BORDEAUX INP Enseirb-Matmeca

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Cerema
CLIMAT & TERRITOIRES DE DEMAIN

Chaire
Mobilité & Transports Intelligents

Rapport PFA

Analyse et visualisation de données mobilités du Campus

Mohamed Taha SANDI
Mohamed Dyae CHELLAF
Mohamed Reda EL KHADER
Omaima CHOUAF
Chaimaa RADIOUSSE
Hassen AKROUT
Youssef MAHJOUB

Département Informatique

S8 – Année 2024/2025

Table des matières

1 Mise en contexte

1.1 Le besoin

Le développement des infrastructures universitaires et la croissance du nombre d'usagers sur le campus universitaire de Talence/Pessac rendent cruciale la compréhension fine des flux de mobilité. En particulier, il devient nécessaire de disposer d'outils d'analyse permettant d'identifier les pics de circulation, les comportements récurrents des usagers et les zones de congestion potentielles. Actuellement, bien que les capteurs de comptage installés sur le campus produisent une quantité importante de données, celles-ci restent souvent sous-exploitées faute de traitements adaptés. Il est donc indispensable de mettre en place un système permettant de transformer ces données brutes en visualisations claires, interactives et exploitables, dans le but de mieux comprendre, anticiper et gérer les déplacements sur le campus.

1.2 Les objectifs

Pour répondre à ces exigences, le projet commandité par le Centre d'études et d'expertise sur les risques, l'environnement, la mobilité et l'aménagement (***Cerema***), représenté par M. Royston FERNANDES en partenariat avec la chaire Mobilité et Transports Intelligents représentée par M. Mohamed MOSBAH, vise à développer une application interactive capable d'analyser et de visualiser les données de mobilité issues des capteurs de comptage présents sur le campus. Les objectifs spécifiques du projet sont :

- **Analyse des données de mobilité :** Intégrer, structurer et enrichir les données de comptage pour permettre l'extraction d'indicateurs pertinents, tels que les flux horaires, journaliers ou hebdomadaires.
- **Développement de visualisations graphiques :** Concevoir des représentations dynamiques et parlantes (histogrammes, courbes de tendance, cartes de chaleur) pour faciliter la compréhension des flux entrants et sortants.
- **Cartographie interactive des points d'entrée et de sortie :** Mettre en place une carte interactive permettant de localiser les capteurs et de visualiser les flux en fonction de la géographie, avec une possibilité d'exploration par périodes choisies ou en temps réel.
- **Interface utilisateur intuitive :** Développer une interface ergonomique permettant de filtrer, naviguer et visualiser les données avec simplicité et efficacité.

1.3 Les résultats

À l'issue du projet, les résultats obtenus ont globalement répondu aux attentes exprimées en début de mission. L'application développée permet désormais de visualiser et d'explorer de manière interactive les données de comptage de véhicules issues des capteurs du campus. L'utilisateur peut observer les flux selon différentes échelles temporelles, comparer les tendances entre points d'entrée et de sortie, et bénéficier d'une interface claire facilitant l'analyse. L'intégration d'une carte interactive enrichit l'expérience en localisant précisément les capteurs et les zones d'affluence. Ce résultat constitue une première étape concrète vers une meilleure compréhension des dynamiques de mobilité sur le campus.

2 Organisation

Pour garantir le bon déroulement de notre projet, nous avons adopté une organisation claire et efficace, combinant les principes de l'agilité avec une gestion bien pensée des ressources humaines. Dans cette section, nous présentons les méthodes que nous avons mises en place, en nous concentrant sur deux aspects essentiels : l'approche Agile et la gestion de la main-d'œuvre.

2.1 Agilité

Au cours de la réalisation du projet, nous avons adopté une approche de gestion **agile** afin de structurer efficacement notre travail, d'encourager la communication au sein de l'équipe, et de garantir un suivi régulier de l'avancement. L'agilité repose sur des itérations courtes, qui permettent d'incrémenter le produit à chaque cycle tout en intégrant les retours du client.

Notre projet s'est ainsi organisé en **sprints hebdomadaires**, à l'issue desquels une réunion était systématiquement tenue en présence du client et de notre superviseur pédagogique, M. Mohamed MOSBAH. Ces réunions suivaient un format structuré :

- **Rétrospective du sprint précédent :** Présentation des tâches accomplies, démonstration des fonctionnalités développées, et évaluation des objectifs atteints.
- **Amélioration continue :** Discussion collective sur les difficultés rencontrées, échanges sur la qualité du code, l'organisation du travail ou la communication au sein de l'équipe.
- **Planification du sprint suivant :** Définition des prochaines priorités techniques et fonctionnelles, en lien avec les objectifs du projet et les retours du client.

Ce cadre nous a permis d'assurer une progression incrémentale du produit, de mieux gérer les imprévus, et de maintenir une implication active du client tout au long du développement. En favorisant la réactivité et la transparence, cette méthode a grandement contribué à la réussite du projet.

2.2 Gestion de la main d'œuvre

Dans le but d'optimiser l'utilisation des ressources humaines disponibles pour ce projet, constitué de sept étudiants, nous avons réparti l'équipe en deux groupes distincts :

- **Équipe *Front-End* :** Spécialisée dans le développement de l'interface
- **Équipe *Back-End* :** En charge de l'intégration de l'algorithme de détection d'objets sur les vidéos, de l'extension du modèle pour inclure la classe des trottinettes, ainsi que du développement de l'application `FastAPI`.

Dès le lancement du projet, un membre de l'équipe a été désigné pour prendre les notes lors des réunions hebdomadaires. Chaque compte rendu, rédigé après la réunion, était stocké dans un dossier Google Drive partagé afin d'assurer un bon suivi des tâches, des décisions prises, et de garantir une traçabilité accessible à tous.

3 Phase de la conception

3.1 Analyse de la problématique

Avant de commencer la conception du *Backend*, notre priorité était de bien cerner les besoins du client. Dans ce cadre, le CEREMA a lancé — et poursuit encore — une vaste collecte de données sur la présence des véhicules dans différentes zones du campus. Ces données sont issues de fichiers CSV générés par des capteurs radars, ainsi que de vidéos enregistrées par plusieurs caméras installées sur le site.

Deux besoins principaux en découlent. Le premier est le développement d'une interface intuitive permettant de visualiser ces données, d'identifier des tendances, et ainsi d'envisager des mesures correctives pour optimiser les flux de circulation. Le second objectif consiste à exploiter les vidéos afin de détecter les véhicules, ce qui permettrait de générer des données dans les zones équipées uniquement de caméras. Dans les zones disposant à la fois de capteurs et de caméras, une comparaison entre les données issues des deux sources pourrait être effectuée afin de valider leur fiabilité.

3.2 Frontend

Lors de cette étape, nous avons débuté par une phase de réflexion créative, où chaque membre de l'équipe a exprimé sa vision du produit final. Cette première phase a été suivie d'un *brainstorming* collectif, au cours duquel nous avons lancé un projet collaboratif sur la plateforme `Figma`. Chaque membre a ainsi pu apporter ses modifications sur différentes versions des maquettes. Après plusieurs itérations, nous avons abouti à une maquette finale, qui servira de référence tout au long de la phase de réalisation.

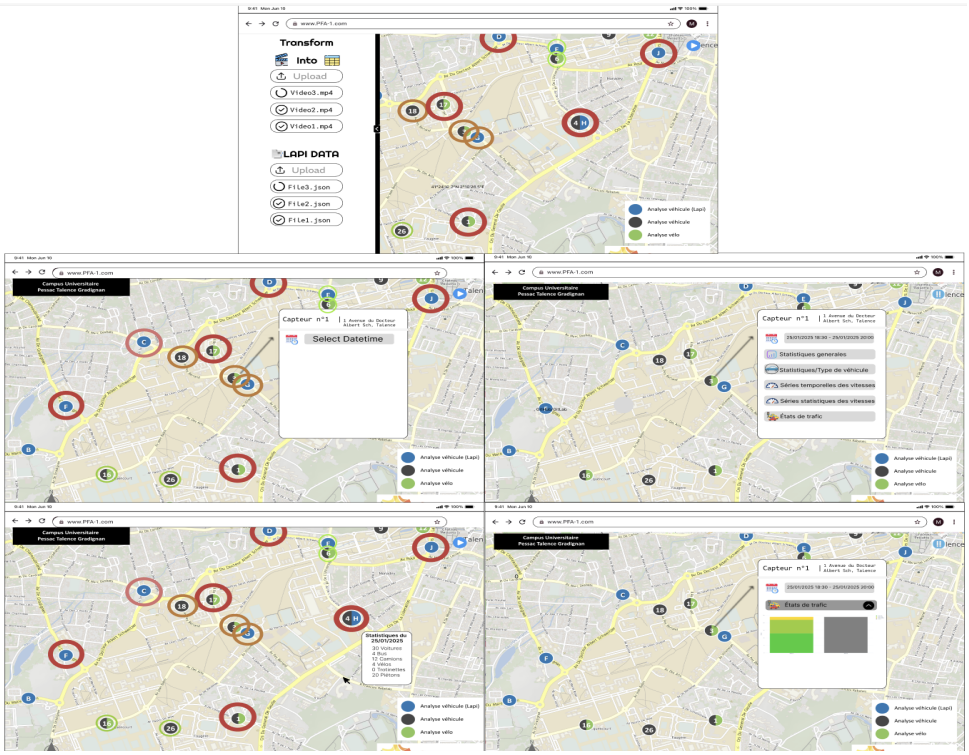


FIGURE 1 – Maquettes

PS : Il est important de noter que ces maquettes ne seront pas nécessairement suivies à la lettre, car au fil de l'avancement du projet, certains éléments pourront être ajustés afin de mieux répondre aux attentes et aux préférences du client.

3.3 Backend

En ce qui concerne le *Backend*, nous avons tenu une réunion d'équipe afin de définir les différents flux de traitement à mettre en place dans l'application. Le premier flux concerne le traitement des vidéos captées par les caméras installées sur le campus. Ces vidéos sont d'abord décomposées en une série d'images (*frames*), qui sont ensuite analysées par un modèle de détection d'objets. Ce modèle permet de générer des fichiers CSV, similaires à ceux produits par les capteurs.

Le second flux porte sur la gestion de la partie serveur de l'application web, ainsi que sur les interactions entre l'utilisateur et le serveur, notamment pour l'envoi, la récupération et la visualisation des données.

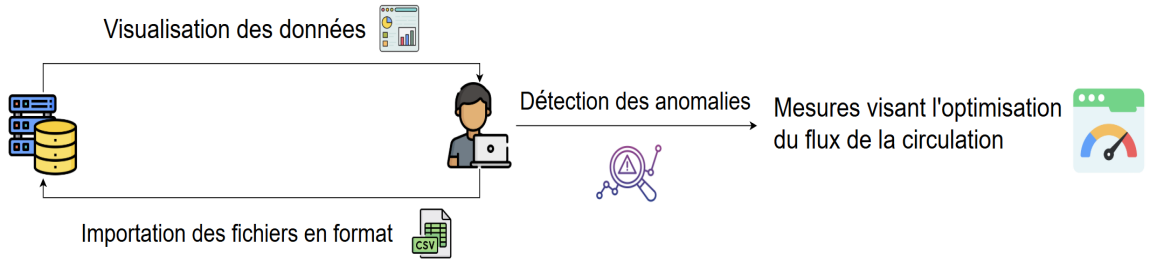


FIGURE 2 – *Workflow* global

En ce qui concerne le stockage des données, celui-ci s'effectuera à l'aide de deux tables, représentées de la manière suivante :

SENSORS		
id	integer	primary key
coordinates	STRING	
type	STRING	
name	STRING	
min_date	DATETIME	
max_date	DATETIME	
←data		

FIGURE 3 – Table des capteurs

DATA		
id	integer	primary key
sensor_id	INTEGER	
datetime	DATETIME	
type	STRING	
direction	STRING	
←sensors		

FIGURE 4 – Table des données importées

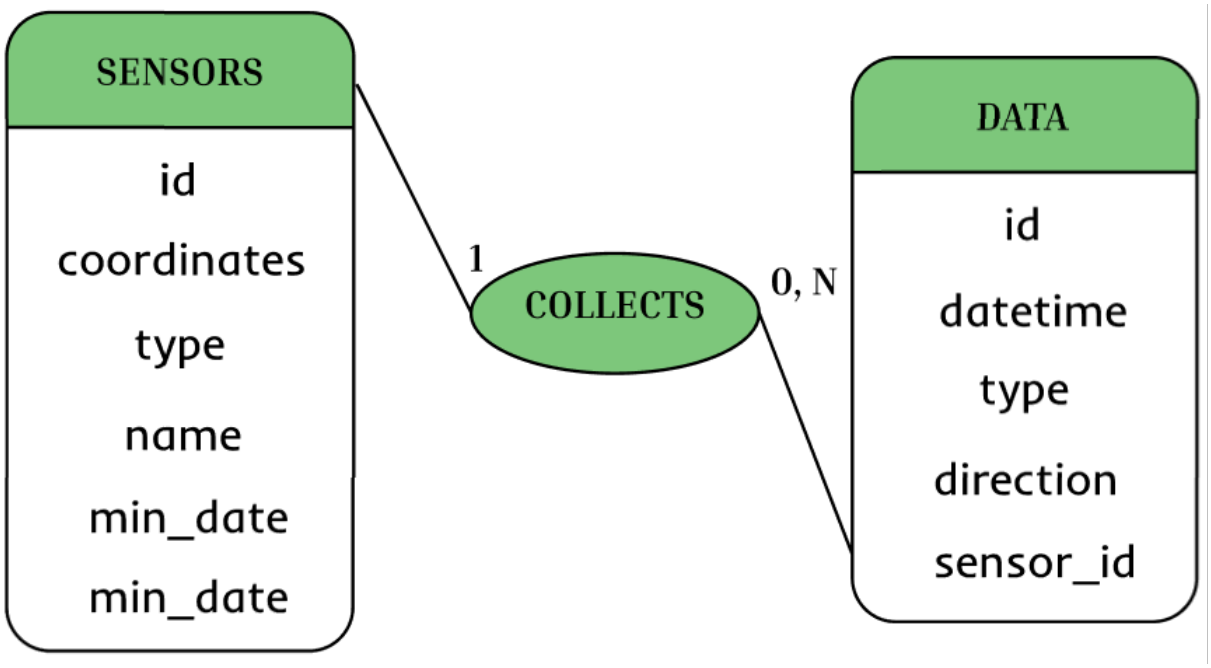


FIGURE 5 – Schéma Entité-Association

La base de données modélise un système de capteurs à travers deux tables principales, ***sensors*** et ***data***, reliées par une relation **un-à-plusieurs**. La table *sensors* représente chaque capteur installé, identifié par un `id` unique, avec ses coordonnées géographiques (`coordinates`), son type (`type`), son nom (`name`), ainsi que les dates limites d'activité (`min_date` et `max_date`). Chaque capteur peut être associé à plusieurs enregistrements de données grâce à la relation *data*, qui assure également la suppression en cascade des données si le capteur est supprimé. La table *data*, quant à elle, enregistre les mesures ou événements captés par les capteurs. Chaque enregistrement possède un `id`, une référence au capteur via `sensor_id`, une date et heure de mesure (`datetime`), un type (`type`), et une direction (`direction`).

3.4 Stack technique

Le projet repose sur une architecture technique moderne et cohérente, choisie stratégiquement par notre groupe afin d'assurer performance, fiabilité et évolutivité de l'application.

Pour le Frontend, nous avons opté pour **`Next.js`**, un framework basé sur **`React`**(voir définition), offrant les avantages de **`React`** en termes de composants dynamiques tout en facilitant le rendu côté serveur (SSR) et l'optimisation du référencement et des performances. La carte interactive a été intégrée à l'interface grâce à la bibliothèque **`Leaflet`**(voir définition), en s'appuyant sur les tuiles ouvertes de **`OpenStreetMap`** pour l'affichage des données géographiques. `D3.js`(voir définition) a été utilisée pour générer les graphes.

Côté Backend, le groupe a choisi d'utiliser **`Python`** avec **`FastAPI`**(voir définition), un framework moderne permettant de développer des API rapides, asynchrones et bien documentées. **`FastAPI`** assure la communication entre le Frontend et le Backend via des échanges **`HTTP`** efficaces, minimisant la latence tout en garantissant une grande scalabilité.

Pour l'analyse visuelle, nous avons intégré le modèle **`YOLO`**(voir définition), qui sert à détecter et classifier les véhicules présents sur les frames extraites des vidéos fournies

par le CEREMA. Ce traitement automatisé permet de générer rapidement des fichiers de comptage fiables.

Enfin, les données relatives aux capteurs et aux flux de véhicules sont stockées de manière pérenne dans une base **PostgreSQL**(voir définition), choisie pour sa robustesse, sa capacité à traiter des volumes importants de données, et sa compatibilité avec les architectures web modernes.

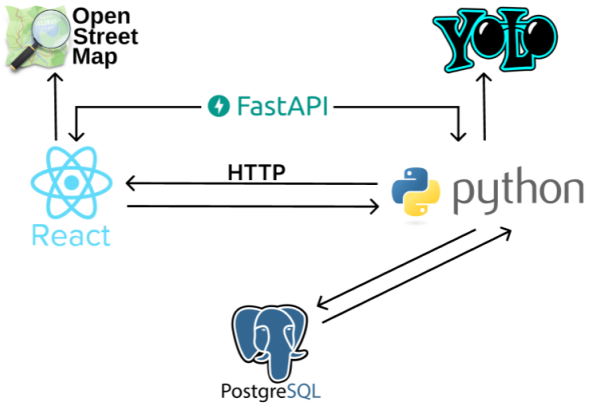


FIGURE 6 – Architecture technique du projet

4 Phase de la réalisation

4.1 Chronologie

Le déroulement du projet a été découpé en plusieurs phases clés, chacune correspondant à un ensemble d'objectifs techniques spécifiques, en appliquant les principes du développement agile. Chaque phase était planifiée sur une durée estimative, mais l'avancement réel a parfois différé selon la complexité rencontrée et la dynamique de l'équipe. Voici un récapitulatif détaillé :

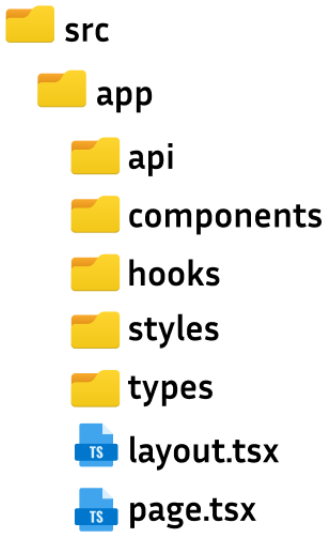
- **Phase 1 – Préparation initiale**
  Cette première phase portait sur la préparation du code YOLO destiné au traitement des vidéos fournies par le CEREMA, du côté Backend, ainsi que sur l'intégration de la carte interactive et l'implémentation des premiers composants du Frontend. Initialement estimée à **trois semaines**, cette phase a nécessité finalement **un mois** complet, en raison de la complexité d'adaptation des environnements techniques et de l'intégration des premières briques de l'interface utilisateur.
- **Phase 2 – Fine-Tuning et structuration de l'interface**
  Cette étape a concerné le *fine-tuning* des modèles YOLO sur les vidéos du CEREMA, l'amélioration des composants frontend existants, ainsi que l'implémentation concrète de la structure à deux modes (voir section 4.1.2). Durant cette phase, nous avons également développé les fonctionnalités avancées de manipulation des capteurs : déplacement, ajout, suppression et interaction contextuelle avec la carte. Bien que cette phase était initialement prévue pour **un mois ou plus**, elle a été réalisée en **trois semaines**, traduisant une progression rapide sur ces aspects critiques.

- **Phase 3 – Connexion Frontend-Backend-Base de données**
  Cette phase a consisté à établir la communication complète entre le frontend, le backend et la base de données, via l'écriture des routes API en FastAPI, ainsi que l'amélioration de la logique applicative côté client. La correction des premiers bugs liés aux échanges de données en temps réel a également été abordée. Cette phase a été finalisée en **trois semaines**.
- **Phase 4 – Finalisation, perfectionnement et tests**
  La dernière phase a porté sur l'exploitation complète des modèles pour traiter le reste des vidéos, comparer les résultats aux hypothèses du CEREMA, mais également finaliser et perfectionner l'interface frontend. Cela incluait l'ajout des dernières fonctionnalités demandées, ainsi qu'une vérification poussée de la stabilité, de l'ergonomie et de l'esthétique de l'application. Estimée à **trois semaines**, cette étape a été achevée dans les temps.

Grâce à cette organisation efficace, le projet a été entièrement finalisé avec une avance confortable dans les **dix derniers jours du mois d'avril**. Cette période restante a été consacrée exclusivement à des rectifications mineures, à l'ajustement de détails visuels ou fonctionnels, et à une poursuite du perfectionnement, garantissant ainsi un produit final à la fois robuste, soigné et aligné avec les attentes initiales.

4.2 Frontend

4.2.1 Architecture



- **api/ :** Récupère les données de la *database*.
- **components/ :** Contient les composants de l'interface.
- **hooks/ :** Intégrer les éléments de l'API avec *ReactQuerry*.
- **styles/ :** Définit les styles des composants.
- **types/ :** Définit l'interface des capteurs.
- ***layout.tsx* :** Englobe le code dans *ReactQuerry*.
- ***page.tsx* :** Fichier principal de l'interface.

4.2.2 Composants

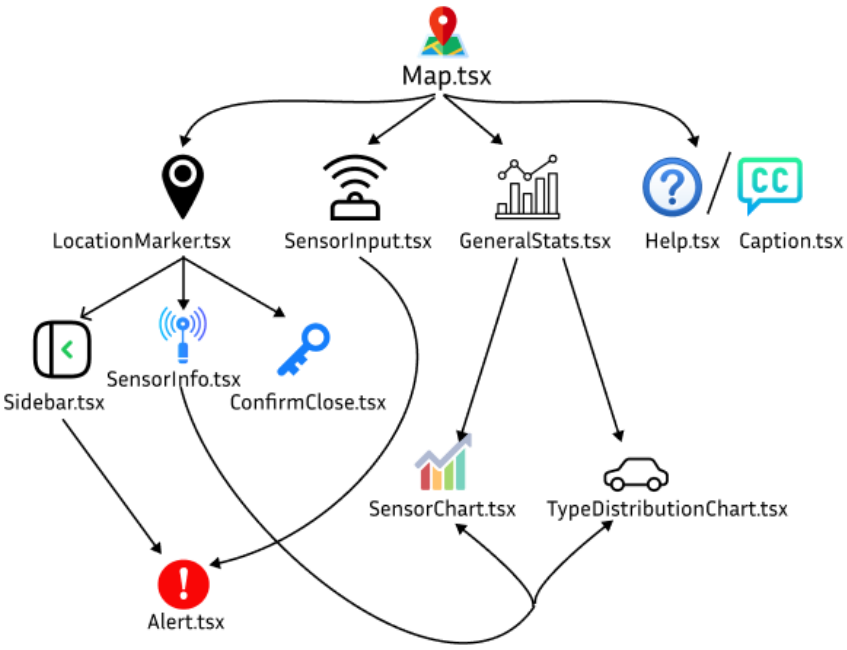


FIGURE 7 – Arbre des composants du Frontend

Le *frontend* de l'application repose sur une **structure modulaire en composants**, conforme aux bonnes pratiques de développement avec `React` et `Next.js`. Chaque fonctionnalité est encapsulée dans des fichiers spécialisés, permettant une lisibilité accrue, une réutilisabilité du code et une maintenance facilitée.

Le composant principal, **`Map.tsx`**, centralise l'affichage de la carte interactive. Il orchestre le comportement de plusieurs sous-composants essentiels :

- **`LocationMarker.tsx`** : gère le marquage des capteurs sur la carte et intègre à son tour :
  - **`Sidebar.tsx`** : interface latérale permettant l'import et la manipulation des données,
  - **`SensorInfo.tsx`** : affichage détaillé d'un capteur sélectionné,
  - **`ConfirmClose.tsx`** : fenêtre de confirmation lors de certaines actions sensibles.
- **`SensorInput.tsx`** : formulaire d'ajout de capteur, avec gestion des types, noms et coordonnées.
- **`GeneralStats.tsx`** : composant affichant les statistiques globales du campus, intégrant deux sous-composants :
  - **`SensorChart.tsx`** : graphique temporel représentant l'évolution des flux,
  - **`TypeDistributionChart.tsx`** : histogramme de répartition des types de véhicules.
- **`Help.tsx`** : composant dédié à l'affichage d'un mode d'emploi succinct de l'interface.
- **`Caption.tsx`** : composant de légende dynamique des capteurs affichés.

Il est également à noter que les composants `SensorInput.tsx` et `Sidebar.tsx` utilisent tous les deux le composant `Alert.tsx`, qui permet de notifier l'utilisateur d'une erreur ou d'un succès lors des actions critiques (ajout, suppression, importation de fichier, etc.).

Cette architecture hiérarchique et modulaire offre une base solide pour le développement, l'évolutivité et la maintenance du projet, en assurant une **séparation claire des responsabilités** entre composants.

4.2.3 Implémentation

La première étape basique était d'intégrer la carte interactive. Pour ce faire, nous avons utilisé la bibliothèque `JavaScript Leaflet`, largement reconnue pour sa légèreté et sa facilité d'utilisation dans les applications web. Couplée aux tuiles cartographiques fournies par `OpenStreetMap`, cette solution nous a permis d'afficher une carte fluide, libre d'utilisation, et parfaitement adaptée à la visualisation des capteurs de trafic sur le campus. Leaflet offre également de nombreuses fonctionnalités natives, comme l'ajout de marqueurs, de popups, ou encore le contrôle du zoom, que nous avons exploitées pour enrichir l'expérience utilisateur.

Quatre types de capteurs ont été définis pour représenter les différents dispositifs de comptage : les **caméras normales**, les **caméras LAPI**, les **capteurs à vélo** et les **capteurs tubes**. Chaque capteur est positionné sur la carte selon ses coordonnées géographiques, avec une couleur ou une forme distincte en fonction de son type. Lorsqu'un nouveau capteur est ajouté à l'interface, sa légende correspondante est automatiquement générée et intégrée à l'interface, assurant ainsi une visualisation claire et cohérente de l'ensemble du système de comptage.

L'interface a été conçue pour fonctionner selon deux modes distincts, controllées par un bouton *switch* : un **mode récepteur**, permettant d'insérer les données des capteurs en format CSV, et un **mode simulateur**, permettant de générer et visualiser les données statistiquement pour chaque capteur ainsi que pour toute la zone du campus.

a) Mode Récepteur :

Lorsque le mode récepteur est activé, l'application est configurée pour recevoir et gérer des données importées par l'utilisateur. Plusieurs fonctionnalités sont alors accessibles via l'interface :

- **Ajout d'un capteur :** L'utilisateur peut insérer un nouveau capteur en renseignant son nom, son type (LAPI, normal, à vélo, ou tube), ainsi que ses coordonnées géographiques.
- **Suppression d'un capteur :** Un capteur peut être supprimé de la carte de manière simple et rapide via un bouton dédié.
- **Déplacement manuel d'un capteur :** Chaque capteur peut être déplacé manuellement à l'aide d'une interaction directe sur la carte.
- **Ajout et traitement des données :** Une barre latérale (*Sidebar*) est associée à chaque capteur, permettant d'importer des fichiers CSV locaux contenant les

données d'un capteur. Une fois le fichier importé, les données sont traitées côté serveur, puis associées et enregistrées dans le capteur sélectionné.

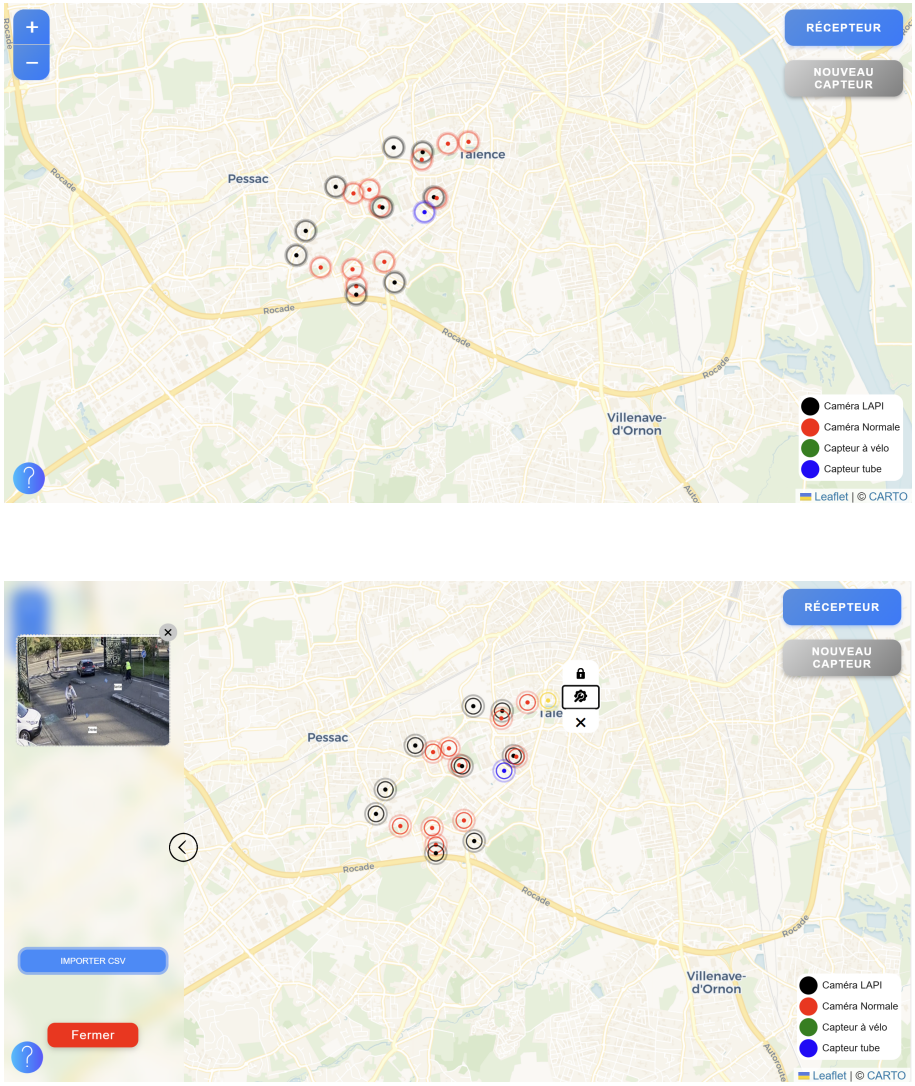


FIGURE 8 – Fonctionnalités du mode récepteur

b) Mode Simulateur :

Le mode simulateur permet à l'utilisateur d'exploiter les données enregistrées dans les capteurs. En cliquant sur un capteur, il peut sélectionner un intervalle de temps valide (correspondant aux périodes pour lesquelles des données sont disponibles) ainsi qu'une période d'affichage (exprimée en minutes). L'application génère alors des analyses spécifiques à l'intervalle choisi, sous forme de séries temporelles et de distributions de types de véhicules.

L'utilisateur peut également consulter des statistiques globales pour l'ensemble du campus, selon les mêmes modalités de visualisation. Enfin, une carte de chaleur permet d'identifier rapidement les zones de forte affluence pour l'intervalle défini.

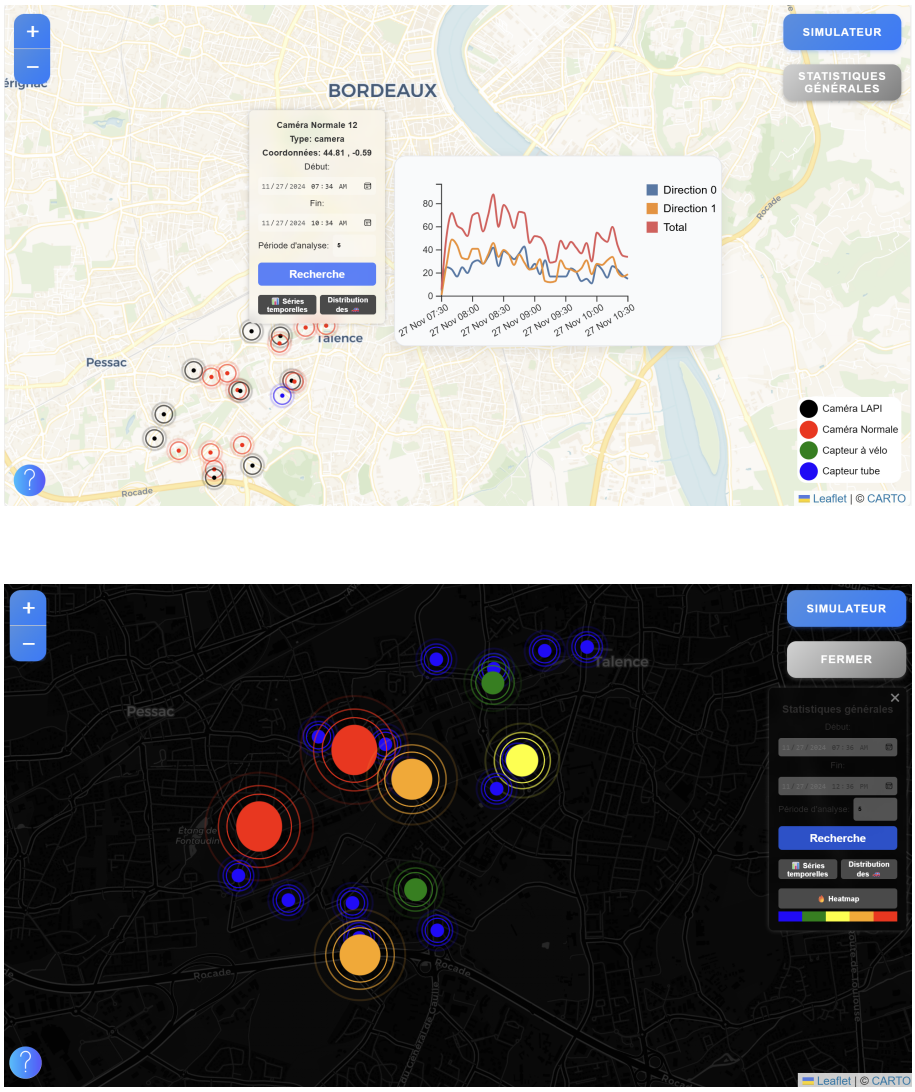


FIGURE 9 – Fonctionnalités du mode simulateur

Pour accompagner l'utilisateur dans la prise en main de l'application, un bouton **« Help »** a été intégré à l'interface. Il permet d'afficher à tout moment un mode d'emploi synthétique, présentant les fonctionnalités principales des deux modes d'utilisation de l'application.

4.3 Backend

4.3.1 Traitement des vidéos

En ce qui concerne le traitement des vidéos, cette étape a été réalisée en implémentant une classe nommée `YoloVideoProcessor`, intégrant le modèle de détection d'objets open-source développé par **Ultralytics**. Nous avons utilisé sa **version 11**, ainsi que le **modèle le plus volumineux** de cette série, doté de **56,9 millions de paramètres**. Ce modèle de grande capacité nous a permis d'approcher le plus fidèlement possible le **nombre réel de véhicules** présents sur chaque capteur, grâce à sa **précision élevée** dans l'identification des objets.

Le traitement des vidéos se déroule de manière **séquentielle** : la vidéo est parcourue *image par image*, et le modèle `YOLO` est appliqué sur chacune d'entre elles pour extraire les différentes classes sélectionnées, à savoir : *person* (personne), *bicycle* (bicyclette), *car* (voiture), *motorcycle* (moto), *bus* (bus) et *truck* (camion).

Comme le modèle détecte les objets à chaque *frame*, un défi majeur apparaît : il est essentiel de ne pas **recompter le même véhicule** plusieurs fois lorsqu'il est détecté sur des images consécutives. Pour répondre à ce besoin, nous avons utilisé l'algorithme de suivi léger et ultra-rapide ***ByteTrack***.

ByteTrack est un algorithme de suivi multi-cibles qui se distingue par sa capacité à exploiter à la fois les objets détectés avec **haute confiance** et ceux détectés avec une **confiance plus faible** (*low-confidence detections*), généralement ignorés par les méthodes classiques. Contrairement aux approches traditionnelles qui ne considèrent que les détections sûres, *ByteTrack* étend l'association en tenant compte également des objets faiblement détectés, ce qui permet de **maintenir le suivi** même en cas de perte temporaire ou de variations rapides.

Cette approche améliore significativement la **robustesse du suivi**, en particulier dans les scènes complexes et denses. Cependant, bien que *ByteTrack* soit extrêmement performant, il n'est pas infaillible : certaines **erreurs de comptage** peuvent subsister, notamment lors de *chevauchements* d'objets ou de *changements brusques de trajectoire*.

Pour pallier ces limitations et affiner le comptage, nous appliquons un **traitement statistique** sur les données collectées. Ce processus repose sur une **analyse de la fréquence d'apparition** des objets dans les différentes *frames*. Pour chaque type d'objet détecté, nous regroupons les données par **identifiant unique** et comptabilisons le **nombre de frames distincts** dans lesquels il apparaît.

Une **normalisation statistique**, sous forme de **score-z**, est ensuite calculée pour mesurer l'écart de chaque objet par rapport à la moyenne. Ce score permet d'identifier les comportements **anormaux**, qu'il s'agisse d'objets apparaissant de manière **trop sporadique** ou, à l'inverse, **trop fréquente**. Un **seuil** est alors appliqué (par défaut entre `-0.1` et `3`) pour ne conserver que les objets considérés comme fiables.

Ceux qui sortent de cette plage sont traités comme des **anomalies** ou des **artefacts**, et sont écartés. Parmi les objets restants, seules les **dernières apparitions** (les plus récentes dans le temps) sont conservées, ce qui permet d'éliminer les doublons. Cette méthode assure un **comptage plus fidèle et cohérent** des véhicules, en se concentrant uniquement sur ceux qui présentent une **présence stable et significative** dans les données vidéo.

4.3.2 Extension des classes détectées aux trottinettes

Dans cette étape, nous avons cherché à étendre les capacités de détection de notre modèle `YOLO` à une nouvelle classe d'objets : les trottinettes. Pour cela, nous avons d'abord constitué un jeu de données conséquent en collectant et annotant un grand nombre d'images contenant des trottinettes. Après application de techniques d'augmentation de données (rotation, changement d'échelle, bruit, etc.), nous avons obtenu un ensemble d'environ **7000 images** annotées.

Nous avons ensuite utilisé un modèle `YOLO` préexistant structuré en 23 modules, où le 23^{e} et dernier module correspond au head de détection, responsable de la prédiction des boîtes englobantes et des classes. Afin de spécialiser le modèle sur la classe « trottinette » tout en minimisant le temps d'entraînement, nous avons adopté une stratégie de fine-

tuning partiel : les modules 0 à 22 (c'est-à-dire l'extracteur de caractéristiques) ont été gelés, et seul le 23^{e} module a été entraîné sur notre jeu de données spécifique.

Une fois ce module ajusté, nous l'avons intégré au modèle initial. Le résultat final correspond donc à la concaténation du module d'origine avec ce nouveau module spécialisé, ce qui permet au modèle d'identifier les trottinettes en plus des classes initialement apprises, tout en conservant une bonne généralisation et un temps d'apprentissage réduit.

Cette approche a été mise en œuvre avec succès en dépit de ressources matérielles limitées, ce qui témoigne de l'efficacité et de la légèreté de la méthode employée.

4.3.3 Format des fichiers CSV

L'application utilise des fichiers `CSV` pour représenter les flux de véhicules détectés ou simulés. Ces fichiers sont exploités à deux niveaux :

- Ils sont générés automatiquement dans le **backend** à partir des vidéos analysées (via les modèles de détection) ;
- Ils peuvent être importés manuellement via l'**interface utilisateur** en mode récepteur, à partir du panneau latéral (Sidebar) d'un capteur.

Structure attendue : Chaque fichier CSV doit contenir exactement trois colonnes, avec les intitulés suivants :

- `horodatage` : la date et l'heure de l'événement, au format `AAAA-MM-JJ HH:MM:SS` ;
- `type` : le type de véhicule détecté (exemples : `voiture`, `bus`, `vélo`, etc.) ;
- `direction` : la direction de déplacement, indiquée soit en texte libre (ex. : `Est`, `Ouest`), soit en valeur numérique (ex. : `0`, `1`).

Exemple de fichier CSV valide

```
horodatage,type,direction
2025-04-18 13:42:00,voiture,1
2025-04-18 13:42:05,camion,1
2025-04-18 13:42:10,bicyclette,0
```

Remarque : lors d'un import via l'interface, les données sont automatiquement associées au capteur sélectionné et stockées dans la base de données après vérification et validation. L'ordre et le nom des colonnes doivent être strictement respectés pour garantir un traitement correct.

4.3.4 Structure et rôle du fichier `database.py`

Le fichier **`database.py`** constitue la pierre angulaire de l'architecture backend du projet. Il est responsable de la gestion de la base de données relationnelle `PostgreSQL` à l'aide de `SQLAlchemy`, une bibliothèque `ORM` (*Object-Relational Mapping*) en `Python`. Ce module centralise la définition des modèles de données, la configuration de la connexion à la base, ainsi que l'initialisation automatique des tables et des données de capteurs.

Connexion à la base de données : La fonction `get_engine` encapsule la création du moteur `SQLAlchemy` à partir des identifiants de connexion (utilisateur, mot de passe, hôte, port, nom de la base). Ces paramètres sont extraits d'un fichier de configuration externe, `config.json`. Ce fichier permet de stocker de manière centralisée et modifiable les informations liées à la base de données.

La fonction **`get_engine`** vérifie ensuite l'existence de la base de données avec **`database_exists()`** et la crée si nécessaire à l'aide de **`create_database()`**. Le moteur `SQLAlchemy` est alors utilisé pour créer un objet **`SessionLocal`** qui gère les transactions `SQL`.

Modélisation des entités : Le fichier définit deux classes principales correspondant aux tables de la base :

- **`Sensor`** : représente un capteur installé sur le territoire. Il comprend un identifiant, un nom, des coordonnées GPS, un type (caméra ou LAPI), des dates de début et fin d'activité optionnelles, un lien vers une image (optionnel), ainsi qu'une relation bidirectionnelle avec les enregistrements de données collectées.
- **`Data`** : représente une mesure collectée par un capteur à un instant donné. Elle contient l'identifiant du capteur (`sensor_id`), la date et l'heure de la mesure, le type de mesure et sa direction. Elle est reliée au capteur associé via une clé étrangère et une relation `SQLAlchemy`.

Ces classes héritent d'une base déclarative commune, ce qui permet de générer automatiquement les tables `SQL` avec leurs relations et contraintes.

Capteurs initiaux et peuplement de la base : Une liste **`initial_sensors`** contient un ensemble de capteurs prédéfinis (caméras normales et capteurs LAPI), chacun décrit par ses coordonnées, son nom et son type. La fonction **`initialize_db`** supprime les anciennes tables, les recrée, puis insère les capteurs initiaux dans la base. Elle est invoquée dans le bloc principal **`if __name__ == "__main__"`** lors de l'exécution directe du fichier, ce qui permet de réinitialiser rapidement la base de données pendant la phase de développement.

4.3.5 Description de l'application FastAPI

L'application FastAPI développée constitue une API RESTful complète dédiée à la gestion de capteurs urbains et à l'analyse des données qu'ils produisent. Construite avec le framework FastAPI, elle offre un ensemble de routes permettant d'interagir avec les capteurs et leurs mesures de manière structurée.

L'API permet de créer, modifier, supprimer et consulter des capteurs, chacun étant défini par un nom, un type, des coordonnées géographiques, et éventuellement une image représentative. Une fonctionnalité permet l'envoi d'images pour illustrer visuellement un capteur. Par ailleurs, il est possible d'importer des fichiers CSV contenant des mesures temporelles associées aux capteurs, notamment le type de détection observée, la direction du flux, et l'horodatage.

Une fois les données importées, plusieurs routes d'analyse permettent d'en extraire des informations utiles. L'utilisateur peut, pour un capteur donné, obtenir l'évolution temporelle des flux par direction sous forme agrégée, ou encore une distribution des types

détectés sur une période donnée. Une vue d'ensemble permet aussi de générer des statistiques globales sur l'ensemble des capteurs, telles que la répartition des types, la densité d'activité dans le temps, ainsi qu'une carte thermique des capteurs colorée selon leur niveau d'activité, basé sur une analyse quantile.

L'application inclut également des routes utilitaires, comme la récupération de l'intervalle temporel commun à tous les capteurs ou la suppression des images associées. L'architecture mise en place permet une extension aisée.

5 Le produit final

5.1 L'application

Le produit final est **une application web complète et aboutie**, associant une interface utilisateur fluide à un serveur backend robuste assurant une gestion efficace des données.

L'application est prête à fonctionner en **mode récepteur** pour recevoir et traiter en temps réel les données issues des capteurs, avec un enregistrement immédiat de toute modification (ajout, suppression ou déplacement de capteur, insertion de nouvelles données) directement dans la base de données du serveur.

En **mode simulateur**, elle exploite les données enregistrées pour proposer des analyses temporelles fines et précises soulignant les données analysées pour les deux sens de circulation ainsi que les donénes globales, des distributions de types de véhicules, ainsi qu'une cartographie thermique des flux de trafic.

Un point fort majeur du produit est **sa rapidité d'exécution**. L'architecture technique soigneusement pensée (combinant un backend `Python` léger et optimisé avec un frontend `React` dynamique) permet un temps de traitement extrêmement court : les opérations côté serveur (comme l'insertion, la mise à jour ou la requête de données) s'effectuent sans latence perceptible, même avec un grand volume de capteurs et d'enregistrements.

De plus, grâce à une gestion rigoureuse des états côté client et à l'utilisation de composants performants (`Leaflet`, composants `React` optimisés), l'interface est **réactive**, **stable**, et **sans bug** constaté lors de l'utilisation normale. Les interactions telles que l'ajout de capteurs, la visualisation des analyses, ou la navigation sur la carte sont instantanées, assurant une expérience utilisateur fluide et satisfaisante.

Afin de garantir la stabilité et la robustesse de l'application, tous les cas d'utilisation possibles ont été anticipés et traités dès la phase de développement. Qu'il s'agisse de l'ajout de capteurs invalides, de la gestion d'intervalle de temps incohérent, ou d'opérations multiples simultanées, des vérifications systématiques et des protections côté serveur et client ont été mises en place. Ce traitement rigoureux a permis de réduire drastiquement les bugs potentiels jusqu'à leur élimination totale dans les scénarios normaux d'utilisation.

Le **choix des couleurs** de l'application a été soigneusement réfléchi afin d'assurer un confort visuel optimal pour l'utilisateur. Les teintes principales ont été sélectionnées pour être à la fois douces pour la vue lors d'une utilisation prolongée et suffisamment

contrastées pour attirer naturellement l'attention sur les éléments essentiels de l'interface, notamment les boutons d'action principaux (ajout, suppression, import de données, visualisations). Cette approche a permis d'optimiser l'ergonomie générale de l'application, en rendant les interactions plus intuitives et agréables.

5.2 Démarrage et exécution

Le déploiement et l'exécution de l'application reposent sur une architecture séparant clairement le Backend et le Frontend, afin d'assurer modularité, fiabilité et facilité de maintenance.

• **Backend :** Le serveur est développé en `Python`. Son lancement s'effectue à l'aide de la commande suivante :

```
python3 main.py
```

Cette commande exécute le fichier principal du serveur, initialise la connexion aux bases de données et prépare l'interface à recevoir les flux de données entrants. Le serveur assure la gestion de :

— l'activation et l'enregistrement des capteurs

— la réception et du traitement des données envoyées ou importées

— la synchronisation en temps réel avec l'interface frontend

• **Frontend :** Le frontend est développé avec `Next.js` (`React` + `TypeScript`). Pour exécuter l'interface utilisateur en mode développement, la commande suivante est utilisée :

```
npm run dev
```

Cette commande lance un serveur de développement local, généralement accessible à l'adresse `http://localhost:3000/`. L'interface permet alors de :

— afficher la carte interactive enrichie par `Leaflet`

— gérer dynamiquement les capteurs

— basculer entre le mode récepteur et le mode simulateur

— importer et visualiser les données

6 Propositions d'amélioration

6.1 Sécurisation de l'accès et authentification

Dans sa version actuelle, l'application est déployée dans un environnement restreint sans mécanisme d'authentification, ce qui est adapté à une phase de développement ou à un usage interne. Toutefois, pour envisager un déploiement plus large ou dans un contexte de production, il serait indispensable d'ajouter une sécurisation robuste de l'accès.

Une première amélioration consisterait à mettre en place un **système d'authentification par identifiants** (nom d'utilisateur et mot de passe), protégeant l'accès aux fonctionnalités sensibles telles que l'ajout, la suppression ou la modification de capteurs et de données.

De plus, il serait pertinent d'implémenter une gestion des rôles pour distinguer différents niveaux d'accès :

- **Administrateurs :** autorisés à ajouter/supprimer des capteurs, importer des fichiers CSV, et consulter les statistiques avancées
- **Utilisateurs standards :** autorisés uniquement à consulter les données et effectuer des analyses sur la carte sans modifier la structure du système.

Pour renforcer la sécurité des échanges, il serait également conseillé de :

- utiliser des sessions sécurisées et des tokens d'authentification (par exemple avec JSON Web Tokens – JWT)
- chiffrer les communications entre le client et le serveur à l'aide du protocole `HTTPS`
- introduire éventuellement un système de traçabilité (*logs*) pour enregistrer les actions sensibles réalisées par les utilisateurs (ajout, suppression, modification de capteurs ou de données)

6.2 Adaptation mobile

Actuellement, l'application est principalement optimisée pour un affichage et une utilisation sur des écrans d'ordinateur de taille moyenne à grande. Afin de rendre l'interface plus accessible à un public plus large et de s'adapter aux usages modernes, il serait pertinent de travailler sur une adaptation spécifique pour les terminaux mobiles (smartphones et tablettes).

Cette amélioration passerait par l'implémentation d'un design responsive, utilisant des techniques telles que :

- des media queries `CSS` adaptées aux différentes résolutions d'écran
- des réorganisations d'interface (par exemple : menu burger pour la *sidebar*, boutons plus gros et espacés pour le tactile)
- l'optimisation de la carte interactive afin d'assurer un zoom, un déplacement et une interaction fluide au doigt

L'adaptation mobile viserait également à :

- réduire la charge visuelle pour éviter de surcharger les petits écrans
- simplifier les formulaires pour l'ajout ou la modification de capteurs
- maintenir une lisibilité parfaite des séries temporelles, des cartes de chaleur et des autres visualisations

Par ailleurs, des éléments spécifiques pourraient être ajoutés, comme :

- une détection automatique du type d'appareil pour ajuster dynamiquement certains comportements
- des optimisations de performances spécifiques aux mobiles (par exemple, chargement différé des couches de la carte)

Cette adaptation mobile rendrait l'application beaucoup plus flexible, attractive et moderne, en permettant une utilisation fluide lors de déplacements sur le campus ou en situations de mobilité professionnelle.

7 Conclusion

7.1 Retour sur le projet

Tout au long du projet, notre équipe a été confrontée à plusieurs défis, en particulier lors de l'intégration des différents composants du système : l'interface frontend développée avec `Next.js`, le serveur backend en `FastAPI`, la base de données `PostgreSQL`, ainsi que les outils de traitement vidéo à l'aide de `YOLO`. La coordination entre ces briques technologiques hétérogènes a exigé une organisation rigoureuse et une communication constante.

En parallèle, la répartition efficace des tâches au sein de notre groupe de sept étudiants a représenté un autre enjeu majeur, notamment pour maintenir un bon rythme de développement sans surcharger certains membres. Ces contraintes nous ont conduits à adapter progressivement notre méthode de travail, avec une forte adhérence aux principes de l'agilité. Les sprints hebdomadaires, les échanges réguliers avec notre client et notre encadrant, ainsi que les ajustements rapides face aux imprévus ont été essentiels pour garantir la progression fluide du projet et la cohésion de l'équipe.

Ce projet s'est révélé **particulièrement stimulant et porteur de sens**, en lien direct avec une problématique réelle de mobilité sur le campus. Grâce à notre rigueur et à notre implication, nous avons su **respecter le cahier des charges initial** et **mener à bien la phase de réalisation**, en livrant une application fonctionnelle, complète et conforme aux attentes exprimées par le client.

7.2 Leçons techniques apprises

L'expérience acquise au cours de ce projet a été particulièrement enrichissante sur le plan technique. Elle nous a permis de consolider nos compétences en programmation, notamment en `Python` pour le développement du serveur backend avec `FastAPI`, et en `TypeScript` pour la conception du frontend à l'aide du framework `Next.js`. Nous avons appris à structurer une architecture client-serveur complète, à gérer l'interaction avec une base de données relationnelle (`PostgreSQL`) et à exploiter une bibliothèque de cartographie dynamique telle que `Leaflet` pour intégrer une carte interactive dans une interface utilisateur fluide.

Nous avons également manipulé des volumes de données réelles issus de vidéos et de capteurs, tout en utilisant des modèles de détection comme `YOLO` pour en extraire des informations pertinentes. Les compétences acquises au travers de cette réalisation renforceront sans aucun doute notre capacité à développer des systèmes logiciels robustes et complexes dans un contexte académique ou professionnel.

7.3 Leçons sur le génie logiciel et l'organisation

Ce projet a mis en lumière l'importance d'une **gestion de projet agile rigoureuse**, particulièrement dans un contexte de collaboration entre plusieurs développeurs aux profils et spécialisations complémentaires. Nous avons pris conscience de la nécessité d'une **communication continue**, d'une **planification souple** et d'une **capacité d'adaptation rapide** face aux imprévus techniques et organisationnels.

Les **réunions hebdomadaires**, organisées en présence de notre encadrant et du client, ont constitué un point d'ancrage essentiel pour assurer la synchronisation des tâches et le respect des objectifs intermédiaires.

Ces enseignements pratiques renforceront à l'avenir notre capacité à organiser efficacement des projets de développement logiciel, à gérer les ressources humaines et techniques de manière plus stratégique, et à garantir une meilleure coordination entre les membres d'une équipe technique.

8 Annexe

A Références

A.1 React.js et Next.js

`React` est une bibliothèque `JavaScript` open-source développée par *Meta* (anciennement *Facebook*), conçue pour créer des interfaces utilisateur dynamiques et réactives. Elle repose sur une architecture de composants réutilisables et sur un mécanisme de mise à jour optimisé appelé *Virtual DOM*, qui améliore significativement les performances de rendu. `React` est largement adopté dans le développement web moderne grâce à sa flexibilité, sa modularité et sa large communauté.

`Next.js`, quant à lui, est un *framework* basé sur `React` qui ajoute des fonctionnalités avancées telles que le rendu côté serveur (SSR), la génération statique (SSG), le préchargement automatique des pages et la gestion intégrée des routes. Il facilite le développement d'applications web performantes, scalables et optimisées pour le référencement. `Next.js` est particulièrement adapté aux projets professionnels ou complexes nécessitant à la fois rapidité, organisation du code et évolutivité.

Pour plus d'informations, voir : https://nextjs.org/ et https://react.dev/.

A.2 D3.js

`D3.js` (pour **D**ata-**D**riven **D**ocuments) est une bibliothèque `JavaScript` puissante permettant de manipuler des documents basés sur des données. Elle utilise les standards web comme `SVG`, `HTML` et `CSS` pour produire des visualisations dynamiques, interactives et hautement personnalisables. D3 se distingue par sa flexibilité : plutôt que de fournir des graphiques prédéfinis, elle offre des outils de bas niveau permettant de créer des visualisations sur mesure à partir de n'importe quel jeu de données.

Grâce à son approche déclarative et à sa gestion fine des échelles, transitions et sélections DOM, D3.js est largement utilisée dans le domaine de la datavisualisation, tant pour des tableaux de bord interactifs que pour des visualisations scientifiques ou journalistiques avancées.

Pour plus d'informations, voir : https://d3js.org/.

A.3 FastAPI

`FastAPI` est un *framework* web moderne pour `Python`, spécialement conçu pour la création d'API REST performantes et asynchrones. Il repose sur `Starlette` pour la gestion du serveur et sur `Pydantic` pour la validation des données. FastAPI se distingue par sa rapidité d'exécution, sa simplicité d'utilisation, et sa capacité à générer automatiquement une documentation interactive des routes (OpenAPI / Swagger UI), ce qui en fait un outil idéal pour les projets orientés services.

Grâce à la prise en charge native de `Python` 3.6+ (types, annotations) et à ses performances proches de celles des *frameworks* en `Node.js` ou `Go`, FastAPI est largement adopté pour le développement de microservices, d'API backend ou d'applications web modulaires.

Pour plus d'informations, voir : https://fastapi.tiangolo.com/.

A.4 YOLO : You Only Look Once

YOLO (**Y**ou **O**nly **L**ook **O**nce) est un algorithme de détection d'objets en temps réel basé sur les réseaux de neurones convolutionnels. Contrairement aux méthodes traditionnelles qui analysent une image par régions successives, YOLO traite l'image entière en une seule passe, ce qui le rend extrêmement rapide et efficace. Il est largement utilisé en vision par ordinateur pour des applications comme la surveillance, la reconnaissance d'objets et la conduite autonome. YOLO existe en plusieurs versions (YOLOv1, YOLOv2, YOLOv3, YOLOv4, YOLOv5, YOLOv7, etc.), chacune améliorant la précision et la vitesse de traitement.
Pour plus de détails, voir : https://pjreddie.com/darknet/yolo/.

A.5 Leaflet

Leaflet est une bibliothèque `JavaScript` open-source conçue pour créer des cartes interactives et légères sur le web. Elle est optimisée pour fonctionner efficacement sur les appareils mobiles et les navigateurs modernes, offrant des fonctionnalités comme le zoom fluide, le marquage de points, la superposition de tuiles et la gestion des événements cartographiques. Compatible avec divers fournisseurs de fonds de carte comme `OpenStreetMap`, elle permet l'intégration et la personnalisation aisée des cartes dans les applications web. Leaflet est largement utilisé pour des applications SIG (Systèmes d'Information Géographique), la visualisation de données géospatiales et les services de navigation.
Pour plus de détails, voir : https://leafletjs.com/.

A.6 PostgreSQL

PostgreSQL est un système de gestion de bases de données relationnel et objet open-source, réputé pour sa robustesse, sa conformité aux standards `SQL` et ses nombreuses fonctionnalités avancées. Il prend en charge des types de données complexes, l'indexation avancée, les requêtes `SQL` complexes et la gestion efficace des transactions (*ACID*). Conçu pour être extensible, PostgreSQL permet d'ajouter des fonctions personnalisées, de gérer des bases de données de grande taille et d'intégrer des technologies comme le `JSON` pour les données semi-structurées. Il est largement utilisé dans des domaines variés tels que les applications web, la finance, la recherche et le big data.
Pour plus de détails, voir : https://www.postgresql.org/.

B Architecture et description du code

Cette section décrit brièvement les fichiers principaux du projet, en précisant leur rôle et les fonctions essentielles qu'ils contiennent. Elle est divisée en deux sous-parties : *Backend* et *Frontend*.

B.1 Backend

`YoloVideoProcessor.py`
Fichier central pour le traitement vidéo par YOLO, à travers la classe `YoloVideoProcessor` :

- `__init__()` : ouvre la vidéo, lit sa résolution, durée et fréquence d'image.
- `get_info(), get_fps(), get_duration_per_frame()` : accesseurs aux métadonnées vidéo.
- `visualize_processing()` : affiche la détection YOLO en direct sur une portion de vidéo.
- `process_video()` : applique YOLO image par image, identifie les véhicules avec ByteTrack, détecte leur direction, et exporte un CSV enrichi.
- `get_datetime()` : lit la date/heure directement depuis la vidéo via OCR (EasyOCR).

`multithreaded_yolo.py`
Permet le traitement parallèle d'une vidéo longue, en plusieurs segments :

- `thread_safe_detection()` : applique `process_video()` à un intervalle donné (en minutes).
- `worker()` : fonction exécutée par chaque thread, traite une tâche (plage de temps), et enregistre un fichier CSV.
- `main()` : découpe la vidéo, planifie les tâches par tranches de minutes, lance les threads et orchestre la synchronisation.

`raw_data_treatment.py`
Nettoie statistiquement les objets détectés à partir d'un fichier CSV :

- `get_valid_object()` : filtre les objets par type (car, bus, etc.) via un score-z appliqué au nombre de frames où ils apparaissent. Ne conserve que la dernière apparition fiable de chaque ID.

`gather_results.py`
Rassemble les résultats d'un batch traité :

- `extract_minute()` : extrait l'index temporel d'un fichier à partir de son nom.
- Script principal : trie les CSV par minute, applique `get_valid_object()` sur chaque, et concatène les résultats filtrés dans `merged_filtered_PXX.csv`.

`final_merge.py`
Fusionne les résultats de plusieurs dossiers batch :

- `concat_csv_files()` : recherche tous les fichiers `merged_filtered_PXX.csv` dans les dossiers `batch_1` à `batch_6`, les lit, supprime les doublons et les fusionne dans un fichier global.

`database.py`
Fichier central de gestion de la base de données PostgreSQL à l'aide de SQLAlchemy :

- `get_engine()` : établit une connexion PostgreSQL à partir du fichier `config.json`, crée la base si elle n'existe pas.
- `Sensor` : modèle représentant un capteur (nom, type, coordonnées, période d'activité, image).
- `Data` : modèle représentant une donnée captée (datetime, direction, type).
- `initialize_db()` : réinitialise les tables, insère une liste prédéfinie de capteurs répartis sur le campus.

`main.py`
Point d'entrée de l'API FastAPI. Ce fichier orchestre les interactions entre le client et la base de données :

- **Configuration du serveur** : via `FastAPI`, configuration CORS, démarrage avec Uvicorn.
- **Capteurs** :
  - `GET /sensors` : liste tous les capteurs avec métadonnées.
  - `POST /sensors` : ajoute un nouveau capteur.
  - `PUT /sensors/{id}` : met à jour les coordonnées.
  - `DELETE /sensors/{id}` : supprime le capteur et ses données.
- **Données** :
  - `POST /upload-csv/{id}` : charge un fichier CSV et l'associe à un capteur.
  - `GET /sensors/{id}/data` : récupère les comptages agrégés par direction sur un intervalle temporel.
  - `GET /sensors/{id}/distribution` : fournit la distribution des types d'objets détectés sur une période.
- **Statistiques globales** :
  - `GET /sensors/general_statistics` : retourne les statistiques globales (courbes d'affluence, distribution des types, carte thermique avec couleur par quantile).
  - `GET /sensors/common-date-range` : donne l'intervalle de dates commun à tous les capteurs.
- **Images de capteurs** :
  - `POST /sensors/{id}/upload-image` : upload d'une image associée à un capteur.
  - `DELETE /sensors/{id}/remove-image` : supprime l'image associée.

B.2 Frontend

B.2.1 Répertoires et fichiers essentiels

— **public/assets/ :** Dossier contenant deux sous répertoires, **svg/** et **png/**, contenant tous les deux les images insérées dans l'interface selon le type (*".svg"* ou *".png"*). Dans le code des fichiers principaux, à chaque insertion d'une image dans le Frontend, le chemin de cette image doit bien être respecté.

— **src/app/ :** Répertoire principal contenant le code `TypeScript` et `CSS` de tous les composants du Frontend. Pour les information sur tous les sous-répertoires, voir le paragraphe 4.2.1. Pour plus de détails sur le répertoire contenant le code `tsx` des composants du Frontend, voir le paragraphe annexe A.2.2.

B.2.2 Code `tsx` des composants du Frontend

`Map.tsx`
Composant principal d'affichage de la carte interactive avec `Leaflet` :

— Initialise la position centrale, les états (mode récepteur/simulateur, *heatmap*, capteur actif...).

— Affiche les capteurs à partir des données récupérées via `useSensors()`.

— Permet de basculer dynamiquement entre deux modes :

  — **Récepteur** : permet d'ajouter, déplacer ou supprimer des capteurs ;

  — **Simulateur** : permet d'afficher les statistiques globales et locales.

— Gère l'interaction avec les capteurs : sélection, affichage des statistiques, ou surcouche de couleur (*heatmap*).

— Composants associés :

  — `LocationMarker` : affiche chaque capteur sous forme de marqueur interactif ;

  — `SensorInput` : formulaire de création de capteur ;

  — `GeneralStats` : panneau de statistiques globales (affiché dans le mode simulateur) ;

  — `Caption` : légende contextuelle.

`SensorInput.tsx`
Formulaire d'ajout manuel d'un nouveau capteur, avec gestion d'alerte utilisateur :

— Gère les champs suivants : nom, type, latitude et longitude.

— Composant contrôlé : chaque champ est lié à l'état `formData`.

— Envoie la requête via le hook `useAddSensor()` pour créer un capteur côté serveur.

— Affiche un message de succès ou d'erreur grâce au composant `Alert`.

`LocationMarker.tsx`
Composant `React` responsable de l'affichage visuel et du comportement de chaque capteur sur la carte :

— Utilise des marqueurs personnalisés (couleurs/types ou taille selon la *heatmap*) avec des effets d'animation via `CSS`.

- Active le déplacement manuel du capteur si le mode récepteur est actif et non verrouillé.
- Met à jour la position du capteur en base via `useUpdateSensor()` lors d'un déplacement.
- Affiche :
  - un `Popup` contextuel avec informations (via `SensorInfo`) ou actions (via `ConfirmClose`) ;
  - un panneau latéral `Sidebar` pour l'import de données ou d'image, lorsque le capteur est sélectionné.

`GeneralStats.tsx`
Composant d'analyse globale de l'ensemble des capteurs dans le mode simulateur :

- Permet de sélectionner une période d'analyse (début, fin, période en minutes) sur un intervalle commun aux capteurs (via `useCommonDateRange()`).
- Trois visualisations disponibles :
  - **Séries temporelles** : évolution du nombre total de véhicules détectés (`SensorChart`) ;
  - **Distribution des types** : parts relatives des véhicules détectés par type (`TypeDistributionChart`) ;
  - **Carte thermique (Heatmap)** : colorisation des capteurs selon leur niveau d'activité, basée sur des quantiles statistiques.
- Intègre des éléments d'interaction :
  - vérification des dates valides, désactivation dynamique de l'interface selon l'état ;
  - possibilité de masquer/afficher les différentes sections de l'analyse.

`Caption.tsx`
Légende dynamique affichée en bas à droite de la carte :

- Affiche uniquement les types de capteurs présents à l'instant donné.
- Associe chaque type (LAPI, caméra, vélo, tube) à une couleur spécifique (noir, rouge, vert, bleu).
- Permet une lecture rapide et intuitive de la carte.

`Sidebar.tsx`
Panneau latéral associé à un capteur sélectionné (en mode récepteur) :

- Permet l'import de :
  - fichiers CSV contenant les données de trafic ;
  - images illustrant le capteur.
- Utilise les hooks :
  - `useUploadCsv` pour importer les données vers le backend ;
  - `useUploadSensorImage` et `useDeleteSensorImage` pour gérer l'image.
- Comportement dynamique : affichage réduit/étendu, fermeture indépendante.
- Affiche un message d'alerte en cas de succès ou d'erreur lors d'une action utilisateur.

`SensorInfo.tsx`
Composant `React` affiché dans un `Popup` (mode simulateur) lors du clic sur un capteur :

- Affiche les informations du capteur : nom, type, coordonnées géographiques.
- Permet à l'utilisateur de :
  - sélectionner une période temporelle (début, fin) dans l'intervalle valide du capteur ;
  - définir une période d'agrégation (en minutes).
- Lance une recherche de données via les hooks :
  - `useSensorData()` pour récupérer l'évolution temporelle des passages ;
  - `useSensorTypeDistribution()` pour obtenir la distribution des types d'objets détectés.
- Affiche dynamiquement les résultats sous forme de :
  - graphique temporel (`SensorChart`) ;
  - graphique de répartition (`TypeDistributionChart`).
- Intègre une logique de validation des dates, avec contrôle sur l'intervalle autorisé.

`SensorChart.tsx`
Composant d'affichage de séries temporelles de trafic, basé sur `D3.js` :

- Affiche dynamiquement une ou plusieurs courbes :
  - une par direction (ex : Est, Ouest...) ;
  - une pour le total des passages.
- Parse les données brutes de comptage (date et direction_counts).
- Configure les axes temporel (X) et quantitatif (Y), avec échelle adaptative.
- Génère un tracé fluide (spline) pour chaque série à l'aide de `d3.line()`.
- Affiche une légende colorée automatique.
- Intègre un **pointeur interactif** :
  - ligne verticale suivant la souris ;
  - boîte d'information montrant les valeurs exactes pour chaque direction au survol.

C Manuel d'utilisation de l'application

Cette annexe présente un guide d'utilisation complet de l'application. L'interface propose deux modes d'interaction principaux : le mode **récepteur**, orienté vers la gestion des capteurs et des données, et le mode **simulateur**, dédié à l'exploration et l'analyse des données enregistrées.

C.1 Mode récepteur

Lorsque le bouton `RÉCEPTEUR` est activés (en haut à droite), l'application permet à l'utilisateur de manipuler dynamiquement les capteurs sur la carte. Les fonctionnalités disponibles dans ce mode sont les suivantes :

— **Ajout d'un nouveau capteur** : l'utilisateur peut cliquer sur le bouton `NOUVEAU CAPTEUR` pour faire apparaître un formulaire de création. Celui-ci demande :

- un nom libre,
- un type de capteur à choisir parmi : `Caméra Normale` (rouge), `Caméra LAPI` (noir), `Capteur à Vélo` (vert) ou `Capteur Tube` (bleu),
- les coordonnées (latitude/longitude) précises du point d'implantation.

Le capteur s'affiche alors immédiatement sur la carte avec une icône et une couleur correspondant au type choisi. La légende se met à jour automatiquement, et une alerte verte s'affiche en haut de l'écran avec un message de succès. Dans le cas d'echec, le capteur ne s'ajoute pas, la légende ne change pas et une alerte rouge s'affiche dans le même endroit avec un message d'echec.

— **Manipulation d'un capteur existant** : Un clic sur un capteur permet d'afficher trois boutons verticalement allignés, chacun servant à un besoin spécifique :

- **Déplacement manuel** : le déverouillage via le bouton "cadenas" permet de déplacer le capteur sur la carte. Une fois dans la position désirée, on le verouille.
- **Suppression** : le clic sur l'icône de suppression sert à supprimer le capteur.

Toute modification est immédiatement enregistrée dans la base de données via le backend `FastAPI`.

— **Import de données CSV** : en cliquant sur un capteur, et en cliquant sur le bouton au milieu des trois boutons qui s'affichent, une `Sidebar` s'affiche à gauche de l'écran. Celle-ci propose :

- l'import d'un fichier CSV avec les données de comptage (date, type, direction) sous le format fixé,
- l'upload d'une image illustrant le capteur,
- un bouton `Fermer` pour replier le panneau.

— **Indications visuelles** :

- une légende dynamique affichée en bas à droite,
- un bouton `"?"` (aide rapide) en bas à gauche,

C.2 Mode simulateur

Lorsque le bouton `SIMULATEUR` est activé, l'utilisateur accède aux outils d'analyse des données déjà enregistrées.

— **Analyse locale par capteur** : en cliquant sur un capteur, une fenêtre s'affiche avec les informations suivantes : nom, type, coordonnées. L'utilisateur peut alors :
  - sélectionner un intervalle de temps **valide**, automatiquement restreint à la période couverte par les données du capteur,
  - choisir une période d'affichage en minutes pour lisser les courbes,
  - lancer une `Recherche` qui va afficher :
    — une série temporelle par direction (*SensorChart*),
    — une distribution des types de véhicules (*TypeDistributionChart*).

— **Analyse globale** : en cliquant sur `STATISTIQUES GÉNÉRALES`, l'utilisateur accède à :
  - la courbe cumulée de trafic total pour tous les capteurs,
  - la distribution globale des types,
  - une carte de chaleur (*Heatmap*) colorant et fixant les diamètres des capteurs selon leur activité relative. Un guide de couleurs accompagne la carte de chaleur quand elle est activée.

— **Fonctions supplémentaires** :
  - Désactivation des analyses en cas de dates invalides,
  - Réfresh automatique des graphiques à chaque modification de période,
  - Affichage conditionnel des sections d'analyse.

Toutes les données affichées sont récupérées en temps réel depuis le serveur `FastAPI`, ce qui garantit leur exactitude et synchronisation avec la base `PostgreSQL`. L'utilisateur peut ainsi exploiter efficacement les données collectées pour produire des visualisations interactives et prendre des décisions informées sur la mobilité du campus.