



# Document de reporting

## Résultats et analyse de la PoC MedHead

### Table des matières :

1. <u>Introduction</u>	02
2. <u>Contexte</u>	03
3. <u>Approche technique et processus de développement</u>	04
1. <u>Technologies utilisées</u>	04
2. <u>Méthodologie de développement</u>	06
4. <u>Résultat</u>	09
1. <u>Fonctionnalités implémentées</u>	09
2. <u>Résultats des tests</u>	10
5. <u>Analyse</u>	12
1. <u>Points forts</u>	12
2. <u>Limites</u>	13
3. <u>Opportunités</u>	14
6. <u>Recommandations</u>	15
1. <u>Améliorations techniques</u>	15
2. <u>Fonctionnalités supplémentaires</u>	17
3. <u>Tests et validation</u>	17
4. <u>Collaboration et documentation</u>	18
7. <u>Références</u>	18

# 1. Introduction

Le projet MedHead est une preuve de concept (PoC) visant à démontrer la faisabilité d'un système innovant de réservation de lits hospitaliers, spécialement conçu pour répondre aux situations d'urgence. L'objectif principal est de fournir une solution efficace et fiable permettant aux professionnels de santé, tels que les médecins de localiser, réserver et valider des disponibilités de lits adaptés à des besoins spécifiques en temps réel.

Cette plateforme s'adresse exclusivement à des utilisateurs professionnels, connectés et identifiés au sein de l'environnement du système grâce à leurs informations personnelles, notamment :

- **Nom et prénom** (identification utilisateur) ;
- **Rôle professionnel** (médecin, infirmière, etc.) ;
- **Localisation géographique** (latitude et longitude).

L'aspect central de cette PoC repose sur l'optimisation des processus critiques liés à la gestion hospitalière d'urgence tout en garantissant :

- **Une rapidité de réponse optimale**, permettant aux utilisateurs de recevoir des résultats instantanés même sous forte charge.
- **La sécurité des données** sensibles via HTTPS et des mécanismes de gestion sécurisée des API ;
- **La conformité réglementaire** en respectant les normes applicables.

**Note importante** : Cette preuve de concept considère un environnement simulé, dans lequel :

- Les aspects d'authentification et d'identification des utilisateurs sont déjà implémentés et disponibles.
- Les API externes, notamment celles utilisées pour le calcul des trajets à partir des coordonnées GPS et les interactions avec les hôpitaux, sont également simulées.

Le développement de ces fonctionnalités spécifiques ne fait donc pas partie des objectifs actuels du projet.

## 2. Contexte

### Objectif du projet :

- **Recherche et tri des hôpitaux** : Fournir une liste d'hôpitaux disponibles en fonction de la spécialité demandée et du délai de trajet estimé.
- **Réservations en temps réel** : Permettre aux utilisateurs de réserver des lits hospitaliers tout en gérant les contraintes des APIs externes.
- **Interface utilisateur fluide** : Offrir une expérience utilisateur simple, intuitive et fonctionnelle, adaptée aux besoins des professionnels de santé.
- **Validation d'une architecture évolutive** : Tester une infrastructure basée sur des microservices indépendants, pouvant évoluer horizontalement grâce à des technologies de conteneurisation.

### Contraintes :

- **Simulation des API externes** : Les interactions avec les API externes (par exemple, pour le calcul de trajets ou l'accès aux bases des hôpitaux) sont entièrement simulées, éliminant ainsi les contraintes des environnements réels.
- **Conformité réglementaire** : Respecter les normes en vigueur, notamment en matière de gestion des données sensibles et de confidentialité, serment d'Hippocrate et NHS (National Health Service).
- **Absence de données réelles** : Travailler dans un environnement simulé, sans interaction directe avec des bases de données ou systèmes réels et des données simplifiées.

## 3. Approche technique et processus de développement

La méthodologie adoptée pour le projet MedHead repose sur une approche moderne combinant des outils technologiques avancés et des pratiques de développement itératives. L'objectif principal est de garantir une livraison rapide et fiable des fonctionnalités clés de la PoC tout en assurant la qualité, la maintenabilité et l'évolutivité du système.

Cette section détaille les technologies utilisées, les étapes méthodologiques suivies, et les pratiques mises en œuvre pour répondre aux exigences du projet, notamment en termes de rapidité, de fiabilité et de conformité aux contraintes simulées.

### 3.1 Technologies utilisées

Le projet MedHead repose sur des technologies modernes adaptées aux besoins de la PoC, couvrant le backend, le frontend, les tests et l'intégration continue pour garantir performance et maintenabilité.

#### Backend :

- **Java 21 avec Spring Boot :**
  - Utilisation de Spring Boot pour faciliter le développement rapide d'applications Java robustes et pour bénéficier de son écosystème riche.
  - Adoption de Java 21 pour tirer parti des dernières fonctionnalités du langage et des améliorations de performance.
- **Swagger (OpenAPI) :**
  - Génération automatique de la documentation interactive de l'API REST.
  - Facilite les tests et l'exploration des endpoints pour les développeurs et les parties prenantes.

#### Frontend :

- **Angular 16 avec TypeScript :**
  - Framework moderne pour construire des applications web dynamiques et réactives.
  - TypeScript apporte un typage statique, améliorant la qualité du code et facilitant la maintenance.

- **Karma et Jasmine :**

- Karma est utilisé comme lanceur de tests pour exécuter les tests unitaires dans différents environnements.
- Jasmine fournit un framework de tests comportementaux pour JavaScript, permettant d'écrire des tests lisibles et faciles à maintenir.

## CI/CD :

- **GitHub Actions :**

- Mise en place de pipelines automatisés pour :
  - Build du backend et du frontend à chaque commit.
  - Exécution des tests unitaires et d'intégration pour garantir la qualité du code.
- Assure une intégration continue fluide et une livraison continue efficace.

## Tests de performance:

- **JUnit :**

- Outil de référence pour l'écriture et l'exécution de tests unitaires en Java.
- Vérification de la logique métier et des services backend dans différents scénarios.
- Complément aux tests de performance pour garantir la fiabilité des composants sous stress.

- **JMeter :**

- Outil open-source pour effectuer des tests de charge et de performance.
- Simulation de multiples utilisateurs simultanés pour évaluer la robustesse de l'API sous stress.
- Création de scénarios réalistes pour mesurer les temps de réponse et identifier les éventuels goulets d'étranglement.

- **JaCoCo :**

- Génération de rapports de couverture de code pour les tests Java.
- Permet d'identifier les parties du code non couvertes par les tests et d'améliorer la qualité globale.

## 3.2 Méthodologie de développement

La méthodologie de développement adoptée pour ce projet repose sur des pratiques modernes, axées sur l'itération rapide, la qualité du code et l'intégration continue. Ces approches permettent d'assurer une progression fluide du projet tout en garantissant la stabilité et la fiabilité des fonctionnalités développées.

### Approche Agile:

- **Sprints courts et itératifs :**
  - Planification de développements sur des cycles courts pour permettre des ajustements rapides en fonction des retours.
  - Priorisation des fonctionnalités clés pour répondre aux objectifs principaux de la PoC.
- **Mise à jour régulière:**
  - Points quotidiens pour suivre l'avancement et lever les obstacles éventuels.
  - Revues de « sprint » pour présenter les fonctionnalités développées les commentaires..

### Respect des principes SOLID et séparation des responsabilités (Separation of Concerns)

Le développement du projet MedHead s'appuie sur les principes SOLID pour garantir un code maintenable, évolutif et bien structuré. Ces principes fondamentaux ont été intégrés dans l'implémentation des services afin d'assurer un découplage optimal et une modularité accrue :

- **Single Responsibility Principle (SRP) :**
  - chaque classe ou module a une responsabilité unique.
- **Open/Closed Principle (OCP) :**
  - Les services sont conçus pour être ouverts à l'extension mais fermés à la modification.
- **Liskov Substitution Principle (LSP) :**
  - Les interfaces définissent des contrats stricts pour les services, permettant de substituer les implémentations sans impact sur le reste du système.

- **Interface Segregation Principle (ISP) :**
  - Les interfaces sont spécifiques à chaque fonctionnalité, évitant aux modules consommateurs d'implémenter ou de dépendre de méthodes inutilisées.
- **Dependency Inversion Principle (DIP) :**
  - Les services ne dépendent pas directement de leurs implémentations, mais plutôt d'abstractions (interfaces).

### Tests à chaque itération:

- **Tests unitaires :**
  - Écriture systématique de tests pour chaque nouvelle fonctionnalité ou correction de bug.
  - Utilisation de frameworks comme JUnit pour le backend et Jasmine pour le frontend.
- **Tests d'intégration:**
  - Vérification que les différents modules fonctionnent correctement ensemble.
  - Tests des endpoints REST avec des outils comme Postman ou Spring MockMvc.
- **Tests de performance:**
  - Exécution de scénarios JMeter pour surveiller l'impact des nouvelles modifications sur les performances.
  - Analyse des résultats pour optimiser le code et améliorer la réactivité de l'application.

### Gestion du code source:

- **Utilisation de Git et GitHub :**
  - Gestion des versions du code avec des branches dédiées pour le développement de fonctionnalités, les corrections de bugs, etc.

- **Changelog maintenu à jour :**
  - Documentation des modifications apportées à chaque version.
  - Facilite le suivi de l'évolution du projet et la communication avec les parties prenantes.

### Collaboration et communication:

- **Documentation Partagée:**
  - Maintien d'un README détaillé pour aider les nouveaux contributeurs à comprendre le projet.
  - Utilisation de Swagger pour partager la documentation de l'API avec l'équipe et les parties externes.

### Qualité et amélioration continue :

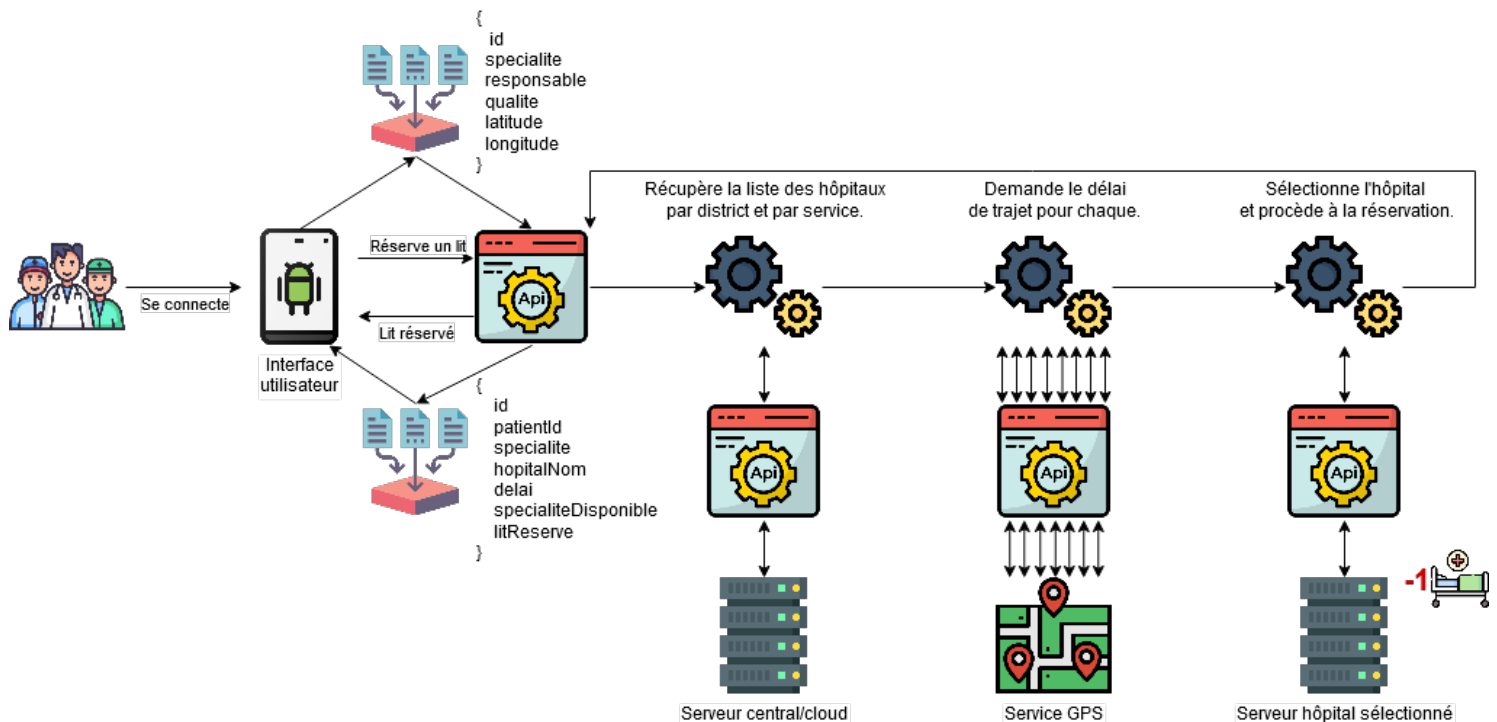
- **Revue de code :**
  - Chaque modification majeure est soumise à une revue pour détecter les erreurs potentielles et garantir la cohérence du code.
- **Gestion des bugs et suivi des tickets :**
  - Utilisation de l'outil de suivi des issues de GitHub pour consigner et prioriser les tâches.
  - Correction immédiate des push contenant des erreurs.



## 4. Résultats

### 4.1 Fonctionnalités implémentées

Le projet MedHead intègre plusieurs fonctionnalités clés permettant de simuler efficacement le processus de réservation de lits hospitaliers, tout en assurant une gestion fluide des interactions entre les utilisateurs et les services externes.



- Figure 1 : Fonctionnements implémentés

1. **Endpoint principal pour la réservation (Service Patient) :**
  - Contacte le service central de la PoC pour effectuer une demande de réservation de lit.
  - Transmet les informations de base, notamment le nom, la qualité, la latitude, la longitude, et la spécialité.
2. **Récupération des informations sur les hôpitaux (Service Hôpital et PopulateHospital) :**
  - Simule un appel à une API externe pour obtenir la liste des hôpitaux disponibles, triés par spécialité et district.
3. **Calcul des délais de trajet (Service GPS) :**
  - Simule un appel à une API externe pour estimer le délai de trajet jusqu'à chaque hôpital proposé.
4. **Réservation auprès de l'hôpital sélectionné (Service Reserve) :**
  - Contacte l'API de l'hôpital sélectionné pour valider et finaliser la réservation du lit.

## 5. Affichage des résultats à l'utilisateur (Service Result) :

- Retourne les informations finales, telles que l'hôpital attribué, le délai estimé et la disponibilité confirmée, directement sur l'interface utilisateur.

## 4.2 Résultats des tests

### 1. Couverture des tests backend : (JUnit & JaCoCo)

- Le taux de couverture global du backend est de 89 % pour les instructions et 74 % pour les branches. Ces résultats montrent que la majorité des scénarios et des cas d'utilisation ont été pris en compte dans les tests unitaires.

PoC medHead

### PoC medHead

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.medHead.poc.config		55 %		50 %	11	21	29	59	10	20	2	6
com.medHead.poc.services		91 %		74 %	17	71	12	126	1	40	0	8
com.medHead.poc.controller		94 %		61 %	14	36	7	97	1	18	0	5
com.medHead.poc.model		94 %		78 %	9	55	5	94	1	34	0	2
com.medHead.poc.entity		97 %		91 %	2	29	1	45	0	17	0	1
com.medHead.poc		100 %		n/a	0	3	0	6	0	3	0	1
Total	193 of 1 899	89 %	42 of 166	74 %	53	215	54	427	13	132	2	23

- Figure 2 : Index.html créer par JaCoCo

### 2. Performances backend avec JMeter :

- Les tests de performance réalisés avec JMeter permettent d'évaluer la robustesse et l'efficacité du backend face à des charges simulées.

#### Scénario de test :

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	60900	3	1	431	8.15	0.04%	874.1/sec	297.60	308.07	348.6
TOTAL	60900	3	1	431	8.15	0.04%	874.1/sec	297.60	308.07	348.6

- Figure 3 : Test de débit avec une condition de réponse de 200 ms

1. **Condition de réponse (SLA) :** Les tests imposent un délai maximum de 200 ms pour la réponse de chaque requête HTTP afin de garantir une expérience utilisateur fluide.

```
PoCMedHeadApplication.java x output_results.txt x
1 ===== Résultats du test =====
2 =                               Taux de succès : 99.94%                               =
3 =                               Délai moyen (pour les réussites) : 6.81 min                               =
4 =                               Succès : Les KPI sont atteints !                               =
5 =====
```

- Figure 4 : Test du délai moyen des réponses et du taux de bon service attribué

2. **Charge simulée :** Simulation de requêtes envoyées simultanément pour mesurer le délai moyen des hôpitaux et la disponibilité des services pour chaque attribution.

**Limites** : Les tests de performance sont limités par la consommation des ports disponibles et leur libération (configuration des tests : MaxPort = 65000 et TimeWait = 30s). Pour des tests plus étendus, il sera nécessaire de conteneuriser avec Docker et d'orchestrer avec Kubernetes. De plus, l'absence de données réelles limite la validité du deuxième test.

### 3. Couverture des tests frontend : (Karma & Jasmine)


- 88,33 % des instructions, 88,88 % des branches et 95,23 % des fonctions sont couvertes.
- Tous les scénarios principaux, tels que la navigation, la validation des champs et les interactions utilisateur, ont été testés.

```
===== Coverage summary =====
Statements   : 88.33% ( 53/60 )
Branches     : 88.88% ( 8/9 )
Functions    : 95.23% ( 20/21 )
Lines        : 87.93% ( 51/58 )
=====
```

- Figure 5 : Test de couverture frontend Karma & Jasmine

**Karma v 6.4.4 - connected; test: complete;** DEBUG

Chrome 131.0.0.0 (Windows 10) is idle

 **Jasmine** 4.6.1 Options

21 specs, 0 failures, randomized with seed 34992 finished in 0.076s

AppComponent

- should navigate to reservation route
- should navigate to reservation and then home
- should redirect empty path to /reservation
- should redirect unknown routes to /home
- should create the app
- should have as title 'angular-frontend'
- should render the router outlet
- should navigate between routes
- should display ReservationComponent for /reservation route

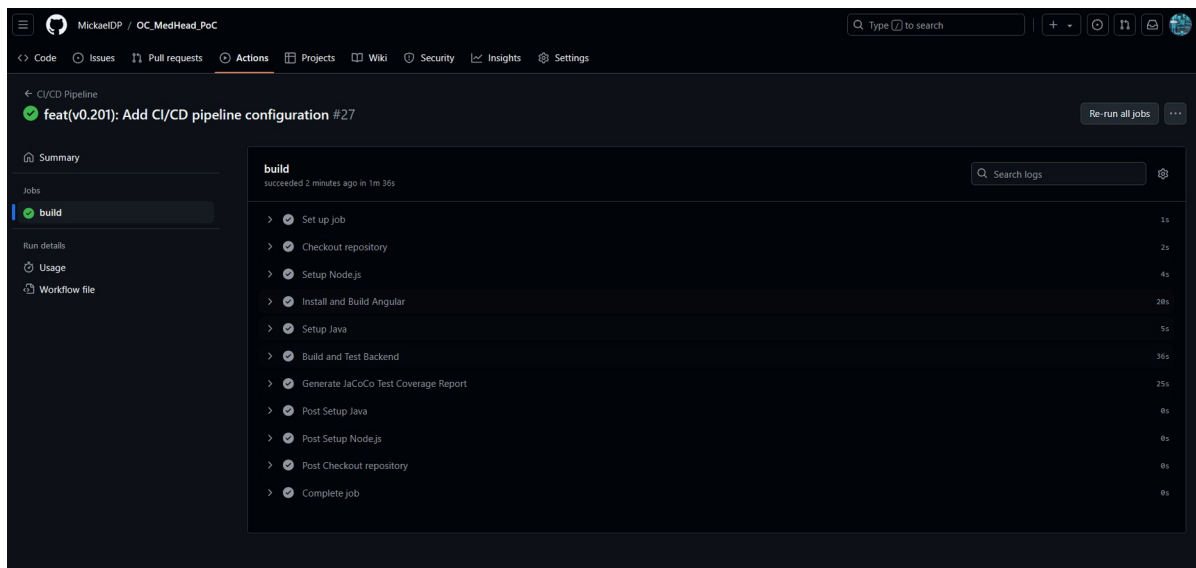
ReservationComponent

- should not proceed if a reservation is already in progress
- should set speciality to emergency and call processReservation
- should reset the popup and clear the form on closePopup
- should not allow form submission if reservationResponse exists
- should create the component
- should capitalize speciality correctly for complex cases
- should handle errors gracefully in reserveInUrgency
- should clear filteredSpecialities when input is empty
- should handle fetch error gracefully
- should update filteredSpecialities on speciality input
- should select a speciality and clear suggestions
- should process a valid reservation and pass correct data to fetch

- Figure 6 : Tests frontend Karma & Jasmine

### 4. CI/CD:

- Les pipelines CI/CD permettent l'exécution automatique des builds, des tests (backend et frontend), ainsi que des rapports de couverture.
- Chaque modification soumise passe par des vérifications rigoureuses avant validation.



- Figure 7 : Pipeline CI/CD avec Git Action

## 5. Analyse

### 5.1 Points forts

#### Architecture modulaire et évolutive :

- Le projet repose sur une architecture basée sur des microservices, permettant une indépendance fonctionnelle et une meilleure scalabilité.
- Bien que la mise en place complète de conteneurs Docker et d'une orchestration Kubernetes ne soit pas réalisée dans la PoC, elle est envisagée comme une solution future pour répondre aux charges croissantes.

#### Performance optimisée :

- Mise en œuvre d'un cache local (ConcurrentHashMap) pour limiter les appels répétés aux APIs externes, réduisant ainsi le temps de réponse global.
- Les tests de performance avec JMeter ont montré un taux de réussite de 99,94 % sous des charges simulées, confirmant une bonne robustesse du backend.

### Respect des standards de sécurité

- Utilisation de certificats pour garantir la sécurisation des échanges via HTTPS, conforme aux standards de confidentialité pour les données sensibles.
- Communications chiffrées et limitation stricte de la diffusion des informations sensibles pour protéger la confidentialité des utilisateurs.

### Documentation technique complète:

- Un ReadMe détaillé accompagne le projet, facilitant la compréhension et l'intégration pour les nouveaux contributeurs.
- La documentation des endpoints via Swagger fournit une vue exhaustive des API disponibles, utile pour le développement et les tests.

### Couverture de tests satisfaisante

- Le backend affiche une couverture de tests de 89 % des instructions et 74 % des branches grâce à JaCoCo, démontrant une prise en compte efficace des scénarios principaux.
- Les tests frontend (Karma & Jasmine) montrent une couverture solide avec 88,33 % des instructions et 95,23 % des fonctions couvertes.

## 5.2 Limites

### Dépendance aux données simulées

- L'utilisation de données en mémoire dans cette PoC ne reflète pas les contraintes opérationnelles des systèmes réels, notamment la synchronisation et la gestion des données dans des bases distribuées ou centralisées.
- Les tests effectués avec des données simplifiées limitent la validation des performances en condition réelle.

### Interface utilisateur minimaliste

- Bien que fonctionnelle, l'interface utilisateur ne propose que des fonctionnalités basiques, sans suivi des réservations, ni tableau de bord analytique pour les performances.
- Un effort supplémentaire serait nécessaire pour rendre l'application plus intuitive et adaptée aux cas complexes. (Nombreuses réservations etc.)

## Tests partiels

- Les tests de charge bien que positifs, ont été limités par les contraintes de ports disponibles sur le système local (MaxPort = 65000 et TimeWait = 30s). Une orchestration avec Docker et Kubernetes serait nécessaire pour des tests à plus grande échelle.
- Le manque de données réelles empêche une validation complète des scénarios de réservation, notamment en termes de fiabilité des temps de réponse ou d'intégrité des processus.

## Conformité réglementaire à valider

- Bien que la sécurité soit respectée (HTTPS, données anonymisées), un audit plus approfondi serait nécessaire pour garantir la conformité aux normes de gestion des données médicales à travers l'ensemble des services utilisés, notamment RGPD, HIPAA et NHS.

## Goulot d'étranglement dans le calcul des trajets

- Problème de scalabilité : Le service GPS, utilisé pour estimer les délais de trajet, devient un goulot d'étranglement dans des scénarios à grande échelle, notamment lorsque des districts comptent plusieurs milliers d'hôpitaux. Le traitement séquentiel actuel des requêtes allonge significativement les temps de réponse.
- Limites des solutions GPS externes : Les solutions gratuites offrent des performances limitées, souvent restreintes en nombre de requêtes ou sujettes à des délais élevés. De plus, la dépendance à des tiers expose également l'application à des interruptions de service ou des variations de qualité non maîtrisées.

## 5.3 Opportunités

### Amélioration des performances et de la scalabilité

- L'intégration de Docker et Kubernetes permettrait d'effectuer des tests de charge étendus et d'assurer une meilleure résilience en production.

### Extension des fonctionnalités

- Ajout d'un suivi des réservations en temps réel pour les utilisateurs professionnels.
- Ajout d'un historique des réservations permettant des annulations ou modifications.

- Intégration de tableaux de bord analytiques pour fournir des statistiques sur les performances des hôpitaux ou la disponibilité des lits.

### Validation avec des données réelles

- Collaboration avec des établissements hospitaliers pour intégrer des flux de données réels, tester les processus à l'échelle et valider les résultats obtenus avec les APIs hospitalières.

### Amélioration de l'expérience utilisateur

- Révision de l'interface frontend pour inclure des éléments interactifs avancés comme des notifications en temps réel, des suggestions basées sur l'historique ou des fonctionnalités d'accessibilité.

### Développement d'une solution GPS interne optimisée

- Indépendance vis-à-vis des tiers, une solution interne supprimerait les restrictions imposées par les services GPS externes, tout en offrant un contrôle total sur la qualité et la disponibilité.
- Optimisations possibles : L'implémentation d'algorithmes multithread pour un traitement parallèle efficace, coupler à la prise en charge du suivi en temps réel des incidents, travaux et trafic pour des délais plus précis. Le tout pourrait être renforcé par l'intégration de modèles prédictifs basés sur les données historiques pour anticiper les conditions de circulation.
- Réduction des coûts à long terme, bien que nécessitant un investissement initial, une solution maison réduirait les coûts récurrents liés aux appels API payants.

## 6. Recommandations

### 6.1 Améliorations techniques

#### Renforcement des mécanismes de sécurité

- Intégrer des audits réguliers pour détecter et corriger les éventuelles vulnérabilités dans les communications API et les flux de données.
- Logs conformes au RGPD pour le monitoring des communications API REST. Implémenter un système de journalisation qui suit les interactions des API tout en respectant les exigences du RGPD. (Anonymisation des données sensibles, traçabilité des opérations, accès contrôlé).

- Rotation et rétention des logs : Mettre en place une stratégie de rétention pour limiter la durée de conservation des logs tout en respectant les obligations réglementaires. Par exemple, conserver les logs uniquement pour une période définie (ex. 30 jours) avant suppression automatique.

### **Kubernetes et extension de cluster :**

- Implémenter la conteneurisation.
- Implémenter une stratégie d'autoscaling basée sur les ressources consommées (CPU, RAM) et le volume des requêtes utilisateurs, assurant ainsi une scalabilité horizontale sans interruption de service.

### **Optimisation des calculs GPS et gestion des trajets :**

- Traitement multithread des requêtes GPS, en implémentant des pools de threads pour paralléliser les calculs et réduire les temps de réponse dans des scénarios de forte charge.
- Utilisation de caches pour les trajets fréquents, stocker temporairement les résultats des calculs de trajets courants pour éviter des recalculs inutiles et réduire la charge.
- Automatisation du suivi des données dynamiques et intégrer des flux en temps réel pour surveiller les conditions de circulation, incidents ou travaux, afin d'ajuster les délais calculés.

### **Optimisation des performances :**

Afin d'améliorer la performance et la résilience de l'application dans un environnement réel, il est recommandé de mettre en œuvre des pratiques avancées de gestion des caches :

- Intégration d'une solution de cache évoluée, comme Caffeine. Avec des fonctionnalités avancées pour l'expiration des données, l'éviction et le monitoring.
- Amélioration de la robustesse avec des caches thread-safe. L'utilisation étendu de « ConcurrentHashMap » pour garantir les accès sécurisés dans un environnement multi-thread

### **Tests et validations d'environnement :**

Pour garantir une expérience utilisateur cohérente sur différents appareils et scénarios de localisation, les points suivants sont recommandés :

- Tests sur appareils mobiles et tablettes, afin de valider la compatibilité et de la réactivité de l'interface utilisateur sur des appareils mobiles (Android, iOS) et tablettes et d'effectuer des tests de la géolocalisation basée sur les capteurs GPS intégrés pour vérifier la précision des résultats.



- Tests sur périphériques fixes avec localisation par IP : Valider les scénarios où les utilisateurs accèdent à l'application depuis des appareils fixes (ordinateurs de bureau, kiosques hospitaliers) et vérifier la précision de la localisation estimée basée sur l'adresse IP.
- Exécuter des tests sur des configurations réseau variées (Wi-Fi, 4G/5G, réseaux à faible latence) pour garantir la stabilité des performances en simulant de multiple environnement.

## 6.2 Fonctionnalités supplémentaires

### Filtres avancés dans la recherche d'hôpitaux :

- Pour les cas spécifiques ajouter des options pour trier les résultats par équipements spécifiques, qualité des soins, spécialités multiples, ou encore capacité d'accueil en fonction du type de patient (pédiatrie, soins intensifs, etc.).

### Gestion multi-langues :

- Développer une interface adaptable en plusieurs langues pour rendre l'application accessible à des utilisateurs internationaux.
- Prendre en charge l'internationalisation dès le frontend et assurer le backend avec des messages API traduits.

## 6.3 Tests et validation

### Extension des tests de charge avec des données hospitalières réelles :

- Réaliser des scénarios de test plus représentatifs avec des volumes de données conformes à ceux des systèmes hospitaliers en production.

### Automatisation des tests end-to-end (E2E) :

- Mettre en place une suite complète de tests automatisés avec des outils comme Selenium ou Cypress pour valider les workflows critiques entre le frontend et le backend.

### Amélioration de la couverture des tests unitaires et d'intégration :

- Augmenter la couverture des branches de code pour le backend (actuellement à 74 %) afin de minimiser les zones non testées et d'assurer une meilleure robustesse.
- Étendre les tests à des cas limites et exceptionnels pour mieux anticiper les comportements inattendus.

## 6.4 Collaboration et documentation

### Amélioration de la documentation technique :

- Maintenir une documentation de déploiement pour permettre une prise en main rapide du projet par des tiers.

### Sensibilisation et formation des utilisateurs :

- Préparer un guide d'utilisation simplifié pour les professionnels de santé afin de faciliter leur adoption de la plateforme.

## 7. Références

### Figures :

- Figure 1 : Fonctionnements implémentés	09
- Figure 2 : Index.html créer par JaCoCo	10
- Figure 3 : Test de débit avec une condition de réponse de 200 ms	10
- Figure 4 : Test du délai moyen des réponses et du taux de bon service attribué	10
- Figure 5 : Test de couverture frontend Karma & Jasmine	11
- Figure 6 : Tests frontend Karma & Jasmine	11
- Figure 7 : Pipeline CI/CD avec Git Action	12