الجمـهورية الجزائرية الديمقراطية الشعبيـة
وزارة التعليم العالي والبحث العلمي
جـامعـة سعـيـدة د. مـولاي الطـاهـر
كلية الرياضيات و الإعلام الآلي و الاتصالات السلكية و اللاسلكية
قسم: الإعلام الآلي

Mémoire de Master

En vue de l'obtention du diplôme de Master Spécialité :

Intelligence Artificielle / Informatiqu

Theme

Conception et réalisation d' un chatbot intelligent pour le e-commerce

▪ *Réalisé par :*

MERZOUGHI Youcef abderrezak
SOUIDI Abdelhakim zakaria

▪ *Encadré par :*

Mokri Miloud Aboubakeur El Sadek

Année universitaire : 2025–2026

Remerciements

Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce mémoire.

Je remercie particulièrement mon encadrant pour son accompagnement, ses conseils précieux, ses orientations scientifiques et sa disponibilité tout au long de ce travail de recherche. Ses directives ont été essentielles pour mener à bien cette étude sur l'intelligence artificielle et son application au commerce électronique.

Mes remerciements vont également à l'ensemble du corps professoral du département d'Informatique de la Faculté de Mathématiques, Informatique et Télécommunications de l'Université Dr Moulay Tahar de Saida, pour la qualité de l'enseignement dispensé durant mon parcours universitaire.

Enfin, je souhaite remercier chaleureusement ma famille, mes amis, ainsi que toutes les personnes qui m'ont soutenu moralement et encouragé tout au long de mes études.

Résumé

Ce mémoire s'inscrit dans le domaine du commerce électronique et de l'intelligence artificielle, avec pour objectif principal l'étude, la conception et l'implémentation d'un chatbot intelligent destiné à améliorer l'interaction entre une entreprise et ses clients. Afin de répondre aux exigences des consommateurs modernes qui attendent des services rapides, personnalisés et disponibles en continu, ce travail propose un assistant conversationnel déployé sur la plateforme Telegram.

Le système repose sur une **Architecture Hexagonale** (Ports et Adaptateurs), garantissant une séparation stricte entre la logique métier, hautement indépendante, et les composants d'infrastructure tels que les frameworks ou les bases de données. Sur le plan technologique, le projet intègre le langage Python, la bibliothèque LangChain, le modèle de langage local Ollama (Llama 3), et une base de données SQLite. Cette conception robuste permet au chatbot de traiter les demandes dans leur contexte via une machine à états intégrée, d'interagir avec le catalogue de produits, de gérer un panier d'achats, et de générer des réponses naturelles et précises.

Mots-clés : chatbot, e-commerce, intelligence artificielle, architecture hexagonale, Telegram, LLM, LangChain, Ollama, NLP.

Abstract

This thesis lies at the intersection of e-commerce and artificial intelligence, aiming to design and implement an intelligent chatbot intended to improve the interaction between a company and its customers. To meet the demands of modern consumers who expect fast, personalized, and continuously available services, this work proposes a conversational assistant deployed on the Telegram platform.

The system is built upon a strict **Hexagonal Architecture** (Ports and Adapters), ensuring a clean separation between the core business logic and infrastructural components such as external frameworks or databases. Technologically, the project integrates Python, the LangChain library, the local Large Language Model Ollama (Llama 3), and an SQLite database. This robust design allows the chatbot to process requests contextually via a built-in state machine, query a product catalog, manage a shopping cart, and generate natural, accurate responses.

Keywords : chatbot, e-commerce, artificial intelligence, hexagonal architecture, Telegram, LLM, LangChain, Ollama, NLP.

Table des matières

Table des figures

Liste des tableaux

Introduction Générale

Contexte et Motivation

La transformation numérique a profondément redéfini les pratiques commerciales mondiales au cours des dernières décennies. L'omniprésence d'Internet, couplée à la démocratisation des terminaux mobiles, a conduit à une explosion du commerce électronique (e-commerce). Cette digitalisation ne se limite pas à la simple transaction d'achat et de vente, mais englobe l'intégralité du parcours client, de la découverte d'un produit à l'assistance post-achat [9].

Dans cet écosystème numérique, le consommateur moderne est devenu plus exigeant. Il attend une accessibilité permanente, une navigation fluide, et surtout, un support client capable de répondre instantanément à ses interrogations, qu'il s'agisse de la disponibilité d'un produit, des modalités de livraison, ou d'un conseil personnalisé [2]. Cependant, maintenir un service client humain disponible 24 heures sur 24 et 7 jours sur 7 représente un coût opérationnel prohibitif pour la grande majorité des entreprises, en particulier pour les startups et les petites ou moyennes entreprises (PME).

C'est dans ce contexte de recherche d'optimisation et d'amélioration de l'expérience utilisateur que l'Intelligence Artificielle (IA) et plus particulièrement le Traitement Automatique du Langage Naturel (NLP - *Natural Language Processing*) ont fait une entrée remarquée. Les agents conversationnels, communément appelés chatbots, se sont imposés comme une solution de choix pour automatiser les interactions de premier niveau [8]. Historiquement limités à des arbres de décision rigides et scénarisés, les chatbots modernes connaissent aujourd'hui une révolution conceptuelle grâce aux Modèles de Langage de Grande Taille (LLM - *Large Language Models*) [13]. Ces nouveaux systèmes sont capables de comprendre le contexte, de générer des réponses naturelles et fluides, et de s'adapter dynamiquement aux requêtes imprévisibles des utilisateurs.

Problématique

Malgré les avancées technologiques spectaculaires des LLMs, leur intégration concrète dans un système de commerce électronique soulève de nombreux défis techniques et ar-

chitecturaux. Un chatbot e-commerce ne peut pas se contenter de générer du texte de manière probabiliste ; il doit être fermement ancré dans la réalité de l'entreprise. Il doit interroger une base de données en temps réel pour connaître le stock, mémoriser l'historique de la conversation, gérer le cycle de vie d'un panier d'achats, et interagir avec des interfaces de messagerie grand public, le tout en garantissant des temps de réponse faibles et une grande fiabilité.

La problématique principale de ce mémoire peut donc se formuler ainsi : **Comment concevoir et implémenter un agent conversationnel intelligent pour le e-commerce, capable d'exploiter la puissance générative des LLMs modernes (via des outils comme Ollama et LangChain) tout en s'intégrant de manière robuste, modulaire et évolutive au sein d'une architecture logicielle stricte (Architecture Hexagonale) ?**

Objectifs de l'Étude

Pour répondre à cette problématique, ce travail se fixe les objectifs suivants :

1. **Étude Théorique :** Analyser l'évolution de l'assistance client dans le domaine du e-commerce et examiner les fondamentaux théoriques sous-jacents aux technologies d'Intelligence Artificielle modernes (Transformers, LLMs).
2. **Conception Architecturale :** Proposer une architecture logicielle découplée (Ports et Adaptateurs) permettant de séparer le raisonnement de l'IA et la gestion des règles métier, des détails d'infrastructure tels que la base de données ou la plateforme de messagerie (Telegram).
3. **Développement et Implémentation :** Réaliser un prototype fonctionnel complet en Python, intégrant un LLM local via Ollama et LangChain, une base de données relationnelle SQLite, et l'API de messagerie Telegram, tout en orchestrant la logique via une machine à états implicite.
4. **Évaluation :** Tester le comportement du système en conditions réelles simulées pour évaluer la pertinence des réponses, la capacité à maintenir le contexte, et l'efficacité de la gestion du panier.

Structure du Mémoire

Afin d'atteindre ces objectifs de manière structurée et méthodique, le présent document est organisé en quatre chapitres principaux :

— **Le Chapitre 1** dresse un état de l'art du commerce électronique, met en exergue l'importance cruciale du service client dans ce domaine, et retrace l'évolution des chatbots comme solution d'automatisation.

— **Le Chapitre 2** est consacré aux fondements théoriques de l'Intelligence Artificielle et du Traitement Automatique du Langage Naturel. Il explore l'architecture des Transformers, le fonctionnement des LLMs, et présente les outils modernes tels que LangChain et Ollama qui facilitent leur intégration.
— **Le Chapitre 3** plonge au cœur de la conception logicielle. Il justifie le choix de l'Architecture Hexagonale, détaille la modélisation conceptuelle des données et des flux d'interaction, et modélise la logique métier régissant le processus d'achat conversationnel.
— **Le Chapitre 4** porte sur la réalisation technique et l'implémentation. Il décrit l'environnement de développement, l'intégration des différents adaptateurs (Telegram, SQLite, In-Memory Session, Ollama), et présente les résultats des scénarios de test.

Une **Conclusion Générale** viendra clôturer ce mémoire en dressant le bilan des travaux réalisés et en esquissant les perspectives d'évolution futures de notre solution.

Chapitre 1

État de l'Art : E-commerce, Expérience Utilisateur et Chatbots

1.1 Introduction

L'essor d'Internet et des technologies de l'information a fondamentalement bouleversé les paradigmes économiques traditionnels. La numérisation des échanges a donné naissance à de nouveaux modèles d'affaires, au cœur desquels figure le commerce électronique, ou e-commerce. Aujourd'hui, les entreprises ne se contentent plus de vendre en ligne ; elles s'engagent dans une course continue pour offrir la meilleure expérience utilisateur possible afin de fidéliser une clientèle de plus en plus volatile et exigeante.

Dans ce contexte hautement concurrentiel, le service client est devenu un élément de différenciation stratégique majeur. La nécessité d'offrir une assistance rapide, personnalisée et disponible de manière ininterrompue a poussé les acteurs du marché à explorer des solutions d'automatisation avancées. Parmi ces solutions, les agents conversationnels, plus connus sous le nom de chatbots, occupent aujourd'hui une place prépondérante.

Ce premier chapitre a pour objectif de dresser un panorama complet du contexte entourant notre étude. Nous commencerons par analyser l'évolution et les tendances du e-commerce, avant de souligner l'importance cruciale de l'expérience utilisateur et du service client. Enfin, nous retracerons l'évolution des chatbots, de leurs versions rudimentaires basées sur des règles strictes jusqu'à leur intégration moderne dans les plateformes de vente en ligne.

1.2 Le Commerce Électronique (E-commerce)

1.2.1 Définition et Concepts Fondamentaux

Le commerce électronique, ou e-commerce, désigne l'ensemble des processus d'achat, de vente et d'échange de biens, de services et d'informations par l'intermédiaire de réseaux informatiques, principalement Internet [9]. Contrairement au commerce traditionnel (les points de vente physiques dits *brick-and-mortar*), le e-commerce abolit les frontières géographiques et temporelles. Laudon et Traver précisent que cette définition ne se restreint pas à la simple transaction financière : elle englobe également les dimensions informationnelles (recherche de produits, avis des consommateurs), relationnelles (service client, marketing digital) et logistiques (suivi de livraison).

Il est essentiel de distinguer le e-commerce du *e-business*. Si le e-commerce est centré sur les échanges transactionnels externes (vers le client ou entre entreprises), le e-business recouvre une notion plus vaste. Il inclut la numérisation complète des processus internes de l'entreprise : gestion des ressources humaines, chaîne d'approvisionnement (Supply Chain Management), ou encore la gestion de la relation client (CRM - *Customer Relationship Management*). Notre étude se concentre spécifiquement sur le volet interactif et relationnel du e-commerce.

1.2.2 Typologie du E-commerce

Le commerce électronique se décline en plusieurs catégories en fonction de la nature des acteurs impliqués dans la transaction :

- **B2C (Business-to-Consumer)** : Le modèle le plus connu, où les entreprises vendent directement aux consommateurs finaux (ex. Amazon, plateformes de retail en ligne).
- **B2B (Business-to-Business)** : Les transactions s'opèrent entre deux entreprises (ex. un grossiste fournissant des détaillants). Le volume financier du B2B dépasse largement celui du B2C.
- **C2C (Consumer-to-Consumer)** : Les particuliers échangent entre eux, souvent via des plateformes agissant comme des tiers de confiance (ex. eBay, Vinted).
- **M-commerce (Mobile Commerce)** : Une sous-catégorie du e-commerce qui se concentre sur les transactions effectuées via des appareils mobiles (smartphones, tablettes).

1.2.3 Évolution et Tendances Actuelles

Les premières plateformes e-commerce, apparues à la fin des années 1990, s'apparentaient à des catalogues statiques numérisés. L'interaction y était asynchrone et très limitée. Avec l'avènement du Web 2.0 et l'amélioration des infrastructures de paiement sécurisé, les sites sont devenus dynamiques et interactifs [2].

Aujourd'hui, le secteur est traversé par plusieurs tendances structurelles lourdes :

1. **L'omnicanalité** : Le consommateur moderne utilise de multiples canaux (site web, application mobile, réseaux sociaux, boutique physique) au cours d'un même parcours d'achat. L'enjeu est d'assurer une continuité fluide de l'expérience entre ces différents points de contact.
2. **L'hyper-personnalisation** : Grâce à l'analyse des mégadonnées (Big Data), les entreprises peuvent recommander des produits de manière ultra-ciblée, en fonction de l'historique de navigation et des préférences du client.
3. **Le commerce conversationnel** : Les utilisateurs achètent de plus en plus via des applications de messagerie (WhatsApp, Telegram, Messenger). Cette tendance fusionne le e-commerce et les réseaux sociaux, créant un environnement propice au déploiement de chatbots.

1.3 L'Expérience Utilisateur (UX) et le Service Client

1.3.1 L'Importance de l'UX dans le E-commerce

L'Expérience Utilisateur (UX - *User Experience*) désigne la qualité de l'interaction globale entre un individu et un environnement numérique. Dans le cadre du e-commerce, l'UX détermine de manière directe le taux de conversion et le taux de rebond. Une interface confuse, un temps de chargement trop long, ou un processus de paiement trop complexe sont autant de frictions qui dissuadent l'achat.

Selon Chaffey et Ellis-Chadwick [2], une bonne UX en e-commerce doit répondre à plusieurs critères :

— **L'utilisabilité** : La plateforme doit être intuitive et facile à prendre en main.
— **La trouvabilité (Findability)** : Le système de recherche et de filtrage doit permettre de trouver le bon produit en un minimum de clics.
— **La crédibilité** : L'affichage d'avis vérifiés, de conditions de retour claires et de protocoles de sécurité rassure l'utilisateur.
— **L'accessibilité** : Le site doit être consultable sur divers supports et par tous les utilisateurs.

1.3.2 Le Rôle Stratégique du Service Client

Le service client est le prolongement naturel de l'expérience utilisateur. En l'absence d'interaction physique et de contact direct avec le produit, l'utilisateur d'un site de e-commerce se pose de nombreuses questions avant, pendant, et après son achat. Le support client intervient à trois phases clés :

1. **En pré-vente** : Rassurer l'utilisateur sur les caractéristiques d'un produit, les compatibilités, ou les délais d'expédition. L'objectif est ici de lever les freins à l'achat.
2. **Pendant l'achat** : Assister lors de problèmes techniques liés au paiement ou au panier.
3. **En post-vente** : Gérer le suivi des colis, les retours (logistique inverse), et les réclamations.

Un service client réactif est un puissant levier de fidélisation. Cependant, l'augmentation constante du volume des ventes en ligne génère un afflux massif de requêtes récurrentes et souvent simples (ex. : *"Où est ma commande ?", "Faites-vous la livraison à domicile ?"*). Traiter ces requêtes manuellement mobilise d'importantes ressources humaines, ce qui représente un coût significatif et allonge les délais de réponse.

C'est face à ce goulot d'étranglement que l'automatisation du support client par l'intelligence artificielle est devenue une nécessité industrielle.

1.4 Les Agents Conversationnels (Chatbots)

1.4.1 Définition et Historique

Un chatbot (mot-valise issu de *chat* "discussion" et *bot* "robot") est un programme informatique conçu pour simuler une conversation avec des utilisateurs humains, que ce soit par texte ou par la voix.

L'histoire des chatbots remonte à bien avant l'essor du e-commerce. Le premier agent conversationnel notable, ELIZA, a été développé au MIT en 1966 par Joseph Weizenbaum. ELIZA utilisait des techniques simples de reconnaissance de modèles linguistiques et de substitution de mots pour simuler un psychothérapeute rogerien. Bien que dépourvu de toute intelligence ou de compréhension réelle, ELIZA a démontré la propension des humains à s'engager émotionnellement avec des interfaces textuelles, un phénomène connu sous le nom d'*Effet ELIZA*.

Dans les années 1990, A.L.I.C.E. (Artificial Linguistic Internet Computer Entity) a introduit le langage AIML (Artificial Intelligence Markup Language), permettant de structurer de manière plus sophistiquée des règles heuristiques de correspondance de modèles (Pattern Matching).

1.4.1.2 Définition Approfondie et Évolution Historique

Un chatbot (contraction des termes anglais chat signifiant « discussion » et bot abréviation de robot) est un programme informatique conçu pour simuler une conversation naturelle avec un utilisateur humain, que ce soit par le biais de texte, de voix, ou d'interfaces graphiques interactives. L'objectif fondamental d'un chatbot est d'automatiser les échanges de manière à répondre aux requêtes de l'utilisateur de façon rapide, pertinente et cohérente, sans intervention humaine directe.

L'histoire des agents conversationnels est intimement liée à l'évolution de l'intelligence artificielle elle-même. Elle débute en 1950, lorsque le mathématicien britannique Alan Turing propose son célèbre test — le Test de Turing — comme critère d'évaluation de l'intelligence des machines. Selon ce test, une machine peut être considérée comme « intelligente » si un juge humain ne parvient pas à la distinguer d'un interlocuteur humain au cours d'une conversation écrite. Ce cadre conceptuel a posé les fondations philosophiques sur lesquelles reposent tous les systèmes conversationnels ultérieurs.

Le premier agent conversationnel notable de l'histoire informatique, ELIZA, fut développé entre 1964 et 1966 au Massachusetts Institute of Technology (MIT) par Joseph Weizenbaum. ELIZA simulait un psychothérapeute rogerien en appliquant des règles simples de reconnaissance de motifs (pattern matching) et de substitution de mots. Son fonctionnement reposait sur un ensemble de scripts (le plus connu étant DOCTOR), qui permettaient au programme de reformuler les phrases de l'utilisateur sous forme de questions. Par exemple, si l'utilisateur saisissait : « Je me sens triste », ELIZA répondait : « Pourquoi vous sentez-vous triste ? ». Bien qu'entièrement dépourvue de compréhension réelle du langage, ELIZA a mis en évidence un phénomène psychologique fascinant : les utilisateurs humains s'engageaient émotionnellement avec le programme et lui attribuaient des qualités humaines, malgré leur connaissance de sa nature artificielle. Ce phénomène, baptisé l'Effet ELIZA, reste un sujet d'étude en psychologie de l'interaction homme-machine.

Dans les années 1970 et 1980, le projet PARRY développa une approche plus sophistiquée en simulant le comportement d'un patient atteint de schizophrénie paranoïaque, introduisant pour la première fois la notion d'un modèle interne d'état émotionnel. Ces travaux furent suivis par les systèmes experts des années 1980, des programmes qui codifiaient le savoir de spécialistes humains sous forme de règles logiques (SI… ALORS…), mais qui se révélaient extrêmement rigides et incapables de s'adapter à des contextes non prévus.

L'année 1995 marqua un tournant avec la création d'A.L.I.C.E. (Artificial Linguistic Internet Computer Entity) par Richard Wallace. A.L.I.C.E. introduisit le langage AIML (Artificial Intelligence Markup Language), un format XML permettant de définir des règles de correspondance de motifs de manière structurée et réutilisable. A.L.I.C.E. remporta le Prix Loebner (une compétition annuelle basée sur le Test de Turing) à trois reprises, démontrant la supériorité relative des approches basées sur des bases de connaissances riches par rapport aux simples scripts de substitution.

L'entrée des grandes entreprises technologiques dans ce domaine, à partir des années 2010, a profondément transformé le paysage des chatbots. En 2011, Apple lance Siri, le premier assistant vocal grand public intégré à un système d'exploitation mobile. En 2012, Google présente Google Now, et en 2014, Amazon commercialise Alexa avec l'enceinte connectée Echo. Ces systèmes marquent le passage des chatbots textuels aux assistants vocaux multimodaux, capables de comprendre la parole naturelle grâce au traitement automatique de la parole (ASR — Automatic Speech Recognition) couplé au NLP.

La véritable révolution conceptuelle commence en 2017 avec la publication de l'article "Attention Is All You Need" par Vaswani et al. chez Google, introduisant l'architecture Transformer. Cette architecture, que nous détaillons au Chapitre 2, a rendu possible le développement des Modèles de Langage de Grande Taille (LLMs), culminant en 2022 avec le lancement public de ChatGPT par OpenAI. Cette démo publique a déclenché une vague d'adoption sans précédent, propulsant les chatbots basés sur les LLMs au cœur des stratégies numériques de milliers d'entreprises.

1.4.2 Typologie Détaillée des Chatbots

La classification des chatbots peut s'effectuer selon plusieurs axes : leur architecture technique, leur domaine d'application, ou leur capacité de compréhension. Nous proposons ici une taxonomie complète et structurée.

A. Classification selon l'Architecture Technique

1. Les Chatbots basés sur des Règles (Rule-Based Chatbots)

Les chatbots à base de règles constituent la première génération de systèmes conversationnels déployés à l'échelle industrielle. Leur fonctionnement repose sur un ensemble fini de règles prédéfinies, souvent organisées sous la forme d'arbres de décision ou de flux de conversation (conversation flows). L'utilisateur est guidé pas à pas au travers d'options prédéterminées, généralement présentées sous forme de boutons cliquables ou de commandes textuelles exactes.

Le moteur de ces systèmes repose sur deux mécanismes fondamentaux :

- La correspondance de mots-clés (keyword matching) : le système recherche des termes ou expressions spécifiques dans la saisie de l'utilisateur.
- Le flux conditionnel (conditional flow) : en fonction du mot-clé détecté ou de l'option sélectionnée, le système dirige la conversation vers une branche prédéfinie.
-

Avantages :

- Comportement 100% prévisible et contrôlable, sans aucun risque de réponse hors-sujet.
- Coût de développement initial relativement faible pour des cas d'usage simples.
- Idéal pour des flux très structurés : collecte d'informations, formulaires de qualification, FAQ à questions limitées.
- Facile à auditer, à tester et à maintenir.
- Ne nécessite aucune infrastructure de calcul intensif (GPU, cloud)

Inconvénients :

- Extrême rigidité : toute reformulation non prévue dans le script entraîne l'échec systématique du chatbot.
- Explosion combinatoire : plus le cas d'usage est complexe, plus le nombre de règles à définir devient ingérable.
- Expérience utilisateur frustrante : l'utilisateur se heurte souvent à des messages du type « Je n'ai pas compris votre demande, veuillez choisir parmi les options suivantes. »
- Incapacité totale à gérer l'ambiguïté, l'humour, les fautes d'orthographe ou les formulations créatives.

2. Les Chatbots basés sur la Récupération d'Information (Retrieval-Based Chatbots)

Ces systèmes, intermédiaires entre les chatbots à règles et les chatbots génératifs, maintiennent une base de données de paires question-réponse. Lorsqu'un utilisateur pose une question, le système calcule la similarité sémantique entre cette question et l'ensemble des questions stockées dans sa base, puis retourne la réponse associée à la paire la plus similaire.

Les techniques de calcul de similarité peuvent aller des simples mesures TF-IDF (Term Frequency-Inverse Document Frequency) jusqu'aux modèles de plongements vectoriels denses (dense vector embeddings) générés par des architectures Transformer légères comme Sentence-BERT. Dans ce dernier cas, la question de l'utilisateur et les questions de la base sont encodées sous forme de vecteurs dans un espace de haute dimension, et la similarité est calculée par le produit scalaire ou la distance cosinus.

Avantages :

- Plus flexible que les chatbots à règles pures, car il peut gérer des reformulations.
- Les réponses sont strictement humaines et validées, sans risque d'hallucination.
- Permet de constituer une base de connaissances contrôlée.
-

Inconvénients :

- Limité aux questions couvertes par sa base : incapable de répondre à une question non référencée.
- La qualité de la réponse dépend entièrement de la richesse et de la couverture de la base de données.
- Ne peut pas générer de réponses originales ni s'adapter à un contexte de conversation multi-tours.

3. Les Chatbots Génératifs (Generative Chatbots)

Les chatbots génératifs représentent la génération actuelle et la plus avancée des systèmes conversationnels. Au lieu de sélectionner une réponse parmi un ensemble prédéfini, ils génèrent la réponse mot par mot (de manière autorégressive) en fonction du contexte de la conversation. Cette capacité repose sur des architectures de deep learning, et plus spécifiquement sur les Modèles de Langage de Grande Taille (LLMs) à base de Transformers.

La génération se fait de la manière suivante : à chaque étape, le modèle prédit la distribution de probabilité sur l'ensemble du vocabulaire pour le prochain token (unité lexicale), et sélectionne le token suivant selon une stratégie d'échantillonnage (greedy decoding, beam search, nucleus sampling). L'ensemble des tokens générés forme une réponse fluide et cohérente.

Avantages :

- Flexibilité extrême : capable de traiter n'importe quelle formulation, y compris les fautes d'orthographe, le langage familier, ou des questions composites.
- Maintien du contexte conversationnel sur de nombreux tours de parole.
- Génération de réponses naturelles, fluides et personnalisées.
- Capacité de raisonnement et de synthèse.
-

Inconvénients :

- Risque d'hallucination : le modèle peut générer des informations fausses mais convaincantes.
- Coût computationnel élevé (nécessite des GPU pour l'inférence en temps réel).
- Nécessite un encadrement rigoureux via le Prompt Engineering et des mécanismes de grounding (ancrage factuel).
- Moins prévisible qu'un système à règles.

4. Les Chatbots Hybrides

Dans la pratique industrielle, la grande majorité des chatbots déployés en production adoptent une approche hybride, combinant les avantages des différentes architectures. Un chatbot hybride typique utilisera :

- Un module NLP pour classifier l'intention (intent classification) et extraire les entités (entity extraction) — par exemple, identifier qu'un utilisateur souhaite « commander une pizza » (intention) de « taille grande » et « saveur fromage » (entités).
- Un flux de dialogue structuré pour guider les étapes transactionnelles où la précision est critique (validation d'une adresse de livraison, confirmation d'un paiement).
- Un LLM pour gérer les questions ouvertes, les cas non prévus, ou pour générer des réponses naturelles à des requêtes d'information générale.

Notre projet adopte précisément cette approche hybride : la machine à états intégrée dans le moteur d'orchestration gère le flux transactionnel structuré (navigation catalogue, gestion du panier, processus de checkout), tandis que le LLM Llama3 via Ollama prend en charge la compréhension du langage naturel et la génération de réponses fluides.

B. Classification selon le Domaine d'Application

Au-delà de leur architecture technique, les chatbots peuvent être classifiés selon leur périmètre fonctionnel :

- Chatbots de Service Client (Customer Service Bots) : Ils traitent les requêtes post-achat (suivi de commande, retours, réclamations) et représentent le cas d'usage le plus répandu dans le commerce en ligne.
- Chatbots de Vente et de Recommandation (Sales Bots) : Ils accompagnent l'utilisateur dans son parcours d'achat, jouant le rôle d'un conseiller de vente virtuel. Notre chatbot appartient principalement à cette catégorie.
- Chatbots RH (HR Bots) : Utilisés en interne par les entreprises pour automatiser la gestion des congés, répondre aux questions des employés sur les politiques internes, ou assister dans le processus de recrutement.
- Chatbots Médicaux (Healthcare Bots) : Permettent le triage initial de symptômes, la prise de rendez-vous, ou la fourniture d'informations médicales générales.
- Chatbots Éducatifs (EdTech Bots) : Tuteurs virtuels capables d'expliquer des concepts, de générer des exercices et de suivre la progression d'un étudiant.
- Chatbots Financiers (FinTech Bots) : Assistent dans la gestion de comptes, la consultation de soldes, ou la réalisation de virements.

C. Classification selon le Niveau d'Intelligence

Une troisième dimension de classification porte sur la sophistication du traitement intellectuel :

- Chatbot Scriptés

Réponses pré-programmées, aucune compréhension du langage

- Chatbots NLP Basiques

Classification d'intentions, extraction d'entités nommées

- Chatbots Contextuels

Maintien de l'état de la conversation sur plusieurs tours

- Chatbots Cognitifs (LLM)

Compréhension profonde, raisonnement, génération fluide

- Agents Autonomes (AI Agents)

Planification, utilisation d'outils externes, auto-correction

1.5.1 Présentation Générale de Telegram

Telegram est une application de messagerie instantanée multiplateforme, fondée en 2013 par les frères russes Pavel Durov et Nikolai Durov, créateurs du réseau social russe VKontakte (VK). Initialement hébergée en Russie, la société a rapidement relocalisé ses serveurs à Berlin, puis à Dubaï pour garantir son indépendance vis-à-vis des pressions gouvernementales, en accord avec la philosophie de liberté et de respect de la vie privée qui constitue l'ADN de la plateforme.

Techniquement, Telegram est fondé sur un protocole de communication propriétaire, le protocole MTProto (Mobile Transport Protocol), développé par Nikolai Durov. Ce protocole, optimisé pour les environnements mobiles à faible bande passante, garantit une vitesse de transmission élevée et un chiffrement bout en bout pour les discussions secrètes (Secret Chats). Pour les conversations standards (cloud-based), Telegram applique un chiffrement client-serveur basé sur AES-256, RSA-2048 et l'échange de clés Diffie-Hellman.

En termes d'adoption, Telegram a connu une croissance exponentielle, passant de 100 millions d'utilisateurs actifs en 2016 à plus de 900 millions d'utilisateurs actifs mensuels en 2024, se positionnant comme l'une des cinq applications de messagerie les plus utilisées au monde. Cette croissance a été notamment amplifiée par plusieurs vagues de migrations massives depuis d'autres plateformes lors de controverses liées à la vie privée (notamment après les changements de politique de confidentialité de WhatsApp en 2021).

1.5.2 Caractéristiques Techniques et Fonctionnelles de Telegram

Telegram se distingue de ses concurrents (WhatsApp, Messenger, Signal) par un ensemble de fonctionnalités avancées qui en font une plateforme particulièrement adaptée au déploiement d'applications automatisées :

a) Les Groupes et Supergroups

Telegram supporte des groupes pouvant accueillir jusqu'à 200 000 membres, une capacité sans équivalent sur les plateformes concurrentes. Les supergroups offrent des fonctionnalités avancées de modération, de sondages, et de gestion de contenu.

b) Les Chaînes (Channels)

Les chaînes Telegram permettent à un émetteur de diffuser des messages vers un nombre illimité d'abonnés. Elles sont utilisées massivement pour la distribution de contenu médiatique, de newsletters, et de flux d'information en temps réel.

c) Les Bots Telegram

C'est la fonctionnalité centrale pour notre projet. Telegram intègre nativement un système de bots, accessibles via l'API Bot de Telegram (Telegram Bot API). Un bot Telegram est un compte spécial géré par un programme informatique externe (et non par un humain). Tout développeur peut créer un bot en interagissant avec @BotFather, l'interface officielle de gestion des bots, qui génère un token d'authentification unique permettant d'appeler l'API.

d) Les Inline Bots et les Réponses Rapides

Telegram supporte les bots inline, qui peuvent être invoqués depuis n'importe quelle conversation en tapant @nom_du_bot suivi d'une requête. Les réponses rapides (quick replies) sous forme de claviers personnalisés (Inline Keyboards et Reply Keyboards) permettent de proposer des options cliquables à l'utilisateur, facilitant les interactions structurées.

e) Les WebApps et Mini-Applications

Depuis 2022, Telegram permet d'intégrer de véritables applications web au sein de la plateforme via les Telegram Web Apps (TWA). Ces mini-applications s'exécutent dans un navigateur intégré à Telegram et peuvent accéder aux données de l'utilisateur (prénom, identifiant) via le SDK JavaScript de Telegram, ouvrant la voie à des expériences e-commerce complètes sans quitter l'application.

f) Telegram Stars et les Paiements

Telegram dispose d'un système de paiement intégré via l'API Payments 2.0, permettant aux bots de recevoir des paiements directement dans les conversations grâce à des fournisseurs de paiement tiers (Stripe, PayPal, etc.). La plateforme a également introduit Telegram Stars, une monnaie virtuelle interne permettant les micro-transactions.

1.5.3 L'API Bot de Telegram : Architecture et Fonctionnement

L'API Bot de Telegram est une interface RESTful qui permet à un programme externe de contrôler un bot Telegram via des requêtes HTTP. Chaque requête est authentifiée par le token unique du bot, inclus dans l'URL de base de l'API :

https://api.telegram.org/bot<TOKEN>/<MÉTHODE>

L'API expose un ensemble de méthodes permettant de :

- Recevoir des messages : via le mécanisme de polling (getUpdates) ou de webhooks (réception push).
- Envoyer des messages : texte (sendMessage), images (sendPhoto), documents (sendDocument), vidéos (sendVideo).
- Gérer les claviers : affichage de boutons inline (InlineKeyboardMarkup) ou de claviers personnalisés au niveau du clavier système (ReplyKeyboardMarkup).
- Répondre aux callbacks : lorsqu'un utilisateur clique sur un bouton inline, un événement CallbackQuery est déclenché.

Polling vs. Webhook :

Il existe deux paradigmes fondamentaux pour recevoir les mises à jour (updates) depuis l'API Telegram :

1. Le Long Polling (getUpdates) : Le bot envoie périodiquement des requêtes HTTP GET au serveur Telegram pour demander les nouvelles mises à jour. C'est l'approche la plus simple à mettre en œuvre, particulièrement adaptée au développement local et aux environnements sans adresse IP publique fixe. C'est l'approche que nous avons retenue dans notre prototype.
2. Le Webhook : Le bot enregistre une URL HTTPS publique auprès de Telegram. Telegram envoie alors automatiquement les nouvelles mises à jour en temps réel à cette URL via des requêtes HTTP POST. Cette approche est plus efficace en production car elle élimine la latence du polling, mais nécessite un serveur accessible publiquement avec un certificat SSL valide.

La Bibliothèque python-telegram-bot :

Dans l'écosystème Python, la bibliothèque python-telegram-bot est la référence de facto pour interagir avec l'API Bot de Telegram. Elle offre une couche d'abstraction haut niveau au-dessus des appels HTTP bruts, fournissant notamment :

- Un système de Handlers pour router les différents types d'événements (message, callback, commande) vers des fonctions de traitement dédiées.
- Un ApplicationBuilder pour configurer et lancer le bot.
- Un support natif pour la programmation asynchrone via le paradigme asyncio de Python 3.10+, permettant de traiter des milliers de conversations en parallèle sur un seul thread.

Dans notre architecture hexagonale, cette bibliothèque constitue le cœur du TelegramAdapter, qui implémente le port IMessagingPlatform.

11.5.4 Pourquoi Telegram comme Plateforme de Déploiement ?

Le choix de Telegram comme canal de déploiement de notre chatbot e-commerce n'est pas le fruit du hasard, ni d'une simple familiarité avec la plateforme. Il résulte d'une analyse comparative rigoureuse, menée selon plusieurs critères techniques, économiques et stratégiques, face aux principales alternatives disponibles sur le marché des plateformes de messagerie. Dans cette section, nous justifions ce choix de manière exhaustive, en examinant successivement les dimensions économique, technique, sécuritaire, et d'adoption utilisateur.

A. Une API Gratuite, Ouverte et Sans Restrictions Commerciales
L'un des facteurs les plus déterminants dans le choix d'une plateforme de déploiement pour un chatbot est le modèle économique de son API. Sur ce critère, Telegram se distingue radicalement de ses concurrents.

L'API Bot de Telegram est entièrement gratuite, sans limite de messages, sans abonnement mensuel, et sans frais à l'utilisation. Tout développeur peut créer un bot fonctionnel en moins de cinq minutes en interagissant avec @BotFather, l'interface officielle de création et gestion des bots. Le processus est entièrement automatisé et ne requiert aucune validation humaine, aucun dossier de candidature, et aucune approbation préalable de la plateforme.

Cette situation contraste fortement avec les alternatives. L'API WhatsApp Business (gérée par Meta), par exemple, adopte un modèle de facturation à la conversation : chaque échange initié par l'entreprise est facturé entre 0,05 et 0,15 dollar selon la région géographique du destinataire. À l'échelle d'un prototype académique, ce coût est prohibitif. De plus, l'accès à l'API WhatsApp Business nécessite de soumettre une demande formelle d'accès, de vérifier le numéro de téléphone de l'entreprise, et d'obtenir l'approbation de Meta, un processus pouvant prendre plusieurs semaines. L'API Messenger de Meta (Facebook) impose elle aussi un processus de validation des applications avant leur déploiement public, et son accès à toutes les fonctionnalités avancées est conditionné à l'approbation d'un examinateur Meta.

Telegram, quant à lui, place la liberté du développeur au cœur de son approche. Il n'existe aucune limite au nombre de bots qu'un développeur peut créer, aucune restriction sur les types de messages envoyés (dans le respect des conditions d'utilisation générales), et aucune revue d'application requise pour un déploiement en production. Cette philosophie de plateforme ouverte est en adéquation parfaite avec les besoins d'un projet de recherche universitaire, où l'agilité de développement et l'absence de contraintes financières sont primordiales.

B. La Richesse des Fonctionnalités Interactives

Au-delà de la gratuité, Telegram offre un ensemble de fonctionnalités interactives nativement intégrées à l'API Bot, qui permettent de construire des expériences conversationnelles riches et fluides, comparables à celles d'une application mobile dédiée. Les Inline Keyboards constituent l'un des outils les plus puissants mis à disposition des développeurs. Il s'agit de claviers de boutons cliquables affichés directement sous un message du bot, permettant à l'utilisateur d'interagir sans avoir à saisir de texte. Chaque bouton peut déclencher un événement CallbackQuery (avec une donnée arbitraire associée) ou ouvrir une URL dans le navigateur intégré. Dans notre chatbot e-commerce, les Inline Keyboards sont utilisés pour afficher le catalogue de produits, les options d'ajout au panier, et les actions de gestion de commande, offrant une expérience utilisateur analogue à celle d'une application mobile.

Les Reply Keyboards offrent une alternative complémentaire : ils remplacent le clavier système du smartphone par un clavier personnalisé de boutons textuels. Contrairement aux Inline Keyboards qui restent attachés à un message précis, les Reply Keyboards persistent tant qu'ils ne sont pas explicitement supprimés, facilitant les interactions guidées étape par étape (comme le processus de checkout dans notre machine à états). Les commandes de bot (/start, /help, /menu, etc.) constituent des points d'entrée standardisés, reconnus et mis en évidence par l'interface Telegram. Lorsqu'un utilisateur saisit /, Telegram affiche automatiquement la liste des commandes disponibles du bot, avec leurs descriptions. Ce mécanisme améliore significativement la découvrabilité des fonctionnalités du chatbot.

La gestion des médias est également un atout de l'API Telegram : le bot peut envoyer et recevoir des images, des fichiers, des vidéos, et des messages vocaux, ouvrant la voie à des fonctionnalités d'e-commerce enrichies, comme l'affichage des photos de produits directement dans la conversation ou la transmission d'une preuve de paiement par image.

Les Callback Queries et l'édition de messages permettent de créer des interfaces dynamiques : le bot peut modifier un message existant (texte et clavier) en réponse à une action de l'utilisateur, sans envoyer un nouveau message. Cette fonctionnalité est particulièrement utile pour afficher une liste de produits paginée, ou pour mettre à jour en temps réel le récapitulatif du panier sans encombrer la conversation.

Aucun de ces mécanismes n'est disponible dans sa totalité sur les plateformes alternatives grand public sans des contournements complexes. La richesse native de l'API Telegram pour les interactions structurées en fait un environnement de développement de premier ordre pour les chatbots transactionnels.

C. La Qualité de l'Écosystème Python

Le troisième pilier justifiant notre choix est la maturité et la qualité de l'écosystème de développement Python autour de Telegram. La bibliothèque python-telegram-bot est, à ce jour, la solution de référence pour l'intégration de Telegram en Python. Maintenue par une communauté active de contributeurs open-source, elle bénéficie d'une documentation exhaustive, d'exemples de code pour chaque fonctionnalité, et d'une mise à jour régulière pour rester synchronisée avec les évolutions de l'API officielle.

La version 20+ de python-telegram-bot a introduit un support natif complet pour la programmation asynchrone via asyncio, en parfaite cohérence avec l'architecture asynchrone de notre système. Le composant Application (successeur du Updater) offre un système de handlers sophistiqué permettant de router les différents types d'événements Telegram (messages, commandes, callbacks, fichiers, etc.) vers des coroutines dédiées de manière déclarative et lisible.

Par ailleurs, d'autres bibliothèques complémentaires sont disponibles dans l'écosystème Python-Telegram, telles que Aiogram (une alternative asynchrone pure, particulièrement performante) et Telethon (permettant d'interagir avec l'API Telegram complète, au-delà de la seule API Bot). Cette abondance de choix témoigne de la vitalité de la communauté de développeurs Telegram, et garantit la pérennité de l'écosystème sur le long terme.
La congruence entre l'écosystème Python-Telegram et les autres technologies de notre stack (LangChain, asyncio, SQLite via aiosqlite) a permis une intégration naturelle et sans friction de tous les composants de notre architecture hexagonale.

D. Sécurité, Confidentialité et Conformité

La dimension sécuritaire est un critère souvent négligé dans le choix d'une plateforme de messagerie pour un chatbot e-commerce, mais elle revêt une importance capitale lorsque des données personnelles et des informations transactionnelles sont en jeu.

E. Adoption Utilisateur et Pénétration du Marché

Enfin, le choix d'une plateforme de déploiement doit tenir compte de son taux de pénétration auprès de la population cible. Un chatbot e-commerce n'a de valeur que s'il est accessible sur une plateforme que les clients utilisent déjà au quotidien.

1.6 Conclusion du Chapitre

Ce chapitre a permis de poser le cadre contextuel de notre étude. Le commerce électronique, en pleine maturation, fait face au dé majeur de l'amélioration de l'expérience utilisateur par un service client réactif et omniprésent. Les chatbots, forts des avancées spectaculaires de l'intelligence arti cielle, s'imposent comme la réponse technologique adéquate à ce besoin d'automatisation

Cependant, comme nous l'avons souligné, les chatbots basés sur de simples règles atteignent rapidement leurs limites face à l'imprévisibilité du langage humain. C'est pourquoi notre projet se tourne vers l'utilisation d'intelligences arti cielles génératives. A n de bien comprendre les mécanismes sous-jacents qui rendent possible cette nouvelle génération d'agents conversationnels, le chapitre suivant sera consacré aux fondements théoriques du Traitement Automatique du Langage Naturel (NLP) et à l'architecture des Modèles de Langage de Grande Taille (LLMs).

Chapitre 2

Fondements Théoriques : Intelligence Artificielle et NLP

2.1 Introduction

L'avènement des chatbots intelligents capables de soutenir des conversations fluides et contextuelles ne s'est pas fait du jour au lendemain. Il est le fruit de décennies de recherche en Intelligence Artificielle (IA) et plus spécifiquement dans la sous-discipline du Traitement Automatique du Langage Naturel (NLP - *Natural Language Processing*).

Ce chapitre vise à établir les fondements théoriques nécessaires à la compréhension de la technologie qui anime le cœur de notre système conversationnel pour le e-commerce. Nous débuterons par un rappel sur l'Intelligence Artificielle et l'Apprentissage Automatique (*Machine Learning*), avant de plonger dans l'évolution historique et technique du NLP.

Nous accorderons une attention particulière à la révolution provoquée en 2017 par l'architecture Transformer [12], qui a rendu possible l'émergence des Modèles de Langage de Grande Taille (LLM - *Large Language Models*). Enfin, nous introduirons les concepts avancés et les outils modernes, tels que le *Prompt Engineering*, l'écosystème LangChain, et le moteur Ollama, qui permettent aujourd'hui d'intégrer concrètement ces LLMs dans des applications logicielles robustes.

2.2 Intelligence Artificielle et Apprentissage Automatique

2.2.1 L'Intelligence Artificielle (IA)

L'Intelligence Artificielle désigne l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine.

Historiquement, l'IA s'est divisée en deux courants majeurs :

— **L'IA Symbolique (ou IA classique)** : Basée sur la logique formelle et des systèmes experts de règles (SI... ALORS). Elle excelle dans des environnements mathématiques et déterministes, mais échoue face à l'ambiguïté inhérente au monde réel et au langage humain.

— **L'IA Connexionniste** : Inspirée de la structure du cerveau humain, elle s'appuie sur des Réseaux de Neurones Artificiels (RNA) capables d'apprendre par l'expérience à partir de données.

2.2.2 L'Apprentissage Automatique (Machine Learning) et l'Apprentissage Profond (Deep Learning)

L'Apprentissage Automatique (*Machine Learning* - ML) est un sous-domaine de l'IA qui permet aux algorithmes de découvrir des modèles récurrents (*patterns*) au sein d'ensembles de données, sans avoir été explicitement programmés pour cette tâche spécifique.

L'Apprentissage Profond (*Deep Learning* - DL) est lui-même une évolution du ML. Il utilise des réseaux de neurones composés de multiples couches cachées (*deep neural networks*). Ces architectures complexes sont capables d'extraire automatiquement des caractéristiques de haut niveau à partir de données brutes. C'est le *Deep Learning* qui a propulsé les performances de la vision par ordinateur et du traitement du langage naturel au cours de la dernière décennie, grâce notamment à l'augmentation exponentielle de la puissance de calcul (cartes graphiques GPU) et à la disponibilité massive de données (Big Data).

2.3 Le Traitement Automatique du Langage Naturel (NLP)

2.3.1 Définition et Défis

Le Traitement Automatique du Langage Naturel est l'intersection de l'informatique, de l'intelligence artificielle et de la linguistique. Son but est de donner aux ordinateurs la capacité de lire, comprendre et générer le langage humain.

Les défis du NLP sont immenses car le langage humain est intrinsèquement complexe, non structuré et hautement ambigu :

— **Ambiguïté lexicale** : Un mot peut avoir plusieurs sens (ex : "avocat" désigne un fruit ou un métier).

— **Ambiguïté syntaxique** : La structure d'une phrase peut prêter à confusion (ex : "Il observe la fille avec un télescope").

— **Ironie, sarcasme et contexte** : Comprendre l'intention sous-jacente nécessite une conscience du contexte et de la pragmatique linguistique.

2.3.2 Tâches Classiques du NLP

Parmi les applications traditionnelles du NLP, on retrouve :

— **La classification de textes** (Analyse de sentiments, détection de spam).

— **L'extraction d'entités nommées (NER)** (Identifier des noms, des lieux, des prix).

— **La traduction automatique**.

— **La modélisation du langage** (Prédire le mot suivant dans une séquence).

Les chatbots modernes reposent fondamentalement sur cette dernière tâche : ils génèrent des réponses en prédisant la séquence de mots la plus probable compte tenu du contexte de la conversation.

2.3.3 L'Évolution des Approches en NLP

1. Approches Statistiques et N-grammes : Avant 2013, les modèles de langage utilisaient des probabilités conditionnelles simples basées sur la fréquence d'apparition de suites de mots (n-grammes). Ces modèles souffraient de la malédiction de la dimensionnalité et d'une incapacité totale à comprendre la sémantique.

2. Plongements Lexicaux (Word Embeddings) : L'introduction de Word2Vec en 2013 a révolutionné le NLP. Les mots sont représentés par des vecteurs denses dans un espace mathématique de grande dimension. Les mots sémantiquement proches sont géométriquement proches. La relation classique $V(\text{Roi}) - V(\text{Homme}) + V(\text{Femme}) \approx V(\text{Reine})$ démontre la capture du sens.

3. Réseaux Récurrents (RNN) et LSTM : Pour traiter des séquences de texte de longueur variable, les architectures récurrentes (RNN) ont été utilisées. Elles traitent le texte mot par mot, de gauche à droite. La variante LSTM (*Long Short-Term Memory*) a permis de mieux retenir le contexte, mais le traitement séquentiel rendait l'entraînement très lent (impossible à paralléliser) et le modèle finissait toujours par "oublier" les informations situées au début de longues phrases.

2.4 La Révolution Transformer et les LLM

2.4.1 L'Architecture Transformer

En 2017, des chercheurs de Google publient l'article fondateur *"Attention Is All You Need"* [12], introduisant l'architecture **Transformer**. Cette architecture a définitivement remplacé les RNN en résolvant le problème de la sequentialité.

L'innovation clé du Transformer réside dans le mécanisme de **Self-Attention** (Attention Scalaire Multi-Têtes). Au lieu de traiter les mots un par un, le Transformer analyse la phrase entière simultanément. Pour chaque mot, le mécanisme d'attention calcule un "score d'importance" par rapport à *tous* les autres mots de la phrase. Ainsi, le mot "il" dans la phrase "Le chat n'a pas traversé la route car il était trop fatigué" sera fortement associé mathématiquement au mot "chat" (et non à "route"), peu importe la distance qui les sépare.

Le Transformer classique est composé de deux blocs :

— **Un Encodeur** : Qui lit le texte en entrée et en extrait une représentation riche et contextuelle. (Modèle dérivé célèbre : BERT, utilisé par Google Search).

— **Un Décodeur** : Qui prend cette représentation et génère du texte de manière autorégressive (un mot après l'autre). (Modèles dérivés célèbres : la famille GPT, LLaMA).

2.4.2 Les Modèles de Langage de Grande Taille (LLM)

Un Modèle de Langage de Grande Taille (LLM) n'est fondamentalement qu'un gigantesque réseau de neurones (généralement un décodeur Transformer) entraîné sur des corpus de textes titanesques (plusieurs téraoctets de données issues de Wikipédia, de livres, de forums, et de code source) [13].

La taille d'un LLM se mesure en nombre de paramètres (les poids des connexions synaptiques du réseau). Par exemple, le modèle GPT-3 possède 175 milliards de paramètres, tandis que le modèle open-source LLaMA-3 (utilisé dans notre étude via Ollama) propose des déclinaisons de 8 milliards de paramètres, optimisées pour fonctionner localement avec de hautes performances.

Le processus d'entraînement d'un LLM se déroule en deux grandes phases :

1. **Le Pré-entraînement (Pre-training)** : Apprentissage non supervisé. Le modèle lit des milliards de phrases et apprend simplement à prédire le mot suivant. Il acquiert ainsi une grammaire parfaite, une vaste culture générale et des capacités de raisonnement basiques.
2. **L'Ajustement Fin (Fine-Tuning / RLHF)** : Le modèle pré-entraîné est "aligné" par un apprentissage supervisé et par renforcement à partir de retours humains (RLHF - *Reinforcement Learning from Human Feedback*). On lui apprend à ne pas simplement compléter un texte, mais à agir comme un assistant utile, poli et sûr.

2.4.3 Le Défi des Hallucinations

Malgré leur puissance, les LLMs souffrent d'un défaut structurel : ils n'ont pas de véritable concept de la vérité ; ce sont des moteurs probabilistes. Lorsqu'ils ne connaissent

pas la réponse, ils ont tendance à inventer des faits de manière très convaincante. Ce phénomène est appelé **Hallucination**. Dans un contexte e-commerce, il est inacceptable qu'un chatbot invente un produit qui n'existe pas ou donne un faux prix. Il est donc indispensable d'encadrer le modèle par des outils logiciels.

2.5 Outils Modernes d'Ingénierie IA

Pour transformer un LLM brut en un système d'information fiable pour une entreprise, de nouveaux concepts et outils sont apparus.

2.5.1 Le Prompt Engineering

Le *Prompt Engineering* (ingénierie d'invite) est la science qui consiste à concevoir et optimiser le texte fourni en entrée (le *prompt*) au LLM pour maximiser la qualité, la précision et le format de la sortie générée. Dans notre projet, le *System Prompt* est crucial : il contraint l'IA à adopter la *persona* d'un assistant de vente, lui interdit formellement d'inventer des produits, et lui impose de répondre de manière concise.

2.5.2 LangChain : Le Framework d'Orchestration

LangChain [3] est un framework open-source conçu spécifiquement pour le développement d'applications basées sur des LLMs. Son principe fondateur est la **composabilité**. Il offre des abstractions standards pour :

- **Les Modèles** : Interfaces unifiées pour appeler OpenAI, Anthropic, ou des modèles locaux comme Ollama.
- **Les Prompts** : Modèles de prompts (Templates) paramétrables dynamiquement.
- **La Mémoire** : Gestion de l'historique de conversation (les LLMs sont par nature sans état ou *stateless*).
- **Les Chaînes (Chains)** : Séquences d'appels (ex. : formater l'historique → injecter les données de la base → appeler le LLM → parser la réponse).

Dans notre architecture, LangChain agit comme la "colle logicielle" entre l'intelligence pure du modèle et la logique métier de l'application.

2.5.3 Ollama et l'IA Locale

L'exécution de LLMs (comme GPT-4) s'appuie traditionnellement sur des API cloud facturées à l'utilisation, ce qui pose des problèmes de coûts opérationnels et de confidentialité des données (RGPD). **Ollama** [10] est un outil révolutionnaire qui simplifie le déploiement de modèles open-source (comme Llama 3, Mistral, Gemma) *en*

local. Grâce à des techniques avancées de **Quantification** (réduction de la précision mathématique des poids du réseau de 32 bits à 4 bits ou 8 bits), Ollama permet d'exécuter de puissants LLMs sur du matériel modeste, tout en maintenant un niveau de performance sémantique élevé. L'utilisation d'Ollama garantit que les données des clients ne quittent jamais l'infrastructure de l'entreprise, tout en s'affranchissant des coûts d'API.

2.5.4 Génération Augmentée par la Récupération (RAG)

Bien que non totalement implémenté sous sa forme vectorielle dans ce prototype, le principe du **RAG** (*Retrieval-Augmented Generation*) inspire notre architecture. Le RAG consiste à récupérer des informations réelles et à jour depuis une source externe (notre base de données SQLite contenant le menu et les prix) et à les injecter dans le contexte du LLM *avant* de lui demander de générer une réponse. Le modèle ne s'appuie donc plus sur ses poids internes (risquant d'halluciner), mais agit comme un synthétiseur intelligent de l'information stricte fournie dans le prompt.

2.6 Conclusion du Chapitre

Ce chapitre a exposé les fondements scientifiques permettant l'émergence des intelligences artificielles génératives modernes. La rupture technologique introduite par l'architecture Transformer a permis le développement des LLMs, capables de comprendre le langage naturel avec une subtilité inédite.

Toutefois, nous avons identifié que le déploiement brut d'un LLM est insuffisant pour répondre aux exigences de fiabilité d'un environnement e-commerce, notamment en raison du risque d'hallucinations et de l'absence de persistance d'état. C'est ici qu'intervient l'ingénierie logicielle. Le chapitre suivant s'attachera à décrire l'architecture logicielle de notre système (l'Architecture Hexagonale) et la manière dont nous concevons le couplage entre la logique transactionnelle (paniers, base de données) et l'intelligence linguistique fournie par le LLM (via LangChain et Ollama).

Chapitre 3

Conception et Architecture du Système

3.1 Introduction

L'intégration d'un Modèle de Langage (LLM) dans un système d'information n'est pas un défi trivial. Contrairement à une API classique qui renvoie des données structurées et prévisibles, un LLM génère du texte de manière probabiliste. De plus, un chatbot e-commerce doit interagir avec de multiples systèmes externes : une base de données pour le catalogue, une plateforme de messagerie (Telegram) pour dialoguer avec le client, et un système de mémoire pour retenir le contexte.

Pour garantir la robustesse, la maintenabilité et l'évolutivité de l'application, une conception logicielle rigoureuse est indispensable. Ce chapitre présente en détail l'architecture retenue pour notre système. Nous y justifierons le choix de l'**Architecture Hexagonale**, nous présenterons la modélisation des entités du domaine, la structure des ports et adaptateurs, et nous décrirons le fonctionnement du moteur d'orchestration qui intègre une machine à états implicite.

3.2 Justification et Choix de l'Architecture Hexagonale

3.2.1 Problématique des Architectures en Couches

Dans une architecture logicielle traditionnelle en couches (N-Tiers), la logique métier dépend souvent de l'infrastructure (la base de données) ou de l'interface utilisateur. Si l'on souhaite changer de base de données (passer de SQLite à PostgreSQL) ou changer d'interface (passer de Telegram à WhatsApp), il est souvent nécessaire de réécrire une grande partie du code métier, car les dépendances sont couplées de haut en bas.

3.2.2 Principes de l'Architecture Hexagonale

Inventée par Alistair Cockburn [6], l'Architecture Hexagonale (ou motif *Ports et Adaptateurs*) repose sur un principe d'inversion des dépendances radical : **L'application (la logique métier) est au centre, et elle ne dépend de rien d'autre qu'elle-même.** Toutes les communications avec l'extérieur (BDD, API Web, Interface Utilisateur) se font via des frontières strictes appelées **Ports**, qui sont implémentées par des **Adaptateurs**.

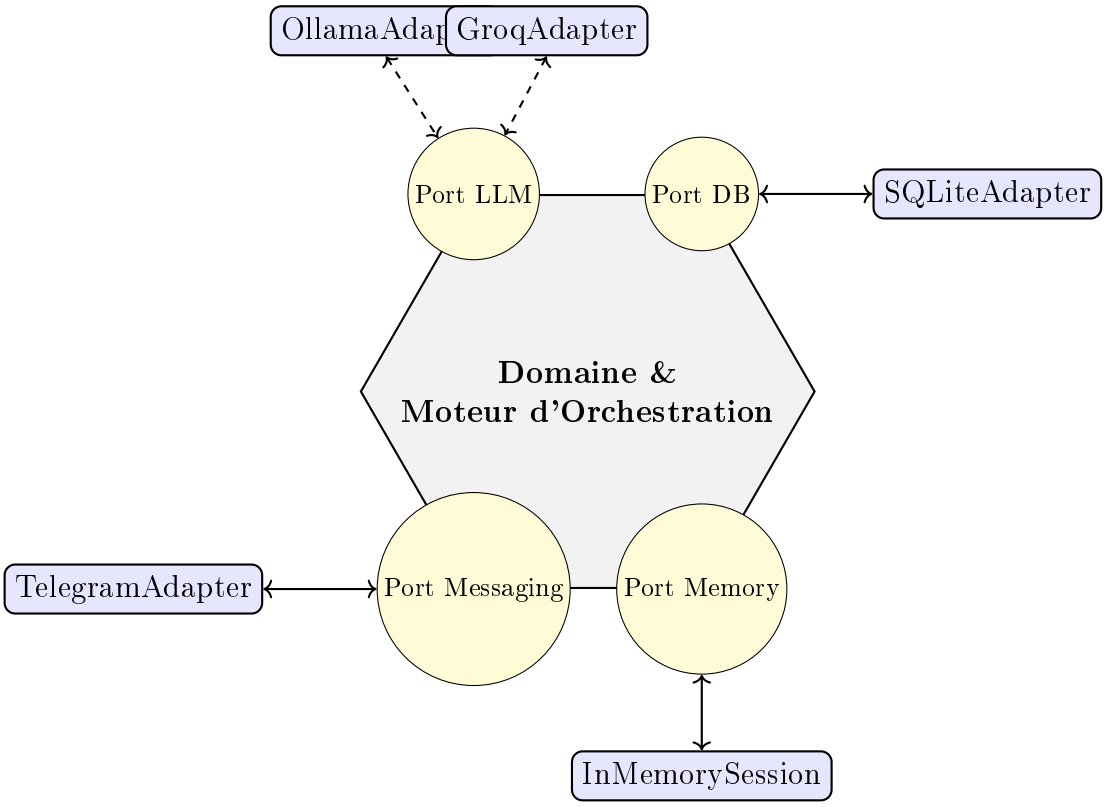

FIGURE 3.1 – Diagramme de l'Architecture Hexagonale du Système

Les avantages de cette architecture pour notre chatbot sont multiples :

— **Agnosticisme Technologique** : Le cœur du chatbot ignore totalement s'il communique sur Telegram, Slack ou un site web. Il sait seulement qu'il envoie et reçoit des messages via le port *IMessagingPlatform*. De même, il ignore s'il utilise Ollama ou l'API Cloud Groq ; il utilise simplement le port *ILLMProvider*.

— **Testabilité** : Il est possible de tester toute la logique du chatbot (intentions, mise au panier) en créant des adaptateurs factices (*mocks*), sans avoir besoin d'allumer une base de données ou de faire des appels réseau coûteux vers l'IA.

3.3 Modélisation du Domaine (Domain-Driven Design)

Conformément à l'approche de conception guidée par le domaine (DDD - *Domain-Driven Design*) [5], nous avons isolé les entités fondamentales du système dans le module `domain.py`. Ces entités sont des *Dataclasses* Python pures.

— `Customer` : Représente l'utilisateur du bot. Il possède un identifiant unique, un `telegram_id`, et un `session_state` (l'état actuel de la conversation).

— `Product` : Représente un article du catalogue. Il dispose d'un `id`, d'un `name`, d'un `price`, et d'un indicateur de disponibilité (`is_available`).

— `UnifiedMessage` : Structure qui standardise le format des messages échangés, indépendamment de la plateforme de messagerie sous-jacente.

Ces entités ne contiennent aucune logique d'accès aux données. Elles définissent le vocabulaire commun (le *Ubiquitous Language*) du projet.

3.4 Les Ports : Définition des Contrats (Interfaces)

Les ports sont définis dans `ports.py` sous la forme de classes abstraites (*Abstract Base Classes* en Python). Ils établissent un "contrat" que les adaptateurs devront respecter.

1. `IDatabaseRepository` : Fournit les méthodes asynchrones `get_available_menu()`, `add_to_cart()`, `get_user_cart_summary()`, etc.
2. `ILLMProvider` : Définit l'unique méthode `generate_reply(prompt, context_data, user_history)`.
3. `IMemoryRepository` : Gère le stockage temporaire du contexte de discussion et la machine à états (`update_memory`, `get_user_state`, `set_user_state`).
4. `IMessagingPlatform` : Permet d'envoyer un message (`send_message`) et d'écouter les événements entrants.

3.5 Les Adaptateurs : Implémentation Technologique

Les adaptateurs (développés dans `adapters.py`) sont le "monde réel". Ils connectent les ports de notre hexagone aux technologies externes.

3.5.1 SQLiteAdapter et Transactions ACID

Cet adaptateur implémente le `IDatabaseRepository`. Il communique avec le fichier `catering.db`. Bien que SQLite soit intrinsèquement synchrone en Python (via

`sqlite3`), nous avons encapsulé les appels SQL via `asyncio.to_thread()` afin de ne pas bloquer la boucle d'événements asynchrone (Event Loop) du chatbot.

Une caractéristique majeure de ce composant est la gestion rigoureuse des stocks en temps réel. Dans le cadre du e-commerce, il est impératif d'éviter la "survente" (vendre plus d'articles qu'il n'y en a en stock). Pour cela, le `SQLiteAdapter` exploite les **propriétés ACID** (Atomicité, Cohérence, Isolation, Durabilité) offertes par SQLite [11]. Lors de l'ajout au panier (`add_to_cart`), une transaction atomique explicite (`BEGIN TRANSACTION`) est initiée. Le système vérifie la disponibilité (`stock_count > 0`), insère l'article dans le panier, et décrémente le stock (`UPDATE menu SET stock_count = stock_count - 1`), le tout dans une opération indivisible qui se termine par un `COMMIT` en cas de succès, ou un `ROLLBACK` en cas de rupture de stock.

3.5.2 OllamaAdapter

Cet adaptateur implémente le port `ILLMProvider`. Il encapsule la complexité de LangChain et d'Ollama. Son rôle est de :

- Initialiser le modèle local (ici *llama3.2 :1b*), configuré pour une faible consommation de RAM (`num_ctx=1512`).
- Construire le *PromptTemplate*, qui injecte les règles de conduite strictes, le contexte du menu formatté en liste à puces, l'historique récent et le message de l'utilisateur.
- Invoquer la chaîne d'exécution LangChain de manière asynchrone (`ainvoke`) pour générer la réponse sans bloquer le processus applicatif principal.

3.5.3 GroqAdapter et le Motif Factory

Afin de pallier les limitations de raisonnement des petits modèles exécutés localement, nous avons développé un deuxième adaptateur : le `GroqAdapter`. Cet adaptateur tire parti de l'API Cloud de Groq et de son architecture matérielle LPU (Language Processing Unit) [1], qui offre des temps d'inférence records par rapport aux GPUs traditionnels.

L'instanciation de `OllamaAdapter` ou de `GroqAdapter` est gérée dynamiquement au démarrage de l'application via la configuration (`ACTIVE_LLM`). Cette approche illustre une implémentation élégante du motif de conception **Factory Method** [7], permettant au cœur de l'application de manipuler l'intelligence artificielle de façon polymorphe, tout en respectant strictement le principe d'ouverture/fermeture (Open-Closed Principle).

3.5.4 InMemorySessionAdapter

Pour garder trace des conversations, cet adaptateur stocke l'historique dans la mémoire vive (RAM) en utilisant une structure `OrderedDict` pour limiter le nombre d'utilisateurs simultanés (afin d'éviter les fuites de mémoire) et restreint l'historique aux 3 derniers messages par utilisateur pour ne pas saturer la fenêtre de contexte du LLM.

3.5.5 TelegramAdapter

Cet adaptateur *primaire* (ou adaptateur "pilote") se connecte à l'API Telegram. Il reçoit les messages des utilisateurs, extrait le `user_id` et le texte, et appelle le moteur d'orchestration. Une fois que le moteur a décidé de la réponse, le TelegramAdapter se charge de l'expédier via `send_message`.

3.6 Le Moteur d'Orchestration (Engine) et Machine à États

Le module `engine.py` (`ChatbotEngine`) contient toute l'intelligence métier. Il reçoit des instances de tous les ports par **Injection de Dépendances** lors de son initialisation dans le `main.py`.

Son flux de traitement (`process_message`) suit une logique séquentielle précise couplée à une **Machine à États** simplifiée, essentielle pour traiter les processus d'achat qui s'étalent sur plusieurs messages.

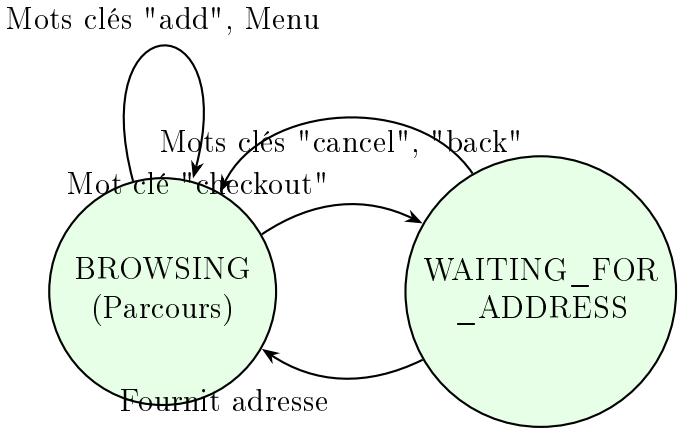

FIGURE 3.2 – Diagramme de la Machine à États de la Session Utilisateur

Le processus d'évaluation s'opère en 3 étapes (Branches) :

1. **Évaluation de l'État (Branche A)** : Le moteur vérifie si l'utilisateur est dans l'état `WAITING_FOR_ADDRESS`. Si oui, le texte reçu n'est pas envoyé au LLM, mais

intercepté comme étant une adresse postale. L'adresse est sauvegardée en base, la commande est confirmée, et l'utilisateur retourne à l'état `BROWSING`.

2. **Interception des Mots-Clés (Branche B et C)** : Si l'utilisateur est en `BROWSING`, le moteur utilise des expressions régulières (Regex) pour détecter des intentions d'achat explicites (ex : "cart", "checkout", "add"). Cette logique déterministe contourne le LLM pour des actions critiques (comme l'ajout effectif au panier en base de données), évitant ainsi que le LLM n'invente de fausses confirmations d'achat.
3. **Génération par LLM** : Si le message ne relève ni de l'état d'adresse ni d'une intention d'achat directe, le contexte métier (menu du jour récupéré via la BDD) et l'historique sont formatés et envoyés au port `ILLMProvider`. Le LLM génère alors une réponse conversationnelle naturelle et empathique.

3.7 Conclusion du Chapitre

L'architecture présentée dans ce chapitre met en lumière l'importance d'une structure logicielle découplée pour la conception d'applications basées sur l'intelligence artificielle générative. En isolant le modèle de langage et l'interface utilisateur derrière des ports abstraits, le système devient robuste face aux changements technologiques et hautement prédictible quant à la gestion des transactions e-commerce (panier, validation). La prochaine étape, détaillée dans le Chapitre 4, consiste à présenter l'implémentation concrète de cette architecture, le déploiement de l'environnement, ainsi que les tests et évaluations effectués pour valider notre approche.

Chapitre 4

Réalisation, Implémentation et Évaluation

4.1 Introduction

Après avoir défini les fondements théoriques (Chapitre 2) et posé l'architecture logicielle de notre système (Chapitre 3), ce chapitre se consacre à l'aspect pratique du projet. Nous détaillerons l'environnement de développement, la configuration matérielle et logicielle, ainsi que l'implémentation des algorithmes clés. Nous conclurons par une série de tests fonctionnels illustrant le comportement du chatbot en situation réelle, et nous analyserons objectivement les limites du système actuel.

4.2 Environnement de Développement

4.2.1 Outils et Technologies

L'ensemble du projet a été développé en utilisant le langage **Python (version 3.10+)**, choisi pour sa maturité, sa vaste bibliothèque standard, et son écosystème dominant dans le domaine de l'Intelligence Artificielle.

Les principales bibliothèques et frameworks utilisés sont :

— **LangChain** (`langchain_ollama` et `langchain_core`) : Pour l'orchestration du LLM local (Ollama).

— `groq` **(AsyncGroq)** : Client Python officiel pour l'intégration asynchrone à l'API cloud Groq LPU.

— `python-telegram-bot` : Un *wrapper* asynchrone complet autour de l'API HTTP de Telegram.

— `sqlite3` : Module intégré à Python pour interagir avec la base de données relationnelle locale.

— `asyncio` : Pour la gestion de la concurrence (I/O asynchrone), indispensable pour un bot devant traiter plusieurs utilisateurs simultanément sans bloquer.

4.2.2 Configuration des Modèles LLM : Local vs Cloud (Groq LPU)

Notre implémentation offre une approche hybride en supportant deux fournisseurs de LLMs, illustrant notre volonté d'allier confidentialité des données (IA locale) et très hautes performances (IA Cloud).

Dans un premier temps, nous avons opté pour le moteur d'exécution local **Ollama** avec le modèle `llama3.2:1b`. Ce modèle a été spécifiquement choisi pour s'adapter à des contraintes matérielles strictes.

Dans un second temps, pour pallier les limitations de raisonnement des petits modèles (1B), nous avons intégré l'API Cloud de **Groq**. Groq utilise des processeurs spécifiques appelés LPU (Language Processing Unit), permettant d'exécuter des modèles gigantesques tels que `llama-3.3-70b-versatile` à une vitesse spectaculaire. Afin de ne pas bloquer le fil d'exécution de l'application, les appels réseau sont effectués avec `AsyncGroq` :

```
self.client = AsyncGroq(api_key=Config.GROQ_API_KEY)
# ... plus loin ...
chat_completion = await self.client.chat.completions.create(
    model="llama-3.3-70b-versatile",
    messages=[...],
    temperature=0.3 # Temperature basse pour eviter l'hallucination
)
```

4.3 Détails d'Implémentation

4.3.1 Le Prompt Engineering et le RAG Simplifié

La clé de la pertinence des réponses du chatbot réside dans le *System Prompt*. Dans l'adaptateur Ollama (`OllamaAdapter`), nous définissons un modèle de prompt (`PromptTemplate`) strict :

```
template=(
    "System: You are an expert catering assistant.\n"
    "CRITICAL RULES:\n"
    "1. ONLY offer and discuss items explicitly listed in the [Live Menu] context.\n"
    "2. NEVER invent items, prices, or delivery times.\n"
    "3. STRICTLY reply in ENGLISH ONLY, regardless of the user's input language. Never use Arabic, French, or any other language.\n"
```

```
    "4. Be polite, natural, and concise.\n"
    "5. If the user asks to see the menu or what is available, YOU
   MUST explicitly list EVERY SINGLE ITEM provided in the [Live Menu]
    context. Do not summarize or skip items.\n\n"
    "[Live Menu & Logistics]:\n"
    "{context_data}\n\n"
    "[Recent History]:\n"
    "{user_history}\n\n"
    "User: {prompt}\n"
    "AI:"
)
```

Ce prompt implémente une forme simplifiée de *Retrieval-Augmented Generation (RAG)*. Avant chaque appel au LLM, le moteur interroge la base de données SQLite (`get_available_menu()`) et injecte dynamiquement le catalogue en temps réel dans la variable `{context_data}`. Pour éviter que le modèle ne tronque l'affichage, le menu est formaté sous la forme d'une liste à puces explicite (ex : `"- Pizza Margherita ($12.50)"`). Ainsi, le modèle connaît le stock exact et agit comme un synthétiseur rigoureux de la vérité contenue dans la base de données.

4.3.2 Transactions Atomiques pour la Gestion des Stocks

L'ajout d'un produit au panier requiert une attention particulière pour éviter que deux utilisateurs ne commandent le dernier article simultanément. Nous avons implémenté des **transactions SQLite explicites** pour assurer l'intégrité ACID des données :

```
cursor.execute("BEGIN TRANSACTION")
# Verification de stock (SELECT ... WHERE name = ?)
if stock_count <= 0:
    conn.rollback()
    return False
# Insertion dans le panier et decrement du stock
cursor.execute("INSERT INTO shopping_cart ...")
cursor.execute("UPDATE menu SET stock_count = stock_count - 1 ...")
conn.commit()
```

Cette logique empêche les conditions de concurrence critique et garantit qu'un achat ne sera finalisé que si le stock est physiquement disponible au moment précis du "commit".

4.3.3 Intégration de l'API Telegram Asynchrone

L'adaptateur Telegram utilise la classe `ApplicationBuilder`. Il déclare deux *Handlers* principaux :

— `CommandHandler("start")` : Répond à la commande initiale `/start`.

— `MessageHandler(filters.TEXT)` : Capture tout texte envoyé par l'utilisateur.

Afin d'éviter qu'une requête lente vers le LLM ne bloque le thread principal (et donc empêche d'autres utilisateurs de communiquer), les gestionnaires de requêtes utilisent intensivement le mot-clé `async/await` natif de Python, et les requêtes SQL (bloquantes par nature) sont déléguées à un thread séparé via `asyncio.to_thread()`.

4.4 Scénarios d'Utilisation et Évaluation

4.4.1 Scénario 1 : Découverte et Ajout au Panier

Lorsqu'un utilisateur exprime le souhait de commander un article, l'expression régulière intercepte l'intention. Si l'historique indique qu'il parlait d'un produit (ex : "Pizza"), et qu'il tape "add", le moteur déclenche `add_to_cart` en base de données.

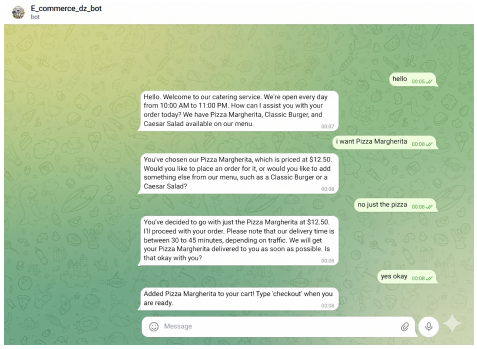

FIGURE 4.1 – Exemple de conversation avec le Chatbot E-commerce Telegram

Comme l'illustre la figure 4.1, le bot est capable de maintenir une conversation cohérente et de comprendre les requêtes de l'utilisateur.

4.4.2 Scénario 2 : Le Processus de Checkout (Machine à États)

Lorsque l'utilisateur tape le mot-clé "checkout" ou "cart", la Branche B du moteur d'orchestration prend le relais. 1. Le bot calcule la somme totale en base de données (`get_user_cart_summary`). 2. Il demande l'adresse de livraison et modifie l'état de l'utilisateur en `WAITING_FOR_ADDRESS`. 3. Au prochain message, au lieu de générer une réponse par IA, la Branche A intercepte le texte, l'enregistre comme adresse, valide la commande, et réinitialise l'état à `BROWSING`.

Cette approche hybride (IA générative + Machine à États stricte) empêche totalement le LLM d'halluciner des confirmations de commandes irréelles ou de manipuler malencontreusement le système de facturation.

4.5 Limites et Perspectives Techniques

Bien que fonctionnel, notre prototype présente certaines limites inhérentes à l'architecture locale et aux contraintes matérielles :

1. **Latence d'Inférence** : L'exécution du modèle Llama 3 (même quantifié et limité à 1B de paramètres) sur une machine grand public sans GPU dédié induit une latence de réponse variant de 2 à 5 secondes.
2. **Fenêtre de Contexte (Context Window)** : Limitée à 512 tokens pour préserver la RAM, cette restriction nous a obligés à tronquer l'historique de conversation (uniquement les 3 derniers messages stockés dans `InMemorySessionAdapter`). Les conversations longues perdent donc leur contexte lointain.
3. **Absence d'Embeddings Vectoriels** : La méthode de RAG utilisée injecte la totalité du catalogue en texte brut. Si la base de données contenait 10 000 produits, cette approche échouerait (saturation du contexte). Il faudrait implémenter une base vectorielle (ex : ChromaDB) pour effectuer une recherche de similarité sémantique préalable.

4.6 Conclusion du Chapitre

L'implémentation de notre solution démontre la faisabilité et la viabilité de notre architecture hexagonale. En combinant l'API asynchrone de Telegram, la robustesse relationnelle de SQLite, et la flexibilité intelligente d'Ollama, nous avons réussi à créer un chatbot e-commerce autonome. Les scénarios de tests confirment que l'hybridation entre une IA générative probabiliste et une logique métier déterministe (Machine à États) est la clé pour obtenir un système d'assistance à la fois naturel dans son dialogue et rigoureux dans ses transactions commerciales.

Conclusion Générale

Le commerce électronique moderne, caractérisé par une concurrence exacerbée et des exigences grandissantes de la part des consommateurs, impose aux entreprises de repenser en profondeur leurs stratégies de relation client. La réactivité, la disponibilité continue, et la personnalisation de l'expérience utilisateur sont devenues des impératifs économiques. C'est dans ce contexte de transformation numérique accélérée que ce mémoire s'est attaché à étudier, concevoir, et implémenter une solution d'automatisation intelligente du support client par l'intermédiaire d'un chatbot intégré à l'application de messagerie Telegram.

La problématique centrale de notre travail résidait dans le défi technique consistant à exploiter la puissance conversationnelle des Modèles de Langage de Grande Taille (LLMs) tout en garantissant la fiabilité transactionnelle exigée par un environnement e-commerce. Contrairement aux approches monolithiques classiques souvent observées dans les preuves de concept (PoC), nous avons opté pour une approche d'ingénierie logicielle rigoureuse.

Bilan des Réalisations

Sur le plan conceptuel, l'adoption de l'**Architecture Hexagonale** (Ports et Adaptateurs) a constitué le fondement de notre réussite. Cette architecture a permis d'isoler parfaitement la logique métier (le moteur d'orchestration et la machine à états) des contraintes d'infrastructure. Ainsi, le système est technologiquement agnostique : le remplacement de Telegram par un module web, ou d'Ollama par une API cloud propriétaire (comme OpenAI), peut s'opérer en modifiant exclusivement les adaptateurs de bordure, sans la moindre altération du cœur métier.

Sur le plan technique, nous avons développé un prototype fonctionnel complet en Python, intégrant dans un premier temps le moteur local **Ollama** (Llama 3.2 1B) pour garantir la confidentialité des données. Par la suite, grâce au patron de conception *Factory*, nous avons démontré la puissance de notre architecture en intégrant dynamiquement l'API Cloud **Groq** (Llama 3.3 70B). Cette évolution a permis d'augmenter drastiquement les capacités cognitives de l'assistant sans impacter la logique transactionnelle existante.

De plus, l'hybridation novatrice du moteur d'orchestration—qui combine la géné-

ration de texte probabiliste pour la conversation courante et des branchements stricts (expressions régulières et Machine à États transactionnelle ACID sur SQLite) pour les actions critiques (ajout au panier, paiement, adresse)—a permis de neutraliser efficacement le risque majeur d'hallucination et de survente inhérent aux systèmes purement génératifs.

Limites et Perspectives

Si notre prototype démontre la faisabilité technique et la pertinence de l'architecture, il n'en demeure pas moins perfectible. L'exécution initiale sur processeur CPU (Ollama) induisait des temps de latence et une fenêtre de contexte restreinte, problèmes que nous avons résolus par l'intégration de Groq LPU. Néanmoins, l'injection globale du catalogue (RAG simplifié par concaténation textuelle) n'est pas *scalable* pour un catalogue e-commerce comprenant des milliers de références.

Parmi les perspectives d'évolution majeures, nous envisageons :

1. **L'implémentation d'une Base de Données Vectorielle (VectorDB)** : Telle que ChromaDB ou FAISS, pour calculer des embeddings (plongements lexicaux) des produits, permettant une recherche sémantique ultra-rapide et l'injection d'un sous-ensemble pertinent du catalogue dans le contexte du LLM.
2. **L'intégration de passerelles de paiement sécurisées** : Connecter le bot à l'API de Stripe ou PayPal pour finaliser les transactions directement au sein de Telegram.
3. **Le multilinguisme automatique** : Exploiter les capacités innées des modèles multilingues pour offrir une assistance en français, arabe, ou anglais, selon la langue utilisée par le client de manière transparente.

En conclusion, l'intégration de l'Intelligence Artificielle générative au sein des architectures logicielles classiques redéfinit les contours du commerce en ligne. Les systèmes de demain ne se contenteront plus d'attendre passivement les clics de l'utilisateur sur une interface graphique ; ils viendront à sa rencontre à travers des expériences conversationnelles riches, proactives, et profondément humaines, tout en s'appuyant sur des fondations techniques immuablement rigoureuses.

Annexe A

Code Source du Projet

Ce chapitre d'annexe contient l'intégralité du code source développé pour ce mémoire, justifiant l'implémentation de l'Architecture Hexagonale et de la logique conversationnelle.

A.1 Le Domaine (`domain.py`)

Le domaine définit les entités pures de notre application, sans aucune dépendance externe.

```python
from dataclasses import dataclass
from datetime import datetime

@dataclass
class Customer:
    id: int
    telegram_id: str
    session_state: str

@dataclass
class Product:
    id: int
    name: str
    price: float
    is_available: bool
    stock_count: int

@dataclass
class UnifiedMessage:
    sender_id: str
    content: str
    timestamp: datetime
```

A.2 Les Ports / Interfaces (ports.py)

Les interfaces abstraites garantissent le découplage entre la logique métier et l'infrastructure.

```python
from abc import ABC, abstractmethod
from typing import List
from domain import Product

class ILLMProvider(ABC):
    @abstractmethod
    async def generate_reply(self, prompt: str, context_data: str, user_history: List[str]) -> str:
        """Generates a reply based on prompt, context data, and user history."""
        pass

class IDatabaseRepository(ABC):
    @abstractmethod
    async def get_available_menu(self) -> List[Product]:
        """Retrieves a list of available products."""
        pass

    @abstractmethod
    async def add_to_cart(self, telegram_id: str, product_name: str) -> bool:
        """Adds a product to the user's shopping cart."""
        pass

    @abstractmethod
    async def get_store_info(self) -> dict[str, str]:
        """Retrieves store logistics info like delivery times and hours."""
        pass

    @abstractmethod
    async def update_cart_address(self, telegram_id: str, address: str) -> bool:
        """Updates the address for the user's latest cart items where address is NULL."""
        pass

    @abstractmethod
    async def get_user_cart_summary(self, telegram_id: str) -> str:
        """Retrieves a formatted summary of the user's cart."""
        pass

```

```python
class IMessagingPlatform(ABC):
    @abstractmethod
    async def start_listening(self):
        """Starts the messaging listener."""
        pass

    @abstractmethod
    async def send_message(self, user_id: str, text: str):
        """Sends a message to the specific user."""
        pass

class IMemoryRepository(ABC):
    @abstractmethod
    async def update_memory(self, user_id: str, message: str) -> None:
        """Updates the memory history for a given user."""
        pass

    @abstractmethod
    async def get_history(self, user_id: str) -> List[str]:
        """Retrieves the memory history for a given user."""
        pass

    @abstractmethod
    async def set_user_state(self, user_id: str, state: str) -> None:
        """Sets the current state of the user."""
        pass

    @abstractmethod
    async def get_user_state(self, user_id: str) -> str:
        """Gets the current state of the user."""
        pass
```

A.3 Les Adaptateurs (adapters.py)

Ce fichier contient les implémentations spécifiques (SQLite, Telegram, Ollama, Mémoire en RAM).

```python
import sqlite3
import logging
import asyncio
from typing import List

from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler,
    MessageHandler, filters, ContextTypes

```

```python
from langchain_ollama import OllamaLLM
from langchain_core.prompts import PromptTemplate

from domain import Product
from ports import ILLMProvider, IDatabaseRepository, IMessagingPlatform, IMemoryRepository
from collections import OrderedDict
from config import Config
from groq import AsyncGroq

logger = logging.getLogger(__name__)

class SQLiteAdapter(IDatabaseRepository):
    def __init__(self, db_path: str = 'catering.db'):
        self.db_path = db_path

    def _query_db(self) -> List[Product]:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("SELECT id, name, price, is_available, stock_count FROM menu WHERE is_available = 1 AND stock_count > 0")
        rows = cursor.fetchall()
        products = [Product(id=row[0], name=row[1], price=row[2], is_available=bool(row[3]), stock_count=row[4]) for row in rows]
        conn.close()
        return products

    async def get_available_menu(self) -> List[Product]:
        try:
            # Run the synchronous sqlite3 operations in a separate thread
            products = await asyncio.to_thread(self._query_db)
            return products
        except Exception as e:
            logger.error(f"Database error: {e}")
            return []

    def _insert_cart(self, telegram_id: str, product_name: str) -> bool:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        try:
            cursor.execute("BEGIN TRANSACTION")

            # Find the price and stock of the product
            cursor.execute("SELECT price, stock_count FROM menu WHERE name = ? COLLATE NOCASE", (product_name,))
```

```
            row = cursor.fetchone()
            if not row:
                conn.rollback()
                return False

            price = row[0]
            stock_count = row[1]

            if stock_count <= 0:
                conn.rollback()
                return False

            cursor.execute(
                "INSERT INTO shopping_cart (telegram_id, product_name, price) VALUES (?, ?, ?)",
                (telegram_id, product_name, price)
            )

            cursor.execute(
                "UPDATE menu SET stock_count = stock_count - 1 WHERE name = ? COLLATE NOCASE",
                (product_name,)
            )

            conn.commit()
            return True
        except Exception as e:
            conn.rollback()
            logger.error(f"Transaction failed: {e}")
            return False
        finally:
            conn.close()

    async def add_to_cart(self, telegram_id: str, product_name: str) -> bool:
        try:
            success = await asyncio.to_thread(self._insert_cart, telegram_id, product_name)
            return success
        except Exception as e:
            logger.error(f"Database error inserting into cart: {e}")
            return False

    def _query_store_info(self) -> dict[str, str]:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("SELECT info_key, info_value FROM store_info")
```

```
93          rows = cursor.fetchall()
94          conn.close()
95          return {row[0]: row[1] for row in rows}
96
97      async def get_store_info(self) -> dict[str, str]:
98          try:
99              return await asyncio.to_thread(self._query_store_info)
100         except Exception as e:
101             logger.error(f"Database error fetching store info: {e}")
102             return {}
103
104     def _update_address(self, telegram_id: str, address: str) -> bool:
105         conn = sqlite3.connect(self.db_path)
106         cursor = conn.cursor()
107         cursor.execute(
108             "UPDATE shopping_cart SET address = ? WHERE telegram_id =
    ? AND address IS NULL",
109             (address, telegram_id)
110         )
111         conn.commit()
112         updated = cursor.rowcount > 0
113         conn.close()
114         return updated
115
116     async def update_cart_address(self, telegram_id: str, address: str
    ) -> bool:
117         try:
118             return await asyncio.to_thread(self._update_address,
    telegram_id, address)
119         except Exception as e:
120             logger.error(f"Database error updating cart address: {e}")
121             return False
122
123     def _query_cart_summary(self, telegram_id: str) -> str:
124         conn = sqlite3.connect(self.db_path)
125         cursor = conn.cursor()
126         cursor.execute(
127             "SELECT product_name, COUNT(*), SUM(price) FROM
    shopping_cart WHERE telegram_id = ? GROUP BY product_name",
128             (telegram_id,)
129         )
130         rows = cursor.fetchall()
131         conn.close()
132
133         if not rows:
134             return "Your cart is empty."
135
```

```
136         summary_lines = []
137         total_price = 0.0
138         for row in rows:
139             product_name = row[0]
140             count = row[1]
141             total = row[2]
142             summary_lines.append(f"{count}x {product_name} - ${total
    :.2f}")
143             total_price += total
144 
145         summary_lines.append(f"Total: ${total_price:.2f}")
146         return "\n".join(summary_lines)
147 
148     async def get_user_cart_summary(self, telegram_id: str) -> str:
149         try:
150             return await asyncio.to_thread(self._query_cart_summary,
    telegram_id)
151         except Exception as e:
152             logger.error(f"Database error fetching cart summary: {e}")
153             return "Error retrieving cart."
154 
155 class InMemorySessionAdapter(IMemoryRepository):
156     def __init__(self, max_users: int = 100, max_history_per_user: int
    = 3):
157         self.user_memory = OrderedDict()
158         self.user_states = {}
159         self.max_users = max_users
160         self.max_history_per_user = max_history_per_user
161 
162     async def update_memory(self, user_id: str, message: str) -> None:
163         if user_id in self.user_memory:
164             self.user_memory.move_to_end(user_id)
165         else:
166             if len(self.user_memory) >= self.max_users:
167                 # Remove the oldest user session
168                 self.user_memory.popitem(last=False)
169             self.user_memory[user_id] = []
170 
171         self.user_memory[user_id].append(message)
172 
173         # Keep ONLY the last N messages
174         if len(self.user_memory[user_id]) > self.max_history_per_user:
175             self.user_memory[user_id].pop(0)
176 
177     async def get_history(self, user_id: str) -> List[str]:
178         return self.user_memory.get(user_id, [])
179 
```

```
180     async def set_user_state(self, user_id: str, state: str) -> None:
181         self.user_states[user_id] = state
182 
183     async def get_user_state(self, user_id: str) -> str:
184         return self.user_states.get(user_id, "BROWSING")
185 
186 class OllamaAdapter(ILLMProvider):
187     def __init__(self):
188         # Initialize Ollama strictly with low RAM constraints
189         self.llm = OllamaLLM(
190             model="llama3.2:1b",
191             num_ctx=1512,
192             num_thread=2
193         )
194 
195         self.prompt_template = PromptTemplate(
196             input_variables=["context_data", "user_history", "prompt"],
197             template=(
198                 "System: You are an expert catering assistant.\n"
199                 "CRITICAL RULES:\n"
200                 "1. NEVER assume an order. If the user says \"pizza\" or \"burger\", list the specific options from the [Menu] and ask which one they want.\n"
201                 "2. If the user asks for something NOT on the menu, apologize and offer an available alternative.\n"
202                 "3. ALWAYS reply in English.\n"
203                 "4. Be concise and polite.\n"
204                 "5. If the user asks to see the menu or what is available, YOU MUST explicitly list EVERY SINGLE ITEM provided in the [Menu] context. Do not summarize or skip items.\n\n"
205                 "[Menu & Logistics]:\n"
206                 "{context_data}\n\n"
207                 "[Recent History]:\n"
208                 "{user_history}\n\n"
209                 "User: {prompt}\n"
210                 "AI:"
211             )
212         )
213 
214         self.chain = self.prompt_template | self.llm
215 
216     async def generate_reply(self, prompt: str, context_data: str, user_history: List[str]) -> str:
217         history_str = "\\n".join(user_history) if user_history else "No previous history."
218         try:
```

```
            response = await self.chain.ainvoke({
                "context_data": context_data,
                "user_history": history_str,
                "prompt": prompt
            })
            return response
        except Exception as e:
            logger.error(f"LLM generation error: {e}")
            return "I apologize, but I am currently having trouble processing your request. Please try again later."

class GroqAdapter(ILLMProvider):
    def __init__(self):
        self.client = AsyncGroq(api_key=Config.GROQ_API_KEY)

    async def generate_reply(self, prompt: str, context_data: str, user_history: List[str]) -> str:
        history_str = "\n".join(user_history) if user_history else "No previous history."

        system_message = (
            "You are an elite E-commerce AI assistant for a catering service. CRITICAL RULES: "
            "1. ONLY offer and discuss items explicitly listed in the [Live Menu] context. "
            "2. NEVER invent items, prices, or delivery times. "
            "3. STRICTLY reply in ENGLISH ONLY, regardless of the user's input language. Never use Arabic, French, or any other language. "
            "4. Be polite, natural, and concise. \n\n"
            f"[Live Menu & Logistics]:\n{context_data}"
        )

        user_message = f"[Recent History]:\n{history_str}\n\nUser: {prompt}"

        try:
            chat_completion = await self.client.chat.completions.create(
                model="llama-3.3-70b-versatile",
                messages=[
                    {"role": "system", "content": system_message},
                    {"role": "user", "content": user_message}
                ],
                temperature=0.3
            )
            return chat_completion.choices[0].message.content
```

```
257         except Exception as e:
258             logger.error(f"Groq API error: {e}")
259             return "I apologize, but I am currently experiencing
    technical difficulties. Please try again in a moment."
260
261 class TelegramAdapter(IMessagingPlatform):
262     def __init__(self, token: str):
263         self.token = token
264         self.app = ApplicationBuilder().token(self.token).build()
265         self.engine = None # Will be injected later
266
267         self.app.add_handler(CommandHandler("start", self.handle_start
    ))
268         self.app.add_handler(MessageHandler(filters.TEXT & (~filters.
    COMMAND), self.handle_message))
269
270     def set_engine(self, engine):
271         self.engine = engine
272
273     async def handle_start(self, update: Update, context: ContextTypes
    .DEFAULT_TYPE):
274         await update.message.reply_text("Welcome to our Catering
    Service! How can I help you today?")
275
276     async def handle_message(self, update: Update, context:
    ContextTypes.DEFAULT_TYPE):
277         user_id = str(update.effective_user.id)
278         text = update.message.text
279         if self.engine:
280             # We don't want to block the handler for too long, so we
    process it.
281             # But await is necessary for sequential processing per
    user.
282             await self.engine.process_message(user_id, text)
283         else:
284             await self.send_message(user_id, "System is not fully
    initialized.")
285
286     async def start_listening(self):
287         logger.info("Starting Telegram Bot asynchronously...")
288         await self.app.initialize()
289         await self.app.start()
290         await self.app.updater.start_polling()
291         # Wait forever
292         await asyncio.Event().wait()
293
294     async def send_message(self, user_id: str, text: str):
```

```
295         try:
296             await self.app.bot.send_message(chat_id=user_id, text=text
    )
297         except Exception as e:
298             logger.error(f"Failed to send message to {user_id}: {e}")
```

A.4 Le Moteur d'Orchestration (engine.py)

Le cœur logique du système qui traite les messages et orchestre les flux.

```
1  import logging
2  from ports import ILLMProvider, IDatabaseRepository,
       IMessagingPlatform, IMemoryRepository
3
4  logger = logging.getLogger(__name__)
5
6  import re
7
8  class ChatbotEngine:
9      def __init__(self, llm: ILLMProvider, db: IDatabaseRepository,
       messaging: IMessagingPlatform, memory: IMemoryRepository):
10         self.llm = llm
11         self.db = db
12         self.messaging = messaging
13         self.memory = memory
14
15     async def process_message(self, user_id: str, text: str):
16         try:
17             logger.info(f"Processing message from {user_id}: {text}")
18
19             # 1. Fetch available menu and store info
20             products = await self.db.get_available_menu()
21             store_info = await self.db.get_store_info()
22
23             if not products:
24                 menu_context = "No products are currently available."
25             else:
26                 menu_context = "\n".join([f"- {p.name} (${p.price:.2f
       })" for p in products])
27
28             logistics_context = " ".join([f"{k.replace('_', ' ').title
       ()}: {v}" for k, v in store_info.items()])
29             context_data = f"Available Menu: {menu_context}\nStore
       Logistics: {logistics_context}"
30
31             # 2. Get user history
```

```
32            user_history = await self.memory.get_history(user_id)
33
34            text_lower = text.lower()
35            words_in_text = re.findall(r'\b\w+\b', text_lower)
36
37            reply = None
38
39            # Branch A: State Machine Escape & Address Handling
40            user_state = await self.memory.get_user_state(user_id)
41
42            if user_state == "WAITING_FOR_ADDRESS":
43                if any(word in words_in_text for word in ["cancel", "
    menu", "back", "stop"]):
44                    await self.memory.set_user_state(user_id, "
    BROWSING")
45                else:
46                    await self.db.update_cart_address(user_id, text)
47                    await self.memory.set_user_state(user_id, "
    BROWSING")
48                    reply = "Thank you! Your address is saved and your
     order is confirmed. It will arrive as per our schedule."
49
50            if not reply:
51                # Branch B: The "Checkout / Cart" Intent
52                if any(word in words_in_text for word in ["cart", "
    checkout", "total", "bill", "pay"]):
53                    cart_summary = await self.db.get_user_cart_summary
    (user_id)
54                    if "empty" in cart_summary.lower():
55                        reply = "Your cart is currently empty. Would
    you like to see the menu?"
56                    else:
57                        reply = f"Here is your order summary:\n{
    cart_summary}\n\nTo confirm and place this order, please type your
     exact delivery address."
58                        await self.memory.set_user_state(user_id, "
    WAITING_FOR_ADDRESS")
59
60                # Branch C: Exact Word "Add" Intent
61                elif any(word in words_in_text for word in ["yes", "
    add", "confirm", "sure"]) and user_history:
62                    last_messages = " ".join(user_history[-2:]).lower
    ()
63                    for p in products:
64                        name_words = p.name.lower().split()
65                        if all(word in last_messages for word in
    name_words):
```

```
66                              success = await self.db.add_to_cart(
    user_id, p.name)
67                              if success:
68                                  reply = f"Added {p.name} to your cart!
     Type 'checkout' when you are ready."
69                              else:
70                                  reply = f"Sorry, {p.name} is
    completely sold out for today!"
71                              break
72
73              # 3. Generate Reply if not already handled
74              if not reply:
75                  reply = await self.llm.generate_reply(prompt=text,
    context_data=context_data, user_history=user_history)
76
77                  # Preserve logistics auto-address logic
78                  logistics_keywords = ["when", "time", "long", "where",
     "delivery", "hours"]
79                  if any(word in words_in_text for word in
    logistics_keywords):
80                      reply += "\n\nTo proceed, please type your exact
    delivery address."
81                      await self.memory.set_user_state(user_id, "
    WAITING_FOR_ADDRESS")
82
83              # 4. Send Message (Only if messaging platform is properly
    configured)
84              if self.messaging:
85                  await self.messaging.send_message(user_id, reply)
86              else:
87                  # If messaging is missing (e.g. during test), just log
     it
88                  logger.info(f"Reply generated (no messaging platform):
     {reply}")
89
90              # 5. Update Memory with the conversation
91              await self.memory.update_memory(user_id, f"User: {text}")
92              await self.memory.update_memory(user_id, f"AI: {reply}")
93
94              # Return the reply in case a caller needs it directly (e.g
    . tests)
95              return reply
96
97          except Exception as e:
98              logger.error(f"Error processing message from {user_id}: {e
    }")
99              if self.messaging:
```

```
100                 await self.messaging.send_message(user_id, "Sorry, I
    encountered an internal error. Please try again.")
101             return "Error occurred"
```

A.5 Le Point d'Entrée (main.py)

Le script de démarrage qui assemble les composants via l'injection de dépendances.

```
1  import logging
2  import asyncio
3  from adapters import SQLiteAdapter, OllamaAdapter, GroqAdapter,
       TelegramAdapter, InMemorySessionAdapter
4  from engine import ChatbotEngine
5  from config import Config
6
7  # Ensure consistent logging configuration
8  logging.basicConfig(
9      level=logging.INFO,
10     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
11 )
12 logger = logging.getLogger(__name__)
13
14 async def main():
15     logger.info("Initializing AI Chatbot System...")
16
17     # 1. Initialize Adapters
18     db_adapter = SQLiteAdapter()
19
20     if Config.ACTIVE_LLM == "groq":
21         logger.info("Starting with Groq API...")
22         llm_adapter = GroqAdapter()
23     else:
24         logger.info("Starting with Local Ollama...")
25         llm_adapter = OllamaAdapter()
26
27     memory_adapter = InMemorySessionAdapter()
28
29     # Load token from config
30     BOT_TOKEN = Config.BOT_TOKEN
31     telegram_adapter = TelegramAdapter(token=BOT_TOKEN)
32
33     # 2. Initialize Engine with Dependency Injection
34     engine = ChatbotEngine(llm=llm_adapter, db=db_adapter, messaging=
       telegram_adapter, memory=memory_adapter)
35
36     # 3. Wire Engine into Telegram Adapter
```

```
37      telegram_adapter.set_engine(engine)
38
39      # 4. Start Listening
40      await telegram_adapter.start_listening()
41
42  if __name__ == "__main__":
43      try:
44          asyncio.run(main())
45      except KeyboardInterrupt:
46          logger.info("Bot stopped by user.")
47      except Exception as e:
48          logger.error(f"Critical error: {e}")
```

Bibliographie

[1] Dennis Abts, John Ross, Jonathan Harvey, et al. Think fast : A tensor streaming processor (tsp) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158. IEEE, 2020.

[2] Dave Chaffey and Fiona Ellis-Chadwick. *Digital marketing*. Pearson uk, 2020.

[3] Harrison Chase. Langchain : Building applications with llms through composability. *Whitepaper*, 2023.

[4] N Chintalapudi et al. The role of artificial intelligence in reshaping e-commerce customer experience. *Journal of Retailing and Consumer Services*, 2025.

[5] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[6] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2012.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Pearson Education, 1995.

[8] PA Landim et al. Chatbots in e-commerce : A systematic literature review. *International Journal of Retail & Distribution Management*, 2021.

[9] Kenneth C Laudon and Carol Guercio Traver. *E-commerce 2021-2022 : business. technology. society*. Pearson, 2021.

[10] Ollama. Ollama : Get up and running with large language models locally, 2024. URL https://ollama.com.

[11] Grant Allen Owens. *The definitive guide to SQLite*. Apress, 2006.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[13] WayneXinZhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. arXiv preprint arXiv :2303.18223, 2023.

[14] Adamopoulou, E., & Moussiades, L. (2020). Chatbots: History, technology, and applications. Machine Learning with Applications, 2, 100006.

[15] Davenport, T., Guha, A., Grewal, D., & Bressgott, T. (2020). How artificial intelligence will change the future of marketing. Journal of the Academy of Marketing Science, 48(1), 24-42.

[16] Hoyer, W. D., Kroschke, M., Schmitt, B., Kraume, K., & Shankar, V. (2020). Transforming the customer experience through new technologies. Journal of Interactive Marketing, 51, 57-71.

[17] Lemon, K. N., & Verhoef, P. C. (2016). Understanding customer experience throughout the customer journey. Journal of Marketing, 80(6), 69-96.

[18] Rese, A., Ganster, L., & Baier, D. (2020). Chatbots in customer service: user acceptance and motivation. Journal of Retailing and Consumer Services, 56, 102176.

[19] Zumstein, D., & Hundertmark, S. (2017). Chatbots--An interactive technology for personalized communication, transactions and services. IADIS International Journal on WWW/Internet, 15(1), 96-109.

[20] Chung, M., Ko, E., Joung, H., & Kim, S. J. (2020). Chatbot e-service and customer satisfaction regarding luxury brands. Journal of Business Research, 117,587-595.

[21] Grewal, D., Roggeveen, A. L., & Nordfält, J. (2017). The future of retailing. Journal of Retailing, 93(1), 1-6.

[22] Jurafsky, D., & Martin, J. H. (2023). Speech and Language Processing (3rd ed. draft). Pearson. (Cite this in Section 2.3 for general NLP definitions).

[23] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459-9474. (Crucial for your Section 2.5.4).

[24] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M. A., ... & Lample, G. (2023). LLaMA: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971. (Cite this when discussing Llama 3/Ollama).

[25] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems, 33, 1877-1901.

[26] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[27] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824-24837. (Cite this in your Prompt Engineering section).

[28] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., ... & Lowe, R. (2022). Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems, 35, 27730-27744. (Cite this in Section 2.4.2 when mentioning RLHF).

[29] Gao, J., Galley, M., & Li, L. (2019). Neural approaches to conversational AI. Foundations and Trends® in Information Retrieval, 13(2-3), 127-298.

[30] Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., ... & Zhang, Y. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv preprint arXiv:2303.12712.

[31] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. Advances in Neural Information Processing Systems, 35, 22199-22213.

[32] Percival, H., & Gregory, B. (2020). Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices. O'Reilly Media. (This is the absolute best book for your exact project—cite it in Chapter 3).

[33] Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall. (Cite this alongside Cockburn [6] in Section 3.2).

[34] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

[35] Freeman, S., & Pryce, N. (2009). Growing Object-Oriented Software, Guided by Tests. Addison-Wesley Professional.

[36] Richards, M., & Ford, N. (2020). Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media.

[37] Ramalho, L. (2021). Fluent Python: Clear, Concise, and Effective Programming. O'Reilly Media. (Cite this in Section 4.2.1 when defending your use of Python).

[38] Lott, S. F. (2018). Mastering Object-Oriented Python. Packt Publishing Ltd.

[39] Grinberg, M. (2018). Flask Web Development: Developing Web Applications with Python. O'Reilly Media.

[40] Bayer, M. (2012). SQLAlchemy. In The Architecture of Open Source Applications (Vol. 2). (Cite this in Section 3.5.1 when discussing Python database wrappers vs. raw SQLite).