Rapport SAE Semestre 6

Chatbot intelligent pour la maintenance automobile

avec RAG et LLM

Réalisé par :
SORBO Samba - CHRIMNI Younes - TEMIZ Arda - FATHI Ilyes
Groupe : BUT3 FA – Groupe Cerbère
Université Sorbonne Paris Nord – IUT de Villetaneuse

Table des matières

1. Introduction du projet

ChatBotAuto est une application web full-stack proposant un chatbot intelligent spécialisé dans le domaine de l'automobile. L'objectif principal du projet est d'offrir aux utilisateurs un assistant virtuel capable de répondre à des questions pointues sur l'automobile (entretien, pannes, caractéristiques techniques, etc.), de manière fiable et contextualisée. Le contexte de ce projet s'inscrit dans l'essor des modèles de langage (LLM) et des chatbots tels que ChatGPT, qui ont démontré l'utilité de l'IA conversationnelle pour de nombreuses tâches. Cependant, un modèle généraliste seul peut manquer de connaissances spécifiques ou à jour, et les entreprises comme les utilisateurs ont besoin de modèles spécialisés sur un domaine précis, capables de comprendre le contexte en temps réel. ChatBotAuto vise justement à combler ce besoin dans le secteur automobile, en combinant un modèle de langage avancé avec une base de connaissances métier.

Conçu comme un assistant pour les passionnés ou professionnels de l'auto, ce chatbot se veut utile et pratique pour ses usagers. Par exemple, un utilisateur pourra demander des conseils de maintenance sur un véhicule, des explications sur le fonctionnement d'une pièce mécanique, ou encore les dernières informations réglementaires, et obtenir une réponse instantanée. L'utilité pour l'usager réside dans le gain de temps et la fiabilité des réponses : au lieu de chercher manuellement dans des documents techniques ou sur Internet, il peut dialoguer naturellement avec ChatBotAuto. Le système est pensé pour fournir des réponses précises et justifiées en s'appuyant sur une base documentaire validée, évitant autant que possible le phénomène d'« hallucination » des LLM (ces réponses inventées qui semblent plausibles). En résumé, ChatBotAuto ambitionne d'être un assistant virtuel de confiance dans le domaine automobile, accessible 24/7 via une simple interface web.

2. Architecture technique

ChatBotAuto repose sur une architecture technique moderne, articulée autour d'un backend Python pour la logique métier et l'IA, d'une base de données NoSQL pour stocker les informations et vecteurs, et d'un frontend web en Angular pour l'interface utilisateur. L'élément central est l'intégration d'un modèle d'intelligence artificielle de type LLM (Large Language Model) nommé *Mistral 7B*, couplé à un système de récupération de connaissances (*Retrieval Augmented Generation* ou RAG) alimenté par des documents PDF thématiques. Cette section détaille chaque composant de l'architecture.

2.1 Backend en Python (Flask, MongoDB)

Le backend de ChatBotAuto est développé en Python en s'appuyant sur le micro-framework Flask. Ce choix permet de mettre en place rapidement une API web RESTful légère et flexible, tout en profitant de l'écosystème Python riche en bibliothèques pour l'intelligence artificielle. Le rôle du serveur Flask est multiple : gérer les requêtes des clients (par exemple l'envoi d'une question au chatbot), orchestrer le processus de génération de réponse en impliquant le modèle d'IA et la base de connaissances, et retourner les réponses formatées au frontend. Flask a été configuré avec des routes spécifiques (endpoints) pour les fonctionnalités du chatbot, comme l'authentification, l'envoi de messages utilisateur, ou la récupération de l'historique de conversation. Grâce à sa simplicité, Flask a permis de

structurer le code backend de manière claire (routes, services, etc.) tout en conservant de bonnes performances pour une application de cette envergure.

Pour la persistance des données, ChatBotAuto utilise MongoDB comme base de données NoSQL. MongoDB a l'avantage d'être orienté document, ce qui convient bien au stockage d'objets JSON représentant par exemple les profils utilisateurs, les messages de conversation, ou encore les vecteurs de textes (embeddings) utilisés par l'IA. La base MongoDB contient ainsi plusieurs collections : une collection pour les utilisateurs (informations d'authentification), une pour les conversations (chaque document pouvant contenir l'historique d'un dialogue), et une pour les documents indexés (chunks issus des PDF, avec leurs vecteurs). Le choix de MongoDB s'explique aussi par l'émergence de fonctionnalités de recherche vectorielle dans cette base. En effet, MongoDB Atlas propose désormais des index spécialisés pour stocker et requêter des vecteurs, permettant d'effectuer des recherches par similarité sémantique directement au sein de la base. ChatBotAuto profite de cette capacité : lorsque le backend reçoit une question, il interroge MongoDB pour trouver, parmi les vecteurs stockés, ceux dont le contenu est le plus proche de la question posée. Cette intégration resserrée entre l'IA et la base de données évite d'avoir à déployer une infrastructure supplémentaire (pas de base vectorielle séparée), tout en maintenant de bonnes performances pour extraire le contexte pertinent.

2.2 Modèle d'IA Mistral 7B (HuggingFace)

Le "cerveau" du chatbot ChatBotAuto est un modèle de langage de grande taille (LLM) nommé Mistral 7B. Développé par la startup française Mistral AI et publié en open source en 2023, Mistral 7B est un modèle comptant environ 7,3 milliards de paramètres, réputé pour offrir une qualité de génération de texte exceptionnelle pour sa taille. Il a notamment été annoncé que Mistral 7B surpasse le modèle LLaMA 2 (13 milliards de paramètres) de Meta sur tous les benchmarks classiques, et rivalise même avec des modèles de 30 milliards de paramètres en efficacité. En pratique, cela signifie que malgré son encombrement mémoire réduit, Mistral 7B est capable de comprendre des questions complexes et de produire des réponses pertinentes, y compris en français. De plus, Mistral 7B possède un contexte étendu (jusqu'à 8192 tokens de texte), le rendant apte à ingérer des questions longues ou du contexte supplémentaire sans perdre le fil.

Dans ChatBotAuto, nous utilisons Mistral 7B en version Instruct (c'est-à-dire fine-tuné pour mieux suivre les instructions et répondre aux questions), via l'API HuggingFace. Concrètement, le backend Flask charge le modèle pré-entraîné depuis HuggingFace (par exemple le modèle `mistralai/Mistral-7B-Instruct-v0.1` ou v0.2) et l'invoque pour générer des réponses. L'inférence se fait soit sur CPU soit sur GPU selon les ressources disponibles, avec possibilité d'optimisations (quantification 4-bit via la librairie *BitsAndBytes*, streaming de tokens, etc.) afin de tenir les performances requises. L'intégration HuggingFace facilite grandement ce processus, car elle fournit les outils de tokenisation et de pipeline de génération. Ainsi, lorsque l'utilisateur envoie une question, le backend prépare une requête au modèle Mistral 7B en lui fournissant un prompt structuré (contenant éventuellement du contexte, voir section RAG ci-dessous), puis récupère la réponse textuelle générée. Mistral 7B ayant été publié sous licence Apache 2.0 et accessible librement, il a pu être déployé sans contrainte dans notre application. Son choix s'est révélé

judicieux pour obtenir un chatbot performant et autonome, sans dépendance à une API tierce payante.

2.3 Système RAG (Retrieval Augmented Generation)

Afin d'assurer des réponses précises et documentées, ChatBotAuto s'appuie sur un mécanisme de Retrieval Augmented Generation (RAG), soit en français une génération de texte augmentée par la récupération d'informations. Le principe du RAG est d'intégrer une étape de recherche documentaire avant la génération de la réponse par le modèle d'IA. En pratique, plutôt que de s'appuyer uniquement sur les connaissances statiques du LLM (qui peuvent être limitées ou obsolètes), le système va extraire dans sa base de connaissances les éléments pertinents liés à la question de l'utilisateur, puis les fournir au modèle sous forme de contexte additionnel. Ainsi, la réponse finale bénéficie à la fois des capacités linguistiques du modèle et de la justesse des données récupérées dans les documents.

Concrètement, lorsque le backend reçoit une question utilisateur, il commence par vectoriser cette question (voir section 3) pour obtenir une représentation mathématique de son sens, puis il effectue une recherche sémantique dans la base MongoDB des vecteurs. Cette recherche retourne les meilleurs chunks de documents (extraits de PDF) qui correspondent sémantiquement à la question posée. Supposons par exemple que l'utilisateur demande : *« À quelle fréquence doit-on changer l'huile d'une voiture diesel récente ? »* ; le système va trouver dans les PDF indexés un paragraphe pertinent (par exemple un extrait de manuel mentionnant les intervalles de vidange recommandés). Une fois ces extraits identifiés, le backend construit un prompt enrichi pour le modèle Mistral 7B : il combine la question de l'utilisateur et le contenu textuel des extraits trouvés, en les formatant de façon appropriée (par exemple *« En se basant sur les informations suivantes : [extraits] répondre à la question… »*). Ce prompt « augmenté » est alors soumis au modèle Mistral 7B, qui génère une réponse en s'appuyant explicitement sur ce contexte. Grâce à ce pipeline, le chatbot peut citer ou exploiter des informations exactes issues des PDF, ce qui améliore nettement la fiabilité de ses réponses. Le RAG permet donc d'atténuer les hallucinations et de fournir des réponses à jour (il suffit de mettre à jour les documents dans la base pour que le chatbot en tienne compte, sans devoir réentraîner le modèle lui-même). Ce fonctionnement confère à ChatBotAuto un avantage certain : il combine la puissance du modèle génératif et la précision d'une base de connaissances dédiée, pour délivrer une assistance experte aux utilisateurs.

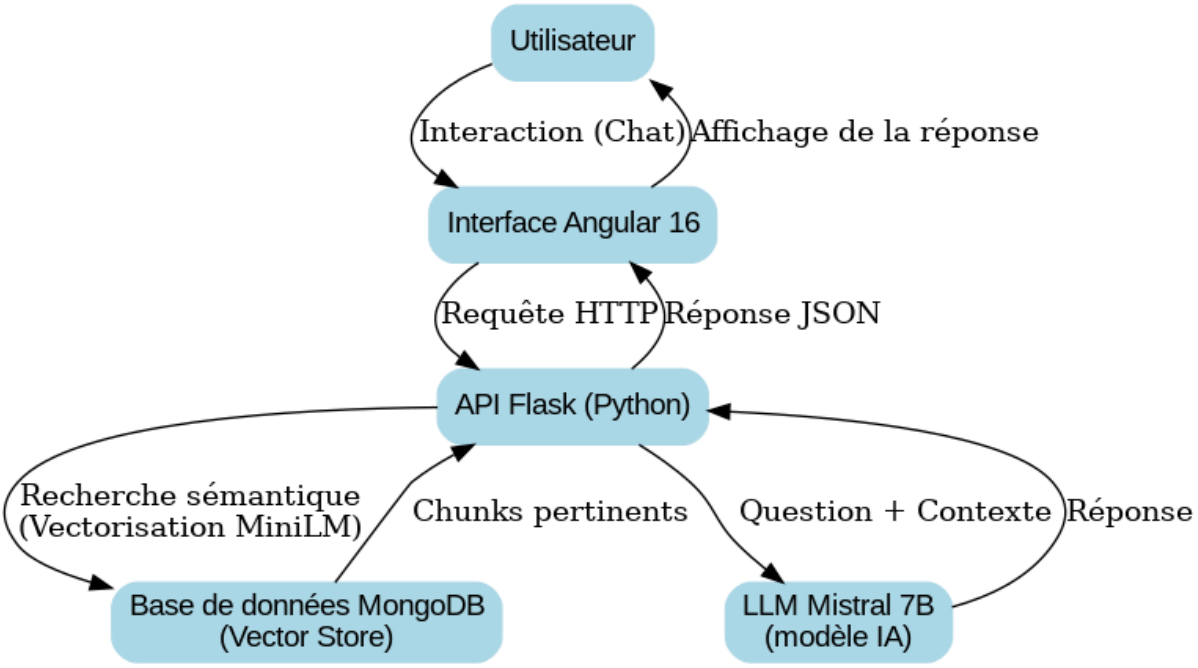

Schéma de l'architecture ChatBotAuto et du flux RAG. Le schéma ci-dessus illustre l'architecture du système ChatBotAuto, incluant le flux de données entre l'utilisateur, le frontend Angular, l'API Flask, la base de connaissances MongoDB et le modèle d'IA Mistral. Le chatbot reçoit d'abord la question de l'utilisateur via l'interface web, puis envoie cette question au serveur Flask. Le backend effectue une recherche RAG : il interroge la base de données MongoDB pour obtenir les chunks pertinents (après vectorisation de la question avec MiniLM), puis transmet la question enrichie de ce contexte au modèle Mistral 7B qui génère une réponse. Enfin, la réponse produite par l'IA est renvoyée à l'interface Angular pour être affichée à l'écran. Ce fonctionnement en boucle permet d'offrir à l'utilisateur une réponse personnalisée et appuyée sur des données fiables extraites des documents PDF.

3. Traitement des documents PDF

ChatBotAuto repose en grande partie sur la qualité de sa base de connaissances, alimentée par des documents PDF thématiques dans le domaine automobile. Pour que le chatbot puisse exploiter ces documents lors des conversations, un prétraitement a été réalisé en amont. Cette étape consiste à extraire le texte des PDF, à le découper en chunks, puis à vectoriser ces segments pour pouvoir les comparer à une question en langage naturel. Les vecteurs obtenus sont finalement stockés dans MongoDB afin d'être interrogés rapidement via le pipeline RAG. Voici comment ce traitement des documents a été mis en place :

3.1 Découpage en chunks et vectorisation (MiniLM)

Chaque document PDF (qu'il s'agisse d'un manuel technique, d'un article ou d'une fiche) peut contenir de nombreuses pages de texte. Les modèles de langage ayant une limite sur la quantité de texte qu'ils peuvent traiter d'un coup, il est nécessaire de découper les documents en morceaux plus petits appelés *chunks*. Un chunk correspond typiquement à un paragraphe ou à quelques phrases cohérentes, de sorte qu'il garde un sens contextuel

autonome. Ce choix de découpage vise un équilibre : les chunks doivent être assez longs pour contenir de l'information utile et du contexte, mais assez courts pour être ingérés individuellement par le modèle d'IA sans dépasser ses limites de tokens. Par exemple, on peut décider qu'un chunk fasse environ 100 à 200 mots (ou ~500 tokens) et découper chaque PDF en suivant cette règle, en veillant à ne pas couper en plein milieu d'une phrase ou d'une section importante.

Une fois les documents splittés en chunks textuels, la prochaine étape est la vectorisation de ces segments. Pour ce faire, ChatBotAuto utilise un modèle d'embedding de phrases de la famille MiniLM (plus précisément *all-MiniLM-L6-v2*). Il s'agit d'un modèle pré-entraîné qui convertit un texte en un vecteur de nombres (une liste de 384 valeurs dans notre cas) représentant son sens sémantique. En appliquant ce modèle à chaque chunk, on obtient ainsi une collection de vecteurs qui « encodent » le contenu des PDF de manière exploitable par machine. Deux textes ayant un sens proche produiront des vecteurs proches dans l'espace vectoriel. Cette propriété est au cœur de la recherche sémantique : elle permet de comparer une question (également vectorisée via MiniLM au moment de la requête) avec tous les chunks vectorisés, et de trouver lesquels sont les plus similaires, c'est-à-dire les plus susceptibles de contenir des éléments de réponse. MiniLM a été choisi car il offre un bon compromis entre performance et rapidité : il est léger et rapide tout en fournissant des embeddings de qualité pour la recherche de similarité sémantique.

3.2 Stockage dans MongoDB

Après l'obtention des embeddings pour tous les chunks de tous les documents PDF, ces vecteurs doivent être stockés dans une base de données pour être facilement requêtés par la suite. Comme évoqué dans la partie architecture, MongoDB sert de stockage des vecteurs. Concrètement, une collection (par exemple nommée `documents_vectorises`) a été créée dans MongoDB, où chaque entrée correspond à un chunk. Un document de cette collection contient notamment les champs suivants : le texte du chunk, éventuellement une référence au document source (ex : nom du PDF et numéro de page), et le vecteur d'embedding du chunk (représenté sous forme d'un tableau de nombres flottants). MongoDB permet de stocker ce type de données sans schéma fixe, ce qui facilite l'insertion des vecteurs.

Pour permettre la recherche par similarité efficace dans ces vecteurs, un index de type spécial a été mis en place. MongoDB Atlas, dans ses versions récentes, offre la possibilité de créer des index de recherche vectorielle sur un champ contenant un tableau de nombres, et de réaliser des requêtes de similarité cosinus ou euclidienne entre un vecteur de requête et les vecteurs indexés. Ainsi, lorsque l'application doit trouver les chunks pertinents pour une question donnée, elle envoie à MongoDB un vecteur (celui de la question) et utilise une requête de recherche vectorielle pour obtenir, par exemple, les 3 ou 5 documents les plus proches dans l'espace vectoriel. MongoDB retourne alors les identifiants et contenus de ces chunks, qui seront utilisés pour composer le contexte envoyé à Mistral 7B. Grâce à ce mécanisme directement dans la base de données, on évite de charger en mémoire tous les vecteurs et de calculer manuellement les distances, ce qui pourrait devenir coûteux. La scalabilité s'en trouve améliorée : si de nouveaux documents sont ajoutés (et leurs chunks vectorisés insérés dans la collection), ils seront automatiquement pris en compte par la recherche sans modification du code, simplement via l'index mis à jour. En résumé,

MongoDB joue le rôle d'une base de données vectorielle intégrée, stockant et retrouvant des embeddings de textes de manière optimisée pour alimenter le RAG.

3.3 Corpus PDF analysé

Le projet ChatBotAuto a utilisé cinq documents PDF thématiques, nommés *auto1* à *auto5*, pour constituer la base de connaissances initiale du chatbot. Ces fichiers couvrent un ensemble varié de sujets dans le domaine automobile afin de doter l'agent conversationnel d'une compréhension large du contexte. Par exemple, l'un des PDF peut être un guide d'entretien de véhicule détaillant les intervalles de maintenance et les bonnes pratiques mécaniques, un autre un recueil de fiches techniques sur des modèles de voitures, un troisième un document sur la réglementation (code de la route, normes environnementales, etc.), et ainsi de suite. Chaque fichier a été passé par le processus de découpage et de vectorisation décrit précédemment. On obtient au final une base de centaines de chunks vectorisés couvrant ces cinq documents.

En phase d'utilisation, cela signifie que pour une question donnée, le chatbot peut puiser des informations dans plusieurs sources PDF différentes. Par exemple, si la question porte sur les émissions polluantes d'un moteur diesel, il est possible que le système trouve un élément de réponse dans le PDF "réglementation" (pour les normes Euro) et un autre dans le PDF "fiches techniques" (pour les caractéristiques du moteur en question). L'agrégation de ces sources permettra au modèle d'IA de formuler une réponse complète et nuancée. Bien que le prototype se limite à cinq documents, l'architecture est conçue pour être évolutive : de nouveaux PDF peuvent être ajoutés à la base (en les vectorisant de la même manière) pour enrichir continuellement les connaissances du chatbot. Cela ouvre la voie à un assistant qui resterait à jour des nouveautés du domaine automobile, simplement en intégrant régulièrement les nouvelles documentations ou publications pertinentes.

4. Fonctionnalités du chatbot

L'application ChatBotAuto propose à l'utilisateur plusieurs fonctionnalités qui rendent l'expérience de chat enrichissante et personnalisée. Voici les principales fonctionnalités disponibles :

- Authentification utilisateur : ChatBotAuto implémente un système de création de compte et de connexion. Chaque utilisateur doit s'authentifier (par exemple via un email et un mot de passe) pour accéder au chatbot. Ceci permet de personnaliser l'expérience et de conserver de manière sécurisée l'historique des interactions propres à chaque compte. L'authentification est gérée côté backend (Flask) avec stockage sécurisé des mots de passe et vérification des jetons de session lors des requêtes.

- Historique des conversations : Une fois connecté, l'utilisateur peut retrouver l'historique de ses échanges avec le chatbot. Chaque question posée et réponse obtenue sont enregistrées, ce qui permet de consulter à tout moment les informations fournies précédemment. Cet historique est affiché sous forme de fil de discussion dans l'interface. D'un point de vue technique, chaque message (utilisateur

ou bot) est stocké en base de données avec un identifiant de conversation, de sorte qu'on peut reconstituer l'ordre chronologique des messages. Conserver le contexte est d'ailleurs utile non seulement pour l'utilisateur, mais aussi pour le chatbot lui-même si l'on souhaite qu'il en tienne compte dans les échanges (par exemple, se souvenir de la question précédente pour une relance).

- Suggestions rapides : L'interface propose des boutons de suggestion ou des phrases prédéfinies que l'utilisateur peut sélectionner en un clic pour poser rapidement une question fréquente. Par exemple, des suggestions comme *« Vérifier la pression des pneus »*, *« Problème de batterie »* ou *« Entretien moteur »* peuvent être présentées. Ces suggestions servent de raccourcis pour engager la conversation, notamment pour les nouveaux utilisateurs qui ne sauraient pas par où commencer. Elles sont le plus souvent définies statiquement en frontend (ou pourraient être générées dynamiquement selon le contexte) et permettent d'améliorer l'ergonomie de l'application.

- Sauvegarde des sessions : ChatBotAuto permet à l'utilisateur de sauvegarder une session de chat particulière et d'y revenir plus tard. Cela signifie que l'on peut avoir plusieurs fils de conversation distincts (par exemple un fil concernant les questions sur une voiture A, un autre fil pour une voiture B). L'interface inclut une sidebar listant les différentes sessions sauvegardées, que l'on peut nommer (par exemple "Discussion entretien Clio") et recharger à volonté. Techniquement, cela se traduit par un champ session ou conversation ID associé à chaque message en base, et par la possibilité de créer une nouvelle session (nouvel ID) sur demande. Cette fonctionnalité apporte une flexibilité d'usage proche de celle d'outils comme ChatGPT, où l'on peut mener plusieurs conversations indépendantes.

En plus de ces points, le chatbot conserve bien sûr les fonctionnalités standard d'un agent conversationnel : compréhension du langage naturel, réponses en langage courant, possibilité de demander des clarifications, etc. L'ensemble de ces fonctionnalités vise à rendre ChatBotAuto à la fois puissant par ses capacités IA et convivial par son interface, de manière à convaincre un utilisateur ou un jury de son utilité pratique.

5. Interface utilisateur

Le frontend de ChatBotAuto est une application web monopage (SPA) réalisée avec Angular 16, une des dernières versions du framework front-end maintenu par Google. Angular a été choisi pour sa robustesse et sa structure modulaire, ce qui a aidé à organiser le code de l'interface en composants réutilisables (barre de navigation, fenêtre de chat, formulaire de login, etc.). L'utilisation de TypeScript et le système de templating d'Angular ont permis de développer une interface dynamique tout en assurant une bonne maintenabilité du code. L'application Angular communique avec le backend Flask via des appels HTTP (par exemple, en utilisant le service HttpClient d'Angular) pour envoyer les questions de l'utilisateur et récupérer les réponses du chatbot en JSON.

Du point de vue design et ergonomie, l'interface utilisateur se veut moderne et intuitive. On retrouve une sidebar (barre latérale) permettant de naviguer entre les différentes sections. La zone principale de l'écran est dédiée à la fenêtre de conversation : les messages de l'utilisateur et ceux du bot y apparaissent sous forme de bulles successives, à la manière des applications de messagerie classiques. Un champ de saisie en bas permet à l'utilisateur de taper sa question, avec un bouton pour envoyer ou activer la saisie vocale.

Par ailleurs, l'interface utilise Angular Material, la bibliothèque officielle de composants UI pour Angular, qui implémente les directives Material Design de Google. Cela a grandement aidé à construire des composants uniformes et accessibles (par exemple l'utilisation de `MatToolbar` pour l'en-tête, de `MatList` pour la liste des sessions, de `MatButton` pour les boutons iconiques, etc.). Angular Material fournit des APIs simples et un comportement cohérent cross-plateforme, tout en permettant une personnalisation poussée du thème. Nous avons tiré parti de cette flexibilité pour appliquer le thème personnalisé inspiré du domaine automobile (voir section suivante) sur les composants Material. En somme, le choix d'Angular 16 combiné à Angular Material et Bootstrap a permis de réaliser une interface utilisateur à la fois esthétique, performante et robuste, digne d'une application professionnelle. Le résultat est une expérience utilisateur fluide, où l'on peut interagir naturellement avec le chatbot ChatBotAuto comme on le ferait avec un assistant humain.

6. Aspects visuels et design

L'identité visuelle de ChatBotAuto a été travaillée pour refléter le domaine de l'automobile et créer une ambiance conviviale rappelant l'univers des garages. Pour ce faire, nous nous sommes inspirés des éléments typiques d'un atelier mécanique dans la charte graphique de l'application. Tout d'abord, un logo a été créé : il représente un petit personnage mascotte à l'allure de mécanicien (casquette et clé à molette à la main) combiné à une bulle de dialogue, symbolisant ainsi le mélange entre la mécanique automobile et la conversation intelligente. Ce logo apparaît notamment sur l'écran de connexion et dans la barre latérale de l'application, pour bien marquer l'identité du chatbot.

La palette de couleurs choisie renforce également le thème du garage. Les couleurs dominantes sont le rouge et le bleu, deux teintes souvent associées à l'univers auto (on pense aux enseignes de garages, aux combinaisons de mécaniciens, ou encore à des marques historiques du secteur). Le rouge est utilisé par petites touches pour les éléments interactifs ou importants (boutons d'action, éléments sélectionnés, avatar du bot), tandis que le bleu plus doux sert de couleur d'arrière-plan ou de fond pour les bulles de messages du bot par exemple. Ce contraste rouge/bleu apporte du dynamisme visuel tout en restant agréable à l'œil. En mode sombre, les mêmes codes couleurs sont conservés mais avec des nuances adaptées (rouge bordeaux foncé, bleu tirant sur le gris anthracite) afin de conserver une bonne lisibilité. Les polices de caractères utilisées sont sobres et sans empattement, pour une lecture aisée des contenus textuels (questions et réponses).

ChatBotAuto n'est pas qu'une application neutre, il a une personnalité visuelle chaleureuse, rappelant un bon mécanicien à l'ancienne sur qui on peut compter. Le retour des premiers utilisateurs tests a été positif, notant que l'interface était attrayante et qu'on identifiait immédiatement le domaine grâce à cette thématique visuelle bien intégrée.

7. Défis rencontrés et solutions

Comme tout projet intégrant des composants variés (IA, base de données, frontend), ChatBotAuto a dû surmonter plusieurs défis techniques au cours de son développement. Nous présentons ici les principaux obstacles rencontrés, regroupés par domaine, ainsi que les solutions apportées pour y remédier.

7.1 Côté IA (modèle et RAG)

Le premier défi majeur concernait l'intégration du modèle Mistral 7B et la gestion de ses contraintes. Avec 7 milliards de paramètres, ce modèle est relativement lourd à charger en mémoire et à exécuter. Sur notre infrastructure de développement, nous avons dû optimiser son utilisation : cela a impliqué l'usage de la quantification en 4 bits (via la bibliothèque `bitsandbytes`) pour réduire l'empreinte mémoire, ainsi que le chargement en mode « autorégressif » sans modifier le modèle (pas de fine-tuning supplémentaire, uniquement de l'inférence). Malgré ces optimisations, le temps de réponse initial du modèle était parfois un peu élevé (quelques secondes par réponse). De plus, nous avons limité la longueur maximale des réponses générées pour éviter que le modèle ne monopolise les ressources trop longtemps sur une seule question.

Un autre défi a été de garantir la pertinence des réponses fournies par le modèle, en particulier d'éviter les erreurs factuelles. Au début, nous avons constaté que si le pipeline RAG ne renvoyait pas de bons extraits ou en renvoyait trop peu, le modèle pouvait combler les trous par de l'invention. La solution a été d'affiner le prompt envoyé à Mistral 7B : nous lui avons ajouté des instructions explicites du type *« Réponds en t'appuyant sur les informations suivantes... Si une information manque, réponds que tu ne sais pas au lieu d'inventer. »*. Cette ingénierie de prompt a significativement réduit les hallucinations. Nous avons également fait des tests pour déterminer le nombre optimal de chunks de contexte à fournir : trop peu conduisait à des réponses incomplètes, trop en donner pouvait noyer le modèle. Fournir 3 extraits pertinents dans le prompt semble donner un bon équilibre entre contexte suffisant et concision.

7.2 Côté base de données

Du côté de la base de données, le principal défi fut la mise en place de la recherche vectorielle avec MongoDB. Au début du projet, nous n'étions pas certains que MongoDB pourrait remplir ce rôle car historiquement il s'agit d'une base orientée documents et non d'un moteur de similarité. Nous avons envisagé l'utilisation de solutions dédiées comme FAISS (Facebook AI Similarity Search) ou ElasticSearch avec plugin vectoriel, voire des services gérés comme Pinecone. Toutefois, en découvrant que MongoDB Atlas proposait désormais nativement des index vectoriels, nous avons décidé d'essayer cette voie pour simplifier l'architecture (éviter une dépendance supplémentaire). La configuration de l'index vectoriel MongoDB nous a demandé un peu de lecture de documentation et de tâtonnements pour choisir la bonne métrique de similarité (cosine) et normaliser les vecteurs. Mais une fois correctement paramétré, le système a fonctionné. Nous avons validé la justesse des résultats en faisant des tests manuels : pour des questions connues, vérifier que les chunks renvoyés étaient logiques. Dans l'ensemble, MongoDB s'est bien comporté,

mais nous gardons à l'esprit que si le volume de données devient très grand ou les performances insuffisantes, nous pourrions migrer vers une solution spécialisée plus tard.

Un autre défi a concerné la structure des données pour l'historique de conversation. Faut-il stocker chaque message comme un document indépendant lié par un identifiant de conversation, ou stocker une conversation entière comme un document contenant un tableau de messages ? Nous avons testé les deux approches. La première (chaque message = un document) est plus souple pour requêter un sous-ensemble de messages, mais la seconde (un document = une conversation complète) facilite la récupération d'un historique entier d'un coup. Nous avons opté pour la seconde option compte tenu du volume modéré de messages par conversation, ce qui simplifie aussi la suppression ou la sauvegarde d'une session en une opération. Ce choix a néanmoins nécessité de faire attention aux concurrences d'accès : si deux messages arrivent quasi simultanément sur la même conversation, il faut s'assurer de ne pas perdre l'un d'eux lors de la mise à jour du tableau. Pour cela, nous avons utilisé une opération atomique (comme `$push` de MongoDB) pour ajouter un message dans le tableau des messages d'une conversation.

Enfin, la gestion des droits d'accès et de la sécurité sur la base MongoDB a requis de la vigilance. Nous avons mis en place un utilisateur MongoDB dédié à l'application avec des droits restreints aux collections nécessaires. De même, les identifiants de connexion à la base sont stockés côté serveur avec les bonnes pratiques (variables d'environnement, jamais en clair dans le code source). Ces mesures assurent que même si le code frontend est exposé, les accès à la base restent protégés.

7.3 Côté affichage

Nous avons porté attention à la gestion des erreurs côté interface. Par exemple, si le serveur ne répond pas (panne de l'IA ou base de données non joignable), le frontend affiche un message d'erreur informatif ("Erreur de communication avec le serveur."). Ce type de cas a été géré en interceptant les erreurs HTTP dans Angular (via un Interceptor global) et en produisant des notifications utilisateur élégantes (message dans la zone de chat). Un défi spécifique a été de prévenir les injections HTML dans les réponses du bot : on souhaite afficher du texte potentiellement long, en conservant les sauts de ligne, mais sans exécuter d'HTML malveillant si jamais il y en avait. Nous avons donc utilisé le binding Angular sécurisé (`<div [textContent]="reponse">` plutôt que `[innerHTML]`) pour afficher la réponse brute et éviter tout problème de sécurité côté client.

Enfin, l'intégration simultanée de Bootstrap et Angular Material a nécessité de régler quelques conflits mineurs de styles CSS. Par exemple, Bootstrap applique un `box-sizing` global qui influençait le rendu de certains composants Material. En ajustant l'ordre d'import des CSS et en ciblant quelques classes spécifiques, nous avons réussi à faire cohabiter les deux frameworks. À l'avenir, il serait sans doute plus propre de n'utiliser qu'un seul système (Material par exemple, qui couvre déjà beaucoup de besoins), mais dans le cadre de ce projet, la combinaison des deux nous a fait gagner du temps.

Malgré ces divers défis, chaque problème a trouvé sa solution au fil du développement, grâce à des recherches (documentation officielle, forums, etc.) et à des tests réguliers. Ce travail de résolution de problèmes fait partie intégrante du projet et nous a permis

d'améliorer continuellement ChatBotAuto jusqu'à obtenir une version stable et aboutie pour la présentation finale.

8. Conclusion et pistes d'amélioration

En conclusion, le projet ChatBotAuto a abouti à la création d'une application web fonctionnelle combinant les avancées en intelligence artificielle et une interface conviviale pour offrir un chatbot spécialisé en automobile. Ce rapport a présenté les objectifs initiaux, l'architecture technique mise en place (backend Flask, base MongoDB, modèle Mistral 7B intégré via RAG), le traitement des documents sources, les fonctionnalités phares pour l'utilisateur, ainsi que les choix de design et les défis surmontés. ChatBotAuto se révèle utile pour quiconque souhaite obtenir rapidement des informations fiables sur des sujets automobiles variés, à travers une simple conversation en langage naturel. Le projet démontre qu'allier un LLM performant à une base de connaissances métier est non seulement possible, mais efficace pour créer des agents conversationnels experts dans un domaine donné.

Bien que pleinement opérationnel, ChatBotAuto offre de nombreuses pistes d'amélioration pour aller encore plus loin. On pourrait envisager d'améliorer le modèle d'IA lui-même : soit en entraînant (fine-tuning) Mistral 7B sur un corpus de questions-réponses automobiles pour le spécialiser davantage, soit en passant à un modèle plus grand (si les ressources le permettent) afin d'obtenir des réponses encore plus précises et nuancées.

Parmi les autres améliorations, on peut citer l'enrichissement de la base de connaissances avec encore plus de documents (par ex. intégrer des manuels constructeurs complets, des FAQs de forums auto, etc.), la mise à jour continue des données (pour que le chatbot reste à jour des évolutions techniques ou réglementaires), ou encore l'ajout d'un module d'analyse d'images pour, pourquoi pas, permettre à l'utilisateur d'envoyer une photo d'une pièce défectueuse et obtenir de l'aide à ce sujet. Sur le plan de l'interface, on pourrait intégrer un système de notation des réponses par l'utilisateur (feedback thumbs-up/down) afin d'identifier les cas où le bot se trompe et affiner le système. Enfin, en termes de déploiement, passer l'application en production sur un serveur cloud plus puissant, avec éventuellement une architecture micro-services (séparer le service d'embedding, le service de génération, etc.), améliorerait la scalabilité pour supporter de nombreux utilisateurs simultanés.

En synthèse, ChatBotAuto est une base solide démontrant l'intérêt d'un chatbot spécialisé et les solutions techniques pour le construire. Les pistes d'amélioration témoignent du potentiel d'évolution du projet vers un produit encore plus abouti. Ce projet aura été pour nous l'occasion d'approfondir des compétences variées (IA, développement web, bases de données, UX design) et de relever le défi de les assembler en un tout cohérent. Nous espérons que ChatBotAuto saura convaincre de l'importance des chatbots à intelligence augmentée (RAG) dans l'avenir des services d'information, y compris dans le secteur automobile.