes du projet m'a offert une vision globale du développement d'un système complet, une expérience que j'ai trouvée particulièrement enrichissante et inspirante pour ma carrière future.

— Éloi Colin : ce projet m'a permis de me conforter dans la voie de l'ingénierie cloud et système. Il m'aura permis de découvrir et de progresser sur des solutions techniques passionnantes qui m'ont donné envie d'y passer du temps personnel dans le futur.

— Florent Hardy : J'ai pu, grâce à ce projet, mettre à l'épreuve les capacités que j'avais apprises lors de mes enseignements. Heureux de constater que je suis capable non seulement de participer à un projet d'envergure mais également de progresser au fil de l'eau. L'expérience fut très éprouvante mais très satisfaisante tout de même.

— Jessica Klotz : Ce projet m'a permis de renforcer mes compétences en développement full stack, en approfondissant le frontend, avec la création d'interfaces interactives, et le backend, axé sur la gestion des données et services. J'ai également enrichi ma compréhension des interconnexions techniques, notamment via les APIs pour l'échange de données et les WebSockets pour la communication en temps réel.

— Maxime Friess : Ce projet a été une expérience très enrichissante pour moi, me permettant de m'adonner au domaine de la programmation embarqué auquel je ne suis pas habitué. Mon passage dans l'équipe Infrastructure m'a aussi permis de développer mes compétences dans ce domaine.

Bibliographie

[Par] Fabio Pardi. *A Timeseries Case Study : InfluxDB VS PostgreSQL to store data.* https://portavita.github.io/2018-07-31-blog_influxdb_vs_postgresql.

[SS14] Alex Xandra Albert Sim et Benhard Sitohang. « OBD-II standard car engine diagnostic software development ». In : *2014 International Conference on Data and Software Engineering (ICODSE)*. 2014, p. 1-5. doi : 10.1109/ICODSE.2014.7062704.

Annexe A

Introduction

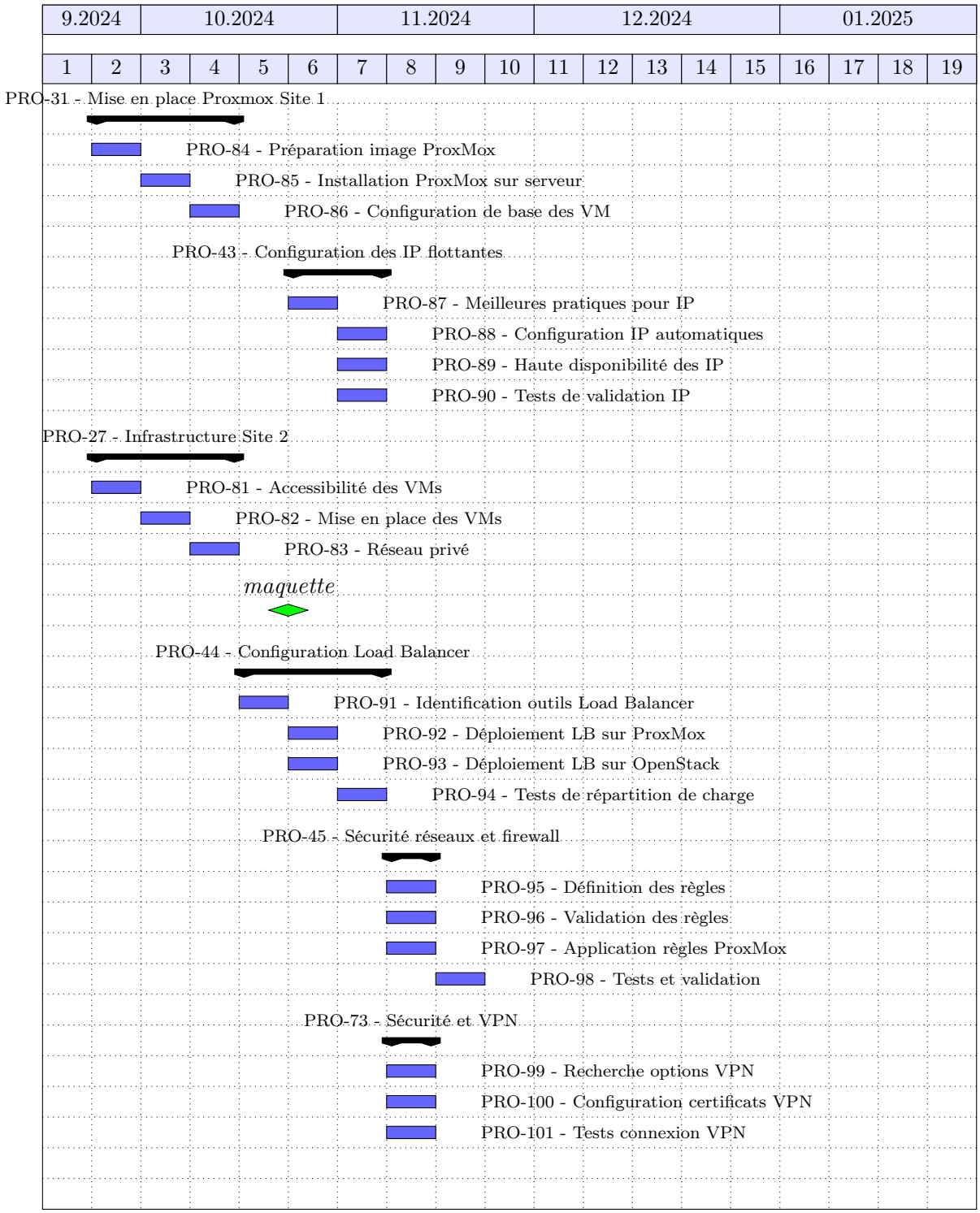
Liste non-exhaustive des contraintes à satisfaire :

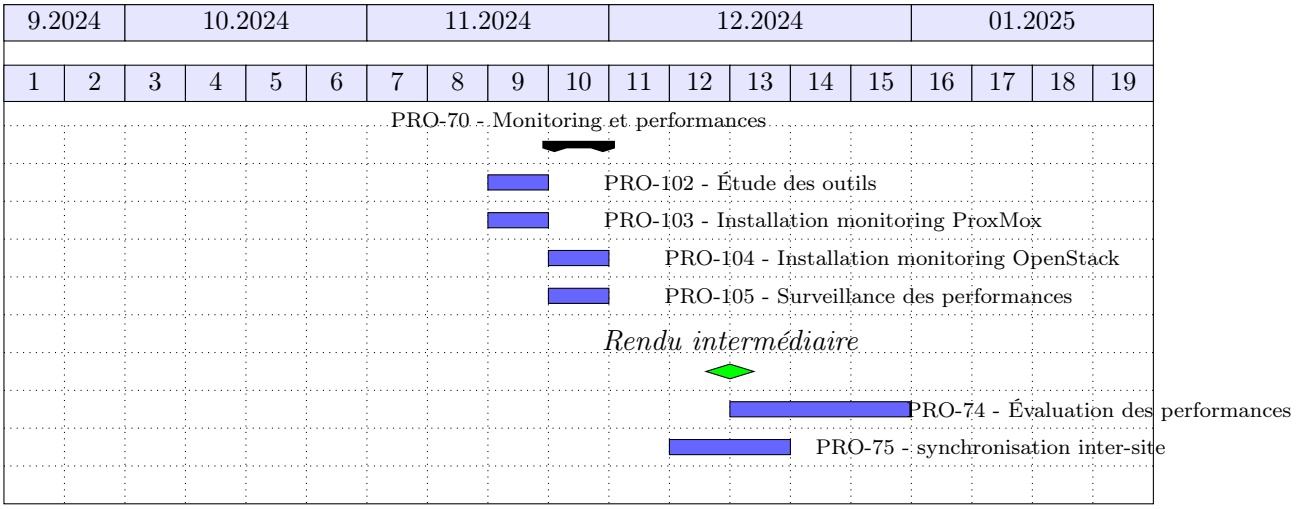
1. Le scénario doit être original ou innovant : il ne suffit pas de chercher à répliquer quelque chose d'existant, vous devez apporter une innovation
2. Votre solution doit reposer sur des objets connectés (par exemple avec les Arduino ou les Raspberry) et incorporer des données du monde physique
3. Le volume de travail doit comprendre entre 30 et 50 % (en temps) de développement logiciel
4. Votre solution doit gérer un flux de données très important et offrir des temps de réponse garantis pour des volumétries données afin d'offrir une qualité d'expérience incomparable
5. Votre solution doit être tolérante à la charge et permettre un redimensionnement le plus automatique possible pour répondre à un afflux soudain de clients
6. L'hébergement doit être réalisé sur un cloud privé, réparti sur au moins deux sites
7. Votre solution doit être tolérante aux pannes d'un site ou d'un composant matériel ou logiciel
8. Votre solution doit comprendre une application web et/ou mobile
9. Votre solution doit être sécurisée avec une analyse des risques (méthode inspirée d'EBIOS, voire très simplifiée comparable à ce que vous avez vu en TD de sécurité)
10. Votre solution doit être supervisée
11. Votre solution doit permettre la communication double pile IPv4/IPv6 et doit également fonctionner en mode « IPv6 only »

Annexe B

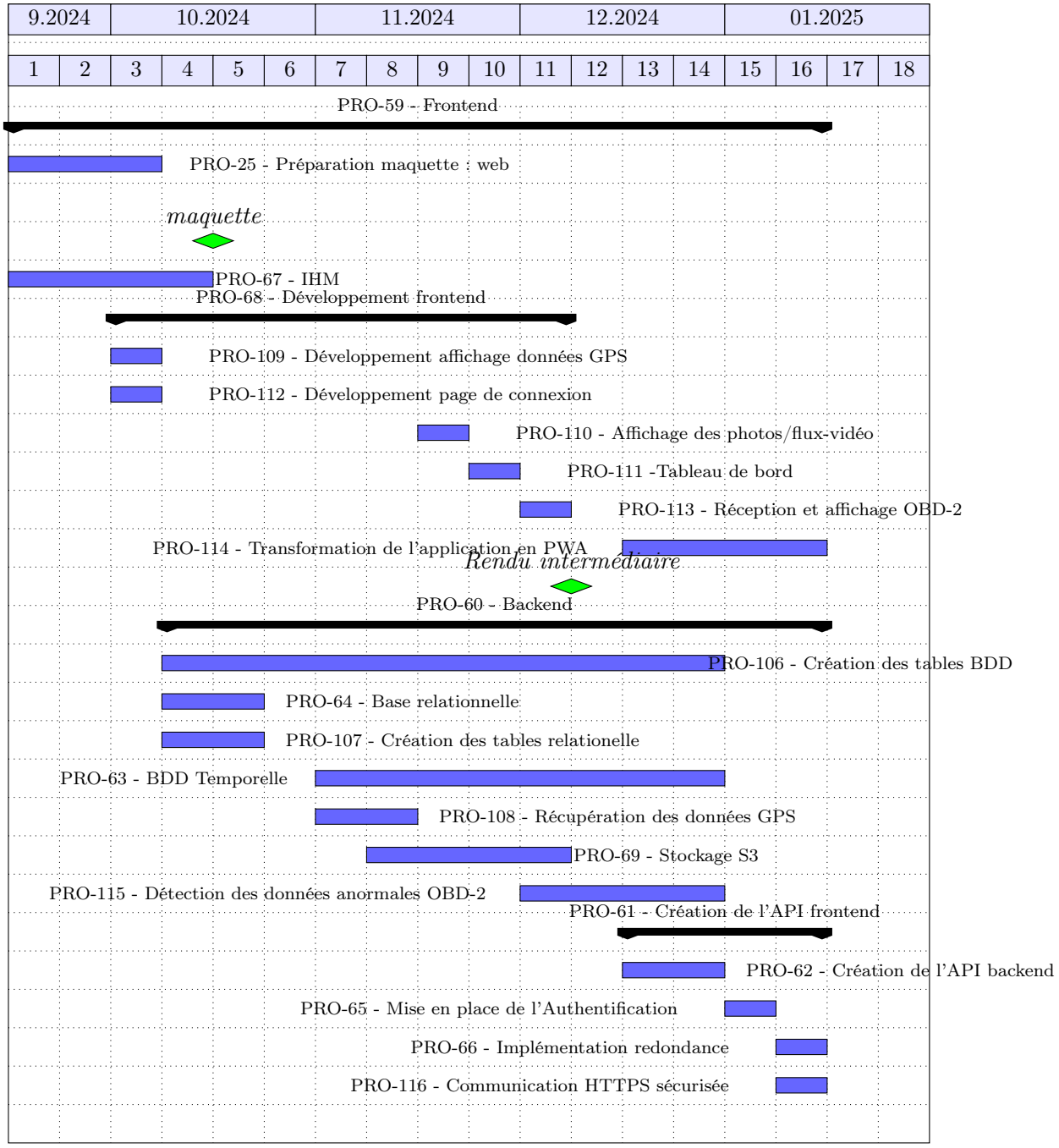
Gestion de projet

B.1 Infrastructure

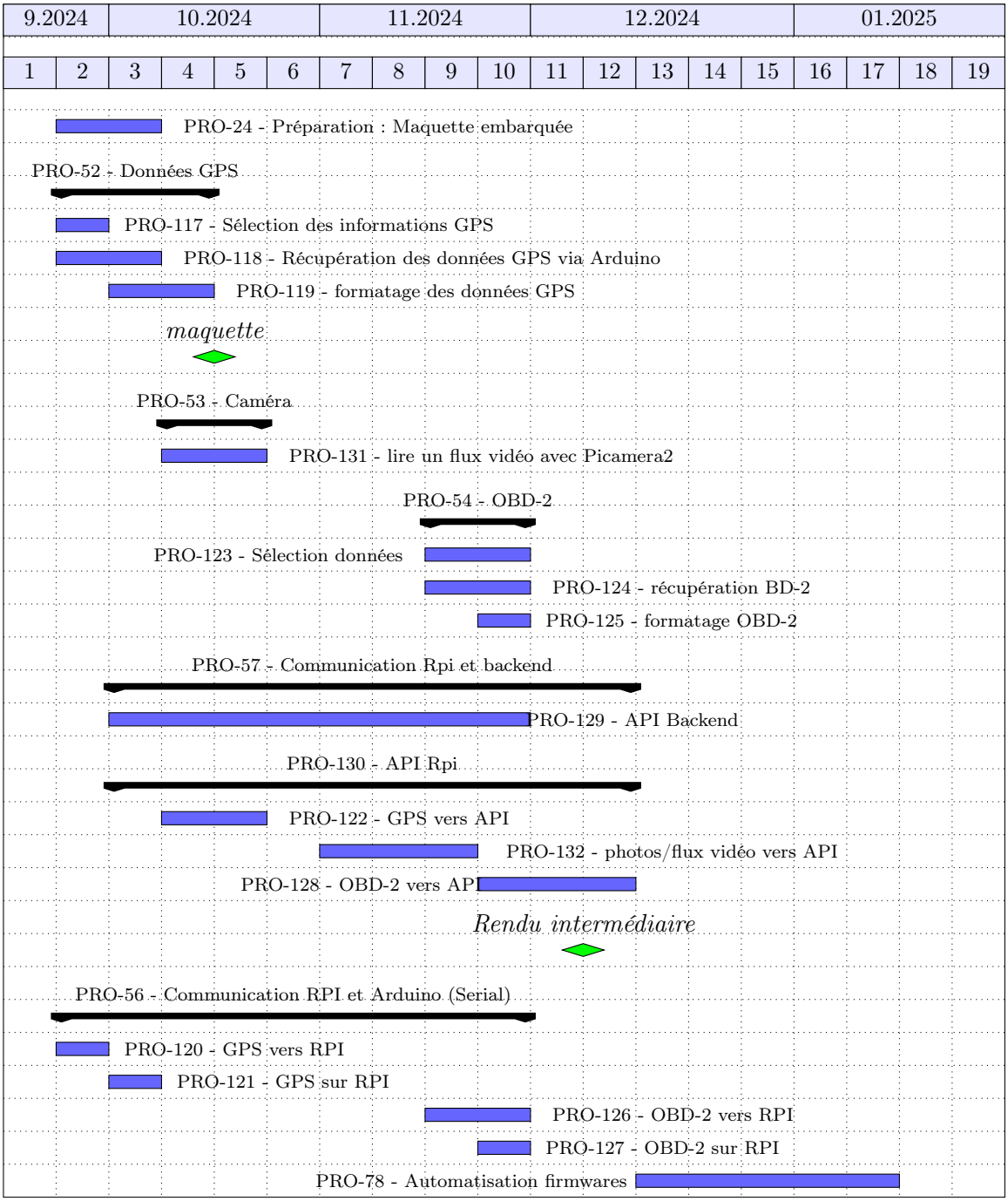



9.2024
10.2024
11.2024
12.2024
01.2025
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
PRO-70 - Monitoring et performances
PRO-102 - Étude des outils
PRO-103 - Installation monitoring ProxMox
PRO-104 - Installation monitoring OpenStack
PRO-105 - Surveillance des performances
Rendu intermédiaire
PRO-74 - Évaluation des performances
PRO-75 - synchronisation inter-site

B.2 Applicatif



B.3 Systèmes embarqués



Annexe C

Structure JSON

```
{
   "localtime":"local_time",
   "records":[
      {
         "time":"local_time",
         "data":{
            "sensors":[
               {
                  "sensor":"arduino",
                  "millis":"time_since_boot_up"
               },
               {
                  "sensor":"gps",
                  "satellites":"gps.satellites.value",
                  "location":[
                     "location_lat",
                     "location_lng"
                  ],
                  "altitude":"gps.meters.altitude",
                  "course":"gps.course.deg",
                  "speed":"gps.speed.kmph",
                  "datetime":"datetime"
               },
               {
                  "sensor":"obd2",
                  "engine_load":"engine_load_percent",
                  "coolant_temp":"coolant_temp_celsius",
                  "intake_pressure":"intake_pressure_kilopascal",
                  "rpm":"rpm",
                  "speed":"speed_kph",
                  "fuel_level":"fuel_level_percent",
                  "ambiant_air_temp":"ambiant_air_temp_celsius"
               }
            ]
         }...
      }
   ],
   "pictures":[
      {
         "car_id":"hashed_vin",
         "time":"local_time",
         "data":"base-64_picture"
      }...
   ]
}
```

Annexe D

Job React

```
job "react-prod" {
  datacenters = ["c315"]
  type = "service"

  group "react" {
    count = 1

    scaling {
      enabled = true
      min     = 1
      max     = 10

      policy {
        evaluation_interval = "5s"
        cooldown            = "1m"

        check "active_connections" {
          source = "prometheus"
          query = "avg(rate(container_cpu_usage_seconds_total{instance=~\"10..+\", name=~\"
              react.+\"}[1m])) * 100"

          strategy "target-value" {
            target = 15  # %CPU
          }
        }
      }
    }
...
    task "react-container" {
      driver = "docker"

      resources {
        cpu = 500
        memory = 700
      }
...
      service {
        name = "react"
        port = "front"
...
        }
      }
    }
  }
}
```

Annexe E

Cartographie de l'infrastructure

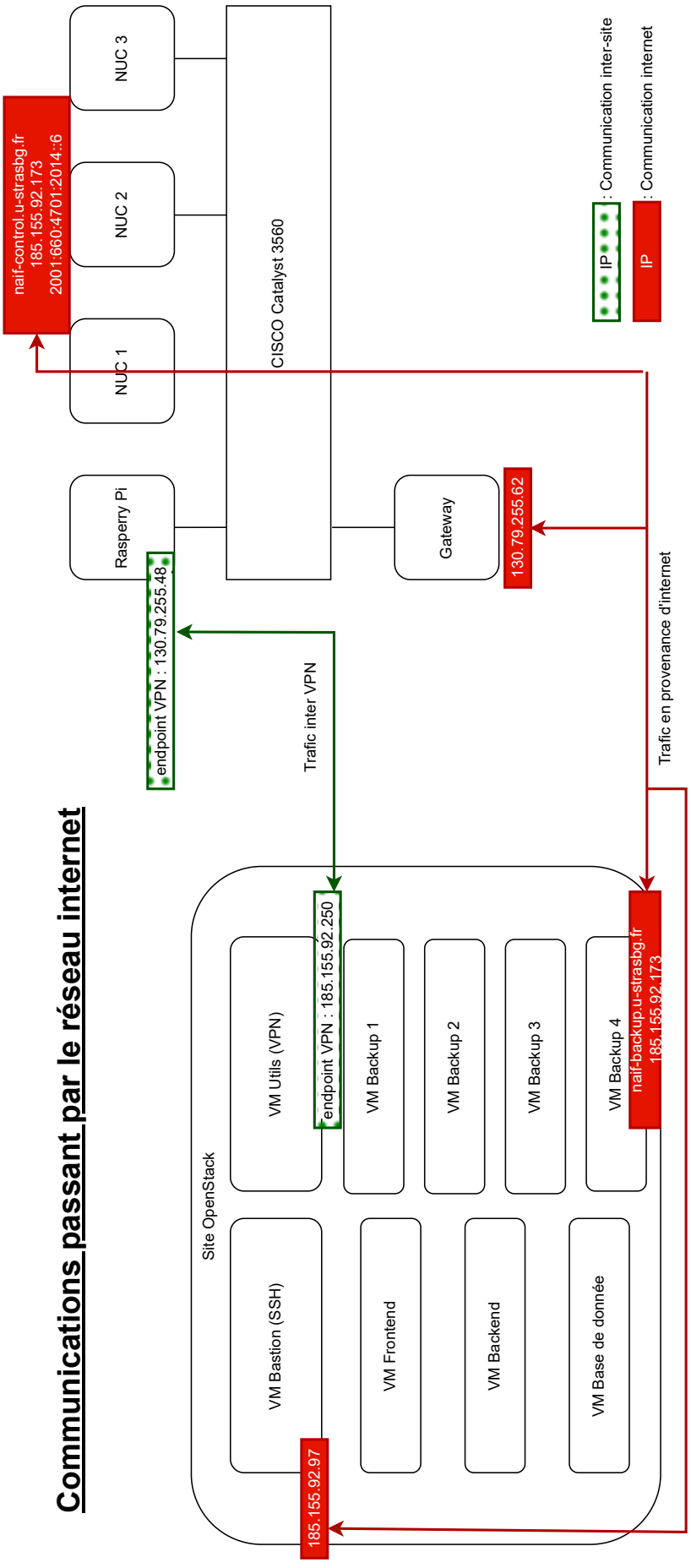


FIGURE E.1 – Cartographie des communications internet de l'infrastructure

Cartographie des services et des communications intra-site

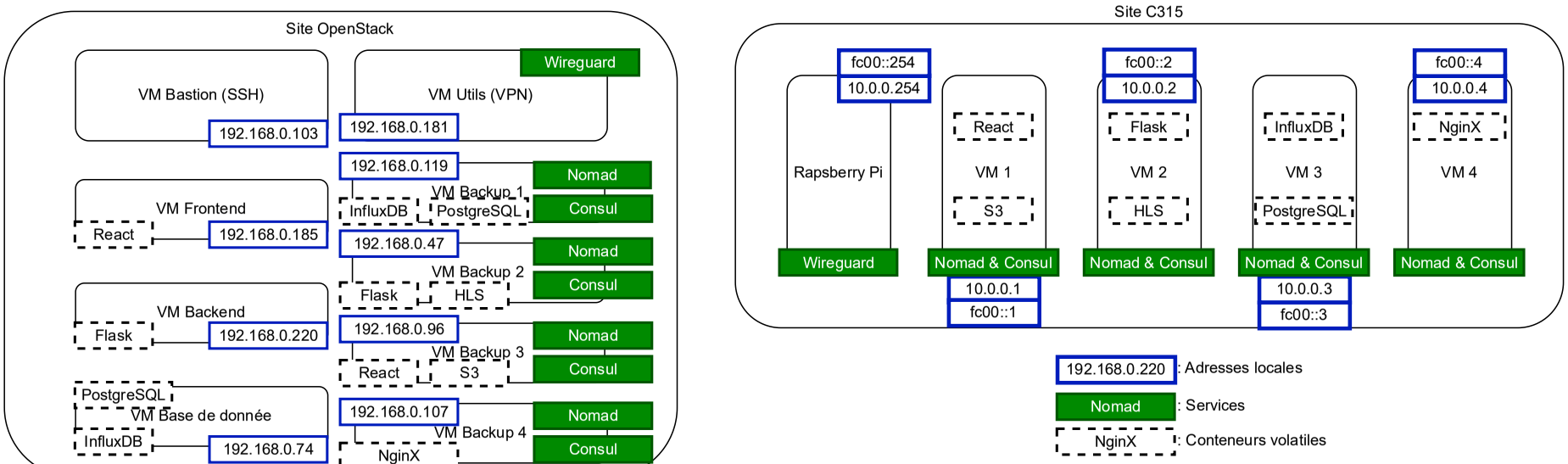


FIGURE E.2 – Schéma de l'architecture du réseau des VM du site C315