

Mastère expert développement logiciel

Seddik Mehdi – Ynov 2025

**Livrable Bloc 4**

CONCEVOIR ET DÉVELOPPER DES APPLICATIONS LOGICIELLES

Seddik Mehdi

Table des matières

[A.4.1 Monitorer l’application logiciellE 4](#_Toc206620605)

[C4.1.1 Gérer les mises à jour des dépendances 4](#_Toc206620606)

[C4.1.2 Concevoir un système de supervision et d’alerte 4](#_Toc206620607)

[A.4.2 Traitement des anomalies détectées en production 4](#_Toc206620608)

[C4.2.1 Consigner les anomalies détectées 4](#_Toc206620609)

[C4.2.2 Créer et déployer un correctif 4](#_Toc206620610)

[A.4.3. Assurer la maintenance du logiciel 4](#_Toc206620611)

[C4.3.1 Proposer des axes d’amélioration 4](#_Toc206620612)

[C4.3.2 Établir un journal des versions déployées 4](#_Toc206620613)

[C4.3.3 Collaborer avec les équipes de support 4](#_Toc206620614)

[Introduction 5](#_Toc206620615)

[A2.1 Préparation des environnements de développement et de test 5](#_Toc206620616)

[C2.1.1 Mettre en œuvre des environnements de déploiement et de test 5](#_Toc206620617)

[Environnement de déploiement 5](#_Toc206620618)

[Protocole de déploiement continue 6](#_Toc206620619)

[Environnement de test 6](#_Toc206620620)

[C2.1.2. Configurer le système d’intégration continue 7](#_Toc206620621)

[Protocole d’integration continue 7](#_Toc206620622)

[A2.2 Développement des fonctionnalités 7](#_Toc206620623)

[C2.2.1. Concevoir un prototype de l’application logicielle 7](#_Toc206620624)

[Technologies, frameworks et bonnes pratiques 7](#_Toc206620625)

[Architecture logicielle 8](#_Toc206620626)

[Présentation d’un des prototypes réalisé 9](#_Toc206620627)

[C2.2.2 Développer un harnais de test unitaire 9](#_Toc206620628)

[Outils de tests 9](#_Toc206620629)

[Couverture des tests 9](#_Toc206620630)

[C2.2.3. Développer le logiciel en veillant à l'évolutivité et à la sécurisation du code 10](#_Toc206620631)

[Mesures de sécurité mises en œuvre 10](#_Toc206620632)

[Mesures d’accessibilité mise en œuvres 11](#_Toc206620633)

[C2.2.4. Déployer le logiciel à chaque modification de code 12](#_Toc206620634)

[Système de gestion de versions 12](#_Toc206620635)

[Historique des évolutions 12](#_Toc206620636)

[A2.3 Recette des fonctionnalités 13](#_Toc206620637)

[C2.3.1 Élaborer le cahier de recettes 13](#_Toc206620638)

[C2.3.2 Élaborer un plan de correction des bogues 13](#_Toc206620639)

[A2.4 Rédaction de la documentation technique 14](#_Toc206620640)

[C2.4.1 Rédiger la documentation technique 14](#_Toc206620641)

[Manuel de déploiement 14](#_Toc206620642)

[Manuel de mise a jour 14](#_Toc206620643)

[Glossaire 15](#_Toc206620644)

[Annexes 15](#_Toc206620645)

# A.4.1 Monitorer l’application logiciellE

## C4.1.1 Gérer les mises à jour des dépendances

Afin de garantir la sécurité et la stabilité de notre application, nous avons mis en place un processus automatisé de gestion des mises à jour des dépendances via **Dependabot**.

Cet outil est configuré pour analyser **quotidiennement** notre dépôt GitHub, aussi bien pour le frontend que pour le backend. Lorsqu’une nouvelle version d’une dépendance est disponible, Dependabot ouvre automatiquement **une Pull Request** (PR) dédiée, ce qui nous permet de :

* Mettre à jour régulièrement les bibliothèques et frameworks que nous utilisons, sans intervention manuelle.
* Identifier et corriger rapidement les failles de sécurité : Dependabot nous signale et propose immédiatement une mise à jour critique lorsqu’une vulnérabilité est détectée dans une version connue.
* Intégrer les mises à jour de manière sécurisée : chaque PR générée est soumise à notre pipeline d’intégration continue (CI), incluant les tests automatisés, afin de s’assurer que les mises à jour n’introduisent pas de régressions.

Grâce à cette organisation, nous maintenons notre application à jour, sécurisée et conforme aux bonnes pratiques de maintenance logicielle.

## C4.1.2 Concevoir un système de supervision et d’alerte

Étant donné que notre logiciel est construit selon une architecture **microservices** et déployé sur **Kubernetes**, nous avons choisi de mettre en place une solution de supervision adaptée à ce type d’environnement distribué.

Pour simplifier l’installation et la configuration, nous avons utilisé un **chart Helm** qui déploie automatiquement :

* **Prometheus**, pour la collecte des métriques,
* **Grafana**, avec des tableaux de bord préconfigurés,
* ainsi que la configuration réseau nécessaire pour assurer leur interconnexion.

Kubernetes expose également des **API natives** qui nous permettent de suivre en temps réel l’état du cluster (nombre de pods actifs, réplicas, containers, etc.). Cela nous donne une vision claire de la santé globale de l’orchestrateur et de nos microservices.

Nous avons identifié comme essentielles les métriques suivantes, suivies au niveau de chaque **nœud du cluster** :

* **Usage CPU**,
* **Usage RAM**,
* **Usage disque**,
* **Usage réseau**.

Pour cela, nous utilisons **Node Exporter**, qui collecte directement les informations via le système d’exploitation et les expose sur une route HTTP compatible avec Prometheus. Prometheus vient ensuite **scraper** ces données afin que Grafana puisse les interroger et les afficher sous forme de graphiques exploitables.

En complément, nous avons intégré **Uptime Kuma**, un outil dédié au suivi de la **disponibilité applicative**. Il nous permet de nous assurer que l’application reste accessible en continu et d’être alertés immédiatement en cas d’interruption.

Grâce à cette organisation, notre système de supervision nous offre une visibilité complète, aussi bien au niveau **infrastructure** (cluster Kubernetes et ses ressources) qu’au niveau **applicatif** (disponibilité des services).

# A.4.2 Traitement des anomalies détectées en production

## C4.2.1 Consigner les anomalies détectées

## C4.2.2 Créer et déployer un correctif

# A.4.3. Assurer la maintenance du logiciel

## C4.3.1 Proposer des axes d’amélioration

## C4.3.2 Établir un journal des versions déployées

## C4.3.3 Collaborer avec les équipes de support

# Introduction

L'objectif principal de ce projet est de concevoir et développer une plateforme qui permet aux « quants » (abréviation pour « quantitatives », des professionnels responsables de la prise de décision financière dans les institutions de finance, notamment via des formules mathématiques complexe et des algorithmes informatique) de visualiser l’état des marchés, et de déployer efficacement leurs algorithmes de trading automatisés. La plateforme Delta répond à deux besoins majeurs: l'affichage de graphiques financiers, ainsi que la gestion complète des algorithmes, incluant le déploiement, le logging et le démarrage/arrêt.

Cette plateforme est conçue pour automatiser les tâches DevOps et de gestion des algos en production, permettant ainsi aux quants de se concentrer sur le développement et l'amélioration des algorithmes. Le but ultime de ce projet est de faciliter la collaboration entre les développeurs, les quants et les opérateurs pour optimiser les performances des algorithmes de trading.

Dans ce dossier, nous présentons la phase de développement du projet.

# A2.1 Préparation des environnements de développement et de test

## C2.1.1 Mettre en œuvre des environnements de déploiement et de test

### Environnement de déploiement

Pour créer un environnement de développement fonctionnel, nous utilisons une combinaison d'outils essentiels :   
En premier lieu, **Git** et **GitHub** pour gérer versionner et collaborer avec d'autres développeurs. Vous pouvez retrouver le dépôt Github à cet url => <https://github.com/mhd-sdk/delta>  
Comme éditeur de code nous utilisons principalement **VSCode**, mais certaines parties ont été écrite sur **Neovim** (il s’agit d’un éditeur de texte terminal beaucoup plus performant et amusant que VSCode)

Pour gérer les dépendances nous avons choisi **Yarn** comme gestionnaire de packages, connu pour être une excellente alternative à npm notamment pour sa rapidité. Pour exécuter les tests unitaires du frontend nous avons utilisé **Jest**, et pour le backend, le langage **GO** a déjà un système de test builtin.  
Pour lancer certains services sans avoir à les compiler nous utilisons **Docker**, ce qui nous permet via la commande Docker compose de démarrer plusieurs services avec leurs dépendances nécessaires en quelques clics.

L’infrastructure repose sur une approche de containerisation via Docker, orchestrée au sein d’un cluster Kubernetes déployé sur un homelab, une machine personnelle servant de serveur. Cette architecture permet d’expérimenter et de mettre en place un pipeline complet, allant de l’intégration continue jusqu’au déploiement continu, tout en garantissant un haut niveau de reproductibilité et de fiabilité.

[PHOTO DE MON HOMELAB]

### Protocole de déploiement continue

A chaque nouvelles versions du logiciel, le serveur de production est mis à jour automatiquement, le protocole est le suivant :

1. Les développeurs écrivent du code, et créent des **Pull request sur la branche main**
2. A chaque Pull Request, l’équipe QA teste les développements, et valident ou invalide le ticket
3. Une fois Validé, la Pull Request est mergée sur main
4. Chaque fin d’itération (un sprint dure 2 semaines), le lead dev crée une **release** sur GitHub, ce qui déclenche une Pipeline de déploiement automatisé
5. La pipeline **build les images docker**, **exécute les tests**, et **déploie l’application** sur le serveur de production

### Environnement de test

Un environnement de **préproduction** est pour nous une étape indispensable dans le cycle de vie du logiciel. Il s’agit d’un espace technique qui reproduit fidèlement les conditions de la production, mais sans impacter les utilisateurs finaux. Nous l’utilisons comme zone de validation avant toute mise en ligne : nous y déployons nos nouvelles fonctionnalités, nous y corrigeons les anomalies, et nous y réalisons des tests de compatibilité, de performance ou encore de sécurité. L’objectif est de disposer d’une image la plus proche possible de la production afin d’anticiper les problèmes et de réduire les risques liés au déploiement final.

Dans notre cas, nous avons choisi d’exploiter un **namespace Kubernetes** dédié à la préproduction. Un namespace est un espace logique isolé à l’intérieur du cluster qui nous permet de séparer et d’organiser nos ressources (pods, services, secrets, configurations, etc.). Le fait d’avoir un namespace propre à la préproduction nous garantit un cloisonnement clair par rapport à la production.

Grâce à un pipeline GitHub, nous mettons à jour automatiquement cet environnement à chaque évolution du code. Dès qu’une fonctionnalité est développée et mise en Pull Request sur GitHub, elle est déployée dans le namespace de préproduction, ce qui nous assure une synchronisation continue entre notre travail de développement et les tests. De cette manière, nous détectons rapidement les régressions éventuelles et nous validons la qualité des livrables avant toute mise en production.

Pour évaluer la pertinence et la robustesse de notre environnement de préproduction, nous suivons plusieurs critères de qualité et de performance :

* Stabilité : l’environnement doit rester fiable et fonctionnel, même lors de déploiements fréquents.
* Reproductibilité : il doit refléter au plus près la configuration de la production pour que les résultats de tests soient significatifs.
* Automatisation : les déploiements, mises à jour et éventuels rollbacks doivent être gérés automatiquement par la pipeline.
* Observabilité : nous devons pouvoir analyser les logs, métriques et traces afin de comprendre rapidement le comportement de l’application et d’identifier les anomalies.
* Sécurité : la gestion des accès, des secrets et des communications doit respecter les mêmes standards que ceux appliqués en production.
* Performances : nous vérifions les temps de réponse, la consommation de ressources (CPU, mémoire) ainsi que la capacité de l’application à monter en charge.

## C2.1.2. Configurer le système d’intégration continue

### Protocole d’integration continue

En parallèle du déploiement continu, une stratégie **d’intégration continue (CI)** à été mise en place afin de garantir la qualité du code livré. Chaque **Pull Request** déclenche automatiquement une série de tests automatisés permettant de détecter d’éventuelles régressions avant la fusion dans la branche principale. De plus, un processus de vérification de la qualité du code est intégré via **ESLint** pour l’analyse statique et **Prettier** pour l’uniformisation du formatage. Ces outils, également configurés comme extensions dans Visual Studio Code, assurent une cohérence de style et une meilleure lisibilité du code. Enfin, ces contrôles sont validés au sein d’une **pipeline d’intégration continue**, qui centralise les résultats et bloque la fusion si les tests ou les règles de linting échouent, garantissant ainsi la robustesse et la maintenabilité du projet. Une image contenant texte, logiciel, Police, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Figure 1: Exemple de pipelines sur une Pull Request

# A2.2 Développement des fonctionnalités

## C2.2.1. Concevoir un prototype de l’application logicielle

### Technologies, frameworks et bonnes pratiques

Pour développer ce projet, nous avons choisi une approche polyvalente qui combine les avantages de différentes technologies. Nous sommes particulièrement fiers de notre choix pour les technologies suivantes :

Le frontend est construit avec React et TypeScript. Nous avons opté pour ces technologies car elles offrent une excellente combinaison de performances, de flexibilité et de facilité d'utilisation. La bibliothèque de composants React Shadcn nous permet également un design UX/UI plus raffiné et une meilleure gestion des interfaces utilisateur complexes.

Le backend est écrit en Golang, un language qui a gagné en popularité ces dernières années en raison de sa vitesse, de son efficacité et de sa facilité d'utilisation. Nous avons choisi Golang pour son écosystème riche, ses bibliothèques puissantes et sa communauté active. Cette langue est également particulièrement adaptée au développement cloud native, car les programmes écrits en GO sont très faciles à containeriser, et la majorité des outils cloud sont écrits avec ce langage (par exemple Kubernetes, Docker, etc...).

Pour stocker les logs des algorithmes, nous avons choisis la base de données PostgreSQL, nous aurions pu utiliser une base de données Timeseries étant donné que nous n’avons pas de donnée relationnelle, mais étant donné que PostgreSQL permet les deux, il nous a semblé judicieux de partir dessus si jamais nous avons besoin d’étendre les fonctionnalités de l’application.

Pour permettre une communication fiable entre le backend et les algorithmes, nous avons choisi le protocole gRPC. Cette technologie offre une haute performance (plus rapide que les interfaces REST, car elle utilise un protocole qui envoie des données compressées en binaire).

Pour ce projet, nous avons fait le choix de déployer l'application dans un cluster Kubernetes pour gérer les conteneurs et assurer leur déploiement automatisé. Cette technologie offre une excellente scalabilité et permet d'automatiser une multitude de tâches telles que le déploiement, le scaling horizontal, la gestion des ressources et la surveillance de l'application.

### Architecture logicielle

Notre projet est basé sur une architecture micro-services, qui consiste en un ensemble d'unités fonctionnelles indépendantes, chacune ayant une responsabilité spécifique. Dans notre cas, le backend est composé d'un service maitre et de plusieurs services esclaves, chacun représentant un algorithme différent. (Voir l’architecture complète en Annexe 1.)

Le service maitre reçoit les requêtes du frontend et les dispatches sur les algorithmes concernés. Ce service est responsable de la gestion des requêtes entrantes et de la coordination entre les différents services esclaves. Il s'assure que les données nécessaires sont disponibles pour chaque algorithme et qu'ils soient exécutés de manière efficace.

Les services esclaves, quant à eux, représentent chaque algorithme individuel. Ils sont responsables de l'exécution spécifique du traitement des requêtes, en fonction de leur domaine d'expertise. Chaque service exclave est containerisé pour assurer une isolation et une sécurité maximales. Le développement de ces algorithmes sont sous la responsabilité des Quants, nous fournirons uniquement des programmes mocké en guise d’exemple.

Le déploiement et la gestion des services sont gérées par un cluster Kubernetes, qui permet de déployer les conteneurs automatiquement et de gérer les ressources en temps réel. Cette architecture nous permet de répondre rapidement aux besoins évoluant du projet et d'améliorer l'efficacité globale de l'application.

Dans ce modèle, chaque service maitre et esclave est conçu pour fonctionner de manière autonome, mais également pour interagir avec les autres services pour atteindre le résultat souhaité. Cette architecture nous permet d'améliorer la flexibilité, la scalabilité et la résilience globale de l'application.

L'utilisation d'une architecture micro-services offre plusieurs avantages, notamment :

Flexibilité : chaque service est indépendant et peut être mis à jour ou modifié sans affecter les autres services, c’est très pratique pour la mise à jour d’un algorithme sans devoir redéployer l’application complète.

Scalabilité : les conteneurs permettent une gestion efficace des ressources et une évolution de la charge de travail, si par exemple un algorithme est surchargé, on peut facilement le dupliquer, voir créer un réplica sur une autre machine du cluster.

Résilience : si un service est indisponible, les autres peuvent prendre son relais pour continuer à fonctionner.

### Présentation d’un des prototypes réalisé

Voici quelques captures d’écran du prototype,

[Capture d’écran du dashboard]

[Capture d’écran de la page des algorithmes]

## C2.2.2 Développer un harnais de test unitaire

### Outils de tests

Pour le **frontend**, nous utilisons **Jest**, un framework de tests JavaScript très populaire dans l’écosystème React et plus largement dans les projets front modernes. Il offre une configuration minimale, une exécution rapide grâce à son runner optimisé, et un écosystème riche permettant d’écrire aussi bien des tests unitaires que des tests d’intégration. Avec ses fonctionnalités comme les *mocks* automatiques, les tests instantanés (*snapshots*) et une interface claire pour les rapports d’erreurs, Jest facilite la mise en place d’une couverture de test robuste et maintenable côté client.

Pour le **backend**, nous nous appuyons sur la **toolchain de Go**, qui fournit nativement un environnement de test intégré. Go propose une commande go test qui permet d’exécuter des tests unitaires et de benchmarker le code sans dépendances externes. Cette approche, fidèle à la philosophie de Go de simplicité et d’efficacité, permet d’écrire des tests lisibles, rapides et directement intégrés au cycle de développement. De plus, l’outillage standard prend en charge la génération de rapports de couverture, le profiling et l’intégration facile dans les pipelines CI/CD.

### Couverture des tests

Pour le frontend avec la commande Yarn test

Une image contenant texte, Police, capture d’écran, nombre

Le contenu généré par l’IA peut être incorrect.

Pour le backend avec la commande go test -cover ./…

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Nous ajoutons des tests à chaque nouvelle features, ces tests sont automatiquement exécutés sur les pipelines à chaque Pull Request et nous souhaitons ajouter des tests supplémentaires de type end to end une fois le produit terminé pour valider la qualité du développement.  
Aussi, Le coverage n’affiche pas 100% partout, mais c’est tout à fait normal, il est difficile de mock des fonctions qui font des appels réseaux, ou qui s’abonnent à des flux de données externe.

## C2.2.3. Développer le logiciel en veillant à l'évolutivité et à la sécurisation du code

### Mesures de sécurité mises en œuvre

La dimension sécurité à été intégrée dès la phase de conception afin de protéger les données sensibles et de garantir la fiabilité, la confidentialité et l’intégrité de l’application. Conformément aux bonnes pratiques de développement sécurisé, plusieurs mesures techniques ont été mises en œuvre :

* **Gestion des secrets** : Les mots de passe, clés API et autres informations sensibles sont stockées dans GitHub Secrets, évitant toute exposition dans le code source ou les commits. Cette pratique respecte les recommandations de sécurité des infrastructures cloud et limite le risque de fuite accidentelle.
* **Authentification biométrique via WebAuthn** : Une fonctionnalité d’authentification renforcée a été développée, permettant aux utilisateurs de se connecter à l’application grâce à leur empreinte digitale. L’intégration de WebAuthn (FIDO2) assure un niveau élevé de sécurité en remplaçant les mots de passe traditionnels par une authentification sans mot de passe, résistante aux attaques par phishing et aux compromissions de base de données.
* **Mise à jour automatisée des dépendances** : L’outil **Dependabot** de GitHub est activé pour surveiller en continu les bibliothèques tierces utilisées dans le projet. En cas de vulnérabilité critique détectée (ex : CVE), un pull request automatique est généré, alertant l’équipe pour une correction rapide. Ce processus contribue à maintenir l’application à jour et sécurisée selon les principes de DevSecOps.
* **Infrastructure sécurisée en production** : Le déploiement en production repose sur un domaine personnalisé (DNS) acquis via un service de nom de domaine (CloudFlare). Un certificat SSL/TLS (via Let’s Encrypt) a été mis en place pour chiffrer toutes les communications entre l’utilisateur et le serveur, garantissant un accès sécurisé via HTTPS. Cette mise en œuvre répond aux exigences de conformité web (RGAA, GDPR) et aux bonnes pratiques de sécurité réseau.
* **Conformité aux 10 failles OWASP** : Le code a été conçu pour minimiser les risques liés aux principales vulnérabilités décrites par l’OWASP (ex : injection SQL, XSS, CSRF). Des contrôles de validation, d’encodage et d’autorisation sont implémentés au niveau des endpoints.

### Mesures d’accessibilité mise en œuvres

Pour veiller à l’accessibilité de l’application, nous avons mis en pratique les recommandations du RGAA, notamment les points suivants :

* **Percevoir l’information :** le contraste entre le texte et le fond est respecté (≥ 4,5:1) et la structure sémantique est correctement organisé (Titres avec les balises h1, h2, h3…, boutons avec la balise <a/>…)
* **Naviguer facilement :** Toute l’application est accessible au clavier, nous avons même mis en place une « omni bar » (voir annexe 4) qui permet de rechercher facilement une page ou une section dans toute l’application
* **Comprendre l’information :** Les champs de texte ont des labels, et la langue de la page est définie.
* **Interagir correctement :** Les erreurs de formulaire sont claires, il n’y a pas de clignotements (pour ne pas déclencher d’épilepsie) et les actions sont cohérente

Pour valider automatiquement l’accessibilité de l’application, nous avons utilisé le scanner open source « **Equal access accessibility checker** » de IBM, il nous a aidé à connaitre les faiblesses de notre application et nous comptons agrandir l’équipe avec des spécialistes pour résoudre les problèmes d’accessibilité manquant

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

## C2.2.4. Déployer le logiciel à chaque modification de code

### Système de gestion de versions

Au fur et à mesure que l’équipe de développement livre de nouvelles fonctionnalités, l’adoption d’une culture **DevOps** devient essentielle afin de garantir la fluidité et la fiabilité du cycle de développement. Le DevOps désigne un ensemble de pratiques visant à rapprocher les équipes de développement et d’exploitation, en automatisant et en intégrant les processus de livraison logicielle. Dans ce projet, un protocole de **déploiement continu (CD)** a été mis en place, permettant de maintenir le serveur de production régulièrement à jour avec la dernière version validée du logiciel. Pour ce faire, nous avons défini un système de versionning suivant le schéma **MAJEUR.MINOR.PATCH** et une stratégie de déploiement sur deux environnements distincts : préproduction et production.

Chaque Pull Request ouverte sur la branche principale déclenche automatiquement le déploiement d’une **release candidate** en préproduction, rendant ainsi les nouvelles fonctionnalités accessibles pour l’équipe QA, qui les valide une à une. Lorsqu’une version est jugée prête pour la mise en production, un **tag de release** est créé sur GitHub, ce qui déclenche le déploiement automatique de l’application en production. Afin d’assurer la traçabilité et la possibilité de **rollback en cas de problème**, les images Docker sont historisées et versionné sur **le registre GitHub Container Registry** (Voir en annexe 2).

### Historique des évolutions

Vous pouvez consulter l’historique des versions sur le dépôt GitHub, et nous avons également mis en place un système de release notes automatisé à chaque release GitHub :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

# A2.3 Recette des fonctionnalités

## C2.3.1 Élaborer le cahier de recettes

Vous pouvez trouver le cahier de recettes dans le même dossier sous le nom de « Cahier de recettes.pdf »

## C2.3.2 Élaborer un plan de correction des bogues

A chaque test effectué, un commentaire est rédigé par l’équipe QA qui valide ou invalide la fonctionnalité.  
Les bugs et axes d’amélioration font l’objet d’un ticket assigné à l’équipe technique, vous pouvez en trouver un exemple en Annexe 3

Voici une liste des bugs et améliorations à traiter ainsi qu’un plan de correction pour chacun d’entre eux

|  |  |  |  |
| --- | --- | --- | --- |
| Fonctionnalité | Test en echec | Priorité | Correction |
| Dashboard |  | | |
|  | **Sauvegarder le Dashboard.** | Élevé | Le Dashboard est pour le moment sauvegardé dans le local storage, il faut changer la structure de donnée pour pouvoir en stocker plusieurs. |
|  | **Charger un Dashboard sauvegardé.** | Élevé | Une fois la sauvegarde terminée, il faut développer un menu qui permet de sélectionner un dashboard. |
| Algorithmes |  | | |
|  | **Discovery des algorithmes sur le réseau** | Élevé | Le service maitre (backend principal) arrive bien à découvrir les algorithmes disponibles sur le réseau, l’équipe frontend est actuellement en train de développer l’interface qui permet de lancer cette feature. |
|  | **Activer un algorithme** | Critique | Le contrat Proto est déjà établi et fonctionnel, le backend expose une route api qui permet de changer l’état d’un algorithme via son id, il s’occupe en suite de router cette requête via gRpc, Il ne manque donc que la requête depuis le frontend. |
|  | **Désactiver un algorithme** | Critique |
| Autre |  | | |
|  | **S’authentifier avec une empreinte digitale** | Critique | La librairie qu’on utilise est disponible sur plusieurs languages, elle fonctionne très bien avec du javascript, mais pas avec du Go. Nous avons pensé dans un premier temps à développer un service d’authentification séparé en NodeJS, mais comme le token de session doit être stocké et accessible en mémoire dans le backend, nous avons reporté cette feature pour ne pas perdre trop de temps et nous concentrer sur les fonctionnalités principales. |

Pour éviter les potentielles régressions, l’équipe QA a également accès au serveur préprod, pour valider manuellement. Il est prévu dans le futur d’ajouter des test end2end avec playwright pour éviter à l’équipe QA de re-tester les mêmes features à l’infinie.

# A2.4 Rédaction de la documentation technique

## C2.4.1 Rédiger la documentation technique

### Manuel de déploiement

Cette application a pour vocation d’être déployé par nos soins sur l’infrastructure du client, nous avons mis en place un manuel de déploiement technique directement sur le dépôt GitHub, de sorte à le versionner et le maintenir en même temps que l’application.

Vous pouvez trouver ce manuel ici => <https://github.com/mhd-sdk/delta/blob/main/README.md>

### Manuel de mise a jour

# Glossaire

Annexes

Une image contenant capture d’écran, diagramme

Le contenu généré par l’IA peut être incorrect.

Figure 1: Schéma haut niveau de l'architecture logicielle

Une image contenant texte, capture d’écran, logiciel, Police

Le contenu généré par l’IA peut être incorrect.

Figure 2: Image docker hébergé sur le github container registry

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Figure 3: Exemple de ticket bug

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

Figure 4: Omni-bar