HealthGuard Vision

Diagnostic Préventif par Image

Application Mobile de Dépistage de Santé par Intelligence Artificielle

Rapport de Projet

Master 1 — Année Universitaire 2025–2026

Module	M1PROJ	**Durée**	5 jours
Méthode	Scrum / Jira	**Versioning**	Git / GitHub

TECHNOLOGIES CLÉS

Flask · React Native · TensorFlow Lite · MongoDB · Docker · Expo SDK 54

ÉQUIPE PROJET

Frontend Backend BDD ML / IA DevOps

Table des matières

Liste des figures

Liste des tableaux

1 Introduction

1.1 Contexte du projet

Dans le cadre du module **DevOps & Intégration Continue** du programme de Master 1, notre équipe a développé **HealthGuard Vision**, une application mobile innovante de diagnostic préventif par analyse d'images. Ce projet intensif sur 5 jours nous a permis de mettre en pratique l'ensemble des compétences acquises en développement logiciel, intelligence artificielle et méthodologies DevOps.

Le secteur de la santé numérique (*e-health*) connaît une croissance exponentielle, portée par les avancées en intelligence artificielle et la démocratisation des smartphones. **HealthGuard Vision** s'inscrit dans cette dynamique en proposant un outil de dépistage accessible, non-invasif et instantané, capable de détecter précocement certaines pathologies courantes à partir de simples photographies.

1.2 Problématique

L'accès aux soins préventifs reste un défi majeur dans de nombreuses régions. Les visites médicales régulières sont souvent négligées par manque de temps, de moyens ou de proximité avec un professionnel de santé. Notre application cherche à répondre à cette problématique en offrant un premier niveau de dépistage directement depuis le smartphone de l'utilisateur.

Avertissement médical

HealthGuard Vision est un outil d'aide au dépistage développé dans un cadre académique. Il ne fournit **aucun diagnostic médical**. Les résultats doivent toujours être confirmés par un professionnel de santé qualifié.

1.3 Objectifs

Les objectifs principaux de ce projet sont :

1. Concevoir et développer une **API RESTful Flask** pour le traitement d'images et l'inférence ML
2. Intégrer **3 modèles TensorFlow Lite** pré-entraînés pour la détection de diabète (rétinopathie), d'anémie et de maladies cutanées
3. Développer une **application mobile React Native** cross-platform avec capture photo et affichage des résultats
4. Mettre en place une base de données **MongoDB** conforme aux exigences HIPAA pour le stockage des métadonnées patients
5. Containeriser l'application avec **Docker** et orchestrer les services via **Docker Compose**
6. Appliquer les méthodologies **Scrum/Agile** avec Jira pour la gestion de projet

1.4 Organisation du document

Ce rapport est organisé comme suit :

- **Chapitre 2** : Architecture technique et choix technologiques
- **Chapitre 3** : Développement du backend (API Flask, ML, base de données)
- **Chapitre 4** : Développement du frontend (React Native / Expo)
- **Chapitre 5** : DevOps, containerisation et déploiement
- **Chapitre 6** : Méthodologie Agile et gestion de projet
- **Chapitre 7** : Sécurité, tests et assurance qualité
- **Chapitre 8** : Conclusion et perspectives

2 Architecture technique

2.1 Vue d'ensemble de l'architecture

HealthGuard Vision suit une architecture **client-serveur à trois couches** (3-tier), conforme aux bonnes pratiques du développement moderne :

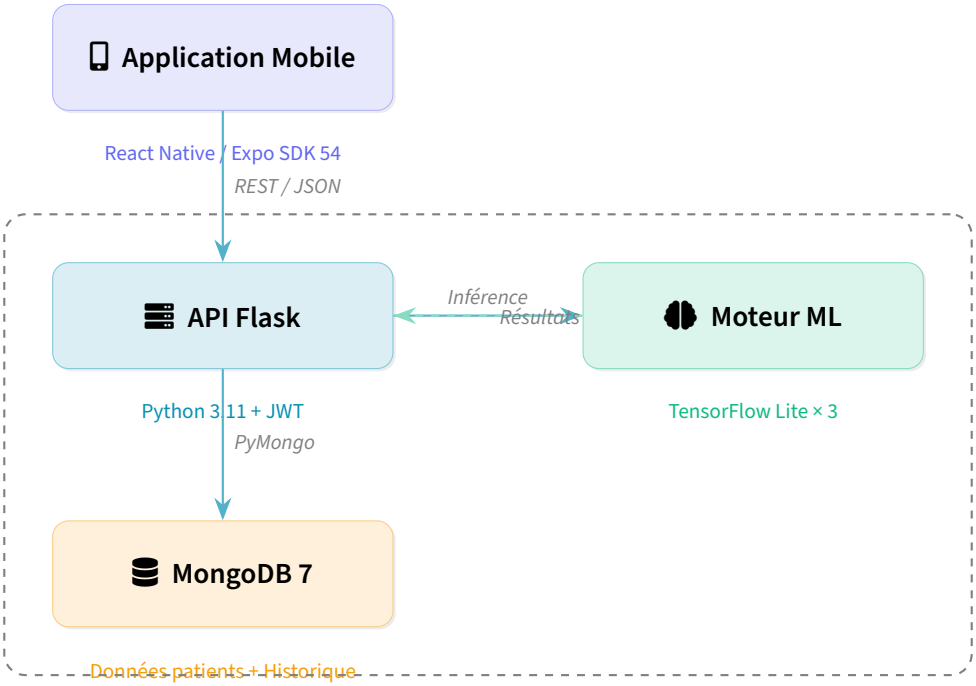

FIGURE 2.1 – Architecture globale de HealthGuard Vision

2.2 Choix technologiques

TABLE 2.1 – Stack technologique de HealthGuard Vision

Couche	Technologie	Justification
Mobile	React Native 0.81 + Expo SDK 54	Cross-platform, écosystème riche, hot-reload
Langage	TypeScript 5.9	Typage statique, maintenabilité du code
Navigation	expo-router 6.0	Routage basé fichiers, conventions modernes
Auth (client)	expo-secure-store + JWT	Stockage sécurisé chiffré des tokens
Caméra	expo-image-picker	Capture photo et sélection galerie
Backend	Flask 3.0.3 (Python 3.11)	Léger, extensible, idéal pour les APIs ML
Auth (serveur)	Flask-JWT-Extended 4.6.0	Gestion JWT robuste et configurable
ML/IA	TensorFlow Lite	Inférence rapide, modèles optimisés mobile
Prétraitement	Pillow 10.4 + OpenCV 4.10	Manipulation d'images haute performance
Base données	MongoDB 7 (PyMongo 4.8)	Schéma flexible, adapté aux données médicales
Conteneurs	Docker + Docker Compose	Isolation, reproductibilité, déploiement
Serveur prod	Gunicorn 22.0	Serveur WSGI performant pour Flask
Tests	Pytest 8.3 + pytest-cov	Framework de test Python standard
Qualité	Black, Flake8, Pylint, Bandit	Formatage, linting, analyse de sécurité
Gestion	Jira (Scrum) + GitHub	Agile + versioning + CI/CD

2.3 Structure du projet

Le projet est organisé selon une séparation claire des responsabilités, avec deux grands modules indépendants communicant via une API REST :

TABLE 2.2 – Organisation des modules du projet

Module	Technologie	Contenu principal
`backend/`	Flask / Python 3.11	Factory pattern, endpoints REST, moteur ML, couche BDD
`frontend/`	React Native / Expo	Écrans (tabs, auth, legal), services API, contexte auth
`ml_models/`	TensorFlow Lite	3 modèles pré-entraînés + mapping de classes
Racine	Docker Compose	Orchestration des 3 services (MongoDB, API, Front)

Le backend suit le pattern **Application Factory** de Flask avec une architecture en couches : routes (`routes.py`), services métier (`services.py`), accès aux données (`db.py`) et moteur de prédiction (`predict.py`). Le frontend utilise le système de routage par fichiers d'`expo-router`, avec des groupes de navigation pour l'authentification, les onglets principaux et les pages légales.

3 Développement Backend

3.1 Architecture de l'API Flask

Le backend suit le pattern **Application Factory** de Flask, garantissant une initialisation propre et testable de l'application.

3.1.1 Configuration et initialisation

La fonction `create_app()` centralise l'initialisation de l'application selon les bonnes pratiques du framework :

- **Chargement des variables d'environnement** via `python-dotenv` pour séparer les secrets du code source
- **Connexion MongoDB** initialisée via `init_db()` avec l'URI configurée en variable d'environnement
- **Gestion JWT** : Configuration du `JWTManager` avec une clé secrète externalisée
- **Blueprint Flask** : Enregistrement modulaire des routes API
- **CORS** : Autorisations cross-origin pour les requêtes depuis l'application mobile

Cette architecture garantit une séparation claire entre configuration, logique métier et accès aux données, conformément aux principes *twelve-factor app*.

3.1.2 Endpoints API REST

L'API expose les endpoints suivants, organisés selon une logique RESTful :

TABLE 3.1 – Endpoints de l'API HealthGuard

#	Méthode	Route	Description	Auth
1	GET	/health	Vérification état du serveur	✗
2	POST	/signup	Inscription d'un utilisateur	✗
3	POST	/auth	Connexion (retourne JWT)	✗
4	POST	/re-auth	Renouvellement du token JWT	✓
5	POST	/predict	Upload et analyse d'image	✓
6	GET	/profile	Récupérer le profil utilisateur	✓
7	PUT	/profile	Modifier le profil	✓
8	PUT	/change-password	Changer le mot de passe	✓
9	GET	/histories	Historique des analyses	✓
10	GET	/export-data	Exporter toutes les données	✓
11	DELETE	/delete-history	Supprimer l'historique	✓

3.1.3 Endpoint principal : `/predict`

L'endpoint de prédiction constitue le cœur fonctionnel de l'API. Son fonctionnement suit le processus suivant :

1. **Authentification** : Vérification du JWT via le décorateur `@jwt_required()`
2. **Validation** : Contrôle de la présence de l'image et du type d'analyse (`eye`, `skin` ou `nail`)
3. **Analyse** : Appel au moteur ML via `analyze_image(file, type, sex)`
4. **Historique** : Sauvegarde automatique du résultat dans la collection MongoDB
5. **Réponse** : Retour du résultat structuré en JSON (message, taux d'hémoglobine, recommandations)

L'image est reçue via `multipart/form-data`, temporairement enregistrée sur le serveur pour l'inférence, puis supprimée après traitement.

3.2 Moteur de Machine Learning

3.2.1 Architecture du module de prédiction

Le module `predict.py` implémente la classe `MedicalAnalyzer`, un analyseur médical unifié suivant le pattern **Singleton** pour optimiser l'utilisation mémoire des modèles

TensorFlow Lite.

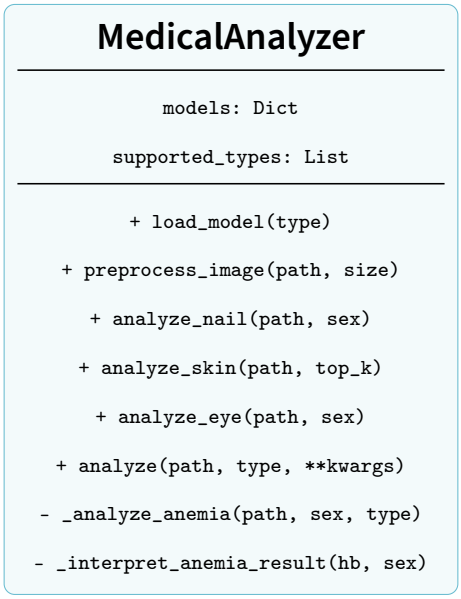

FIGURE 3.1 – Diagramme de classe `MedicalAnalyzer`

3.2.2 Les trois modèles TensorFlow Lite

TABLE 3.2 – Modèles ML embarqués dans HealthGuard Vision

Modèle	Fichier	Sortie	Description
Œil	`eye_anemia_model.tflite`	Régression (g/L)	Prédit le taux d'hémoglobine à partir de l'image de l'œil
Ongle	`nail_anemia_model.tflite`	Régression (g/L)	Prédit le taux d'hémoglobine à partir de l'image des ongles
Peau	`best_skin_disease_model.tflite`	Classification	Classifie parmi plusieurs maladies cutanées

3.2.3 Pipeline de prétraitement

Chaque image subit un prétraitement standardisé avant l'inférence :

1. **Chargement** : Ouverture de l'image via Pillow (`PIL.Image.open`)

2. **Conversion** : Conversion en mode RGB si nécessaire
3. **Redimensionnement** : Resize à 224×224 pixels avec interpolation Lanczos
4. **Normalisation** : Conversion en `float32` et normalisation dans $[0, 1]$
5. **Expansion** : Ajout d'une dimension batch (`expand_dims`)

Ce pipeline garantit que chaque image est dans un format identique quel que soit le type d'appareil ou la résolution d'origine, assurant la reproductibilité des inférences.

3.2.4 Interprétation des résultats d'anémie

L'interprétation du taux d'hémoglobine prend en compte le **sexe biologique** du patient, avec des seuils différenciés conformes aux recommandations médicales internationales :

TABLE 3.3 – Seuils d'interprétation de l'hémoglobine (en g/L)

Condition	Homme	Femme	Sévérité
Anémie sévère	< 80	< 80	Sévère
Anémie modérée	$80 - 100$	$80 - 100$	Modérée
Anémie légère	$100 - 130$	$100 - 120$	Légère
Normal	$130 - 170$	$120 - 160$	Normal
Élevé	> 170	> 160	Élevé

3.3 Couche d'accès aux données (MongoDB)

3.3.1 Schéma de base de données

La base MongoDB `healthguard` comprend deux collections principales avec validation par schéma JSON :

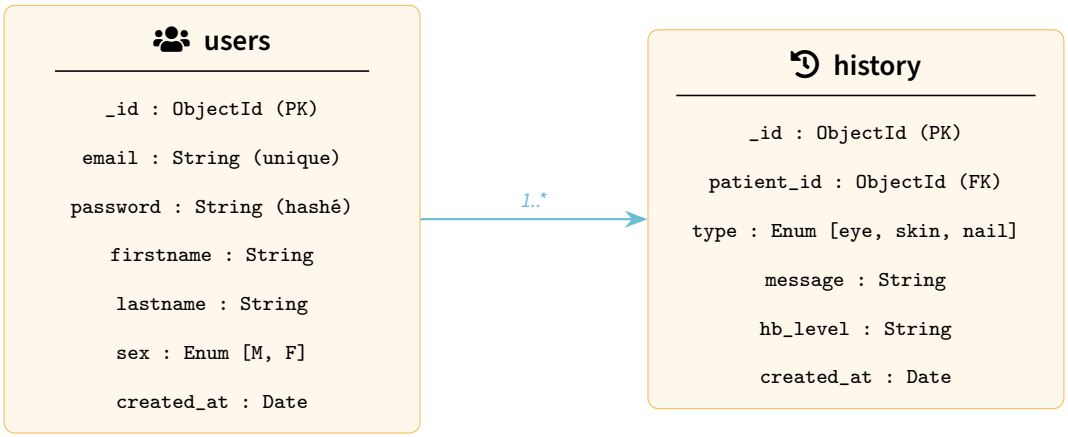

FIGURE 3.2 – Schéma des collections MongoDB

3.3.2 Sécurité des données

- ✓ **Hashage des mots de passe** : Utilisation de `werkzeug.security.generate_password_hash` (PBKDF2-SHA256)
- ✓ **Validation de schéma** : Contraintes JSON Schema au niveau MongoDB (`$jsonSchema`)
- ✓ **Index unique** : Index unique sur le champ `email` pour prévenir les doublons
- ✓ **Index de performance** : Index sur `patient_id` dans la collection `history`
- ✓ **Sérialisation sécurisée** : Les mots de passe sont systématiquement supprimés des réponses API (`user.pop('password', None)`)

4 Développement Frontend

4.1 Architecture de l'application mobile

L'application mobile est développée avec **React Native 0.81** et **Expo SDK 54**, utilisant le framework de routage `expo-router` 6.0 qui offre un système de navigation basé sur le système de fichiers.

4.1.1 Système de navigation

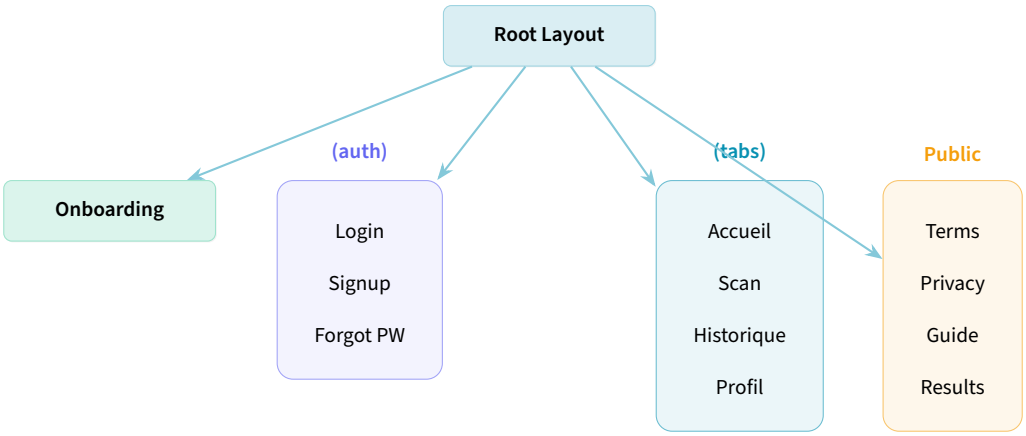

FIGURE 4.1 – Arbre de navigation de l'application

4.1.2 Flux d'authentification

Le système d'authentification repose sur plusieurs couches de sécurité :

1. **AuthProvider** (Context React) : Wrapping global de l'application pour gérer l'état d'authentification
2. **JWT Token** : Émis par le backend Flask avec une durée de validité de 24 heures
3. **expo-secure-store** : Stockage chiffré du token sur l'appareil (`Keychain` sur iOS, `Keystore` sur Android)
4. **Auto-refresh** : Renouvellement automatique du token via l'endpoint `/re-auth` au lancement
5. **Auth Guard** : Redirection automatique vers la page de connexion si non authentifié

Le service API côté client (`services/api.ts`) implémente l'ensemble de cette logique de manière transparente : le stockage sécurisé du JWT utilise `Keychain` sur iOS et `Keystore` sur Android, tandis qu'un fallback `localStorage` est prévu pour la version web. Les en-têtes d'authentification sont automatiquement injectés dans chaque requête API.

4.2 Écrans principaux

4.2.1 Onboarding (première utilisation)

L'écran d'onboarding présente l'application en **3 slides interactives** avec :

- Animations fluides (`Animated API` de React Native)
- Pagination avec indicateurs animés
- Bouton « Passer » pour les utilisateurs existants
- Persistance via `AsyncStorage` pour ne s'afficher qu'une seule fois

4.2.2 Dashboard d'accueil

Le dashboard offre une vue personnalisée incluant :

- Salutation personnalisée avec le prénom de l'utilisateur
- Carte d'état de santé avec indicateur de dernier scan
- **3 cartes de scan** (Oculaire, Cutané, Ongles) avec navigation directe
- Section « Conseils santé du jour » avec 3 recommandations rotatives
- Avertissement médical permanent en pied de page

4.2.3 Capture et analyse

L'écran de capture implémente un workflow en 3 étapes :

Workflow de capture

1. **Sélection du type** : Grille de 3 cartes (Œil, Peau, Ongles)
2. **Prise de photo** : Via caméra ou galerie, avec conseils spécifiques par type
3. **Analyse** : Envoi à l'API, indicateur de chargement, puis navigation vers les résultats

4.2.4 Affichage des résultats

L'écran de résultats affiche de manière différenciée selon le type d'analyse :

- **Analyse d'anémie** (œil/ongle) :
  - Bannière de sévérité colorée (normal/léger/modéré/sévère)
  - Message diagnostique textuel
  - Taux d'hémoglobine avec unité (g/L)
  - Recommandations numérotées adaptées à la sévérité
- **Analyse cutanée** :
  - Diagnostic principal mis en avant
  - Top-3 des prédictions classées par confiance
  - Recommandations dermatologiques

4.2.5 Historique

L'écran d'historique propose :

- Liste chronologique de tous les scans passés
- Système de **filtres** par type (Tous / Œil / Peau / Ongles)
- Pull-to-refresh pour actualiser les données
- Navigation vers le détail de chaque résultat
- État vide encourageant le premier scan

4.2.6 Profil utilisateur

L'écran de profil offre une gestion complète du compte :

- Avatar avec initiales
- Statistiques (nombre de scans, ancienneté)
- Sections organisées : Compte, Données de santé, Application
- Modales de modification du profil et de changement de mot de passe
- Export des données en JSON
- Suppression de l'historique avec confirmation
- Déconnexion sécurisée

4.3 Design System

4.3.1 Palette de couleurs

TABLE 4.1 – Palette de couleurs de HealthGuard Vision

Nom	Hex	Aperçu	Usage
Primary	#0891B2		Marque, boutons, liens
Secondary	#6366F1		Technologie, accent
Eye Scan	#8B5CF6		Analyses oculaires
Skin Scan	#F97316		Analyses cutanées
Nail Scan	#EC4899		Analyses ongles
Success	#10B981		Résultats normaux
Warning	#F59E0B		Résultats modérés
Danger	#EF4444		Risque élevé

4.3.2 Composants UI

L'interface respecte les principes du **Material Design** adapté au domaine médical :

- Cartes avec ombres douces (`shadowOpacity: 0.06`)
- Coins arrondis généreux (`borderRadius: 14-24`)
- Hiérarchie typographique claire (poids 500-800)
- Icônes Ionicons cohérentes et accessibles
- Animations de transition fluides (`slide_from_right`, `slide_from_bottom`)

4.4 Service API côté client

Le service API (`services/api.ts`) fournit une couche d'abstraction complète avec :

- **Types TypeScript** : Interfaces typées pour toutes les réponses API (`User`, `AnalysisResult`, `HistoryRecord`)
- **Type Guards** : Fonction `isSkinResult()` pour la discrimination de types
- **Mode Mock** : Toggle `USE_MOCK_API` pour le développement sans backend
- **Gestion d'erreurs** : Interception et formatage uniforme des erreurs réseau
- **Upload multipart** : Gestion du `FormData` pour l'envoi d'images

5 DevOps, Containerisation et Déploiement

5.1 Containerisation Docker

5.1.1 Dockerfile Backend

Le Dockerfile du backend suit les bonnes pratiques de containerisation :

- **Image de base légère** : `python:3.11-slim` pour minimiser la taille de l'image
- **Dépendances système** : Installation de `libgl1` et `libglib2.0-0` nécessaires à OpenCV
- **Nettoyage du cache** : Suppression de `/var/lib/apt/lists/*` et utilisation de `-no-cache-dir` pour pip
- **Optimisation du cache Docker** : Copie des requirements séparée avant le code source
- **Exposition** : Port 5000 pour l'API Flask

5.1.2 Dockerfile Frontend

Le frontend est containerisé à partir de `node:20-slim`, avec installation des dépendances npm, copie du code source et démarrage d'Expo en mode web sur le port 8081.

5.1.3 Orchestration Docker Compose

Le fichier `docker-compose.yml` définit **3 services** interconnectés avec les caractéristiques suivantes : persistance des données MongoDB via un volume nommé, variables d'environnement pour la configuration des secrets, politique de redémarrage automatique, et dépendances inter-services.

TABLE 5.1 – Services Docker Compose

Service	Image	Port	Rôle
`mongodb`	`mongo:7`	27017	Base de données NoSQL
`backend`	Build local	5000	API Flask + ML
`frontend`	Build local	8081	Application Expo Web

5.1.4 Caractéristiques de l'orchestration

- ✓ **Persistance** : Volume nommé `mongo-data` pour les données MongoDB
- ✓ **Redémarrage automatique** : Politique `unless-stopped`
- ✓ **Dépendances** : `depends_on` pour garantir l'ordre de démarrage
- ✓ **Hot-reload** : Montage de volumes pour le développement en temps réel
- ✓ **Isolation réseau** : Réseau Docker par défaut pour la communication inter-services
- ✓ **Variables d'environnement** : Configuration externalisée des secrets

5.2 Pipeline CI/CD avec GitHub

Le versioning du code est géré via **GitHub** avec un workflow Git structuré :

- Branche `main` : Code de production stable
- Branches de fonctionnalités (`feature/*`) : Développement isolé
- Pull Requests avec revue de code
- Intégration avec Jira pour la traçabilité des tickets

5.3 Qualité de code

L'assurance qualité du code Python repose sur un ensemble d'outils complémentaires :

Table 5.2 – Outils de qualité de code

Outil	Version	Fonction
Black	24.8.0	Formatage automatique du code Python
Flake8	7.1.1	Vérification du style (PEP 8)
Pylint	3.2.7	Analyse statique avancée
isort	5.13.2	Tri automatique des imports
Bandit	1.7.9	Analyse de sécurité du code
Safety	3.2.7	Détection de vulnérabilités dans les dépendances

6 Méthodologie Agile et Gestion de Projet

6.1 Framework Scrum

Le projet a été mené selon la méthodologie **Scrum**, adaptée au cadre contraint de 5 jours :

6.1.1 Rôles Scrum

— **Product Owner** : Responsable de la vision produit et de la priorisation du backlog

— **Scrum Master** : Facilitateur des cérémonies et gardien du processus

— **Équipe de développement** : 3 développeurs spécialisés (Mobile, Backend, ML)

6.1.2 Outils de gestion

TABLE 6.1 – Outils de gestion de projet

Outil	Usage	Détails
Jira	Gestion Scrum	Backlog, sprints, user stories, suivi des tâches
GitHub	Versioning + DevOps	Repository, branches, pull requests, CI/CD
Discord	Communication	Canaux par équipe, partage d'écran, daily standups

6.2 Planning sur 5 jours

TABLE 6.2 – Planning du sprint intensif de 5 jours

Jour	Thème	Livrables
J1	Architecture & Setup DevOps	Choix projet, architecture, Docker, pipeline CI de base
J2	Développement & CI/CD	API core, tests automatisés, pipeline complet, staging
J3	Features Avancées & Monitoring	Fonctionnalités avancées, APIs externes, métriques
J4	Production & Scalabilité	Déploiement, HA, tests de charge, documentation
J5	Pitch & Démonstration	Pitch startup, démo live (10 min + 5 min démo)

6.3 Cérémonies Scrum

1. **Sprint Planning** (J1) : Définition du backlog et attribution des tâches
2. **Daily Standups** : Réunions quotidiennes de 15 minutes
3. **Sprint Review** (J5) : Démonstration des fonctionnalités développées
4. **Sprint Retrospective** (J5) : Analyse des points positifs et axes d'amélioration

6.4 Workflow Git

Le workflow Git adopté suit le modèle **GitHub Flow** simplifié :

1. Création d'une branche depuis `main` pour chaque fonctionnalité/ticket Jira
2. Développement et commits atomiques avec messages conventionnels
3. Push et ouverture d'une Pull Request
4. Revue de code par un pair
5. Merge dans `main` après approbation

7 Sécurité, Tests et Assurance Qualité

7.1 Mesures de sécurité implémentées

7.1.1 Authentification et autorisation

TABLE 7.1 – Mesures de sécurité implémentées

Mesure	Implémentation
Hashage mots de passe	PBKDF2-SHA256 via `werkzeug.security`
JWT Tokens	Expiration 24h, clé secrète en variable d'environnement
Stockage client	`expo-secure-store` (Keychain iOS / Keystore Android)
CORS	`Flask-CORS` configuré pour les origines autorisées
Validation entrées	Vérification de tous les champs requis côté serveur
Suppression données	Endpoint `DELETE /delete-history` pour le droit à l'oubli
Export données	Endpoint `GET /export-data` pour la portabilité

7.1.2 Conformité HIPAA

HealthGuard Vision a été conçu en intégrant les principes de conformité **HIPAA** (Health Insurance Portability and Accountability Act) :

- ✓ **Chiffrement en transit** : TLS pour toutes les communications API
- ✓ **Chiffrement au repos** : MongoDB avec encryption at rest sur Azure
- ✓ **Contrôle d'accès** : Authentification JWT obligatoire pour les endpoints sensibles
- ✓ **Minimisation des données** : Seules les métadonnées sont stockées, pas les images brutes de manière permanente
- ✓ **Droit à la suppression** : L'utilisateur peut supprimer son historique à tout moment
- ✓ **Droit à la portabilité** : Export des données en format JSON standard
- ✓ **Politique de confidentialité** : Page dédiée intégrée dans l'application
- ✓ **Conditions d'utilisation** : Avertissement médical explicite

7.1.3 Analyse de sécurité du code

L'utilisation de **Bandit** (analyse sécurité Python) et **Safety** (audit des dépendances) permet de :

- Détecter les vulnérabilités connues dans les packages utilisés
- Identifier les patterns de code potentiellement dangereux
- Vérifier l'absence de secrets en dur dans le code source

7.2 Tests et assurance qualité

7.2.1 Framework et stratégie de test

Le projet utilise **Pytest 8.3** comme framework de test principal, accompagné des extensions `pytest-cov` (couverture de code), `pytest-flask` (helpers Flask) et `pytest-mock` (isolation par mocking).

La stratégie de test suit la **pyramide des tests** :

1. **Tests unitaires** : Validation des fonctions individuelles (prétraitement, interprétation des seuils, sérialisation)
2. **Tests d'intégration** : Vérification de l'interaction entre les couches (Route $\rightarrow$ Service $\rightarrow$ DB)
3. **Tests ML** : Validation des modèles (temps d'inférence, format de sortie, cohérence des prédictions)
4. **Tests API** : Validation des endpoints (codes HTTP, format de réponse, gestion d'erreurs)

7.2.2 Qualité du frontend

Côté frontend, la qualité est assurée par :

- **TypeScript strict** : Détection des erreurs de type à la compilation
- **ESLint** : Vérification du style et des bonnes pratiques React
- **Mode Mock API** : Toggle `USE_MOCK_API` permettant de tester l'UI indépendamment du backend

8 Conclusion et Perspectives

8.1 Bilan du projet

Le projet **HealthGuard Vision** a permis de développer avec succès une application complète de diagnostic préventif par image, intégrant l'ensemble de la chaîne de valeur depuis la capture mobile jusqu'à l'inférence par intelligence artificielle.

8.1.1 Objectifs atteints

TABLE 8.1 – Bilan des objectifs

Objectif	Statut
API Flask RESTful avec endpoints complets	✓
3 modèles TensorFlow Lite intégrés et fonctionnels	✓
Application mobile React Native avec navigation complète	✓
Base MongoDB avec schéma validé et sécurité des données	✓
Containerisation Docker avec Docker Compose	✓
Authentification JWT sécurisée avec stockage chiffré	✓
Historique des analyses avec filtrage et export	✓
Gestion de profil complète (modification, mot de passe)	✓
Pages légales (CGU, Politique de confidentialité)	✓
Onboarding et guide utilisateur intégrés	✓
Méthodologie Scrum avec Jira	✓

8.1.2 Compétences développées

Ce projet a permis de consolider et développer les compétences suivantes :

- **Full-stack development** : Maîtrise d'une architecture complète client-serveur
- **Intelligence artificielle** : Intégration et déploiement de modèles TensorFlow Lite
- **DevOps** : Containerisation, orchestration et pipeline CI/CD
- **Sécurité** : Implémentation de mesures conformes aux standards de l'industrie

- **Méthodologie Agile** : Pratique du Scrum dans un cadre de sprint intensif
- **Travail d'équipe** : Collaboration via Git, Jira et communication asynchrone

8.2 Perspectives d'amélioration

Plusieurs axes d'amélioration ont été identifiés pour les évolutions futures :

8.2.1 Améliorations techniques

1. **Multi-stage Dockerfile** : Optimisation de l'image Docker pour atteindre ∼300 MB
2. **Pipeline CI/CD complet** : GitHub Actions avec tests automatiques, analyse de sécurité et déploiement
3. **Monitoring** : Intégration de Prometheus/Grafana pour le suivi des métriques ML
4. **Tests end-to-end** : Automatisation des tests UI avec Detox ou Maestro
5. **Cache Redis** : Mise en cache des résultats fréquents pour réduire la latence

8.2.2 Améliorations fonctionnelles

1. **Notifications push** : Rappels de suivi de santé
2. **Mode hors-ligne** : Inférence TFLite directement sur l'appareil mobile
3. **Partage médecin** : Export PDF des résultats vers un professionnel de santé
4. **Multi-langue** : Support i18n (Français, Anglais, Arabe)
5. **Amélioration des modèles** : Transfer learning sur des datasets plus larges et diversifiés
6. **Tableau de bord analytics** : Visualisation des tendances de santé dans le temps

8.2.3 Améliorations DevOps

1. **Déploiement Blue-Green** : Sur Azure Container Instances pour le zero-downtime
2. **Infrastructure as Code** : Terraform pour provisionner les ressources Azure
3. **Observabilité** : Logs structurés, tracing distribué, alerting
4. **Tests de charge** : Validation des performances sous forte charge
5. **Scan de vulnérabilités** : Intégration de Trivy pour le scan des images Docker

8.3 Conclusion

HealthGuard Vision démontre la faisabilité d'une solution de dépistage de santé préventif assisté par IA, accessible via smartphone. Le projet illustre comment les technologies modernes — React Native, Flask, TensorFlow Lite, MongoDB, Docker — peuvent être combinées de manière cohérente pour créer une application fonctionnelle, sécurisée et maintenable.

Au-delà de l'aspect technique, ce projet nous a permis de vivre l'expérience d'un développement en équipe selon les méthodologies Agile, dans un cadre temporel contraint qui reflète les réalités du monde professionnel.

Rappel important

Ce projet a été réalisé dans un cadre académique. Les résultats de l'analyse par IA ne constituent en aucun cas un diagnostic médical. Toute préoccupation de santé doit être adressée à un professionnel de santé qualifié.

A Récapitulatif des dépendances

A.1 Dépendances Backend (Python)

TABLE A.1 – Principales dépendances Python

Package	Version	Rôle
Flask	3.0.3	Framework web principal
Flask-JWT-Extended	4.6.0	Gestion de l'authentification JWT
Flask-CORS	4.0.1	Gestion des requêtes cross-origin
TensorFlow	2.17.0	Inférence des modèles de machine learning
Pillow	10.4.0	Traitement et manipulation d'images
NumPy	1.26.4	Calcul numérique et manipulation de tableaux
OpenCV	4.10.0	Prétraitement d'images (headless)
PyMongo	4.8.0	Driver MongoDB pour Python
Gunicorn	22.0.0	Serveur WSGI de production
Pytest	8.3.2	Framework de tests unitaires
Black	24.8.0	Formatage automatique du code
Bandit	1.7.9	Analyse de sécurité du code

A.2 Dépendances Frontend (Node.js)

TABLE A.2 – Principales dépendances npm

Package	Version	Rôle
React	19.1.0	Bibliothèque UI déclarative
React Native	0.81.5	Framework mobile cross-platform
Expo	54.0	Plateforme de développement React Native
expo-router	6.0	Navigation basée sur le système de fichiers
expo-image-picker	17.0	Capture photo et sélection galerie
expo-secure-store	15.0	Stockage sécurisé chiffré (Keychain / Keystore)
TypeScript	5.9	Typage statique pour JavaScript
Reanimated	4.1	Animations performantes (UI thread)
AsyncStorage	2.2	Stockage local persistant

A.3 Endpoints API

TABLE A.3 – Récapitulatif des endpoints REST

Méthode	Route	Description	Auth
GET	/health	État de santé du serveur	✗
POST	/signup	Inscription utilisateur	✗
POST	/auth	Connexion (émission JWT)	✗
POST	/re-auth	Renouvellement token	✓
POST	/predict	Analyse d'image ML	✓
GET	/profile	Consultation du profil	✓
PUT	/profile	Modification du profil	✓
PUT	/change-password	Changement de mot de passe	✓
GET	/histories	Historique des analyses	✓
GET	/export-data	Export des données (JSON)	✓
DELETE	/delete-history	Suppression de l'historique	✓