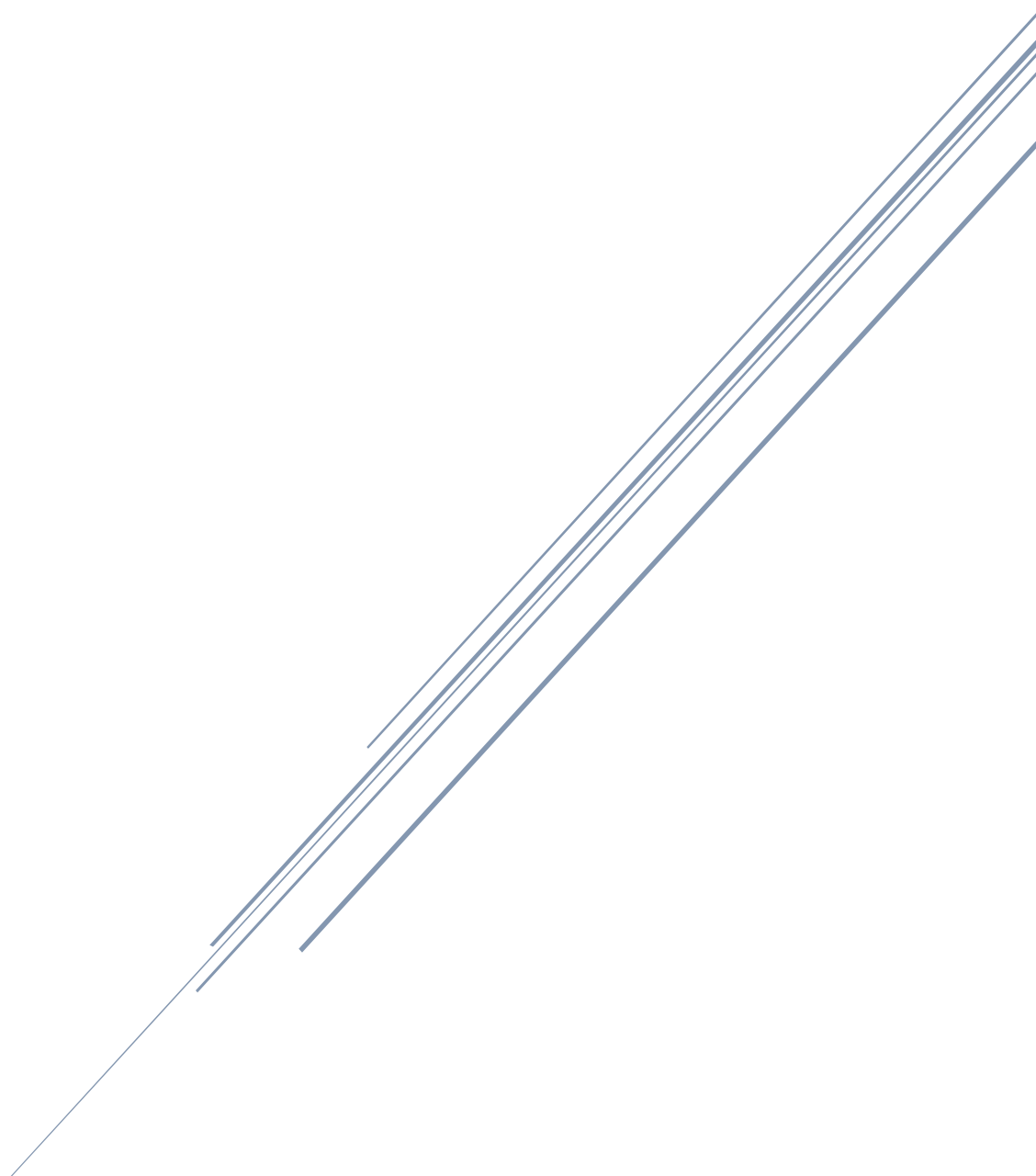


# RAPPORT DE LA POC

Architecte logiciel P11



OpenClassroom

Faite adhérer les parties prenantes avec un POC

# Rapport de la POC

## Table des matières

I.	Objet du document .....	2
II.	Stack technique .....	2
1.	Principe d'Architecture .....	2
2.	Justification des Technologies employées .....	2
3.	Conformité aux Normes .....	3
III.	Évaluation de la POC .....	4
4.	Tests unitaires et d'intégration .....	4
5.	Tests de charge.....	4
6.	Pipeline CI/CD .....	5
IV.	Sécurité .....	6
7.	Disponibilité .....	6
8.	Intégrité.....	6
9.	Confidentialité .....	6
10.	Traçabilité .....	7
V.	Résultat et enseignement de la POC .....	7
11.	Faisabilité de la solution.....	7
12.	Piste d'amélioration de la POC .....	7
VI.	Conclusion.....	8

# I. Objet du document

Ce document présente les résultats de la Preuve de Concept (POC) du système de routage hospitalier, développé dans le cadre de l'évaluation de faisabilité d'une plateforme de gestion et d'orientation des patients vers les établissements de santé.

L'objectif de ce POC est de démontrer la viabilité technique d'une architecture micro-services capable de gérer la recherche d'hôpitaux par spécialité, le calcul de distances routières en temps réel, et la réservation de lits disponibles, tout en respectant des contraintes de performance strictes (800+ requêtes/seconde avec un temps de réponse inférieur à 200ms).

Ce rapport détaille l'architecture mise en œuvre, les choix technologiques retenus (Spring Boot, Vue.js, OSRM), les résultats des tests de performance et de sécurité, ainsi que les recommandations pour une éventuelle mise en production. Il s'adresse aux décideurs techniques et métiers chargés d'évaluer la pertinence d'un déploiement à plus grande échelle du système.

## II. Stack technique

### 1. Principe d'Architecture

**Micro-services** : La POC est découpée en trois micro-services distincts correspondant chacun à une fonctionnalité unique :

- hospital-ui : Interface utilisateur
- hospital-service : Service traitant les informations des hôpitaux
- distance-service : Service effectuant des calculs de distances routières

**API REST** : Les différents services communiquent entre eux via une API dont les endpoints ont été standardisés de sorte qu'il n'y a pas « d'état session » côté serveur.

**Conteneurisation** : Chaque service est isolé dans un conteneur unique, leur environnement d'exécution ne dépendant donc pas du serveur qui les héberge mais de la configuration du conteneur associé. De plus, le déploiement s'en retrouve facilité puisque nous pouvons, en une commande à l'aide d'une certaine configuration, déployer l'ensemble de l'architecture.

**CI/CD** : Le projet est versionné sur git et utilise une CI/CD afin d'éprouver le code qui sera prochainement déployé. Cette CI éprouve la qualité du code via des tests unitaires et d'intégration pour chaque service, effectue des tests de sécurité et de charge, tout en vérifiant la qualité du code et la couverture des tests.

### 2. Justification des Technologies employées

**Backend – Spring Boot** : Il s'agit d'un framework java open source facilitant le développement de micro-services en offrant en outre une grande flexibilité, qui est très largement documenté et robuste.

**Base de données – H2 :** Cette base de données est régénéré à chaque déploiement (suffisant pour une POC) et se configure d'elle-même en fonction du code java associé. En outre un simple fichier sql permet d'ajouter facilement des données d'exemples.

**Frontend – Vue.js :** Framework javascript open-source utile pour la création d'interface utilisateur monopage, ce qui est le cas de notre POC. Il est en outre très largement documenté et simple à apprendre.

**Conteneurisation – Docker :** Le plus gros avantage d'utiliser docker est d'offrir une portabilité solide à la solution puisqu'elle n'est plus dépendante de la configuration de la machine qui l'héberge mais des conteneurs qui l'encapsule, soit un environnement dont nous pouvons contrôler l'installation des dépendances, suivre le statut (sain, arrêté, crash, etc) et permettant des échanges sur un réseau séparé interne à docker. De plus cela offre la possibilité de mettre un orchestrateur automatisant le déploiement et l'extensibilité de la solution, un exemple de déploiement autogérer est déjà disponible via les « docker compose ».

**Calcul de distance – OSRM (Open Source Routing Machine):** Service open source disposant d'une image docker afin de le monter en local dans l'architecture permettant un temps de réponse optimal (à noter qu'il existe également une api publique disponible qui suffit pour tester les fonctionnalités mais qui ne convient pas pour des tests de charge entre autre) dont les données bruts (carte routière) peuvent être obtenu gratuitement et traité dans des docker afin de les préparer pour le service local.

**Test de charge – K6 :** Disponibilité d'une image docker officiel, k6 est un outil open source gratuit qui peut suivre des tests rédigés en javascript, tout en offrant des métriques et des critères de validation configurable afin de mettre en place des validations automatiques.

### 3. Conformité aux Normes

**Respect du RGPD :** Pour l'heure cette POC ne stocke pas de données personnelles de quelque sorte que ce soit, cependant il est important de réfléchir au différent système à mettre en place afin de toujours respecter cette norme et de garantir la sécurité des données des utilisateurs, c'est pourquoi les communications sont d'ores et déjà chiffrées avec un chiffrement TLS permettant de n'utiliser que des requêtes https et offrant une protection contre les attaques de type « man in the middle ».

Par la suite, nous pourrions mettre en place d'autres mesures comme le chiffrement des données au repos lorsque des bases de données séparées seront mises en place, l'authentification des utilisateurs de la solution ou encore la pseudonymisation des données.

### III. Évaluation de la POC

#### 4. Tests unitaires et d'intégration

Afin de garantir la fiabilité de la solution, des tests unitaires et d'intégrations ont été développés pour chaque service afin de s'assurer que les différentes fonctionnalités de ces derniers fonctionnent comme prévu.

Les tests unitaires ont pour objectif de vérifier le comportement d'une unité de code isolé (méthode ou classe) indépendamment de ses dépendances externes. Nous validons ainsi qu'indépendamment tous les services fonctionnent correctement.

Les tests d'intégration vérifient quant à eux que plusieurs composants fonctionnent correctement ensemble, en testant leurs interactions réelles. L'objectif est de valider que l'intégration entre modules est correcte.

Ces deux types de tests font partie de la pyramide des tests, les tests unitaires en sont la base, puis vient les tests d'intégration et enfin les tests E2E (end to end).

Les tests sont automatiquement lancés à chaque build afin de s'assurer continuellement que le code est fonctionnel, pour ce faire il faut évidemment développer les tests en même temps que les fonctionnalités auxquels ils se rapportent. Il est important à noter également que les tests peuvent être positifs (vérifier si le comportement est celui attendu avec des paramètres attendus) ou négatifs (vérifier le comportement en cas de paramètre erroné ou de mauvais type).

#### 5. Tests de charge

Dans le but d'éprouver la solution dans son ensemble des tests de charges ont été réalisés via l'outil K6. Les critères de la POC étaient un temps de réponse de moins de 200ms pour un taux de requête de 800 par seconde.

Ils en retournent que les résultats dépendent du nombre d'hôpitaux contenu dans la BDD de test. Voici quelques données de comparaison :

- Avec 2 hôpitaux : 30ms à 50ms et + de 1000 req/s
- Avec 10 hôpitaux : 100ms à 150ms et ~430 req/s
- Avec 32 hôpitaux : + de 200ms et ~250 req/s

La raison de données aussi hétérogène vient du fait de la procédure de test. Afin de garantir que les requêtes soient traitées par OSRM, un client local devait être mis en place et héberger dans docker au même titre que la solution et le conteneur de k6 générant les requêtes.

La puissance de calcul de la machine de test a donc été le limitant absolu ici puisqu'il fallait partager les ressources entre OSRM, la solution et k6 (k6 n'étant clairement pas le plus gros demandeur).

La raison pour laquelle les résultats dépendent du nombre d'hôpitaux viens du fait que nous devons parcourir la liste de tous les hôpitaux possédant la spécialité cible dans le but de déterminer celui qui est le plus proche des coordonnées spécifiées.

Chaque hôpital supplémentaire ajoute un appel de calcul de distance à OSRM et démultiplie donc la charge. Pour 2 hôpitaux, le service hospital reçoit 1000 requêtes qui engendre 2000 demandes de calcul de distance à OSRM, pour 10 hôpitaux, il s'agit de 10000 demandes à OSRM, etc.

Une piste pour améliorer cette valeur serait :

- Augmenter le nombre d'instance d'OSRM : En répartissant la charge sur plusieurs instances le temps de réponse s'en trouvera diminuer. Prenons un exemple, 10 instances d'OSRM et 10 hôpitaux où le calcul de distance de chaque hôpital est confié à une instance différente, le temps de réponse sera réduit de 10 fois le temps de calcul à un seul grâce à de la parallélisation.

## 6. Pipeline CI/CD

Le pipeline CI/CD mis en place apporte plusieurs avantages pour la solution et la gestion du projet dans son ensemble :

- Exécutions des tests unitaires/intégrations
- Scan de vulnérabilité
- Analyse de qualité de code
- Build des images docker
- Test de charge

Avec toutes ces phases automatisées, nous nous assurons que la version de la solution qui est testé est fiable, viable et sécurisé. Nous pouvons ainsi détecter rapidement toute faille avant d'en faire les frais une fois déployé en production. Le pipeline s'inscrit dans la dynamique du workflow git (cf le README du git du code pour plus de détail) de sorte que ce dernier se déclenche automatiquement sur :

- Toute branche de développement de fonctionnalité/bugfix qui ferait l'objet d'une pull request vers la branche « dev-main »
- Tout push (nouveau commit) sur la branche « dev-main »

Comme la branche « dev-main » est la seule à pouvoir être fusionné avec la branche « main » (la branche de release/production) alors nous sommes sûr que le pipeline à déjà été lancer sur le code qui sera livrer.

En outre, chaque exécution du pipeline est documentée avec divers artefacts comme le taux de couverture des tests, les résultats des différents tests effectuer, etc.

## IV. Sécurité

### 7. Disponibilité

Pour garantir une haute disponibilité à la solution, nous préconisons de d'utiliser un moyen de conteneurisation comme docker offrant plusieurs avantages :

- Déploiement sur n'importe quelle machine pour peu que docker soit installé
- Suivi de l'état des services individuellement
- Intégration d'un orchestrateur pour redémarrage automatisé d'une instance défaillante et répartition de la charge entre les différentes instances (instance pouvant être montée spécialement pour répondre à un pic d'influence)

L'architecture micro-service est celle qui convient le mieux à ce type d'outil puisque chaque fonctionnalité est compartimentée dans un service, lui-même compartimenté dans un conteneur. Ils peuvent être reliés par un réseau interne et non exposé à la machine hôte (cela ne dispense pas de chiffrer les communications inter-service, ce n'est qu'une sécurité supplémentaire).

### 8. Intégrité

L'intégrité des livraisons de version est éprouvée par les différents tests que chaque version subit au cours du déroulement du pipeline de CI/CD. Cela nous permet de réduire au maximum la livraison/mise en production d'une version défaillante, cependant ces mesures ne sont pas absolues et doivent être préservées d'une certaine déviance comme :

- La création de test pour atteindre 100% de couverture de code : Cette statistique ne doit pas devenir un but en soi, il s'agit d'une indication mais les tests doivent avant tout être créés pour s'assurer du bon fonctionnement de la solution.
- L'oubli de test : L'inverse est également vrai, il ne faut pas déployer du code qui n'aura pas eu de tests créés spécifiquement pour lui. Le pipeline pourrait annoncer que les tests ont réussi, sans préciser que ces derniers ne couvrent pas les nouvelles fonctionnalités.

Il faut également garantir l'intégrité des BDD qui seraient déployés avec la réalisation de sauvegarde cyclique et la mise en place d'un plan de remontée de ces sauvegardes pour minimiser l'impact de toute défaillance dans les données.

### 9. Confidentialité

La confidentialité est très importante dans toute solution, mais encore plus lorsque l'on traite de données médicales, aussi plusieurs mesures sont ou seront mises en place afin de protéger la solution :

- Chiffrement des données en transit : Chiffrement TLS mis en place dans le cadre de la POC mais il s'agit là de certificat autosigné

insuffisant pour un déploiement en production. Il faudra passer à un chiffrement mTLS (mutual TLS) où chaque point terminal d'une requête (émetteur et récepteur) vérifie l'identité de leur correspondant.

- Chiffrement des données au repos : Il faudra, lorsque des BDD séparé seront mis en place, chiffrer les données lorsqu'elles sont au repos afin que, même en cas de fuite, elle soit inutilisable.
- Mise en place d'une gestion utilisateur et RBAC : Avec des profils utilisateurs gérer et défini nous pourrons limiter l'accès de chaque utilisateur en fonction de son rôle, de même que ses droits sur la solution (principe du moindre privilège).

## 10. Traçabilité

Toutes les actions menées sur la solution devront être documenter dans des logs (il faudra donc mettre en place un service de centralisation des logs), pour ce faire nous pouvons imaginer rendre l'architecture hybride avec de l'évènementiel. Toute action menée fera alors l'objet d'un événement qui pourra être récupérer par le service de centralisation des logs. De plus, il est facile d'imaginer héberger un bus d'évènement dans un conteneur docker, ce qui facilitera la mise en place d'un tel procédé.

# V. Résultat et enseignement de la POC

## 11. Faisabilité de la solution

La POC qui a été développer répond aux différentes exigences qui ont été émise par le consortium MedHead sont soit respecté, soit font l'objet d'amélioration possible pour y répondre. Le workflow git et le pipeline CI/CD permettent de développer efficacement la solution, là où docker permet de la déployé partout. Nous avons donc une solution portable, disponible, permettant une extensibilité et une grande évolution.

Nous pensons donc que la POC valide la faisabilité du système d'intervention d'urgence en temps réel.

## 12. Piste d'amélioration de la POC

Bien évidemment, il ne s'agit là que d'une POC et non d'une solution complète donc beaucoup de fonctionnalité ont été ignorer et sont manquante pour l'utiliser en tant que tel, cependant voici des améliorations intéressantes pour la rendre fonctionnels :

- Mise en place d'une gestion des utilisateurs (login/mot de passe)
- Mise en place d'une gestion des rôles (RBAC)
- Tests E2E (end to end)
- Hybridation de l'architecture avec de l'évènementiel

- Mis en place d'outil pour améliorer la traçabilité
- Mise en place de BDD dans des conteneurs séparé
- Chiffrement des BDD
- Sauvegarde cyclique et plan de remontage à partir d'une sauvegarde des BDD
- Anonymisation/pseudonymisation des données
- Mise en place d'un orchestrateur comme Kubernetes

## VI. Conclusion

La preuve de concept (POC) développé à donc réussi à démontrer la faisabilité technique de ce projet de système d'intervention d'urgence, tout en démontrant la pertinence des technologies choisies. De plus, il sera possible pour une équipe de développement de prendre la suite de cette POC afin de développer la version aboutie de la solution en suivant les méthodologies appliquées lors de ce projet, ainsi que les outils employés.