Institut National des Postes et Télécommunications

PROJET PERSONNEL ET PROFESSIONNEL
Rapport de PROJET

Ai Counsult : Chatbot Hybride Intelligent

Réalisé par :
Salmane BABA
Yassin NAJMI
Abderrahmane OUARACH

Encadrant :
Pr H. KAMAL IDRISSI

Table des matières

1 Introduction **2**

2 Application Android **3**
2.1 Objectifs de l'Application Android . . . 3
2.2 Choix Technologiques . . . 3
2.2.1 Langage de Programmation : Kotlin . . . 3
2.2.2 Interface Utilisateur : Jetpack Compose . . . 3
2.2.3 Architecture : Clean Architecture et MVVM . . . 3
2.3 Communication Réseau et Backend . . . 4
2.3.1 Retrofit et OkHttp . . . 4
2.3.2 Authentification et Sécurité . . . 4
2.4 Système de Chat avec Agent IA Unique . . . 4
2.5 Gestion de l'État et Performance UI . . . 4
2.6 Apports de cette Première Étape . . . 5

3 Application Android : Implémentation du Client Mobile **5**
3.1 Objectifs de l'Application Android . . . 5
3.2 Choix Technologiques . . . 5
3.2.1 Langage de Programmation : Kotlin . . . 5
3.2.2 Interface Utilisateur : Jetpack Compose . . . 6
3.2.3 Architecture : Clean Architecture et MVVM . . . 6
3.3 Communication Réseau et Backend . . . 6
3.3.1 Retrofit et OkHttp . . . 6
3.3.2 Authentification et Sécurité . . . 7
3.4 Système de Chat avec Agent IA Unique . . . 7
3.5 Gestion de l'État et Performance UI . . . 7
3.6 Apports de cette Première Étape . . . 7

4 Backend : Conception et Implémentation **8**
4.1 Technologies Utilisées . . . 8
4.2 Architecture Générale . . . 8
4.3 Authentification et Sécurité . . . 8
4.4 Flux d'Authentification . . . 8
4.5 Gestion des Données . . . 8
4.6 API REST . . . 9
4.7 Gestion des Erreurs et Logs . . . 9
4.8 Lien avec l'Évolution Multi-Agents . . . 9

5 Prototype du Juge Multi-Agents (Première Implémentation) **10**
5.1 Objectif du Prototype . . . 10
5.2 Architecture Globale . . . 10
5.3 Détails du Backend (Le Cerveau) . . . 10
5.3.1 Le Juge et le Routeur (`brain.py`) . . . 10
5.3.2 La Synthèse des Experts (`brain.py`) . . . 11
5.3.3 Les Travailleurs (`ai.py`) . . . 11
5.3.4 Le Serveur (`server.py`) . . . 11

1 Introduction

L'essor spectaculaire de l'intelligence artificielle conversationnelle a profondément transformé la manière dont les utilisateurs interagissent avec les systèmes numériques. Les chatbots modernes ne se limitent plus à fournir des réponses prédéfinies ou à exécuter des règles simples : ils sont désormais capables de raisonner, de synthétiser de l'information complexe et de s'adapter dynamiquement au contexte de l'utilisateur. Cependant, malgré ces avancées, une limite majeure subsiste dans la majorité des solutions existantes : la dépendance à un modèle d'intelligence artificielle unique.

Dans les architectures classiques, un seul modèle est responsable de l'analyse de la requête, du raisonnement et de la génération de la réponse finale. Cette approche présente plusieurs inconvénients. D'une part, chaque modèle possède ses propres forces et faiblesses : certains excellent dans le raisonnement logique, d'autres dans la créativité ou la concision, tandis que certains sont plus performants pour le code ou l'analyse technique. D'autre part, s'appuyer sur un seul modèle augmente le risque d'erreurs, d'hallucinations ou de réponses incomplètes, en particulier pour des requêtes complexes ou ambiguës.

C'est dans ce contexte que s'inscrit notre projet de chatbot intelligent basé sur une architecture multi-agents. L'idée centrale du projet est de ne plus considérer l'intelligence artificielle comme une entité unique, mais comme un ensemble collaboratif de plusieurs intelligences artificielles spécialisées, travaillant en parallèle. Dans notre implémentation, lorsqu'un utilisateur envoie un message, celui-ci n'est pas traité par un seul modèle, mais diffusé simultanément à trois intelligences artificielles distinctes. Chaque IA analyse la requête de manière indépendante et génère sa propre réponse, selon ses capacités et son mode de raisonnement.

Une fois ces réponses produites, un quatrième composant intelligent intervient. Son rôle n'est pas de répondre directement à l'utilisateur, mais d'agir comme un juge et un évaluateur. Ce composant compare les différentes réponses générées par les trois IA, analyse leur pertinence, leur cohérence et leur qualité globale, puis sélectionne — ou synthétise — la meilleure réponse possible. Cette approche permet de tirer parti des points forts de chaque modèle tout en réduisant l'impact de leurs faiblesses individuelles.

L'objectif principal de ce projet est donc de concevoir et d'implémenter un chatbot hybride, plus fiable, plus précis et plus robuste qu'un chatbot traditionnel. En combinant le parallélisme des modèles, la comparaison intelligente des réponses et une logique de décision centralisée, notre solution vise à offrir à l'utilisateur final une réponse optimale, aussi bien pour des questions simples que pour des problématiques complexes nécessitant du raisonnement avancé.

2 Application Android

Cette section présente la conception et le développement de l'application Android du projet. Lors de la première étape du travail, l'objectif était de mettre en place une application mobile fonctionnelle intégrant un **seul agent d'intelligence artificielle**. Cette étape a permis de valider l'architecture mobile, la communication avec le backend et l'expérience utilisateur, avant l'évolution vers une architecture multi-agents plus avancée.

2.1 Objectifs de l'Application Android

L'application Android constitue le point d'entrée principal pour l'utilisateur final. Elle a été conçue afin de :

- Fournir une interface de messagerie simple, fluide et moderne.
- Permettre l'authentification sécurisée des utilisateurs (inscription et connexion).
- Assurer la communication avec un backend exposant une API REST.
- Intégrer un premier agent IA (DeepSeek-R1 via Ollama) pour valider le fonctionnement du chatbot.
- Préparer une base logicielle extensible pour l'intégration future de plusieurs agents IA.

2.2 Choix Technologiques

Le développement de l'application mobile repose sur des technologies Android modernes, garantissant performance, maintenabilité et évolutivité.

2.2.1 Langage de Programmation : Kotlin

L'ensemble de l'application est développé en **Kotlin**. Ce langage offre une syntaxe concise, une meilleure gestion de la nullabilité et un excellent support de la programmation asynchrone via les **Coroutines**. Ces caractéristiques sont essentielles pour gérer les appels réseau vers le backend et l'agent IA sans bloquer l'interface utilisateur.

2.2.2 Interface Utilisateur : Jetpack Compose

L'interface graphique est construite avec **Jetpack Compose**, un framework déclaratif moderne. Contrairement aux interfaces XML traditionnelles, Compose permet de lier directement l'état de l'application à l'affichage, rendant l'UI plus réactive et plus facile à maintenir. Les écrans principaux incluent :

- Écran de connexion et d'inscription.
- Écran principal de discussion (chat).
- Gestion dynamique de l'historique des messages.

2.2.3 Architecture : Clean Architecture et MVVM

L'application suit les principes de la **Clean Architecture** combinés au pattern **MVVM (Model-View-ViewModel)**. Cette organisation permet une séparation claire des responsabilités :

- **Couche Présentation :** Composants Jetpack Compose et ViewModels responsables de l'état de l'interface.

— **Couche Domaine :** Contient la logique métier et les modèles principaux (messages, conversations).
— **Couche Data :** Gère la communication réseau et l'accès aux sources de données distantes.

2.3 Communication Réseau et Backend

La communication entre l'application Android et le backend est réalisée via une API REST.

2.3.1 Retrofit et OkHttp

La librairie **Retrofit** est utilisée pour définir les endpoints réseau et sérialiser les données échangées. **OkHttp** complète cette configuration en gérant les timeouts étendus nécessaires aux réponses de l'agent IA, ainsi que l'ajout automatique des en-têtes de sécurité.

2.3.2 Authentification et Sécurité

L'authentification repose sur un mécanisme de jeton **JWT**. Après l'inscription, un processus d'*auto-login* est déclenché afin de récupérer immédiatement un jeton valide. Celui-ci est stocké de manière sécurisée à l'aide de **Jetpack DataStore**, garantissant une persistance fiable et asynchrone.

2.4 Système de Chat avec Agent IA Unique

Dans cette première version, chaque message envoyé par l'utilisateur est transmis à un **unique agent IA** (DeepSeek-R1 exécuté localement via Ollama). Le workflow est le suivant :

1. L'utilisateur saisit un message dans l'interface de chat.
2. Le ViewModel envoie la requête au backend via le Repository.
3. Le backend transmet la requête à l'agent IA DeepSeek-R1.
4. La réponse générée est renvoyée à l'application Android.
5. L'interface met à jour l'historique de la conversation en temps réel.

Cette approche mono-agent a permis de valider la stabilité du flux de données, la gestion des états de chargement et l'affichage fluide des réponses.

2.5 Gestion de l'État et Performance UI

L'état du chat est centralisé dans un objet `ChatState` observé par l'interface. L'utilisation de `LazyColumn` pour l'affichage des messages permet d'optimiser les performances, même en présence d'un grand nombre de messages, en ne rendant que les éléments visibles à l'écran.

2.6 Apports de cette Première Étape

Le développement de cette application Android mono-agent a constitué une étape clé du projet. Il a permis :

- De valider les choix d'architecture mobile.
- De tester l'intégration d'un agent IA réel dans une application Android.
- D'identifier les contraintes de latence et de performance côté client.
- De poser des bases solides pour l'évolution vers une architecture multi-agents, présentée dans les sections suivantes.

Cette application Android servira ainsi de socle pour l'intégration du backend avancé et du système de comparaison intelligente entre plusieurs agents IA.

3 Application Android : Implémentation du Client Mobile

Cette section présente la conception et le développement de l'application Android du projet. Lors de la première étape du travail, l'objectif était de mettre en place une application mobile fonctionnelle intégrant un **seul agent d'intelligence artificielle** basé sur **Ollama** et le modèle **DeepSeek-R1**. Cette étape a permis de valider l'architecture mobile, la communication avec le backend et l'expérience utilisateur, avant l'évolution vers une architecture multi-agents plus avancée.

3.1 Objectifs de l'Application Android

L'application Android constitue le point d'entrée principal pour l'utilisateur final. Elle a été conçue afin de :

- Fournir une interface de messagerie simple, fluide et moderne.
- Permettre l'authentification sécurisée des utilisateurs (inscription et connexion).
- Assurer la communication avec un backend exposant une API REST.
- Intégrer un premier agent IA (DeepSeek-R1 via Ollama) pour valider le fonctionnement du chatbot.
- Préparer une base logicielle extensible pour l'intégration future de plusieurs agents IA.

3.2 Choix Technologiques

Le développement de l'application mobile repose sur des technologies Android modernes, garantissant performance, maintenabilité et évolutivité.

3.2.1 Langage de Programmation : Kotlin

L'ensemble de l'application est développé en **Kotlin**. Ce langage offre une syntaxe concise, une meilleure gestion de la nullabilité et un excellent support de la programmation asynchrone via les **Coroutines**. Ces caractéristiques sont essentielles pour gérer les appels réseau vers le backend et l'agent IA sans bloquer l'interface utilisateur.

3.2.2 Interface Utilisateur : Jetpack Compose

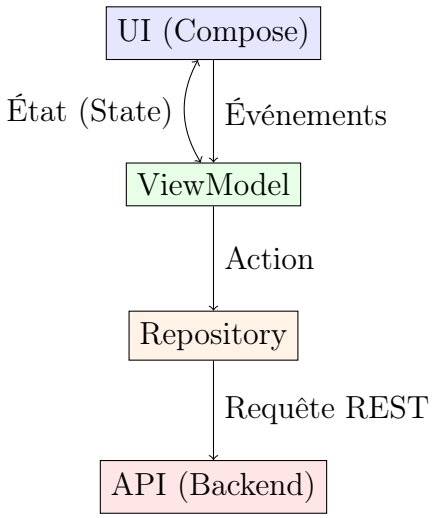
L'interface graphique est construite avec **Jetpack Compose**, un framework déclaratif moderne. Contrairement aux interfaces XML traditionnelles, Compose permet de lier directement l'état de l'application à l'affichage, rendant l'UI plus réactive et plus facile à maintenir. Les écrans principaux incluent :

— Écran de connexion et d'inscription.
— Écran principal de discussion (chat).
— Gestion dynamique de l'historique des messages.

3.2.3 Architecture : Clean Architecture et MVVM

L'application suit les principes de la **Clean Architecture** combinés au pattern **MVVM (Model-View-ViewModel)**. Cette organisation permet une séparation claire des responsabilités :

— **Couche Présentation :** Composants Jetpack Compose et ViewModels responsables de l'état de l'interface.
— **Couche Domaine :** Contient la logique métier et les modèles principaux (messages, conversations).
— **Couche Data :** Gère la communication réseau et l'accès aux sources de données distantes.



3.3 Communication Réseau et Backend

La communication entre l'application Android et le backend est réalisée via une API REST.

3.3.1 Retrofit et OkHttp

La librairie **Retrofit** est utilisée pour définir les endpoints réseau et sérialiser les données échangées. **OkHttp** complète cette configuration en gérant les timeouts étendus nécessaires aux réponses de l'agent IA, ainsi que l'ajout automatique des en-têtes de sécurité.

3.3.2 Authentification et Sécurité

L'authentification repose sur un mécanisme de jeton **JWT**. Après l'inscription, un processus d'*auto-login* est déclenché afin de récupérer immédiatement un jeton valide. Celui-ci est stocké de manière sécurisée à l'aide de **Jetpack DataStore**, garantissant une persistance fiable et asynchrone.

3.4 Système de Chat avec Agent IA Unique

Dans cette première version, chaque message envoyé par l'utilisateur est transmis à un **unique agent IA** (DeepSeek-R1 exécuté localement via Ollama). Le workflow est le suivant :

1. L'utilisateur saisit un message dans l'interface de chat.
2. Le ViewModel envoie la requête au backend via le Repository.
3. Le backend transmet la requête à l'agent IA DeepSeek-R1.
4. La réponse générée est renvoyée à l'application Android.
5. L'interface met à jour l'historique de la conversation en temps réel.

Cette approche mono-agent a permis de valider la stabilité du flux de données, la gestion des états de chargement et l'affichage fluide des réponses.

3.5 Gestion de l'État et Performance UI

L'état du chat est centralisé dans un objet `ChatState` observé par l'interface. L'utilisation de `LazyColumn` pour l'affichage des messages permet d'optimiser les performances, même en présence d'un grand nombre de messages, en ne rendant que les éléments visibles à l'écran.

3.6 Apports de cette Première Étape

Le développement de cette application Android mono-agent a constitué une étape clé du projet. Il a permis :

— De valider les choix d'architecture mobile.
— De tester l'intégration d'un agent IA réel dans une application Android.
— D'identifier les contraintes de latence et de performance côté client.
— De poser des bases solides pour l'évolution vers une architecture multi-agents, présentée dans les sections suivantes.

Cette application Android servira ainsi de socle pour l'intégration du backend avancé et du système de comparaison intelligente entre plusieurs agents IA.

4 Backend : Conception et Implémentation

Le backend constitue le cœur logique et fonctionnel du système SudChat. Il assure l'authentification des utilisateurs, la gestion des conversations, la persistance des données et la communication entre l'application Android et les agents d'intelligence artificielle. Son architecture a été pensée pour être modulaire, sécurisée et évolutive, afin de supporter la transition progressive d'un système mono-agent vers une architecture multi-agents.

4.1 Technologies Utilisées

Le backend repose sur une stack moderne orientée performance et maintenabilité :
- **NestJS** : Framework Node.js permettant de construire des API REST robustes et modulaires.
- **TypeScript** : Typage statique garantissant une meilleure maintenabilité du code.
- **MongoDB** : Base de données NoSQL orientée documents.
- **Mongoose** : ODM pour la définition des schémas et la validation des données.
- **JWT** : Authentification stateless sécurisée.
- **Docker** : Conteneurisation et portabilité du backend.

4.2 Architecture Générale

Le backend suit une architecture modulaire conforme aux bonnes pratiques NestJS. Chaque fonctionnalité est isolée dans un module indépendant.
- `auth/` : Gestion des utilisateurs, authentification JWT et guards.
- `chat/` : Gestion des conversations et messages.
- `schemas/` : Modèles MongoDB (User, Conversation, Message).
- `app.module.ts` : Module principal de l'application.

4.3 Authentification et Sécurité

L'authentification repose sur des tokens JWT signés et à durée de vie limitée. Les mots de passe sont stockés sous forme hashée à l'aide de **bcrypt**. Les routes sensibles sont protégées par des guards JWT.

4.4 Flux d'Authentification

1. Envoi des identifiants à `/auth/login`.
2. Validation des données via DTOs.
3. Vérification des identifiants et génération du JWT.
4. Retour du token au client.
5. Accès sécurisé aux routes protégées.

4.5 Gestion des Données

La persistance est assurée par MongoDB :
- **User** : email, mot de passe hashé, date de création.
- **Conversation** : participants, messages, dates.
- **Message** : auteur, contenu, horodatage, statut.

4.6 API REST

Le backend expose une API RESTful structurée :
— Authentification : inscription, connexion, refresh token.
— Chat : création de conversations, envoi et récupération des messages.
— Utilisateurs : gestion des profils.

4.7 Gestion des Erreurs et Logs

Les erreurs sont gérées de manière centralisée grâce aux filtres d'exceptions NestJS. Les logs permettent de tracer les requêtes et les erreurs critiques.

4.8 Lien avec l'Évolution Multi-Agents

Dans cette première phase, le backend agit comme un intermédiaire entre l'application Android et un seul agent IA (DeepSeek-R1 via Ollama). Sa conception modulaire permet d'intégrer ultérieurement plusieurs agents IA, de diffuser les requêtes en parallèle et de comparer les réponses sans modifier la base existante.

5 Prototype du Juge Multi-Agents (Première Implémentation)

Cette section présente une première implémentation expérimentale du **Juge multi-agents**, conçue comme un prototype fonctionnel permettant de valider le concept central du projet. Cette étape ne représente pas une solution finale optimisée, mais une *première tentative* visant à tester la faisabilité, la pertinence et les performances d'un mécanisme de comparaison intelligente entre plusieurs intelligences artificielles.

5.1 Objectif du Prototype

L'objectif principal de ce prototype est de démontrer qu'il est possible de :
— Diffuser une même requête utilisateur vers plusieurs agents IA en parallèle.
— Collecter leurs réponses indépendantes.
— Évaluer ces réponses selon des critères qualitatifs.
— Sélectionner automatiquement la réponse la plus pertinente pour l'utilisateur final.
Ce prototype constitue ainsi une **preuve de concept (PoC)** du système de décision multi-agents.

5.2 Architecture Globale

Le projet repose sur une architecture client-serveur moderne utilisant les technologies suivantes :
— **Backend :** Python avec le framework `FastAPI` (Asynchrone).
— **Communication :** WebSockets pour une interaction temps réel bidirectionnelle.
— **Frontend :** HTML5, CSS3 et JavaScript pur (Vanilla JS).
— **IA :** Intégration via API de Gemini 1.5 Flash, GPT-4o-mini et DeepSeek-R1.

Flux de Données

1. L'utilisateur envoie un message depuis l'interface web.
2. Le serveur FastAPI reçoit le message via WebSocket.
3. Le module `brain.py` (Le Juge) analyse l'intention.
4. Si la requête est simple → Réponse immédiate.
5. Si la requête est complexe → Appel parallèle aux 3 modèles experts, puis synthèse.
6. La réponse finale est renvoyée au frontend et affichée.

5.3 Détails du Backend (Le Cerveau)

Le cœur du système réside dans le dossier `backend/`. Voici l'explication détaillée de chaque module.

5.3.1 Le Juge et le Routeur (`brain.py`)

Ce fichier contient la logique décisionnelle. La fonction `judge_and_route` agit comme un garde-fou intelligent.

```
async def judge_and_route(user_input: str, history=[]):
    # Prompt syst me d finissant les r gles
    system_instruction = """
    You are the Router for an AI System.
    TASK: Decide if this needs "Simple" processing or "Complex"
   reasoning.
    CRITERIA FOR 'SIMPLE': Greetings, Facts, Personal questions.
    CRITERIA FOR 'COMPLEX': Coding, Math, Debate, Reasoning.

    OUTPUT RULES:
    1. If SIMPLE: Respond directly.
    2. If COMPLEX: Output ONLY "COMPLEX_MODE".
    """
    # Appel    Gemini Flash (rapide et peu co teux) pour d cider
    res = await gemini_client.generate_content(...)
    if "COMPLEX_MODE" in res.text:
        return "COMPLEX", None
    else:
        return "SIMPLE", res.text
```

Listing 1 – Logique du Juge (brain.py)

5.3.2 La Synthèse des Experts (`brain.py`)

Si le mode complexe est activé, la fonction `synthesize_final_answer` est appelée. Elle prend les réponses brutes de Gemini, GPT-4o et DeepSeek, et utilise un modèle pour les fusionner en une réponse parfaite, éliminant les hallucinations et combinant les meilleurs points de chaque modèle.

5.3.3 Les Travailleurs (`ai.py`)

Ce fichier gère les connexions aux APIs externes. L'utilisation de `asyncio` est cruciale ici pour la performance.

```
# Exemple d'appel asynchrone pour ne pas bloquer le serveur
async def ask_gemini(prompt, history=[]):
    res = await asyncio.to_thread(
        gemini_client.models.generate_content,
        model="gemini-1.5-flash",
        contents=format_history(prompt, history)
    )
    return res.text
```

Listing 2 – Exécution Asynchrone

5.3.4 Le Serveur (`server.py`)

Point clé : Le Parallélisme. Pour le mode complexe, nous n'attendons pas les modèles l'un après l'autre. Nous les lançons tous en même temps grâce à `asyncio.gather` :

```
# Lancement simultan  des 3 requ tes
r1, r2, r3 = await asyncio.gather(
    ask_gemini(user_text, chat_history),
    ask_openai(user_text, chat_history),
    ask_deepseek(user_text, chat_history)
)
```

Cela divise le temps d'attente par 3 par rapport à une exécution séquentielle.

Conclusion Générale

Ce projet a permis de concevoir et de mettre en œuvre une solution complète de chatbot intelligent, allant d'une application mobile Android moderne jusqu'à un backend robuste intégrant des mécanismes avancés d'intelligence artificielle. L'approche adoptée s'inscrit dans une démarche progressive et méthodique, débutant par une architecture mono-agent afin de valider les choix techniques fondamentaux, puis évoluant vers une architecture multi-agents plus ambitieuse.

Le développement de l'application Android a constitué une étape essentielle du projet. Il a permis de mettre en place une interface utilisateur fluide, sécurisée et performante, reposant sur des technologies modernes telles que Kotlin, Jetpack Compose et la Clean Architecture. Cette première version mono-agent, basée sur DeepSeek-R1 via Ollama, a permis de tester concrètement l'intégration d'un agent IA dans un environnement mobile réel, tout en identifiant les contraintes liées à la latence, à la gestion de l'état et à l'expérience utilisateur.

Le backend, développé avec NestJS et MongoDB, a fourni une base solide, modulaire et sécurisée pour la gestion des utilisateurs, des conversations et des messages. Son architecture découplée, l'utilisation de JWT pour l'authentification, ainsi que la conteneurisation via Docker garantissent une bonne maintenabilité, une sécurité renforcée et une évolutivité adaptée à des déploiements à plus grande échelle.

L'apport majeur de ce projet réside dans l'introduction d'un prototype de juge multi-agents. Cette première implémentation expérimentale a permis de démontrer la pertinence d'une approche collaborative, dans laquelle plusieurs intelligences artificielles sont mises en concurrence afin de produire une réponse finale de meilleure qualité. Bien que ce mécanisme reste perfectible, il valide l'hypothèse selon laquelle la comparaison et la synthèse de réponses issues de différents modèles peuvent surpasser une approche mono-agent classique.

En perspective, plusieurs axes d'amélioration peuvent être envisagés, notamment l'optimisation des critères de jugement, l'introduction de métriques quantitatives, l'apprentissage adaptatif du juge, ainsi qu'une intégration plus avancée côté mobile. Ainsi, ce projet constitue une base technique et conceptuelle solide pour le développement futur d'assistants intelligents plus fiables, plus performants et mieux adaptés aux besoins complexes des utilisateurs.